Shared-Memory Parallelism Can Be Simple, Fast, and Scalable

Julian Shun

CMU-CS-15-108

May 2015

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This work is supported by the National Science Foundation under grant numbers CCF-1018188 and CCF-1314590, the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program, the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and a Facebook Graduate Fellowship.

Keywords: Parallel Computing, Shared-Memory, Multicore, Programming Techniques, Programming Frameworks, Large-Scale, Deterministic Parallelism, Graph Algorithms, String Algorithms

To my family.

Abstract

Parallelism is the key to achieving high performance in computing. However, writing efficient and scalable parallel programs is notoriously difficult, and often requires significant expertise. To address this challenge, it is crucial to provide programmers with high-level tools to enable them to develop solutions efficiently, and at the same time emphasize the theoretical and practical aspects of algorithm design to allow the solutions developed to run efficiently under all possible settings. This thesis addresses this challenge using a three-pronged approach consisting of the design of shared-memory programming techniques, frameworks, and algorithms for important problems in computing. The thesis provides evidence that with appropriate programming techniques, frameworks, and algorithms, shared-memory programs can be simple, fast, and scalable, both in theory and in practice. The results developed in this thesis serve to ease the transition into the multicore era.

The first part of this thesis introduces tools and techniques for deterministic parallel programming, including means for encapsulating nondeterminism via powerful commutative building blocks, as well as a novel framework for executing sequential iterative loops in parallel, which lead to deterministic parallel algorithms that are efficient both in theory and in practice.

The second part of this thesis introduces Ligra, the first high-level shared-memory framework for parallel graph traversal algorithms. The framework allows programmers to express graph traversal algorithms using very short and concise code, delivers performance competitive with that of highly-optimized code, and is up to orders of magnitude faster than existing systems designed for distributed memory. This part of the thesis also introduces Ligra+, which extends Ligra with graph compression techniques to reduce space usage and improve parallel performance at the same time, and is also the first graph processing system to support in-memory graph compression.

The third and fourth parts of this thesis bridge the gap between theory and practice in parallel algorithm design by introducing the first algorithms for a variety of important problems on graphs and strings that are efficient both in theory and in practice. For example, the thesis develops the first linear-work and polylogarithmic-depth algorithms for suffix tree construction and graph connectivity that are also practical, as well as a work-efficient, polylogarithmic-depth, and cache-efficient shared-memory algorithm for triangle computations that achieves a 2–5x speedup over the best existing algorithms on 40 cores.

Acknowledgments

There are many people without whom this thesis would not have been possible, and I would like to thank all of them.

First and foremost, I thank my advisor Guy Blelloch for the guidance and inspiration that he gave me during my graduate studies. He introduced me to parallel computing, taught me the knowledge necessary for writing this thesis, and gave me a lot of useful career advice. I am very grateful to Guy for spending hours with me every week answering all of my questions, going through problems on the board, and even helping me improve my code.

I am also thankful to the rest of my thesis committee members for helping me throughout my graduate studies and providing me with useful feedback to improve this thesis. I thank Phil Gibbons for being a great collaborator, always sparking interesting discussions during our regular research meetings. His broad knowledge has given me a better perspective on the "big picture". I also thank Phil for listening to many of my practice talks and giving me advice on job applications. I thank Jeremy Fineman for being an excellent collaborator, and his expertise in algorithms helped me in developing many of the results in this thesis. I am grateful to Christos Faloutsos for many interesting discussions on graphs, which inspired me to do research on large-scale graph processing. I thank Gary Miller for our many interesting conversations on algorithm design, and for teaching me about parallel algorithms, spectral graph theory, and computational geometry. Finally, I thank Charles Leiserson for hosting my visit to MIT and encouraging me to think more about the importance of my thesis statement.

During my graduate studies, I also had the opportunity to interact with many other faculty members at CMU. I had the opportunity to work as a teaching assistant for Umut Acar, Todd Mowry, Margaret Reid-Miller, Anthony Rowe, and Danny Sleator, and from them I learned how to become more effective at teaching. I am grateful to Umut for trusting me to give a lecture in class and helping me prepare for it, as well as helping me with job applications. I thank Kayvon Fatahalian, Alan Frieze, Anupam Gupta, and Frank Pfenning for helping me with my speaking and writing skills requirements.

I thank my fellow CMU students Laxman Dhulipala, Yan Gu, Aapo Kyrola, Richard

Peng, Harsha Simhadri, Yihan Sun, Kanat Tangwongsan, and Fuyao Zhao for valuable discussions and research collaborations. I am thankful to all of the friends I met at CMU for making my Ph.D. journey more enjoyable. Thanks to Jason Chen, Zhuo Chen, Dalong Cheng, Vincent Chu, Ina Fiterau, Yu Gong, Favonia Hou, Tzu-Kuo Huang, Antonio Juarez, Gunhee Kim, Haison Le, Seunghak Lee, Nan Li, Rui Liu, Qian Mao, Xi Tan, Yu-Ting Weng, Gus Xia, Lianghong Xu, Min Xu, Yang Xu, Shoou-I Yu, Huanchen Zhang, Yin Zhang, Yingjie Zhang, Yuan Zhou, and many others.

I am especially thankful to my wonderful girlfriend, Wenlu Hu. She stood by my side during all of my ups and downs, and her constant support and encouragement enabled me to finish writing this thesis. Finally, I am grateful to my parents and sister for always motivating me to pursue my dreams and being there whenever I needed them. Without them, I would not have come this far.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's data-driven world with rapidly increasing data sizes, performance has become more important than ever before. Reducing the running time of programs lowers overall costs—for example, the rental costs of machines on Amazon EC2[1] is proportional to the usage time. In addition, reducing the time-to-completion of tasks has been shown to increase worker productivity as well as end-user experience. Alternatively, one can view improving performance as enabling more computation to be performed in a given amount of time, effectively increasing one's computing budget.

Traditionally, high-performance computing solutions have been developed and used by only a small community, as these solutions rely on expensive and specialized computing environments. In recent years, in an effort to bring performance computing closer to the rest of the community, large-scale computing solutions using distributed clusters of commodity machines have emerged. However, within the past decade, commodity multicore machines have become prevalent, and today these machines support up to terabytes of memory,[2] more than enough for a majority of applications. *This thesis contends that a single shared-memory machine is sufficient for solving many problems in large-scale computing.* The thesis shows that large-scale shared-memory solutions can be simple, scalable to the largest data sets considered by distributed-memory solutions for many problems, and significantly more efficient on a per-core, per-dollar, and per-joule basis than existing distributed-memory solutions. *The goal of this thesis is to bring high-performance computing to the masses via parallel programming frameworks, techniques, and algorithms for shared-memory multicore machines.*

Why have multicore machines become so widespread in just the past decade? Moore's

[1] `http://aws.amazon.com/ec2/pricing/`

[2] For example, the Intel Sandy Bridge-based Dell PowerEdge R920 can be configured with up to 60 cores and 6 Terabytes of memory.

law states that the transistor density doubles approximately every 18 months [337], and along with Dennard scaling, which states that transistor power density is constant [132], this has historically corresponded to increases in clock speeds of single core machines of roughly 30% per year since the mid-1970's [295]. However, since around the mid-2000's, Dennard scaling no longer continued to hold due to physical limitations of hardware, and as a result hardware vendors have turned to developing processors with multiple cores to deliver improved performance. These machines are referred to as *shared-memory multicore machines*,[3] as the different cores have access to a shared global memory. This shift in processor technology has often been referred to as the "multicore revolution" [295]. Multicore technology has become ubiquitous today, with most personal computers, and even most cellular phones containing multiple cores. Therefore, writing parallel programs to take advantage of the multiple cores on a machine is crucial to obtaining scalable performance and enabling large-scale data to be processed.

In addition to multicore technology, parallel computing can come in the form of distributed systems as mentioned above, graphics processing units (GPUs), and field programmable gate arrays (FPGAs). Unlike multicores, distributed systems can solve problems that do not fit in the memory of a single machine. However, compared to multicore shared-memory systems, communication and data replication in distributed systems often leads to high additional overheads. Therefore, for problems that can fit in memory, shared-memory multicores are generally significantly more efficient on a per-core, per-dollar, and per-joule basis than distributed-memory systems. For example, this thesis shows that the exact triangle count of the Yahoo! Web graph with over 6 billion edges can be computed in under 1.5 minutes and a suffix tree can be constructed on the 3 gigabyte human genome in under three minutes on a modern 40-core machine, much faster than previous distributed-memory solutions (both in absolute performance and on a per-core basis) for the same problem. The data sets in these examples are among the largest considered in the literature for the corresponding problems, and easily fit on a multicore machine. While GPUs and FPGAs may be more efficient for certain problems, multicore machines are much more general-purpose, support larger memory sizes (useful for scaling to large data), and are considerably easier to program.[4] This thesis argues that shared-memory multicores offer a *sweet spot* between programmability and efficiency. There has been a large body of work on developing efficient algorithms and frameworks for regular problems, where the parallelism is relatively well-structured (e.g., problems in dense numerical linear algebra and scientific simulations), while less work has been done for irregular problems,

[3]These are sometimes also referred to as *manycore* machines when the number of cores is large enough.

[4]The techniques developed in this thesis are also applicable to Intel's new Xeon Phi coprocessors, which support higher memory bandwidth than traditional multicore machines. However, currently their memory sizes are not sufficient for some of the larger data sets studied in this thesis.

where the parallelism is much less well-structured and highly dependent on input data (e.g., problems on graphs and strings). *This thesis studies shared-memory programming techniques, frameworks, and algorithms for a wide class of irregular problems and shows that shared-memory parallelism can be simple, fast, and scalable.*

The thesis adopts a three-pronged approach of studying shared-memory parallelism from the perspective of programming techniques, algorithm design, and performance analysis. Furthermore, significant attention will be paid to both the theoretical aspects as well as the practical implications of the solutions developed. The work in this thesis builds on ideas from previous research on shared-memory parallelism, but the comprehensive approach used in the thesis enables simplicity, efficiency, and scalability, both in theory and in practice, to be achieved for a variety of important problems for the first time. The remainder of this chapter is organized as follows:

- Section 1.1 introduces nested fork-join parallelism, which is the type of parallelism this thesis studies. This section then describes challenges in shared-memory programming, including obtaining determinism, controlling shared access, and developing high-level programming abstractions. The reader will obtain an overview of the contributions of this thesis to addressing these challenges.

- Section 1.2 describes the Parallel Random Access Machine (PRAM) and work-depth models for analyzing parallel algorithms. This is followed by some highlights of the thesis's contribution in bridging the gap between theory and practice in parallel algorithms via designing theoretically-efficient algorithms that perform well on modern multicore machines.

- Section 1.3 describes performance factors of multicore programs, including caching, memory contention, scalability, and memory bandwidth. This section introduces techniques developed in this thesis that take into account these factors to improve performance.

- Section 1.4 introduces a benchmark suite developed in this thesis to comprehensively evaluate solutions to given problems in terms of simplicity as well as theoretical and practical efficiency.

- The thesis statement is presented in Section 1.5.

- The contributions of this thesis are summarized in Section 1.6.

1.1 Shared-Memory Programming

Languages. While shared-memory parallelism has many advantages, writing correct, efficient, and scalable shared-memory multicore programs is notoriously difficult. Traditionally, shared-memory parallel programs are written with explicit assignment of tasks to threads (e.g., using `pthreads`). This low-level approach requires the programmer to carefully consider the many possible interleavings of threads, and it is generally difficult to write a correct program let alone an efficient and scalable one. For programs in which there is no clear way to evenly split the work among threads, scheduling for good performance is a big challenge. Such programs generally require extensive tuning to obtain good performance.

Another method for writing shared-memory multicore programs is to use simple constructs that indicate which parts of the program are safe to run in parallel, and allow a run-time scheduler to assign work to threads and perform load balancing on-the-fly. This approach is known as *dynamic multithreading*. Using languages such as Cilk [158], OpenMP [360], Intel Threading Building Blocks [237], Habanero [76], and X10 [88] that support dynamic multithreading, one can write clean programs while letting the run-time scheduler perform the work allocation and load balancing. This approach frees the programmer from the low-level details of explicit thread management, leading to simpler code, while delivering comparable or improved performance. With advances in scheduling, it is now possible to write a wide class of parallel programs in this framework that are efficient, both in theory and in practice [65], without having to tune the program to achieve balanced workloads.

Nested Fork-Join Parallelism. All of the algorithms and techniques studied in this thesis are designed for *nested fork-join parallelism*, in which procedures can be called recursively in parallel via a *fork* construct, and synchronized via a *join* construct [49]. Nested parallel computations can be defined inductively in terms of the composition of sequential and parallel components, and modeled as a directed acyclic graph (computation DAG). Dynamic multithreading languages such as Cilk support low-overhead primitives to implement fork-join parallelism [294]. A broad class of parallel programs can be expressed with fork-join parallelism, and the programming techniques and frameworks developed in this thesis aim to enable programs written within this paradigm to be simpler and more efficient.

Determinism. While dynamic multithreading languages free the programmer from scheduling and load balancing, there are still many challenges in writing correct and fast parallel programs. *One of the key challenges in parallel programming is dealing with nondeterminism arising from the parallel program and/or the parallel machine and its runtime environment.* Nondeterminism arises from race conditions in the program (concurrent accesses to the same data with at least one being a write), and makes it hard for programmers to debug and reason about the correctness/performance of their code. One way to obtain determinism in nested parallel programs is to not have any races. While this approach is reasonable for certain problems, in general it can be overly restrictive as it is often useful and efficient to have shared data. The goal in this thesis is to develop less restrictive and more efficient ways to obtain determinism.

There has been significant previous work on obtaining determinism using various approaches, including using special-purpose hardware, modifying compilers, runtime systems and/or operating systems, and designing new programming languages (see Chapter 3 for references). In contrast to most previous work, this thesis designs building blocks and programming techniques for simplifying deterministic parallel programming that can be used with the *existing computing stack*, making determinism more accessible. In other words, programmers do not have to install special programming languages, compilers, runtime systems or operating systems, nor do they need access to special-purpose hardware. This thesis advocates a form of determinism called *internal determinism*. Informally, given an abstraction level, a program is internally deterministic if key intermediate steps of the program are deterministic with respect to the abstraction level. Internal determinism has many benefits, including leading to external determinism and implying a sequential semantics, which in turn leads to many advantages such as ease of reasoning about code, verifying correctness and debugging.

One of the main approaches to developing efficient deterministic parallel solutions in this thesis is the *deterministic reservations* framework for parallelizing greedy sequential algorithms (Chapter 3). The approach consists of two phases—in the *reserve* phase, the iterates concurrently mark all of the data that they affect, and in the *commit* phase, iterates whose mark is still written on all of its affected data proceed with the computation on the data. Determining successful reservations is done in a deterministic manner, so that for a given round the same iterates succeed/fail on every execution. Parallel algorithms written in this framework return the same answer as their sequential counterparts, which gives determinism, and allows the parallel and sequential algorithms to be interchanged when necessary. The algorithms developed are also very simple, as the user only needs to specify the `reserve` and `commit` functions called by each iterate in the two corresponding phases, as well as corresponding data structures. For example, Figure 1.1 shows the `C++` code for a spanning forest algorithm using deterministic reservations. `disjointSet` is a deterministic union-find data structure developed in this thesis, and `speculative_for` executes the deterministic reservations framework using the user-defined `reserve` and `commit` functions (more details will be discussed in Chapter 3).

Part I of this thesis describes tools for writing internally deterministic parallel code [53, 423, 421], drawing heavily on using *commutative operations*. This part also describes internally deterministic solutions to a broad set of benchmark problems using these tools, and shows that these solutions are efficient (competitive with existing nondeterministic

```
struct STStep {
  int u;  int v;
  edge *E;  res *R;  disjointSet F;
  STStep(edge* _E, disjointSet _F, res* _R)
    : E(_E), R(_R), F(_F) {}

  bool reserve(int i) {
    u = F.find(E[i].u);                                              //find component
    v = F.find(E[i].v);                                              //find component
    if (u == v) return 0;           //skip edge if endpoints belong to the same component
    if (u > v) swap(u,v);
    R[v].reserve(i);                                       //reserve larger component
    return 1;}

  bool commit(int i) {
    if (R[v].check(i)) { F.link(v, u); return 1;}     //link if reservation was successful
    else return 0;  }
};

void ST(res* R, edge* E, int m, int n, int psize) {
  disjointSet F(n);                              //deterministic union-find data structure
  speculative_for(STStep(E, F, R), 0, m, psize);       //deterministic reservations driver
}
```

Figure 1.1: C++ code for spanning forest using deterministic reservations (with its operations `reserve`, `check`, and `speculative_for`), where m is the number of edges and n is the number of vertices in the graph.

solutions and achieve good parallel speedup), scalable to large inputs, natural to reason about, not complicated to code [53], and also have good theoretical guarantees [55, 427].

Controlling Shared Access. Many parallel programs use *locks* to control access to shared resources. The granularity of locking (e.g., locking an entire data structure versus locking a small part of the data structure) affects the performance, scalability, and programmability of a solution, with coarser-grained locking leading to simpler solutions and finer-grained locking leading to higher efficiency and scalability. Programming with locks, however, has disadvantages such as leading to deadlock or livelock, and writing efficient fine-grained lock-based programs is often very tedious. There has been significant work on writing parallel programs without locks by making use of atomic operations (e.g., compare-and-swap and fetch-and-increment) supported in hardware [225]. Proper use of atomics can lead to more efficient programs than fine-grained locking and has the advantage of having progress guarantees. All of the programming techniques, algorithms, and data structures developed in this thesis are lock-free, making use of atomic operations when necessary, while also being simple. An extremely useful atomic primitive called *priority update* for controlling shared access in deterministic programs [423] is introduced in Chapter 6, and is used throughout the algorithms in this thesis.

Transactional memory (TM) is a technique to simplify shared-memory programming by allowing users to specify regions of code that will execute atomically (see, e.g., [216] for an overview). This frees the programmer from having to lock critical sections in code, leading to simpler programs. There has been significant research in implementing transactional memory both in software and in hardware. However, the techniques developed in this thesis are unlikely to benefit from TM for two reasons: (1) the order in which transactions succeed in TM is not deterministic, and (2) the algorithms in this thesis have no lock-based critical sections—shared accesses are protected using only a single atomic instruction.

Programming Frameworks. Another effort in simplifying shared-memory programming has been in developing higher-level frameworks and interfaces for writing parallel solutions. These range from general parallel programming libraries such as the Parallel Boost Graph Library [197], Multi-Core Standard Template Library (MCSTL) [432], SWARM [22], Galois [379], and algorithms/containers provided as part of the Intel Thread Building Blocks, to domain-specific frameworks/languages such as GraphLab [306, 186] and Green-Marl [229]. The solutions all vary in programmability, efficiency, and coverage.

Graph processing frameworks have received significant recent interest due to their importance in large-scale data analytics. Part II of this thesis introduces Ligra, the first high-level shared-memory graph processing framework that targets graph traversal algorithms (i.e., algorithms that visit a small subset of the graph in each iteration). The framework is very simple and lightweight. In addition to a graph data structure, it requires only one data structure, used for representing a subset of vertices (vertexSubset), and two functions, one for mapping user-defined functions over vertices (VERTEXMAP) and the other for mapping over edges (EDGEMAP). For example, Figure 1.2 shows a concise implementation of a parallel breadth-first search (BFS) algorithm in Ligra. Each iteration of the BFS algorithm applies an EDGEMAP to the current frontier of vertices (Line 10), in which the user-defined UPDATE function is applied to all outgoing edges of the frontier vertices such that the applying the COND function on the target of the edge returns *true*. Here, the COND function simply checks if a vertex is unexplored, and if so, the UPDATE function atomically marks the neighbor as visited with a compare-and-swap.

This thesis shows that Ligra can process the largest publicly-available real-world graphs in shared-memory, is much faster than existing graph processing systems, and competitive with highly-optimized code for the same applications. This work advocates performing large-scale graph analytics on a single shared-memory server instead of using distributed memory, and since the development of Ligra, there have been several other large-scale graph processing frameworks [351, 399, 247, 471] developed for shared-memory multicores, as well as a graph processing framework for GPUs sharing ideas with Ligra [457].

Concurrency. There has been a large body of research on *concurrency* in parallel programming, which studies how different threads interact with each other. Dealing with concurrency often requires considerable effort from the programmer because the behavior of concurrent programs is almost always nondeterministic due to the nondeterministic order in which the threads execute. The goal of this thesis is to hide the concurrency in

```
1:  Parents = $\{-1, \dots, -1\}$                        ▷ initialized to all -1's, indicating unexplored
2:  procedure UPDATE($s$, $d$)
3:      return (CAS(&Parents[$d$], $-1$ , $s$ ))                      ▷ atomically explore vertex
4:  procedure COND($i$)
5:      return (Parents[$i$] $==$ $-1$)                               ▷ check if unexplored
6:  procedure BFS($G$, $r$)                           ▷ $G$ is the graph and $r$ is the source vertex
7:      Parents[$r$] $= r$
8:      Frontier $= \{r\}$                          ▷ vertexSubset initialized to contain only $r$
9:      while (SIZE(Frontier) $\neq 0$) do
10:         Frontier $=$ EDGEMAP($G$, Frontier, UPDATE, COND)              ▷ visit next frontier
```

Figure 1.2: Pseudocode for breadth-first search (BFS) in Ligra. The compare-and-swap function CAS(*loc*,*oldV*,*newV*) atomically checks if the value at location *loc* is equal to *oldV* and if so it updates *loc* with *newV* and returns *true*. Otherwise it leaves *loc* unmodified and returns *false*.

parallel programs from the programmer by raising the level of abstraction and developing deterministic tools at this higher level of abstraction (e.g., deterministic reservations described in Chapter 3 and priority updates described in Chapter 6) and data structures (e.g., a deterministic phase-concurrent hash table described in Chapter 5) that the user can simply call in their programs. By raising the level of abstraction, the implementations of the tools can be nondeterministic (but hidden to the programmer), giving more flexibility and efficiency. This approach leads to deterministic parallel solutions that are simple to reason about, and that are also efficient at the same time.

Memory consistency issues often arise in concurrent programs as instructions can be reordered on multicore processors. However, in all of the solutions developed in this thesis, reads and writes to the same memory location are either separated by a synchronization point or use a compare-and-swap, which implicitly issues a memory barrier to prevent consistency issues. All of the solutions are *sequentially consistent*, which means that their results are consistent with some valid sequential execution of the program [291].

Thesis Scope. In summary, the algorithms, frameworks, and techniques developed in this thesis are for nested fork-join parallelism, and use only the fork and join primitives, parallel for-loops (which can be implemented with fork and join), and atomic instructions supported in hardware. This set of primitives was sufficient for all of the problems considered in this thesis. Furthermore, designing algorithms within this paradigm allows for clean theoretical analysis in the work-depth model, described in Section 1.2, and good performance in practice using a work-stealing runtime scheduler. Solutions in this thesis do not use techniques such as locks, transactional memory, pipelining, futures, or message passing, as they were not necessary in developing simple and efficient solutions for the problems considered.

1.2 Shared-Memory Algorithm Design

Parallel Random Access Machine. Algorithm designers have traditionally used the Parallel Random Access Model (PRAM) to analyze parallel algorithms for shared memory. In this model, every core has unit-time access to the shared global memory. An algorithm's complexity is characterized by its asymptotic time T and number of cores P, with the total number of operations being the product of the two terms. They can also be analyzed in the Work-Time Framework [243], in which the total number of operations W and number of parallel time steps T is specified. PRAM algorithms are written using *flat parallelism*, in which parallel operations over a single array is done synchronously at every time step. The algorithm must specify how work can be efficiently allocated among the cores on each step (known as the *processor allocation problem*). Using Brent's scheduling principle [73, 243], an algorithm with W work and T time can be run in $W/P + T$ time with P cores. Nested fork-join parallel algorithms cannot be directly expressed in the PRAM, and the parallelism in such algorithms must be flattened to work for the PRAM. Different classes of PRAM models differ in whether concurrent reads or writes are allowed, how to resolve write conflicts, and how to deal with contention (see, e.g., [243, 171]). There have also been variants proposed that allow for asynchrony among the cores [168, 107, 356, 170], as well as a related model that provides parallel primitives on vectors [48].

Work-Depth Model. The *work-depth model* is a model supporting nested fork-join parallelism.[5] As discussed in Section 1.1, a nested parallel computation can be modeled as a computation DAG. An algorithm's complexity is analyzed by computing its work W, which is the sum of the costs of all the tasks in the computation DAG, and its depth D, which is the maximum sum of costs of tasks on a directed path in the DAG (the longest sequential dependence). The maximum possible amount of parallelism (i.e., the maximum number of cores the computation can take advantage of) is W/D. The complexity of PRAM algorithms translate to results in the work-depth model, however they can often be simplified, as the processor allocation step is not necessary and divide-and-conquer can be used. The work-depth model underlies the design of programming languages such as NESL [49] and Cilk [158], and algorithms designed for the model can take advantage of dynamic multithreading languages. For example, a computation with work W and depth D using Cilk's randomized work-stealing scheduler gives an expected running time of $W/P + O(D)$ when running on P cores [65]. The algorithms developed in this thesis are analyzed in the work-depth model, but they can easily be translated into PRAM algorithms.

Traditional Design Goals. The main goal in developing efficient parallel algorithms is to have an algorithm with low (polylogarithmic) depth and work matching that of the best

[5]This contrasts with the Work-Time Framework, which is a framework for analyzing PRAM algorithms and does not allow for nested parallelism.

sequential algorithm for the same problem (*work-efficient*). Being work-efficient is desirable in that the parallel algorithm does not perform asymptotically more operations than the best sequential algorithm for the same problem, and so is efficient even when there is not much parallelism available. Having depth that is polylogarithmic ($O(\log^c n)$ for an input size of n and any constant c) is desirable in that it allows for ample parallelism.[6] Work-efficient and polylogarithmic-depth algorithms have been developed for many fundamental problems in computing. Many of these algorithms, however, are not practical as they involve many sophisticated machinery and have large hidden constant factors in their complexity.

Bridging Theory and Practice. Because the goal of this thesis is to develop parallel algorithms that are efficient and scalable on real shared-memory machines, the simplicity and practicality of the algorithms are also important. Therefore, *in addition to designing work-efficient algorithms with low depth, this thesis also strives for simple solutions that perform well in practice.* Having algorithms that are efficient both in theory and in practice allows for good performance across all possible inputs, scalability across a wide range of core counts, and graceful scalability to larger data sets. There has traditionally been a gap between theory and practice in parallel algorithms, with many theoretically-efficient algorithms not being practical and many algorithms used in practice lacking strong theoretical guarantees. This thesis seeks to bridge this gap by developing large-scale shared-memory algorithms for various well-studied problems on that are simple, and efficient both in theory and in practice.

Chapter 4 presents the theoretical guarantees and empirical performance of several simple parallel algorithms developed using the technique of deterministic reservations. The chapter shows that, perhaps surprisingly, several natural sequential iterative algorithms inherently have high parallelism, both in theory and in practice, leading to very simple and practical deterministic parallel implementations. Parts III and IV of this thesis introduce the first parallel algorithms for a variety of problems on graphs and strings that are both theoretically-efficient and practical. The theoretical bounds of the algorithms developed are shown in Table 1.1, and an experimental analysis on modern multicore machines of each of the algorithms is presented in their respective chapters of the thesis.

We will now briefly look at the performance of two of the algorithms developed in this thesis—triangle counting and suffix tree construction. For triangle counting, this thesis develops the first work-efficient, polylogarithmic-depth, and cache-friendly shared-memory algorithm (Chapter 10), which outperforms existing shared-memory algorithms by a factor of 2–5x on 40 cores with two-way hyper-threading and achieves a parallel speedup ranging from 22x to 49x [428]. The speedup of the algorithm with respect to the fastest existing

[6]Polylogarithmic-depth algorithms are also desirable for computational complexity reasons, as they fall in the class NC (Nick's Class) containing problems that can be solved on circuits with polylogarithmic depth and polynomial size [15].

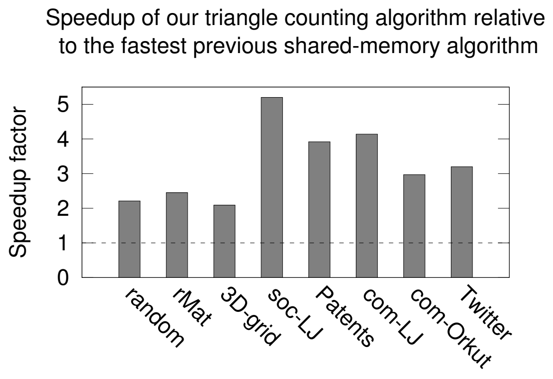
Problem	Work	Depth
Maximal Independent Set (Chapter 4)	$O(m)$	$O(\log^3 n)$
Maximal Matching (Chapter 4)	$O(m)$	$O(\log^3 m)$
Random Permutation (Chapter 4)	$O(n)$	$O(\log^2 n)$
List Contraction (Chapter 4)	$O(n)$	$O(\log^2 n)$
Tree Contraction (Chapter 4)	$O(n)$	$O(\log^2 n)$
Connected Components (Chapter 9)	$O(m)$	$O(\log^3 m)$
Triangle Counting (Chapter 10)	$O(m^{3/2})$	$O(\log^{3/2} m)$
Cartesian Tree/Suffix Tree† (Chapter 11)	$O(n)$	$O(\log^2 n)$
Longest Common Prefixes (Chapter 12)	$O(n)$	$O(\log^2 n)$
Lempel-Ziv Factorization† (Chapter 13)	$O(n)$	$O(\log^2 n)$
Wavelet Tree Construction‡ (Chapter 14)	$O(n \log \sigma)$	$O(\log n \log \sigma)$

Table 1.1: Work and depth bounds for the (randomized) algorithms developed in this thesis. For the graph problems, n = number of vertices and m = number of edges. For the other problems, n is the input size. †Bounds are for constant-sized alphabets. ‡σ = alphabet size. The depth of some of these algorithms can be improved with approximate compaction [174], as described in their respective chapters.

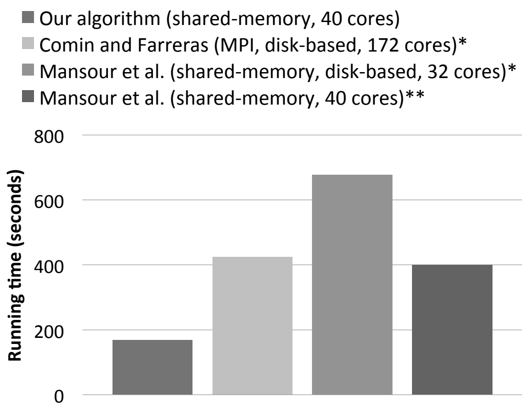
shared-memory implementation on various graphs is shown in Figure 1.3(a). Additionally, this algorithm has stronger theoretical bounds than previous shared-memory algorithms. Compared to existing distributed-memory solutions, the algorithm is faster by *at least an order of magnitude* on a per-core basis on the largest graphs studied in the literature. For suffix tree construction, this thesis develops the first parallel algorithm with linear work and polylogarithmic depth that is also practical (Chapter 11) [422]. On 40 cores with two-way hyper-threading, the algorithm achieves a 5.4–50.4x speedup over the best sequential algorithm [285] on a variety of inputs. The algorithm can construct in under 3 minutes the suffix tree for the 3 gigabyte human genome, one of the largest data sets reported in the literature for suffix tree construction. Compared to the fastest numbers reported in the literature for suffix tree construction on the human genome, the algorithm is at least two times faster in practice, as shown in Figure 1.3(b), in addition to being theoretically more efficient.

1.3 Shared-Memory Performance

Cache Performance. Due to the high latency to access main memory, modern multicore machines have *caches*, which are smaller memories that support faster access times. Multicore machines can have multiple levels of caches, each with different sizes and access times, and furthermore caches may either be shared among cores or private to a single core. The caches thus form a hierarchy, and designing algorithms that make efficient use of the cache hierarchy is crucial for performance. The algorithms studied in this thesis involve many memory accesses, and thus their performance is largely determined by the number of cache misses. While this thesis does not explicitly analyze the cache performance of



(a) Speedup of our triangle counting algorithm relative to the fastest shared-memory algorithm (varies between the implementation in GraphLab [186] and the one by Green et al. [192]) on various synthetic graphs from [424] and real-world graphs from [298, 288] on 40 cores with two-way hyper-threading.

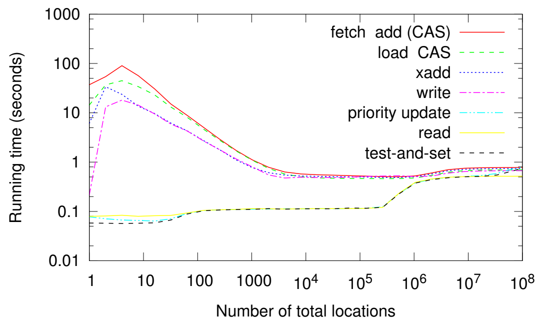


(b) Parallel running times of suffix tree construction on the 3 GB human genome. *Reported time from the literature [320, 108]. **Code from [320] run on our 40-core machine with a memory budget of 160 GB.

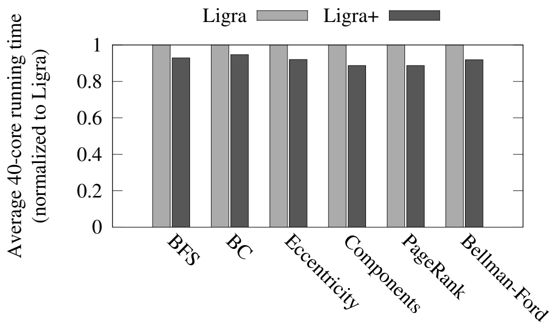
Figure 1.3: Experimental evaluation of triangle counting and suffix tree construction.

algorithms (with the exception of Chapter 10, which analyzes cache performance of triangle computations), they are all implemented to be cache-friendly, maximizing spatial and temporal locality when possible. Cache misses can also be factored into an algorithm's theoretical complexity (see, e.g., [157, 431]), although this is not the focus of this thesis.

Contention. On multicore machines, different private caches may reference the same objects in memory, and so there is the challenge of making sure that the cores' views of the data are consistent. A *cache coherence protocol* dictates how this consistency is maintained among the caches (see, e.g., [121] for more details). Cache coherence protocols have a significant effect on the performance of shared memory accesses (see, e.g., the recent study by David et al. [123]). In general, when updates are performed to a shared location concurrently by many different cores, the *memory contention* causes performance to worsen as the cache coherence protocol must perform significant work to ensure consistency among different caches. To reduce contention in shared-memory programs, Chapter 6 of this thesis develops and advocates the usage of the priority update operation, which performs an actual update only when the value written has "higher priority" than the existing value, for a large class of applications. The thesis studies its performance both experimentally and theoretically under varying degrees of sharing, showing that it is much more efficient than many commonly-used operations, and comparable in performance to other, less powerful operations. Figure 1.4(a) shows an experiment measuring the performance of commonly



(a) Impact of sharing on a variety of operations. Times are for 5 runs of 100 million operations to varying number of memory locations on a 40-core Intel Nehalem machine (log-log scale). Since the number of operations is fixed, fewer locations implies more operations sharing those locations.



(b) Speedup of Ligra+ relative to Ligra on a variety of graph applications on 40 cores with two-way hyper-threading.

Figure 1.4: (a) Experiments measuring contention of various parallel operations and (b) average performance of Ligra+ relative to Ligra on 40 cores with two-way hyper-threading.

used operations on varying numbers of shared locations (fewer locations implies more sharing). Observe that when there is a high degree of sharing (e.g., only 8 locations) the priority update is competitive with reads and test-and-sets (less powerful operations), and *over two orders of magnitude faster* than standard writes and other atomic operations. The priority update operation also has the added benefit of giving determinism and guaranteeing progress when used appropriately.

Scalability. The goal in parallel computing is to design solutions that scale well both with an increasing number of cores and also with increasing input size. The shared-memory solutions developed in this thesis are able to achieve both of these goals. They achieve good parallel scalability on the multicore machines used in this thesis (limited by memory bandwidth, as discussed next), and due to their low depth complexities are likely to scale well on future multicore machines with many more cores. The solutions are also scalable to large data sets—for example, the Ligra framework and the graph algorithms introduced in Part III are able to process the largest publicly-available real-world graphs (with billions of vertices and edges) in the order of seconds to minutes, and the string algorithms developed in Part IV scale to texts with billions of symbols, such as the human genome. This thesis proposes the use of graph compression in Chapter 8 to reduce space usage and allow even larger graphs to be processed in shared-memory.

Memory Bandwidth. Due to the irregular nature of the problems that studied in this thesis, random access is often unavoidable, and the parallel scalability of solutions is often limited by the bandwidth of the memory interconnect (using more cores increases the load on the

Basic Building Blocks	Prefix Sum, Integer Sort, Comparison Sort, Remove Duplicates, Dictionary, Sparse Matrix-Vector Multiply, Random Permutation, List Contraction, Tree Contraction
Graphs	Breadth-First Search, Connected Components, Spanning Forest, Minimum Spanning Forest, Maximal Independent Set, Maximal Matching, Triangle Counting, Graph Separators
String/Text Processing	Suffix Array, Burrows-Wheeler Transform, Longest Common Prefixes, Sequence Alignment
Computational Geometry and Graphics	Quad/Oct Tree, Delaunay Triangulation, Delaunay Refinement, Convex Hull, k-Nearest Neighbors, N-Body, Ray Casting

Table 1.2: Benchmarks in the Problem Based Benchmark Suite.

memory interconnect, which often becomes saturated before all cores are fully utilized). To alleviate this problem, this thesis uses graph compression techniques in Chapter 8 to reduce memory usage, thus reducing the impact of the memory bandwidth bottleneck, and as a result improving parallel performance and scalability. The thesis develops Ligra+ by integrating the graph compression techniques into Ligra, and shows that *reduced space usage and improved parallel performance can be achieved at the same time* [426]. The graph sizes are reduced to about half of the original size on average, and performance increases by about 14% on average on 40 cores. Figure 1.4(b) shows the average relative performance of Ligra+ compared to Ligra on various graph applications using 40 cores. Ligra+ is the first high-level graph processing system to support in-memory compression.

1.4 The Problem Based Benchmark Suite

To measure the programming simplicity, theoretical efficiency, and empirical performance among different solutions for given problems, my co-authors and I have developed a benchmark suite, called the Problem Based Benchmark Suite (PBBS) [424], containing a set of well-known fundamental problems that is representative of a broad class of non-numeric applications arising in computing. Table 1.2 shows the problems currently in the benchmark suite (the definitions of these problems can be found in Section 2.6).[7] Unlike most existing benchmarks, which are based on specific code, the PBBS benchmarks are defined in terms of the problem specifications—a concrete description of valid inputs and corresponding valid outputs, along with some specific inputs. Any algorithms, programming methodologies, specific programming languages, or machines can be used to solve the problems. The benchmark suite is designed to compare the benefits and shortcomings of different algorithmic and programming approaches, and to serve as a dynamically improving set of educational examples of how to parallelize applications. The PBBS has

[7]The table has been modified from [424] to reflect the problems currently in the benchmark suite.

enabled comparisons in terms of simplicity, and theoretical/practical performance among various algorithms and programming techniques for the problems studied in this thesis.[8] Many of the implementations developed in this thesis are part of the PBBS.

1.5 Thesis Statement

This thesis seeks to address the three types of challenges arising in multicore programs, as outlined in Sections 1.1, 1.2, and 1.3, to make large-scale shared-memory parallelism more accessible. Programming techniques, algorithm design, and performance analysis are closely interrelated, and therefore effective solutions require attention to all three areas. Throughout the development of this thesis, I have used my knowledge in each of these areas to improve my understanding of issues arising in the other areas, and thus the thesis contains contributions cutting across all three areas.

This thesis provides evidence to support the following statement:

Thesis statement: With appropriate programming techniques, frameworks, and algorithms, shared-memory programs can be simple, fast, and scalable, both in theory and in practice.

I believe that the frameworks, tools, algorithms and ideas developed in this thesis will enable more people to write efficient shared-memory parallel programs and take advantage of the power of multicore machines to perform large-scale computations. The code developed as part of this thesis is publicly available, and has already been used by various researchers for benchmarking and developing their own shared-memory solutions.

1.6 Thesis Contributions

This thesis uses a three-pronged approach studying programming techniques, algorithm design, and performance analysis for shared-memory multicores. These three areas are highly interrelated, and so each of the chapters of this thesis will inevitably cut across the different areas. An illustration placing each of the topics of this thesis into the closer two among the three categories is shown in Figure 1.5. I have developed the results of this thesis in collaboration with various co-authors: Guy Blelloch, Laxman Dhulipala, Jeremy Fineman, Phillip Gibbons, Yan Gu, Aapo Kyrola, Harsha Simhadri, Kanat Tangwongsan, and Fuyao Zhao. The following paragraphs describe the organization and contributions of this thesis.

Chapter 2 introduces the necessary definitions and notation used throughout the thesis. Then, Part I of the thesis describes frameworks and techniques for simplifying deterministic parallel programming. The contributions of this part include:

[8]While the thesis focuses on multicore solutions, this is not a constraint of the PBBS.

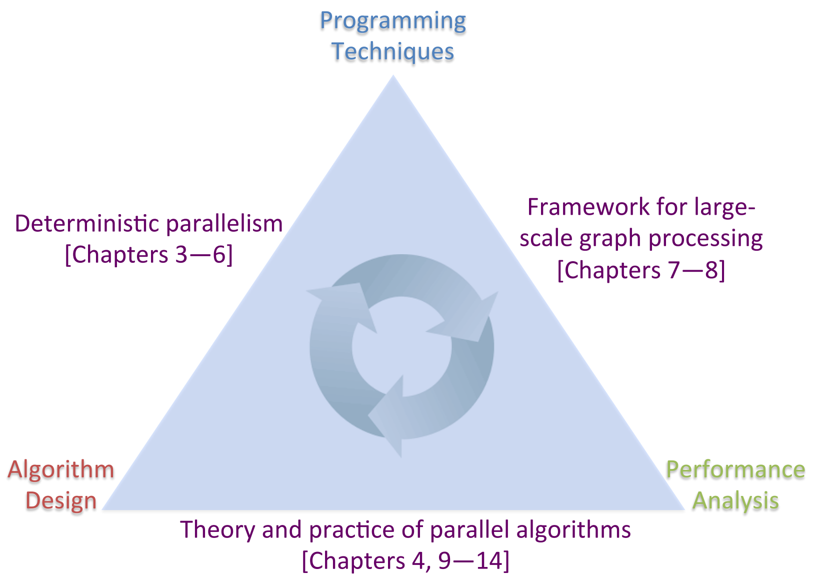

Programming Techniques
Deterministic parallelism [Chapters 3—6]
Framework for large-scale graph processing [Chapters 7—8]
Algorithm Design
Performance Analysis
Theory and practice of parallel algorithms [Chapters 4, 9—14]

Figure 1.5: A pictorial organization of this thesis. The topics touch upon programming techniques, algorithm design, and performance analysis, and are placed in the closer two among the three areas in the figure.

- A new approach for writing efficient deterministic parallel programs using building blocks based on commutativity, and the design of several building blocks including priority updates, dictionaries, and disjoint sets (Chapters 3, 5, and 6).

- A novel technique called deterministic reservations for taking sequential loops with dependencies among iterations and parallelizing them deterministically (Chapters 3 and 4).

- A suite of deterministic parallel algorithms and data structures, including comparison sorting, a hash-based dictionary, remove duplicates, random permutation, list contraction, tree contraction, breadth-first search, spanning forest, minimum spanning forest, maximal independent set, maximal matching, suffix arrays, Delaunay triangulation, Delaunay refinement, quad/oct trees, k-nearest neighbors, N-body, and triangle ray intersect, along with experiments showing they are *fast, scalable, and competitive with the best nondeterministic code for the same problem* (Chapters 3–6).

- The first proofs that the lexicographically first maximal independent set and maximal matching problems on random inputs have polylogarithmic depth, as well as efficient linear-work parallel algorithms for the problems (Chapter 4).

- The first proofs that the standard sequential random permutation algorithm and natural sequential iterative algorithms for list contraction and tree contraction on random inputs have logarithmic depth, as well as efficient linear-work parallel implementations of the algorithms (Chapter 4).

- The first application of Nisan's pseudorandom generator for space-bounded computations [354] to reducing the amount of randomness in low-depth parallel algorithms, in particular reducing the amount of randomness in the random permutation and list contraction algorithms from $O(n \log n)$ to a polylogarithmic number of random bits (Chapter 4).

- The formalization of the concept of phase-concurrency in deterministic parallel programs to simplify the design of data structures and improve their performance (Chapter 5).

- A deterministic phase-concurrent hash table that is *faster than all existing concurrent hash tables*, and has many applications in deterministic parallel programs, such as in removing duplicates, Delaunay refinement, suffix trees, edge contraction, breadth-first search, and spanning forest (Chapter 5).

- The generalization of special cases of the priority update operation in the literature, an efficient contention-reducing implementation of the operation, as well as the first theoretical analysis of its performance (Chapter 6).

- The first comprehensive experimental study of the priority update operation versus other widely-used operations under varying degrees of sharing, demonstrating that it is up to *orders of magnitude faster* on modern Intel and AMD multicore machines (Chapter 6).

- Many applications of the priority update operation in deterministic parallel programs, enabling good performance even under a high degree of write sharing (Chapter 6).

Part II of the thesis describes the Ligra/Ligra+ graph processing framework and includes the following contributions:

- The Ligra shared-memory graph processing framework containing just two simple functions—one for mapping computation over a subset of vertices and one for mapping computation over a subset of edges—sufficient to concisely express a broad class of graph traversal algorithms in shared-memory (Chapter 7).

- The generalization of the direction-optimizing idea used in breadth-first search [32] to a large class of graph traversal algorithms to improve performance (Chapter 7).

- An experimental evaluation showing that the Ligra implementations are efficient and scalable to the largest publicly-available real-world graphs in the literature, and *outperform existing systems by up to orders of magnitude* (Chapter 7).

- The first high-level shared-memory graph processing system (Ligra) to process (*in under a minute*) the largest publicly-available real-world graph, the Yahoo! Web graph with *over 6 billion edges*, showing the benefits of shared-memory for large-scale graph processing, and subsequently leading to several other shared-memory graph processing systems [351, 399, 247, 471, 457] (Chapter 7).

- Ligra+, the first high-level shared-memory graph processing system to use graph compression to reduce in-memory space usage, improving the scalability of shared-memory graph processing (Chapter 8).

- An efficient implementation and experimental evaluation of Ligra+ showing that graph compression *both reduces the space usage and also improves the parallel performance* of graph traversal algorithms (Chapter 8).

Part III of the thesis describes practical large-scale parallel algorithms with strong theoretical guarantees for solving problems on graphs. The contributions of this part include:

- The first *practical* linear-work and polylogarithmic-depth parallel algorithm for graph connectivity, a problem that has been open for over a decade (Chapter 9).

- Extensive empirical evaluation of the parallel connectivity algorithm, showing that it is competitive with existing parallel implementations, none of which are linear-work and polylogarithmic-depth (Chapter 9).

- The first work-efficient, polylogarithmic-depth, and cache-efficient shared-memory algorithms for exact and approximate triangle computations that are both simple and practical (Chapter 10).

- Comprehensive empirical evaluation of the running time and cache performance of the triangle computation algorithms showing that they are faster than distributed implementations by up to *orders of magnitude* and shared-memory implementations by up to a factor of 5, and scale to the largest publicly-available real-world graphs (Chapter 10).

Part IV of the thesis describes large-scale parallel string algorithms that have strong theoretical guarantees and also perform well in practice, scaling to the largest data sets considered in the literature for the problems. This part includes the following contributions:

- A new and simple linear-work, polylogarithmic-depth parallel algorithm for building multiway Cartesian trees using divide-and-conquer, and various applications of Cartesian trees (Chapter 11).
- The first *practical* linear-work and polylogarithmic-depth parallel algorithm for suffix tree construction, developed using suffix arrays and multiway Cartesian trees (Chapter 11).
- The *state-of-the-art parallel suffix tree implementation for shared-memory*, achieving good parallel speedup (up to 24x on 40 cores) and outperforming existing parallel implementations by at least a factor of 2 (Chapter 11).
- New theoretically-efficient and practical parallel algorithms for computing longest common prefixes, a useful primitive in suffix array (and suffix tree) construction (Chapter 12).
- The first comprehensive experimental evaluation of parallel longest common prefix algorithms, showing that the new algorithms achieve good parallel speedup, are up to 2.3x faster than the best existing algorithm on 40 cores, and lead to improved performance for suffix array construction (Chapter 12).
- The first *practical* linear-work and polylogarithmic-depth parallel algorithm for Lempel-Ziv factorization (based on suffix arrays), an essential operation in many data compression methods (Chapter 13).
- An extensive experimental study of the Lempel-Ziv factorization algorithm showing that it achieves good parallel speedups (up to 23x on 40 cores) and outperforms the sequential algorithm with just 2 or more threads (Chapter 13).
- The first polylogarithmic-depth parallel algorithms for constructing wavelet trees, an essential component to many compressed data structures (Chapter 14).
- A comprehensive empirical evaluation of the wavelet tree algorithms showing that they achieve good speedup over the sequential algorithm (up to 27x on 40 cores) and are up to 5.6 times faster than existing parallel implementations (Chapter 14).

Finally, Chapter 15 concludes the thesis and describes directions for future work.

Chapter 2

Preliminaries and Notation

This chapter presents the definitions, models, and notation that will be used throughout the thesis. Individual chapters have additional definitions and notation that are specific to the chapter.

2.1 Parallel Programming Model

All of the algorithms, frameworks, and tools in this thesis can be implemented using ***nested fork-join parallelism***, in which a ***fork*** specifies procedures that can be called in parallel, and a ***join*** specifies a synchronization point among procedures. The fork and join constructs can be nested, making this type of parallelism particularly useful for divide-and-conquer algorithms.

More formally, nested parallel computations can be defined inductively in terms of the composition of sequential and parallel components. At the base case, a ***strand*** is a sequential computation. A ***task*** is then a sequential composition of strands and parallel blocks, where a ***parallel block*** is a parallel composition of tasks starting with a fork and ending with a join.

A nested parallel computation can be modeled (a posteriori) as a series-parallel ***computation DAG*** over the operations of the computation: the tasks in a parallel block are composed in parallel, and the operations within a strand as well as the strands and parallel blocks of a task are composed in series in the order they are executed. All operations are assumed to take a state and return a value and a new state (any arguments are part of the operation). Vertices in the computation DAG are labeled by their associated operation (including arguments, but not return values or states). An operation (vertex) u ***precedes*** v if there is a directed path from u to v in the DAG. If there is no directed path in either direction between u and v, then u and v are ***logically parallel***, meaning that they *may* be executed in parallel.

The support of nested parallelism dates back at least to Dijkstra's `parbegin-parend` construct. Many parallel languages support nested parallelism including NESL [49], Cilk [158], the Java fork-join framework [244], OpenMP [360], X10 [88], Habanero [76], Intel Threading Building Blocks [237], and the Task Parallel Library [441]. Although not appropriate for certain types of parallelism, e.g., pipeline parallelism, nested parallelism has many theoretical and practical advantages over more unstructured forms of parallelism, including simple schedulers for dynamically allocating tasks to cores, compositional analysis of work and depth, and good space and cache behavior (e.g., [2, 64, 50, 59]).

Programs in this thesis are written with the Cilk programming language, which is a dynamic multithreading language for shared memory that supports nested fork-join parallelism [64]. Simple constructs are used to indicate which parts of the program are safe to run in parallel, and a run-time scheduler assigns work to threads and performs load-balancing. The Cilk constructs used are `cilk_for`, used to indicate that iterates of a for-loop may execute in parallel, `cilk_spawn`, used to indicate a procedure may be called in parallel (fork), and `cilk_sync`, used to indicate that the current procedure must wait for all procedures that it spawned to complete before proceeding (join). A `cilk_for` loop is implemented using `cilk_spawn` and `cilk_sync`. There is an implicit `cilk_sync` at the end of each procedure.

2.2 Algorithmic Complexity Model

This thesis uses the ***work-depth*** model to analyze the complexity of algorithms. As discussed in Section 2.1, a computation can be modeled using a computation DAG. The thesis assumes unbounded in-degree and out-degree of the vertices in the DAG, although other variants of the model assume bounded degree.[1] The ***work*** W of an algorithm is equal to the sum of the costs of all tasks in the computation DAG, which is equivalent to the number of operations the algorithm performs. The ***depth*** D of an algorithm is equal to the maximum sum of costs of tasks over all directed paths in the computation DAG, which is equivalent to the number of time steps the algorithm requires if an infinite number of cores were available. This model makes it particularly convenient for analyzing nested parallel algorithms. Using the randomized work-stealing scheduler of Cilk gives an expected running time of $W/P + O(D)$ when using P cores [65]. Note that for sequential algorithms, the work and the depth terms are equivalent. A parallel algorithm is defined to be ***work-efficient*** if its work is asymptotically equal to the work of the fastest sequential algorithm for the same problem. The goal of this thesis is to design work-efficient parallel algorithms with polylogarithmic depth.

The traditional parallel random access machine (PRAM) model [243] for analyzing par-

[1]This increases the overall depth by at most a logarithmic factor.

allel algorithms differs from the work-depth model in that nested parallelism is not allowed (parallelism must be flattened), and on each time step the algorithm must specify how work is allocated to the cores (known as the ***processor allocation problem***). Algorithms are analyzed using the Work-Time Framework [243], where work is the same as in the work-depth model and time is equivalent to depth in the work-depth model. For an algorithm with work W and time T, Brent's scheduling theorem [73, 243] bounds the running time by $W/T + P$ using a greedy scheduler with P cores. Most of the algorithms in this thesis can be easily translated into PRAM algorithms with the same work and depth (time) complexities as they use parallel primitives that have equivalent complexities (see Section 2.3) in both the work-depth and PRAM models, parallelism can be flattened when necessary, and there is enough parallel slackness in each iteration to perform processor allocation efficiently. There are four versions of the PRAM that are used in the thesis: (1) the exclusive-read exclusive-write (EREW) model, which does not allow for concurrent reads or writes; (2) the concurrent-read exclusive-write (CREW) model, which allows for concurrent reads but not concurrent writes; (3) the concurrent-read concurrent-write (CRCW) model, which allows for both concurrent reads and writes; and (4) the scan PRAM [47], a version of the EREW PRAM in which scan (prefix sum) operations take unit depth. For the CRCW model, concurrent writes to a shared location results in either an arbitrary write being recorded (arbitrary CRCW), or the minimum (or maximum) value being recorded (priority CRCW).

Randomization. Many of the algorithms make use of randomization. For randomized algorithms, the thesis states that a result holds ***in expectation*** if it holds on average over all possible random choices made by the algorithm (the input can be adversarially chosen). Similarly, a result holds ***with high probability (w.h.p.)*** for an input of size n if it holds with probability at least $1 - 1/n^c$, for any constant $c > 0$, over all possible random choices made by the algorithm.

2.3 Parallel Primitives

The thesis makes use of the basic parallel primitives, prefix sum (scan), reduce, filter, and merge [62]. ***Prefix sum (scan)*** takes a sequence A of length n, an associative binary operator $\oplus$, and an identity element $\perp$ such that $\perp \oplus a = a$ for any a, and returns the sequence $(\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus A[0] \oplus A[1] \oplus \ldots \oplus A[n-2])$ as well as the resulting "sum" $\perp \oplus A[0] \oplus A[1] \oplus \ldots \oplus A[n-1]$. ***Reduce*** takes the same arguments as prefix sum, but only returns the resulting sum $\perp \oplus A[0] \oplus A[1] \oplus \ldots \oplus A[n-1]$. ***Filter*** takes a sequence A of length n, and a predicate function f, and returns a sequence A' of length n' containing the elements in $a \in A$ such that $f(a)$ returns true, in the same order that they appear in A. Filter can be implemented using prefix sum, and both require $O(n)$

work and $O(\log n)$ depth [62].[2] ***Merge*** takes sorted sequences A and B of lengths n and m, respectively, and returns a sorted sequence containing the union of the elements in A and B. It can be implemented in $O(n+m)$ work and $O(\log(n+m))$ depth [62]. Merge can be modified to return the intersection of the elements of two sorted sequences in the same complexity bounds. The above primitives all run on the EREW PRAM in the stated bounds. Cilk implementations of the primitives are available in the Problem Based Benchmark Suite.

A ***compare-and-swap*** (CAS) is an atomic instruction that takes three arguments—a memory location (*loc*), an old value (*oldV*) and a new value (*newV*); if the value stored at *loc* is equal to *oldV* it atomically stores *newV* at *loc* and returns *true*, and otherwise it does not modify *loc* and returns *false*. CAS is supported in hardware by modern multicore machines. The implementations in this thesis use CAS's both directly and as a subroutine to other atomic functions. The notation $\&x$ is used to refer to the memory location of variable x.

2.4 Graphs

A directed unweighted graph is denoted by $G = (V, E)$, where V is the set of vertices and E is the set of (directed) edges in the graph. The thesis uses the convention of denoting the number of vertices in a graph by $n = |V|$ and number of edges in a graph by $m = |E|$. The vertices are assumed to be indexed from 0 to $n-1$. A weighted graph is denoted by $G = (V, E, w)$, where w is a function which maps an edge to a real value, and each edge $e \in E$ is associated with the weight $w(e)$. $N^+(v)$ denotes the set of out-neighbors of vertex v in G and $d^+(v)$ denotes the out-degree of v in G. Similarly, $N^-(v)$ and $d^-(v)$ denote the in-neighbors and in-degree of v in G. For an undirected graph, $d(v)$ is used to denote the degree of vertex v. The thesis uses $N(V)$ to denote the set of all neighbors of vertices in V, and $N(E)$ to denote the neighboring edges of E (ones that share a vertex). $N(v)$ is used as a shorthand for $N(\{v\})$ when v is a single vertex. $G[U]$ is used to denote the ***vertex-induced subgraph*** of G by vertex set U, i.e., $G[U]$ contains all vertices in U along with edges of G with both endpoints in U. $G[E']$ is used to denote the ***edge-induced subgraph*** of G, i.e., $G[E']$ contains all edges E' along with the incident vertices of G.

The ***adjacency list*** format for graph representation stores for each vertex an array of indices of other vertices that it has an edge to as well as the vertex's degree. The arrays are assumed to be stored consecutively in memory. This representation requires $O(n+m)$ space.

[2]This thesis uses $\log x$ to be the base 2 logarithm of x, unless stated otherwise.

2.5 Strings

A string is denoted by S, its length by n, the i'th character (using zero-based indexing) of a string S by $\mathsf{S}[i]$, and the sub-string starting at the i'th character and ending at the j'th character of S by $\mathsf{S}[i, \ldots, j]$. The alphabet of S is denoted by $\Sigma = [0, \ldots, \sigma - 1]$, where $\sigma = |\Sigma|$ is the alphabet size. The thesis assumes that a string ends with a special character $\$$, lexicographically smaller than all characters in Σ. suf_i of a string S is defined to be the suffix of S starting at position i (i.e., $\mathsf{S}[i, \ldots, n-1]$).

2.6 Problem Definitions

This section defines the various problems studied in the thesis.

2.6.1 Sequences

Comparison Sort. For a sequence S and comparison function $<$ defining a total order on elements of S, return the values of S sorted by $<$.

Remove Duplicates. For a sequence of elements, a comparison function f, and a hash function h that maps elements to integers, return a sequence in which any duplicates (equal-valued elements) are removed.

Random permutation. For a sequence S, return a random ordering of the elements of S such that each of the $|S|!$ possible orderings is equally likely.

2.6.2 Lists, Trees, and Graphs

List Contraction. For an input of a collection of linked lists represented by an array L ($L[i]$ stores the predecessor and successor of node i), contract each list into a single node, possibly combining values on the nodes during contraction.

List Ranking. For an input of a collection of linked lists represented by an array L ($L[i]$ stores the predecessor and successor of node i), compute the distance from each node to the end of its linked list.

Tree Contraction. For a tree represented by an array T ($T[i]$ stores pointers to the parent and the two children of node i), contract the tree down to the root node, possibly combining values on the nodes during contraction.

Breadth First Search. For an undirected graph G and a source vertex r, return a breadth-first-search (BFS) tree, rooted at r, containing all of the vertices reachable from r in G.

Connected Components. For an undirected graph G, return a labeling L such that for two vertices u and v, $L(u) = L(v)$ if u and v belong in the same connected component (i.e., there exists a path between u and v), and $L(u) \neq L(v)$ otherwise.

i	$\mathsf{S}[i]$	$\mathsf{SA}[i]$	$\mathsf{LCP}[i]$	suf_i
0	b	6	0	$\$$
1	a	5	0	$a\$$
2	n	3	1	$ana\$$
3	a	1	3	$anana\$$
4	n	0	0	$banana\$$
5	a	4	0	$na\$$
6	$\$$	2	2	$nana\$$

Figure 2.1: Example: SA and LCP arrays for $\mathsf{S} = banana\$$.

Spanning Forest. For an undirected graph $G = (V, E)$, return edges $F \subseteq E$, such that for each connected component $C_i = (V_i, E_i)$ in G, a spanning tree T_i ($|T_i| = |V_i| - 1$) of C_i is contained in F. Furthermore, $|F| = \sum_{C_i \in G}(|V_i| - 1)$.

Minimum Spanning Forest. For an undirected graph $G = (V, E)$ with weights $w : E \rightarrow \Re$, return a spanning forest of minimum total weight.

Maximal Independent Set. For an undirected graph $G = (V, E)$, return $U \subseteq V$ such that no vertices in U are neighbors and all vertices in $V \setminus U$ have at least one neighbor in U.

Maximal Matching. For an undirected graph $G = (V, E)$, return $E' \subseteq E$ such that no edges in E' share a vertex and each edge in $E \setminus E'$ shares at least a vertex with an edge in E'.

Single-source Shortest Paths. For a weighted graph $G = (V, E, w)$ and a source vertex r, compute either the shortest path distance from r to each vertex in V (if a vertex is unreachable from r, then the distance returned is ∞), or report the existence of a negative cycle.

2.6.3 Strings

Suffix Array and Longest Common Prefixes. The ***suffix array*** [319] SA of S is a permutation of the integers $[0, \ldots, n-1]$ such that $\mathsf{suf}_{\mathsf{SA}[0]} < \mathsf{suf}_{\mathsf{SA}[1]} < \ldots < \mathsf{suf}_{\mathsf{SA}[n-1]}$, where "$<$" means lexicographically smaller. The ***longest common prefix*** array is an array LCP of length n such that $\mathsf{LCP}[0] = 0$ and for $i > 0$, $\mathsf{LCP}[i]$ contains the length of the longest common prefix (lcp) between $\mathsf{suf}_{\mathsf{SA}[i-1]}$ and $\mathsf{suf}_{\mathsf{SA}[i]}$. As an example, Figure 2.1 shows the SA and LCP arrays for the string $\mathsf{S} = banana\$$.

Trie. For a set of strings $\mathcal{S}$, return a tree where (1) each edge stores a character, (2) the concatenation of the characters on any path from the root to a node in the tree is a prefix of at least one string in $\mathcal{S}$, and (3) every string in $\mathcal{S}$ corresponds to concatenation of labels for a path from the root to a leaf.

Patricia Tree. For a set of strings $\mathcal{S}$, return a modified (compacted) trie in which (1) edges can be labeled with a sequence of characters instead of a single character, (2) no node has a single child, and (3) every string in $\mathcal{S}$ corresponds to concatenation of labels for a path from the root to a leaf [340].

Suffix Tree. For a string S, return the patricia tree storing the n suffixes of S [460].

2.6.4 Geometry

Triangle Ray Intersect. For a set of triangles T and rays R in three dimensions, return the first triangle each ray intersects, if any.

Delaunay Triangulation. For a set of n points in two dimensions, return a triangulation such that no point is contained in the circumcircle of any triangle in the triangulation [127].

Delaunay Refinement. For a Delaunay Triangulation on a set of n points, and an angle α, add new points such that in the resulting Delaunay Triangulation, no triangle has an angle less than α.

N-body. For a set of n point sources in three dimensions, each point p with coordinate vector $\vec{p}$ and a mass m_p, return the force induced on each one by the others based on the Coulomb force $\vec{F}_p = \sum_{q \in P, q \neq p} m_q m_p (\vec{q} - \vec{p}) / ||\vec{q} - \vec{p}||^3$.

K-Nearest Neighbors. For n points in two or three dimensions, and a parameter k, return for each point its k nearest neighbors (Euclidean distance) among all the other points.

2.7 Experimental Environment

This section summarizes the shared-memory multicore machines and compilers used for experimental evaluation throughout this thesis. The experimental setup varies among different chapters as the development of this thesis took several years, and different machines and compilers were available at different points in time. The specifications of the three machines and the compilers that were used are given below.

32-core Intel machine. A 32-core (with two-way hyper-threading) Dell PowerEdge 910 with 4×2.26GHz Intel 8-core X7560 Nehalem Processors. Each processor has a 1066MHz bus and a 24MB L3 cache. Each core has a 256KB L2 cache, a 32KB L1 data cache, and a 32KB L1 instruction cache. The processors are connected via an Intel QuickPath Interconnect (QPI) with a theoretical peak bandwidth of 25.6GB/second. The machine has a total of 64GB of main memory.

40-core Intel machine. A 40-core (with two-way hyper-threading) machine with $4 \times$ 2.4GHz Intel 10-core E7-8870 Xeon processors. Each processor has a 1066MHz bus and 30MB L3 cache. Each core has a 256KB L2 cache, a 32KB L1 data cache, and a 32KB L1 instruction cache. This machine also uses the Intel QPI and has a total of 256GB of main memory.

64-core AMD machine. A 64-core AMD machine with 4×2.4GHz 16-core 6278 Opteron processors. Each processor has a 1600MHz bus and 16MB L3 cache, 8$\times$2MB shared L2 caches, 8$\times$64KB shared L1 instruction caches, and 16$\times$16KB private L1 data caches. The interconnect uses HyperTransport with a theoretical peak bandwidth of 25.6GB/second. There is a total of 188GB of main memory on the machine.

Compilers. The three compilers used to compile parallel code are the `cilk++` compiler (build 8503) with the `-O2` flag, `icpc` compiler (version 12.1.0) with the `-O3` flag, and the `g++` (version 4.8.0, which supports Cilk) compiler with the `-O2` flag. The sequential programs were compiled using `g++` with the `-O2` flag. The optimization flags were chosen to give the best performance overall.

Part I

Programming Techniques for Deterministic Parallelism

Introduction

This part of the thesis introduces techniques and primitives for deterministic parallel programming, as well as deterministic algorithms and data structures. Chapter 3 studies a form of determinism, known as internal determinism, which requires the result of the computation as well as certain intermediate states to be deterministic. The chapter demonstrates that for a wide body of problems, there exist efficient internally deterministic algorithms, and moreover that these algorithms are natural to reason about and not complicated to code. Programming at a higher level of abstraction using *commutative building blocks*, and the technique of *deterministic reservations* for parallelizing sequential loops with dependencies among iterations are introduced as useful tools for deterministic parallel programming. Chapter 4 studies the theoretical properties of natural sequential algorithms for maximal independent set, maximal matching, random permutation, list contraction, and tree contraction, and shows that they actually exhibit high parallelism. The chapter designs simple parallel algorithms for these problems that obey the same dependencies as the corresponding sequential algorithms, and hence are deterministic. Experiments show that the implementations perform well in practice, outperforming the corresponding sequential algorithms with just a modest number of cores. Chapter 5 describes a deterministic *phase-concurrent hash table* in which operations of the same type are allowed to proceed concurrently, but operations of different types are not. Phase-concurrency guarantees that all concurrent operations commute, guaranteeing that the state of the table at any quiescent point is independent of the ordering of operations (and is hence deterministic). Furthermore, restricting the hash table to be phase-concurrent enables it to support operations more efficiently than previous concurrent hash tables. Chapter 6 presents a detailed study of the *priority update* operation, a useful primitive for deterministic parallel programming. The chapter shows both experimentally and theoretically that if implemented appropriately, priority updates greatly reduce memory contention over standard writes or other atomic operations when locations have a high degree of sharing. Various applications of the priority update in deterministic parallel programs are presented.

The results in this part of the thesis have appeared in the following publications:

- Guy Blelloch, Jeremy Fineman, Phillip Gibbons and Julian Shun. Internally Deterministic Parallel Algorithms Can Be Fast, *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 181–192, 2012.

- Guy Blelloch, Jeremy Fineman and Julian Shun. Greedy Sequential Maximal Independent Set and Matching are Parallel on Average, *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 308–317, 2012.

- Julian Shun, Yan Gu, Guy Blelloch, Jeremy Fineman and Phillip Gibbons. Sequential Random Permutation, List Contraction and Tree Contraction are Highly Parallel. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 431–448, 2015.

- Julian Shun and Guy Blelloch. Phase-concurrent Hash Tables for Determinism. *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 96–107, 2014.

- Julian Shun, Guy Blelloch, Jeremy Fineman and Phillip Gibbons. Reducing Contention Through Priority Updates. *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 152–163, 2013.

Chapter 3

Internally Deterministic Parallelism: Techniques and Algorithms

3.1 Introduction

One of the key challenges of parallel programming is dealing with nondeterminism. For many computational problems, there is no inherent nondeterminism in the problem statement, and indeed a serial program would be deterministic—the nondeterminism arises solely due to the parallel program and/or due to the parallel machine and its runtime environment. The challenges of nondeterminism have been recognized and studied for decades [371, 210, 168, 436]. Steele's 1990 paper, for example, seeks "to prevent the behavior of the program from depending on any accidents of execution order that can arise from the indeterminacy" of asynchronous programs [436]. More recently, there has been a surge of advocacy for and research in determinism, seeking to remove sources of nondeterminism via specially-designed hardware mechanisms [134, 135, 230], runtime systems and compilers [35, 37, 359, 469, 120, 119, 303, 118, 352, 247, 308], operating systems [36, 235], and programming languages [66, 322, 284, 283].

While there seems to be a growing consensus that determinism is important, there is disagreement as to what degree of determinism is desired (worth paying for). Popular options include:

- *Data-race free* [4, 165], which eliminate a particularly problematic type of nondeterminism: the data race. Synchronization constructs such as locks or atomic transactions protect ordinary accesses to shared data, but nondeterminism among such constructs (e.g., the order of lock acquires) can lead to considerable nondeterminism in the execution.

- *Determinate* (or *external determinism*), which requires that the program always produces the same output when run on the same input. Program executions for a given input may vary widely, as long as the program "converges" to the same output each time.

- *Internal determinism*, in which key aspects of intermediate steps of the program are also deterministic, as discussed in this chapter.

- *Functional determinism*, where the absence of side-effects in purely functional languages make all components independent and safe to run in parallel.

- *Synchronous parallelism*, where parallelism proceeds in lock step (e.g., SIMD-style) and each step has a deterministic outcome.

There are trade-offs among these options, with stronger forms of determinism often viewed as better for reasoning and debugging but worse for performance and perhaps programmability. Making the proper choice for an application requires understanding what the trade-offs are. In particular, is there a "sweet spot" for determinism, which provides a particularly useful combination of debuggability, performance, and programmability?

This chapter advocates a particular form of *internal determinism* as providing such a sweet spot for nested-parallel computations in which there is no inherent nondeterminism in the problem statement. As discussed in Chapter 2, an execution of a nested-parallel program defines a computation DAG with vertices representing computations and edges representing control dependencies among them. This DAG when annotated with the operations performed at each vertex (including arguments and return values, if any) is referred to as the *trace*. Informally, a program/algorithm is *internally deterministic* if for any input there is a *unique* trace. This definition depends on the level of abstraction of the operations in the trace. At the most primitive level the operations could represent individual machine instructions, but more generally, and as used in this chapter, it is any abstraction level at which the implementation is hidden from the programmer. Note that internal determinism does not imply a fixed schedule since any schedule that is consistent with the DAG is valid.

Internal determinism has many benefits. In addition to leading to external determinism [371] it implies a sequential semantics—i.e., considering any sequential traversal of the dependence DAG is sufficient for analyzing the correctness of the code. This in turn leads to many advantages including ease of reasoning about the code, ease of verifying correctness, ease of debugging, ease of defining invariants, ease of defining good coverage for testing, and ease of formally, informally and experimentally reasoning about performance [134, 135, 230, 37, 359, 469, 36, 66, 35]. Two primary concerns for internal determinism, however, are that it may restrict programmers to a style that (i) is complicated to program, unnatural, or too special-purpose and (ii) leads to slower, less scalable programs than less restrictive forms of determinism. Indeed, prior work advocating less restrictive forms of determinism has cited these concerns, particularly the latter concern [219].

This chapter seeks to address these two concerns via a study of a set of the benchmark problems in the Problem Based Benchmark Suite (refer to Section 1.4 and Figure 1.2), which cover a reasonably broad set of applications including problems involving sorting, graphs, geometry, graphics, and string processing. The main contribution of this chapter is demonstrating that *for this wide body of problems, there exist fast and scalable internally deterministic algorithms, and moreover that these algorithms are natural to reason about and not complicated to code*.

This thesis's approach for implementing internal determinism for these benchmarks is to use nested parallel programs in which concurrent operations on shared state are required to commute [459, 436] in their semantics and to be linearizable [227] in their implementation. Many of the algorithms implemented use standard algorithmic techniques based on nested data parallelism where the only shared state across concurrent operations is read-only (e.g., divide-and-conquer, map, reduce, and scan) [50]. However, a key aspect to several of the algorithms is the use of non-trivial commutative operations on shared state. The notion of commutativity has a long history, dating back at least to its use in analyzing when database transactions can safely overlap in time [459]. A seminal paper by Steele [436] discusses commutativity in the context of deterministic nested-parallel programs, showing that when applied to reads and writes on memory locations, commutativity of concurrent operations is sufficient to guarantee determinism.

Although there has been significant work on commutativity, there has been little work on the efficacy or efficiency of using non-trivial commutativity in the design of deterministic parallel algorithms. Much of the prior work on commutativity focuses on enforcing commutativity assuming the program was already written within the paradigm (e.g., using type systems [67]), automatically parallelizing sequential programs based on the commutativity of operations [396, 437, 383], or using commutativity to relax the constraints in transactional systems [224, 280], an approach that does not guarantee determinism. In contrast, this chapter identifies useful applications of non-trivial commutativity that can be used in the design of internally deterministic algorithms.

This chapter describes, for example, an approach called *deterministic reservations* for parallelizing certain greedy algorithms. In this approach, the user implements a loop with potential loop carried dependencies by splitting each iteration into *reserve* and *commit* phases. The loop is then processed in rounds in which each round takes a prefix of the unprocessed iterates applying the reserve phase in parallel and then the commit phase in parallel. Some iterates can fail during the commit due to conflicts with earlier iterates and need to be retried in the next round, but as long as the operations commute within the reserve and commit phases and the prefix size is selected deterministically, the computation is internally deterministic (for a given round, the same iterates always succeed/fail on every execution).

This chapter describes algorithms for the benchmark problems using these approaches and presents performance results for Cilk implementations of these algorithms on a 32-core machine. Perhaps surprisingly, for all problems, the internally deterministic implementations achieve good speedup and good performance even relative to prior nondeterministic and externally deterministic solutions, implying that the performance penalty of internal determinism is quite low. The experiments show parallel speedups of up to 31.6 on 32 cores with two-way hyper-threading (for sorting), and almost all of the speedups are above 16. Compared to good sequential implementations of the problems, the internally deterministic parallel implementations range from being slightly faster on one core (sorting) to about a factor of 2 slower (spanning forest). All of the internally deterministic algorithms are quite concise (20–500 lines of code), and are "natural" to reason about (understandable, not complicated, not special purpose). This combination of performance and understandability provides significant evidence that internal determinism is a sweet spot for a broad range of computational problems.

3.2 Programming Model

This chapter focuses on achieving *internally deterministic* behavior in nested-parallel programs through "commutative" and "linearizable" operations. Each of these terms limits the programs permitted by the programming model, but as Section 3.4 exhibits, the model remains expressive. This section defines each of these terms.

3.2.1 Nested parallelism

As discussed in Chapter 2, nested-parallel computations achieve parallelism through the nested instantiation of fork-join constructs, such as parallel loops, parallel map, parbegin/-parend, parallel regions, and spawn/sync. Figure 3.1 shows an example of a nested-parallel program using a syntax similar to Dijkstra's `parbegin` [137]. Languages with nested parallelism rely on runtime schedulers to assign sub-computations to cores. Whereas these runtime schedulers are inherently nondeterministic to handle load balancing and changes in available resources, the goal of this chapter is to guarantee that the program nevertheless behaves deterministically.

3.2.2 Internal determinism

This chapter adopts a strong notion of determinism here, often called internal determinism [144, 348]. Not only must the output of the program be deterministic, but all intermediate values returned from operations must also be deterministic. Note that this does not

```
1. $x := 0$
2. **in parallel do**
3.     { $r_3 :=$ AtomicAdd$(x, 1)$ }
4.     { $r_4 :=$ AtomicAdd$(x, 10)$
5.         **in parallel do**
6.             { $r_6 :=$ AtomicAdd$(x, 100)$ }
7.             { $r_7 :=$ AtomicAdd$(x, 1000)$ }
       }
8. return $x$
```

Figure 3.1: An example nested-parallel program. The **in parallel** keyword means that the following two $\{\ldots\}$ blocks of code may execute in parallel. $\texttt{AtomicAdd}(x, v)$ atomically updates x to $x := x + v$ and returns the new value of x.

preclude the use of pseudorandom numbers, where one can use, for example, the approach of Leiserson et al. [297] to generate deterministic pseudorandom numbers in parallel from a single seed, which can be part of the input.

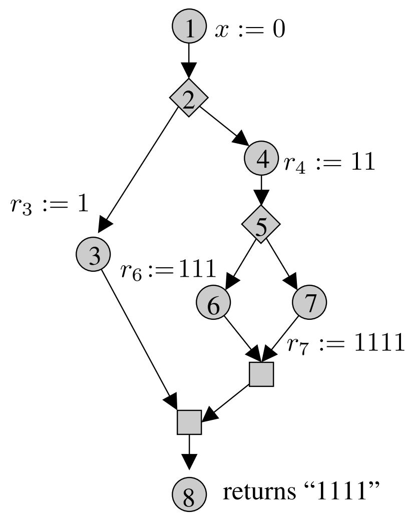
This chapter defines determinism with respect to abstract operations and abstract state, not with respect to machine instructions and memory state. Nevertheless, the definition supplied here is general and applies to both cases. The difference hinges on the notion of "equivalence." Various levels of abstraction have been considered in the literature (see [309] for a discussion). Given a definition of equivalent operations, states, and values, internal determinism is defined as follows.

For a (completed) computation, its ***trace*** is the final state along with the computation DAG on which operation vertices are (further) annotated with the values returned (if any). Figure 3.2 shows two traces corresponding to executions of the program shown in Figure 3.1. Two computation DAGs are equivalent if they have the same graph structure and corresponding vertices are labeled with equivalent operations. Two traces are ***equivalent traces*** if they have equivalent final states, equivalent computation DAGs, and corresponding DAG vertices are annotated with equivalent return values.

Definition 1. *A program is* **internally deterministic** *if for any fixed input I, all possible executions with input I result in equivalent traces.*

Note that since the parallelism is dynamic, a nondeterministic program may result in dramatically different DAGs. Because all decisions in a computation are based only on the result of operations performed, however, if operations return equivalent results despite different schedulings, then the structure of the DAG is guaranteed to remain the same.

For primitive types like integers, it is clear what equivalence means. When working with objects and dynamic memory allocation, however, a formal definition of equivalent objects and states becomes more complicated, and not within the scope of this thesis. Informally,



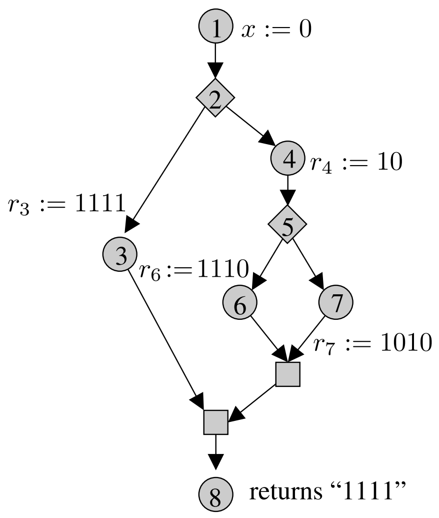


Figure 3.2: Two possible traces for the program in Figure 3.1. The diamonds, squares, and circles denote forks, joins, and data operations, respectively. Vertices are numbered by line number, as a short hand for operations such as `AtomicAdd`$(x, 1)$. The left trace corresponds to the interleaving/schedule $1, 2, 3, 4, 5, 6, 7, 8$, whereas the right trace corresponds to $1, 2, 4, 5, 7, 6, 3, 8$. Because the intermediate return values differ, the program is not internally deterministic. It is, however, externally deterministic as the output is always the same. If `AtomicAdd` did not return a value, however, then the program would be internally deterministic.

when we say that states or values are equivalent, we mean semantically equivalent, i.e., that no sequence of valid operations can distinguish between them (see, e.g., [224]).

3.2.3 Commutativity

Internally deterministic programs are a subset of parallel programs, and thus programming methodologies that yield internal determinism restrict a program's behaviors. The methodology adopted in this chapter is to require all logically parallel accesses of shared objects to use operations that commute. The fact that this restriction yields internally deterministic programs is observed in many works, see, for example, [436, 396, 90] among others.

This chapter adopts Steele's notation and definition of commutativity [436]. We use $f(S) \rightarrow S' \Rightarrow v$ to denote that when the operation f is executed (without any concurrent operations) starting from system (object) state S, the system transitions to state S' and f returns the value v. To simplify notation, operations not returning values are viewed as returning $v = \emptyset$.

Definition 2. *Two operations f and g* **commute with respect to state** S *if the order in which they are performed does not matter. That is, if*

$$f(S) \rightarrow S_f \Rightarrow v_f$$
$$g(S_f) \rightarrow S_{fg} \Rightarrow v_g$$

and

$$g(S) \to S'_g \Rightarrow v'_g$$
$$f(S'_g) \to S'_{gf} \Rightarrow v'_f$$

then f and g commute with respect to S if and only if $S_{fg} = S'_{gf}$, $v_f = v'_f$, and $v_g = v'_g$, where "$=$" here denotes equivalence. (Note that there is no requirement that $S_f = S'_g$.)

Moreover, that two operations are said to ***commute*** if they commute with respect to *all* valid states S. It is possible to relax this definition (e.g., [459, 224]), but this definition is sufficient for the purposes in this chapter.

Linearizability. Commutativity is not a sufficient condition for deterministic behavior, as commutativity alone does not guarantee that the implementation of the operations work correctly when their instructions are interleaved in time. To guarantee safety of concurrent execution of operations this chapter uses the standard definition of linearizability [227], which enforces atomicity of the operations. In this setting, operations are concurrent if and only if they are logically parallel. Thus, linearizability guarantees that there is a total order (or *history*), H, of the annotated operations in a trace T such that H is a legal sequential execution of those operations, starting from the initial state. That is, (i) H is a valid scheduling of T's computation DAG, and (ii) each annotated operation in T remains legal (including its return value) when executed atomically in the order of H. Note that linearizability is a property of the implementation and not the semantics of the operation (e.g., two insertions into a dictionary might semantically commute, but an implementation might fail when interleaved). One way to guarantee linearizability is to use a lock around all commuting operations, but this is inefficient. This chapter uses only non-blocking techniques to achieve linearizability among commuting operations. We however do not guarantee that all commuting operations are linearizable, just that the logically parallel ones are.

Summary. The model this chapter uses for internally deterministic behavior is summarized by the following theorem.

Theorem 1. *Let P be a nested-parallel program. If for all inputs, all logically parallel operations commute and are linearizable, then P is internally deterministic.*

Proof. (Sketch) Consider any fixed input I and any fixed (completed) execution of P with input I. Let G (T) be the resulting computation DAG (trace, respectively), and let H be its linearizability history. The proof will show that T is equivalent to a canonical trace T^* obtained by executing P with input I using only a single core. Let G^* and H^* be the computation DAG and linearizability history, respectively, for T^*. The proof shows by induction on the length of H^* that (i) G and G^* are equivalent and (ii) H permuted to match the order in H^* of equivalent vertices is also a linearizability history for T, implying equivalent return values. Construct such a permutation, H', inductively, with $H' = H$ initially. Assume inductively that (i) the subgraph of G^* corresponding to the vertices in $H^*[1 \ldots i]$ has an equivalent subgraph in G, and (ii) H' is a linearizability history for T such that $H'[1 \ldots i]$ and $H^*[1 \ldots i]$ are equivalent ($[j \ldots k]$ denotes subsequence). Consider $i+1$, and let σ^* be the $i+1$'st annotated vertex in H^*. It follows inductively that there is a vertex σ in T with equivalent parent(s) and an equivalent operation, say the j'th vertex in H'. If $j = i+1$, the proof is done, so assume $j > i+1$. None of the vertices in $H'[i+1 \ldots j-1]$ can precede or be preceded by σ, so σ must commute with each such vertex. Thus, σ can be pairwise swapped up to position $i+1$ in H' while preserving a linearizability history, establishing both inductive invariants. The argument is readily extended to show the equivalence of the final states by augmenting each execution with operations that read the final state. The theorem follows. □

The approach of this chapter is similar to previous models for enforcing deterministic behavior [436, 90], except that in Steele [436] commutativity is defined in terms of memory operations and memory state, and in Cheng et al. [90] commutativity is defined with respect to critical sections and memory state. In this work, commutativity is defined in terms of linearizable abstract operations and abstract state.

3.3 Commutative Building Blocks

Achieving deterministic programs through commutativity requires some level of (object or operation) abstraction. Relying solely on memory operations is doomed to fail for general-purpose programming. For example, requiring a fixed memory location for objects allocated in the heap would severely complicate programs and/or inhibit parallelism, possibly requiring all data to be pre-allocated. Instead, this section defines some useful higher-level operations that are used as commutative operations in many of the algorithms presented later. They are all defined over abstract data types supporting a fixed set of operations. This section also describes non-blocking linearizable implementations of each operation. These implementations do not commute at the level of single memory instructions and hence the abstraction is important.

Priority write. The most basic data type is a memory cell that holds a value, and supports a priority write and a read. The priority write on a cell x, denoted by $x.\texttt{pwrite}(v)$ updates x to be the maximum of the old value of x and a new value v. It does not return any value. $x.\texttt{read}()$ is just a standard read of the cell x returning its value. Priority write is often used to select a deterministic winner among parallel choices, e.g., claiming a next-step neighbor in breadth first search (Section 3.4.4).

Any two priority writes $x.\texttt{pwrite}(v_1)$ and $x.\texttt{pwrite}(v_2)$ commute, in accordance with Definition 2, because (i) there are no return values, and (ii) the final value of x is the maximum among its original value, v_1, and v_2, regardless of which order these operations execute. A priority write and a read do not commute since the priority write can change the value at the location. We implement non-blocking and linearizable priority writes using a compare-and-swap. With this implementation, the machine primitives themselves do not commute. The implementation, further applications, and a detailed experimental study of priority writes will be presented in Chapter 6.

Priority reserve. In the "deterministic reservations" approach described later in Section 3.4, multiple program loop iterates attempt to reserve the same object in parallel, and later the winner operates on the reserved object. Deterministic reservations uses a data type that supports three operations, a priority reserve ($x.\texttt{reserve}(p)$), a check ($x.\texttt{check}(p)$), and a check-and-release ($x.\texttt{checkR}(p)$), where p is a priority. As with a priority write, a higher priority value overwrites a lower priority and hence the highest priority will "reserve" the location. The one difference is that a unique priority tag $\perp$ is required to denote when the location is currently unreserved. The priority $\perp$ has the lowest priority, and it is invalid to make a `pwrite` call with $p = \perp$. As with `pwrite`, any number of `reserves` commute, and we implement a linearizable non-blocking version using compare-and-swap.

The $x.\texttt{checkR}(p)$ call requires $p \neq \perp$. If the current value at location x has priority p, then the reservation is released (i.e., the value $\perp$ is written to x), and TRUE is returned to indicate that p was the highest priority reservation on x. If the current priority is not p, then the state does not change and FALSE is returned. Operations $x.\texttt{checkR}(p_1)$ and $x.\texttt{checkR}(p_2)$ commute if and only if $p_1 \neq p_2$. A `check` is the same as a `checkR` without the release and commutes in the same way. A priority reserve and either form of check do not commute.

The deterministic algorithms in this thesis ensure that for any given location, (i) priority reserves are not called logically in parallel with either form of check, and (ii) all logically parallel operations use distinct priorities. Thus, the commutativity and resulting internal determinism extend to those algorithms.

Dynamic map. The purpose of a dynamic map is to incrementally insert keyed elements and, once finished inserting, to return an array containing a pseudorandom permutation of these elements, omitting duplicates. A dynamic map supports two operations: $M.\texttt{insert}(x)$, which inserts keyed element x into the map M without returning any value, and $M.\texttt{elements}()$, which returns an arbitrary, but deterministic, permutation of all the elements in the map M. The map removes duplicate keys on insert: if elements y and x have the same key and y is already in the map when $M.\texttt{insert}(x)$ is called, one of the elements (chosen deterministically based on a user specified priority) is discarded.

This thesis implements a dynamic map using a parallel version of a history-independent hash table by Blelloch and Golovin [58]. The implementation, proofs of correctness, along with an experimental study of the hash table will be presented in Chapter 5. Chapter 5 shows that two `inserts` commute, however, the $M.\texttt{insert}(x)$ operation does not commute with the $M.\texttt{elements}()$ operation since for some states of S, x is not in M and will affect the result of `elements`.

Disjoint sets. The spanning forest algorithms in this section rely on a structure for maintaining a collection of disjoint sets corresponding to connected components. Each set is associated with a unique element acting as the identifier for the set. A disjoint set data type supports two operations: a find and a link. For an instance F, the $F.\texttt{find}(x)$ operation returns the set identifier for the set containing x. The $F.\texttt{link}(S, x)$ operation requires that S be a set identifier and the set containing x be disjoint from the set S. It logically unions the set S with the set containing x such that the identifier for the resulting unioned set is the identifier of the set containing x. Here, x and S denote references or pointers to elements in the sets.

This section implements an instance F of the disjoint set data type as a collection of trees with parent pointers, where the root of each tree acts as a unique identifier for the set [112]. A $F.\texttt{find}(x)$ operation simply follows parent pointers up the tree and returns the root. It may also perform path compression [112], which points vertices along the query-to-root path directly to the root, thereby accelerating future queries. A $\texttt{link}(S, x)$ operation is implemented by pointing S to the root vertex of the set containing x.

Two `find` operations commute with each other as they cause no semantic modifications—i.e., any changes to the pointer structure caused by path compression cannot be discerned by future operations on F. Two `link` operations commute with each other as long as they do not share the same first argument. That is to say, $F.\texttt{link}(S_1, x_1)$ and $F.\texttt{link}(S_2, x_2)$ commute as long as $S_1 \neq S_2$; having x_1 and x_2 be equal or from the same set is allowed, as is having x_1 in set S_2 or x_2 in set S_1. The $\texttt{link}(S_1, x_1)$ and $\texttt{find}(x_2)$ only commute if $x_1 = x_2$.

Let us now consider linearizability. Even with path compression, `find` operations are linearizable (and non-blocking) since there is only one possible update to each pointer (the a priori root of the tree). This requires no compare-and-swap or any other special memory operations. Logically parallel `link` operations with distinct first arguments, and no cycles among the linked sets, are also linearizable and non-blocking with no special memory operations since they only require updating a pointer which is not shared by any other logically parallel operation. In the implementation, `find`'s and `link`'s are not guaranteed to be linearizable. Hence, in the algorithms that use disjoint sets, `find`'s are never logically parallel with `link`'s: they alternate phases of only `find`'s and only `link`'s.

Note that we are using an asymmetric `link` operation instead of the standard symmetric `union`. This is because `union` does not commute according to Definition 2, which requires two operations to commute for all start states. In a more relaxed definition of commutativity, `union` can be made to commute [280].

3.4 Internally Deterministic Parallel Algorithms

3.4.1 Benchmark Problems

For testing the utility of nested parallel internally deterministic algorithms, this chapter uses a set of benchmarks from the Problem Based Benchmark Suite (described in Section 1.4). It is important that the benchmarks are *problem-based* since it might be that very different algorithmic approaches are suited for a deterministic algorithm versus a nondeterministic algorithm. The problems studied in this chapter are shown in Figure 3.1, and their definitions can be found in Section 2.6. The benchmarks are selected to cover a reasonable collection of fundamental problems. The focus, however, is on problems involving unstructured data since there is already very good coverage for such benchmarks for linear algebra and typically deterministic algorithms are much simpler for these problems.

The rest of this section describes the approaches we use when designing internally deterministic parallel algorithms for the benchmark problems and outlines the resulting algorithms for each of the benchmarks. Many of the approaches used are standard, but this section introduces a new approach for greedy algorithms, called deterministic reservations. The approach plays a key role in the implementation of several of the problems. The algorithms also make use of our commuting and linearizable implementations of various operations. Table 3.1 summarizes what approaches/techniques are used in which of the algorithms.

3.4.2 Nested Data Parallelism and Collection Operations

The most common technique throughout the benchmark implementations is the use of nested data parallelism. This technique is applied in a reasonably standard way, particularly in the use of fork-join and parallel loops (with arbitrary nesting) in conjunction with parallel operations on collections. For the operations on collections, the implementations use a library of operations on sequences, developed as part of the Problem Based Benchmark Suite. The operations make heavy use of divide-and-conquer. In the divide-and-conquer algorithms, the implementations almost always use parallelism within the divide step (to partition the input data), and/or the merge step (to join the results), typically using the collection operations in the sequence library. The three primitives, `reduce`, `scan` and `filter` are used throughout the algorithms, and are defined in Section 2.3. The PBBS implementations of `reduce` and `scan` are deterministic even if f is not associative—e.g., with floating point addition.

Reduce is used to calculate various "sums": e.g., to calculate the bounding box (max-

Problem	**D&C**	**Reduce**	**Scan**	**Filter**	**DR**	**CL**
Comparison Sort	yes		yes			
Remove Duplicates				yes		DM
Breadth First Search			yes	yes		PW
Spanning Forest				yes	yes	DS
Min Spanning Forest	sub			yes	yes	DS
Triangle Ray Intersect	yes		yes	yes		
Suffix Array	sub	yes	yes	yes		
Delaunay Triangulation	sub	yes	sub	yes	yes	
Delaunay Refine		yes		yes	yes	DM
N-body	yes	yes	yes			
K-Nearest Neighbors	sub			yes		

Table 3.1: Techniques used in the algorithms for each of the benchmarks. D&C indicates divide-and-conquer; Reduce, Scan and Filter are standard collection operations; DR indicates deterministic reservations; and CL indicates the use of a non-trivial commutative and linearizable operation other than reservations: dynamic map (DM), disjoint sets (DS), or priority write (PW). **sub** indicates that it is not used directly, but inside a subroutine, e.g., inside a sort.

imum and minimum in each coordinate) of a set of points. Filter is used in most of the algorithms. In the divide-and-conquer algorithms, it is typically used to divide the input into parts based on some condition. In the other algorithms, it is used to filter out elements that have completed or do not need to be considered. It plays a key role in deterministic reservations. Scan is used in a variety of ways. In the sorting algorithm it is used to determine offsets for the sample sort buckets, in the suffix array algorithm it is used to give distinct elements unique labels, and in the breadth-first search algorithm it is used to determine the positions in the output array to place distinct neighbor arrays.

3.4.3 Deterministic Reservations

Several of the deterministic algorithms in this thesis (spanning forest, minimum spanning forest, Delaunay triangulation, Delaunay refinement, maximal independent set, maximal matching, random permutation, list contraction, and tree contraction) are based on a greedy sequential algorithm that processes elements (e.g., vertices) in linear order. These can be implemented using speculative execution on a sequential loop that iterates over the elements in the greedy order.

Various studies have suggested both compiler [396, 383] and runtime techniques [437, 219] to automate the process of simulating in parallel the sequential execution of such a loop. These approaches rely on recognizing at compile and/or run time when operations in the loop iterates commute and allowing parallel execution when they do. Often the programmer can specify what operations commute. We are reasonably sure that the compiler-only

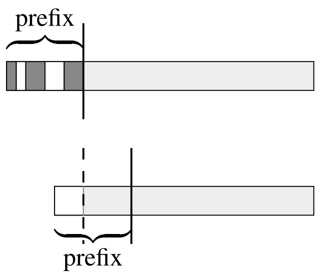


Figure 3.3: A generic example of deterministic reservations. The top and the bottom depict the array of iterates during consecutive rounds. In each round, a prefix of some specified size is selected. All of these prefix iterates perform the reserve component. Then they all perform the commit component. The dark regions in the top array represent iterates that successfully commit. All uncommitted iterates (shown in white) are packed towards the right, as shown in the bottom array. The next round then begins by selecting a prefix of the same size on the bottom array.

techniques would not work for the benchmark problems in this chapter because the conflicts are highly data-dependent and any conservative estimates allowing for all possible conflicts would serialize the loop. The runtime techniques typically rely on approaches similar to software transactional memory: the implementation executes the iterations in parallel or out-of-order but only commits any updates after determining that there are no conflicts with earlier iterations. As with software transactions, the software approach is expensive, especially if required to maintain strict sequential order. In fact, in practice the suggested approaches typically relax the total order constraint by requiring only a partial order [383], potentially leading to nondeterminism. A second problem with the software approach is that it makes it very hard for the algorithm designer to analyze efficiency—it is possible that subtle differences in the under-the-hood conflict resolution could radically change which iterates can run in parallel.

This section presents an approach, called ***deterministic reservations***, that gives more control to the algorithm designer and fits strictly within the nested parallel framework (needing neither special compiler nor runtime support). In this approach, the algorithm designer controls exactly on what data the conflicts occur and these conflicts are deterministic for a given input. The generic greedy algorithm for deterministic reservations works as follows, illustrated in Figure 3.3. It is given a sequence of iterates (e.g., the integers from 0 to $n-1$) and proceeds in rounds until no iterates remain. Each round takes any prefix of the remaining unprocessed iterates, and consists of two phases that are each parallel loops over the prefix, followed by some bookkeeping to update the sequence of remaining iterates. The first phase executes a *reserve* component on each iterate, using a priority reserve (`reserve`) with the iterate priority, in order to reserve access to data that might interfere (involve non-commuting or non-linearizable operations) with other iterates. The second phase executes a *commit* component on each iterate, using a `check` to see if the reservations succeeded, and if the required reservations succeed then the iterate is processed, otherwise it is not. Typically updates to shared state (at the abstraction level available to the programmer) are only made if successful. After running the commit phase, the processed iterates are removed. In the implementation of deterministic reservations, the unprocessed iterates are kept in a contiguous array ordered by their priority. Selecting a prefix can therefore just use a prefix of the array, and removing processed iterates can be implemented with a `filter` over the boolean results of the second phase.

The specifics of the reserve and commit components depend on the application. The work done by the iterate can be split across the two components. We have found that in the unstructured problems in the benchmarks, just determining what data might interfere involves most of the work. Therefore, the majority of the work ends up in the reserve component. In most cases, all of the reservations are required to succeed, but we have encountered cases in which only a subset need to succeed (e.g., the minimum spanning-forest code reserves both endpoints of an edge but only requires that one succeeds).

It is worth noting that the generic approach can select any prefix size including a single iterate or all of the iterates. There is a trade-off, however between the two extremes. If too many iterates are selected for the prefix, then many iterates can fail. This not only requires repeated effort for processing those iterates, but can also potentially cause high contention on the reservation slots. On the other hand, if too few iterates are selected then there might be insufficient parallelism. Clearly the amount of contention depends on the specific algorithms and also on the input data. The effect of contention in deterministic reservations is studied in more detail in Chapter 6.

As long as the prefix size is selected deterministically, and all operations commute and are linearizable within the reserve phase and separately within the commit phase, a program will be internally deterministic. This means the algorithm designer only needs to analyze commutativity/linearizability within each phase. In our code, we have implemented a function `speculative_for` that takes four arguments: a structure that implements the `reserve` and `commit` components (both taking an index as an argument), a start index, an end index, and a prefix size.

The next section includes several algorithms (spanning forest, minimum spanning forest, Delaunay triangulation, and Delaunay refinement) that use the deterministic reservations approach. Chapter 4 introduces several additional algorithms (maximal independent set, maximal matching, random permutation, list contraction, and tree contraction) implemented using deterministic reservations that have provably strong work and depth bounds.

3.4.4 Algorithms

This section describes each of the algorithms used to implement the benchmarks discussed in Section 3.4.1. In all cases, my co-authors and I considered a variety of algorithms and selected the one we felt would perform the best. In many cases, we arrived at the algorithm discussed after trying different algorithms. In all cases, the algorithms are either motivated by or directly use results of many years of research on parallel algorithm design by many researchers.

Comparison Sort. We use a low-depth cache-efficient sample sort [57]. The algorithm (1) partitions the input into $\sqrt{n}$ blocks, (2) recursively sorts each block, (3) selects a global sample of size $\sqrt{n}\log n$ by sampling across the blocks, (4) sorts the sample, (5) buckets each of the blocks based on the sample, (6) transposes the keys so keys from different blocks going to the same bucket are adjacent, and (7) recursively sorts within the buckets. The transpose uses a cache-efficient block-transpose routine. When the input is small enough, quicksort is used. The algorithm is purely nested parallel. There is nesting of the parallelism (divide-and-conquer) in the overall structure, in the merge used for bucketing blocks, in the transpose, and in the quicksort.

Remove Duplicates. We use a parallel loop to concurrently `insert` the elements into the dynamic map described in Section 3.3. This data structure already removes all duplicates internally and returns the distinct elements with a call to `elements` (which internally uses a `filter`). The ordering returned by the routine is deterministic, but does not correspond to the input ordering in any natural way and different hash functions will give different orderings. The hash table size is set to be twice the size of the input rounded up to the nearest power of 2.

Breadth First Search (BFS). We use a level-ordered traversal of the graph. In level-order traversal, each vertex u adds each of its unvisited neighbors v to the next frontier and makes u the parent of v in the BFS tree. In standard parallel implementations of BFS [296, 383], each level is processed in parallel and nondeterminism arises because vertices at one level might share a vertex v at the next level. These vertices will attempt to add v to the next frontier concurrently. By using a compare-and-swap or similar operation, it is easy to ensure that a vertex is only added once. However, which vertex adds v depends on the schedule, resulting in internal nondeterminism in the BFS code and external nondeterminism in the resulting BFS tree.

We avoid this problem by using a priority write. The vertices in the frontier are prioritized by their ID and each level involves two rounds. In the first round, each vertex in the frontier writes its priority to all neighbors that have not been visited in previous rounds. In the second round, each vertex v in the frontier reads from each neighbor u the priority. If the priority of u is v (v is the highest priority neighbor in the frontier), then the implementation makes v the parent of u and adds u to the next frontier. The neighbors are added to the next frontier in the priority order of the current frontier. This uses a `scan` to open enough space for each neighbor list.

Spanning Forest. Sequentially, a spanning forest can be generated by greedily processing

```
struct STStep {
  int u;  int v;
  edge *E;  res *R;  disjointSet F;
  STStep(edge* _E, disjointSet _F, res* _R)
    : E(_E), R(_R), F(_F) {}

  bool reserve(int i) {
    u = F.find(E[i].u);                                              //find component
    v = F.find(E[i].v);                                              //find component
    if (u == v) return 0;           //skip edge if endpoints belong to the same component
    if (u > v) swap(u,v);
    R[v].reserve(i);                                         //reserve larger component
    return 1;}

  bool commit(int i) {
    if (R[v].check(i)) { F.link(v, u); return 1;}     //link if reservation was successful
    else return 0;  }
};

void ST(res* R, edge* E, int m, int n, int psize) {
  disjointSet F(n);                              //deterministic union-find data structure
  speculative_for(STStep(E, F, R), 0, m, psize);       //deterministic reservations driver
}
```

Figure 3.4: C++ code for spanning forest using deterministic reservations (with its operations `reserve`, `check`, and `speculative_for`).

the edges in an arbitrary order using a disjoint set data structure. When an edge is processed, if the two endpoints are in the same component (which can be checked with `find`) then it is removed, otherwise the edge is added to the spanning forest and the components are joined (with `union`). This algorithm can be run in parallel using deterministic reservations prioritized by the edge ordering and will return the exact same spanning forest as the sequential algorithm. The idea is simply to reserve both endpoints of an edge and check that both reservations succeed in the commit component. Indeed this is how we implement minimum spanning forest, after sorting the edges. However there is an optimization that can be made with spanning forests that involves only requiring one of the reservations to succeed. This increases the probability a commit will succeed and reduces the cost. This approach returns a different forest than the sequential version but is internally deterministic for a fixed schedule of prefix sizes.

The C++ code is given in Figure 3.4. For an iterate i corresponding to the edge $E[i]$, the reserve component does a `find` on each endpoint (as in the sequential algorithm) returning u and v (without loss of generality, assume $u \leq v$). If $u = v$, the edge is within a component and can be dropped returning 0 (false),[1] otherwise the algorithm reserves v with the index i ($R[v]$.`reserve`(i)). The commit component for index i performs a $R[v]$.`check`(i) to see if its reservation succeeded. If it has, it links v to u and otherwise the commit fails. At the

[1]If false is returned by `reserve`(), then the iterate is dropped without proceeding to the commit.

end of the algorithm the edges $E[i]$ in the spanning tree can be identified as those where $R[i] \neq \perp$. The only difference from the sequential algorithm is that after determining that an edge goes between components, instead of doing the union immediately it reserves one of the two sides. It later comes back to check that the reservation succeeded and if so does the union (link).

Note that in a round the reservation guarantees that only one edge (the highest priority) will link a vertex v to another vertex. This is the condition required in Section 3.3 for commutativity of `link`. Also because the `link` and `find` are in different phases they are never logically parallel, as required. Finally, note that because the algorithm links higher to lower vertex numbers, it will never create a cycle. In this algorithm our code sets `psize`, the size of the prefix, to be $.02m$ and we have observed that on our test graphs less than 10% of the reservations fail.

Minimum Spanning Forest (MSF). We use a parallel variant of Kruskal's algorithm [112]. The idea of Kruskal's algorithm is to sort the edges and then add them one-by-one using disjoint sets as in the spanning forest code. Therefore, deterministic reservations prioritized by the sorted order to insert the edges can be used. Unlike the spanning forest described above, however, both endpoints of an edge need to be reserved to guarantee the edges are inserted in "sequential" order. However, during the commit component, only one of the two endpoint needs to succeed because to commute `link` only requires that one of the two arguments is unique. If v succeeds, for example, then the code uses `link`(v, u). Note this is still internally deterministic because which endpoints succeed is deterministic. The code uses a further optimization: It sorts only the smallest k edges ($k = \min(m, 4n/3)$ in the experiments) and runs MSF on those, so that the remaining edges can be filtered out avoiding the need to sort them all. The baseline sequential MSF algorithm also uses the same optimization.

Triangle Ray Intersect. We use a k-d tree with the surface area heuristic (SAH) [312] to store the triangles. The algorithm is similar to the parallel algorithm discussed in [96] and makes use of divide-and-conquer and heavy use of `scan` and `filter`.

Suffix Array. We use a parallel variant of the algorithm of Karkkainen and Sanders [256]. It uses sorting and merging as subroutines, which involves nesting, but otherwise only makes use of `reduce`, `scan`, and `filter`.

Delaunay Triangulation. We use a Bowyer-Watson style incremental Delaunay triangulation algorithm [127] with deterministic reservations. The points are used as the elements. To reduce contention, the prefix is always selected to be smaller than the current size of the mesh. The algorithm therefore starts out sequentially until enough points have been added. The reserve component of the code, for a point p, identifies all triangles that contain p in their circumcircle, often referred to as the hole for p. Adding p requires removing the hole and replacing it with other triangles. The reserve component therefore reserves all vertices around the exterior of the hole. The majority of the work required by a point p is in locating p in the mesh and then identifying the triangles in the hole. The commit component checks if all the reserved vertices of the mesh have succeeded, and if so, removes the hole and replaces it with triangles surrounding p and filling the hole. The reservations ensure that all modifications to the mesh commute since the triangles in the mesh only interact if they share a vertex. In fact, reserving the edges of the hole would be sufficient and reduce contention, but our mesh implementation has no data structures corresponding to edges on which to reserve. For efficiently locating a point p in the mesh, the nearest neighbor structure described below is used.

Delaunay Refinement. This algorithm uses the same routines for inserting points as the Delaunay triangulation. However, it does not need a point location structure but instead needs a structure to store the bad triangles. A dynamic map is used for this purpose.

N-body. We use a parallel variant of the Callahan-Kosaraju algorithm [85]. This is a variant of Greengard and Rokhlin's well-known FMM algorithm [194] but allows more flexibility in the tree structure. The algorithm makes use of traditional nested parallelism with divide-and-conquer, as well as `reduce` and `scan`.

K-Nearest Neighbors. We use a quad- and oct-tree built over all input points for 2d and 3d inputs, respectively. As with the k-d tree used in triangle-ray intersection, the tree is built using only divide-and-conquer and nested parallelism. Once built, the tree is static and used only for queries of the points.

3.5 Experimental Results

This section reports experimental results for the internally deterministic algorithms on the 32-core Intel machine described in Section 2.7. The parallel programs were compiled using the `cilk++` compiler, and sequential programs were compiled using `g++`. Experiments are presented for all of the benchmarks described in Section 3.4, except for remove duplicates, which will be discussed in detail in Chapter 5. The results are summarized in Table 3.2, which reports the average timings over all inputs for each implementation.

Four of the benchmarks will be discussed in detail, and their performance is compared to other published results at that time of the publication of this work [53]. For each benchmark, given core count, and input, Table 3.3 reports the median time over three trials.

For comparison sort, the experiments use a variety of inputs all of length 10^7. This includes sequences of doubles in three distributions and two sequences of character strings. Both sequences of character strings are the same but in one the strings are allocated in order (i.e., adjacent strings are likely to be on the same cache line) and in the other they are randomly permuted. The internally deterministic sample sort is compared to three other

Application Algorithm	**1 thread**	**64 threads** (32h)	**Speedup**
Comparison Sort			
serialSort	3.581	–	–
*stlParallelSort	3.606	0.151	23.88
sampleSort	2.812	0.089	31.6
quickSort	3.043	0.68	4.475
Breadth First Search			
serialBFS	3.966	–	–
**ndBFS	5.4	0.28	19.29
deterministicBFS	7.136	0.314	22.73
**LS-PBFS†	4.357	0.332	13.12
Spanning Forest			
serialSF	2.653	–	–
deterministicSF	6.016	0.326	18.45
**Galois-ST$	12.39	1.136	10.91
Minimum Spanning Forest			
serialMSF	8.41	–	–
parallelKruskal	14.666	0.785	18.68
Triangle Ray Intersect			
kdTree	8.7	0.45	19.33
Suffix Array			
parallelKS	13.4	0.785	17.07
Delaunay Triangulation			
serialDelaunay	56.95	–	–
deterministicDelaunay	80.35	3.87	20.76
*Galois-Delaunay	114.116	39.36	2.9
Delaunay Refine			
deterministicRefine	103.5	6.314	16.39
**Galois-Refine‡	81.577	5.201	15.68
N-body			
parallelCK	122.733	5.633	21.79
K-Nearest Neighbors			
octTreeNeighbors	37.183	3.036	12.25

Table 3.2: Weighted average of running times (seconds) over various inputs on a 32-core machine with hyper-threading (32h). A "*" indicates an internally nondeterministic implementation and a "**" indicates an externally (and hence internally) nondeterministic implementation. All other implementations are internally deterministic. †LS-PDFS does not generate the BFS tree, while the programs in this chapter do. $Galois-ST generates only a spanning tree, while the code in this chapter generates the spanning forest. ‡Galois-Refine does not include the time for computing the triangle neighbors and initial bad triangles at the beginning while the code in this chapter does (takes 10-15% of the overall time).

sorting routines: the standard template library (STL) sort, the parallel STL sort [432], and a simple divide-and-conquer quicksort that makes parallel recursive calls but partitions the keys sequentially. The results are summarized in Tables 3.2 and 3.3(a), and Figure 3.5(a). Due to the cache-friendly nature of the sample sort algorithm, on average it is more efficient than any of the algorithms even on one core, and it gets an average parallel speedup of

(a) **Comparison Sort Algorithm**	10^7 random (1)	10^7 random (32h)	10^7 exponential (1)	10^7 exponential (32h)	10^7 almost sorted (1)	10^7 almost sorted (32h)	10^7 trigram (1)	10^7 trigram (32h)	10^7 trigram (permuted) (1)	10^7 trigram (permuted) (32h)
serialSort	1.42	–	1.1	–	0.283	–	4.31	–	5.5	–
*stlParallelSort	1.43	0.063	1.11	0.057	0.276	0.066	4.31	0.145	5.57	0.236
sampleSort	2.08	0.053	1.51	0.042	0.632	0.028	3.21	0.095	3.82	0.131
quickSort	1.58	0.187	1.06	0.172	0.357	0.066	3.35	0.527	4.78	1.31

(b) **BFS Algorithm**	random local graph $n = 10^7$ $m = 5 \times 10^7$ (1)	(32h)	rMat graph $n = 2^{24}$ $m = 5 \times 10^7$ (1)	(32h)	3d grid $n = 10^7$ (1)	(32h)
serialBFS	4.14	–	4.86	–	2.9	–
**ndBFS	6.07	0.226	6.78	0.294	3.35	0.322
deterministicBFS	7.13	0.255	9.25	0.345	5.03	0.343
**LS-PBFS	4.644	0.345	5.404	0.426	3.023	0.225

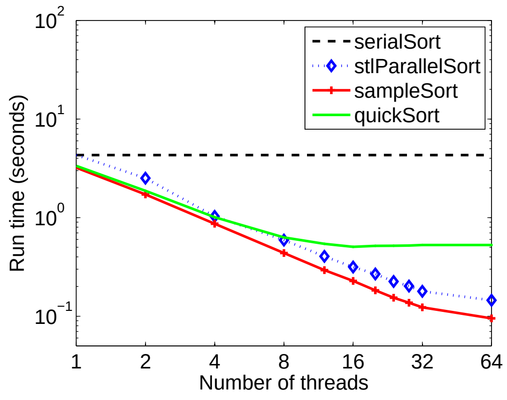
(c) **MSF Algorithm**	random local graph $n = 10^7$ $m = 5 \times 10^7$ (1)	(32h)	rMat graph $n = 2^{24}$ $m = 5 \times 10^7$ (1)	(32h)	2d grid $n = 10^7$ (1)	(32h)
serialMSF	8.47	–	11.2	–	5.56	–
parallelKruskal	14.3	0.78	19.7	1.08	10.0	0.49
*Galois-Boruvka†	–	–	–	–	35.128	7.159

(d) **Delaunay Triangulation Algorithm**	2d in cube $n = 10^7$ (1)	(32h)	2d kuzmin $n = 10^7$ (1)	(32h)
serialDelaunay	55.1	–	58.8	–
deterministicDelaunay	76.7	3.5	84.0	4.24
*Galois-Delaunay	110.705	39.333	117.527	36.302

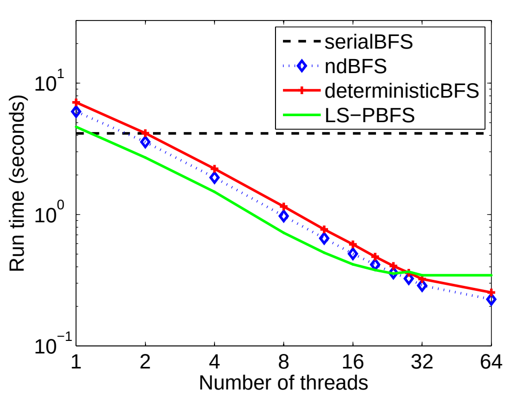
Table 3.3: Running times (seconds) of algorithms over various inputs on a 32-core machine (with hyper-threading). A "*" indicates an internally nondeterministic implementation and a "**" indicates an externally (and hence internally) nondeterministic implementation. †Galois-Boruvka did not terminate in a reasonable amount of time for the first two inputs.

31.6x on 32 cores with hyper-threading. It is not quite as fast on the double-precision values since there the cache effects are less significant. As expected, the quicksort with serial partitioning does not scale.

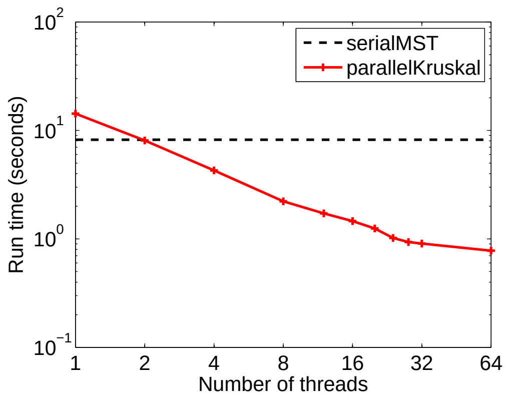
For breadth-first search (BFS), and all of the graph algorithms, three types of graphs were used: random graphs, grid graphs, and rMat graphs [87]. The rMat graphs have a power-law distribution of degrees. All edge counts are the number of undirected edges—the



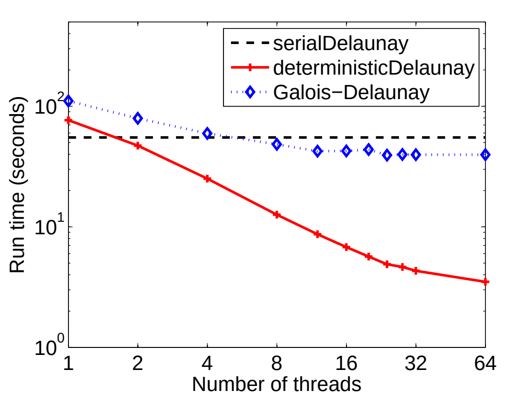
(a) comparison sorting algorithms with a **trigram string** of length 10^7



(b) BFS algorithms with a **random local graph** $(n = 10^7, m = 5 \times 10^7)$



(c) MST algorithms with a **weighted random local graph** $(n = 10^7, m = 5 \times 10^7)$



(d) Delaunay Triangulation algorithms with a **2d in cube graph** $(n = 10^7)$

Figure 3.5: Log-log plots of running times on a 32-core machine (with hyper-threading). The deterministic algorithms are shown in red.

implementations actually store twice as many since they store the edge in each direction. The experiments compare the internally deterministic BFS (deterministicBFS) to a serial version (serialBFS) and a nondeterministic version (ndBFS). The results are summarized in Tables 3.2 and 3.3(b), and Figure 3.5(b). The nondeterministic version is slightly faster than the deterministic version due to the fact that it avoids the second phase when processing each round. The average parallel speedups on 32 cores of the deterministic and nondeterministic versions are 22.7x and 19.3x, respectively. The experiments also compare to published results at the time of this work. We ran the parallel breadth-first search algorithm from [296] and our performance is very close to theirs (their algorithm is labeled LS-PBFS in the tables and figures). Our performance is 5 to 6 times faster than the times reported in [219] (both for 1 thread and 32 cores), but their code is written in Java instead of C++ and is on a Sun Niagara T2 processor which has a clock speed of 1.6GHz instead of 2.26GHz so it is hard to compare directly. Since the publication of this work [53], there have been faster (nondeterministic) implementations of BFS developed [420, 32, 467, 468]. One such implementation is discussed in Chapter 7.

For minimum spanning forest (MSF), the experiments compare the internally deterministic parallel algorithm to an optimized version of Kruskal's serial algorithm (see Section 3.4). The results are shown in Tables 3.2 and 3.3(c), and Figure 3.5(c). Our parallel code is about 1.7x slower on a single thread, and achieves 18–20x speedup on 32 cores. The experiments also compare to the parallel version of Boruvka's algorithm from the C++ release (2.1.0) of the Galois benchmark suite [379] (labeled as Galois-Boruvka in the table). Their code did not terminate in a reasonable amount of time on the random and rMat graphs; for the 2D-grid graph, our code is much faster and achieves much better speedup than their algorithm.

For Delaunay triangulation, the experiments use two point distributions: points distributed at random and points distributed with the Kuzmin distribution. The latter has a very large scale difference between the largest and smallest resulting triangles. The experiments compare the internally deterministic algorithm to a quite optimized serial version. The results are shown in Tables 3.2 and 3.3(d), and Figure 3.5(d). On one thread, the parallel code is a factor of about 1.4 slower, but it gets a speedup of 20–22x on 32 cores. The experiments also compare to the implementations in the Galois benchmark suite [379] (labeled as Galois-Delaunay and Galois-Refine in the tables and figures), and our triangulation code is faster and achieves better speedup on the same machine.[2] Note, however, that on the Delaunay refinement problem our code achieves almost the same running time as the Galois benchmarks (after subtracting the time for computing the initial processing of triangles from our times, which is about 10–15% of the overall time, since this is not part of the timing in the Galois code). Since the time for the refinement code is dominated by triangle insertion and the code for triangulation is dominated by point location, it would appear that the reason for our improved performance is due to our point location data structure, and that triangle insertion performs about equally well in both cases.

[2]The Galois code has been improved since the publication of this work.

Chapter 4

Deterministic Parallelism in Sequential Iterative Algorithms

4.1 Introduction

Over the past several decades there has been significant research on deriving new parallel algorithms for a variety of problems, with the goal of designing highly parallel (polylogarithmic depth), work-efficient algorithms. For some problems, however, one might ask if perhaps a standard sequential algorithm is already highly parallel if sub-computations are simply executed opportunistically when they no longer depend on any other uncompleted sub-computations. This approach is particularly applicable in iterative or greedy algorithms that iterate (loop) once through a sequence of *steps* (or elements), each step depending on the results or effects of only a subset of previous steps. In such algorithms, instead of waiting for its turn in the sequential order, a given step can run immediately once all previous steps it depends on have been completed. The approach allows for steps to run in parallel while performing the same computations on each step as the sequential algorithm, and consequently returning the same result. Surprisingly, this question has rarely been studied.

Beyond the intellectual curiosity of whether sequential algorithms are inherently parallel, the approach has several important benefits for the design of parallel algorithms. Firstly, it can lead to very simple parallel algorithms. In particular, if there is an easy way to check for dependencies, then the parallel algorithm will be very similar to the sequential one. Iterative/greedy parallel algorithms can be naturally implemented in the deterministic reservations framework described in the previous chapter (Section 3.4.3). Secondly, the approach can lead to very efficient parallel algorithms. Using deterministic reservations, this chapter shows that if a sufficiently small prefix of the uncompleted iterations are processed at a time, then most steps do not depend on each other and can run immediately. This reduces the overhead for repeated checks and leads to work which is hardly any greater than that of the sequential algorithm. Finally, the parallelization of the sequential algorithm will be deterministic, returning the same result on each execution (assuming the same source of random numbers). The result of the algorithm will therefore be independent of how many threads are used, how the scheduler works, or any other nondeterminism in the underlying hardware and software, which can make debugging and reasoning about parallel programs much easier, as discussed in Chapter 3.

This chapter studies the theoretical properties of several of these algorithms—maximal independent set, maximal matching, random permutation, list contraction, and tree contraction. The chapter also presents a detailed experimental study of these algorithms implemented using the deterministic reservations framework introduced in Section 3.4.3. Background and previous work for each of the problems, and our new results for the problem are described below.

Maximal Independent Set. The maximal independent set (MIS) is a fundamental problem in parallel algorithms with many applications [310] (recall the definition from Section 2.6). For example, if the vertices represent tasks and each edge represents the constraint that two tasks cannot run in parallel, then the MIS finds a maximal set of tasks to run in parallel. Parallel algorithms for the problem have been well-studied [260, 310, 8, 185, 182, 184, 183, 111, 84]. Luby's randomized algorithm [310], for example, runs in $O(\log n)$ depth on $O(m)$ cores of a CRCW PRAM and can be converted to run in linear work. The problem, however, is that on a modest number of cores it is very hard for these parallel algorithms to outperform the very simple and fast sequential greedy algorithm. Furthermore, the parallel algorithms give different results than that of the sequential algorithm. This can be undesirable in a context where one wants to choose between the algorithms based on platform but wants deterministic answers.

This chapter shows that, perhaps surprisingly, a trivial parallelization of the sequential greedy algorithm is in fact highly parallel (polylogarithmic depth) when the order of vertices is randomized. In particular, removing a vertex as soon as an earlier neighbor is added to the MIS, or adding it to the MIS as soon as no earlier neighbors remain gives a parallel linear-work algorithm. The MIS returned by the sequential greedy algorithm, and hence also its parallelization, is referred to as the ***lexicographically first*** MIS [110]. In a general undirected graph and an arbitrary ordering, the problem of finding a lexicographically first MIS is P-complete [110, 195], meaning that it is unlikely that any efficient low-depth parallel algorithm exists for this problem.[1] Moreover, it is even P-complete to approximate the size of the lexicographically first MIS [195]. The results in this chapter show that for any

[1]Cook [110] shows this for the problem of finding the lexicographically first maximal clique, which is equivalent to finding the MIS on the complement graph.

graph and for the vast majority of orderings, the algorithm for finding the lexicographically first MIS has polylogarithmic depth.

Our results generalize the work of Coppersmith et al. [111] (CRT) and Calkin and Frieze [84] (CF). CRT provide a greedy parallel algorithm for finding a lexicographically first MIS for a random graph $G_{n,p}$, $0 \leq p \leq 1$, where there are n vertices and the probability that an edge exists between any two vertices is p. It runs in $O(\log^2 n / \log\log n)$ expected depth on a linear number of cores. CF give a tighter analysis showing that this algorithm runs in $O(\log n)$ expected depth. They rely heavily on the fact that edges in a random graph are uncorrelated, which is not the case for general graphs, and hence their results do not extend to our context. This chapter, however, uses a similar approach of analyzing prefixes of the sequential ordering.

Maximal Matching. The maximal matching (MM) of G can be solved by finding an MIS of its line graph (the graph representing adjacencies of edges in G), but the line graph can be asymptotically larger than G. Instead, the efficient (linear-work) sequential greedy algorithm goes through the edges in an arbitrary order, adding an edge if no adjacent edge has already been added. As with MIS, this algorithm is naturally parallelized by adding in parallel all edges that have no earlier neighboring edges. The results for MIS directly imply that this algorithm has polylogarithmic depth for random edge orderings with high probability. This chapter also shows that with appropriate prefix sizes the algorithm runs in linear work. Previous work has shown polylogarithmic-depth and linear-work algorithms for the MM problem [239, 238] but as with MIS, the MM algorithm in this chapter returns the same result as the sequential algorithm and leads to very efficient code. Subsequent to this work, Birn et al. [43] have developed a simple parallel maximal matching algorithm, although again it does not return the same result as the sequential algorithm.

Random Permutation. This chapter considers Durstenfeld's well-known algorithm for randomly permuting a sequence of n values [139, 270]. The algorithm iterates through the sequence from the end to the beginning (or the other way) and for each location i, it swaps the value at i with the value at a random target location j at or before i. In the algorithm, each step can depend on previous steps since on step i the value at i and/or its target j might have already been swapped by a previous step. The question is: What does this dependence structure look like? Also, can the above approach be used to derive a highly parallel, work-efficient parallelization of the sequential algorithm?

Generating random permutations in parallel has been well-studied, both theoretically [10, 11, 122, 169, 173, 174, 203, 205, 335, 388] and experimentally [109, 204]. Many of these algorithms do linear work and have polylogarithmic depth. As far as we know, however, none of this previous work has considered the parallelism available in Durstenfeld's sequential algorithm, and none of them return the same permutation as it does, given the same source of randomness.

This chapter shows that Durstenfeld's random permutation algorithm as described above has a dependence structure that follows the same distribution over the random choices as random binary search trees. This implies an algorithm with $\Theta(\log n)$ depth with high probability. A straightforward linear-work polylogarithmic-depth implementation of the algorithm is also presented. Therefore the "sequential" algorithm is effectively parallel.

List Contraction. The list contraction problem is to contract a set of linked lists each into a single node (possibly combining values), and has many applications including list ranking and Euler tours [259, 243, 395]. The sequential algorithm considered in this chapter simply iterates over the nodes in random order splicing each one out.[2] This chapter shows that for this algorithm, each linked list has a dependence structure that follows the same distribution as random binary search trees, giving a $O(\log n)$ depth parallel algorithm w.h.p. Again, a straightforward linear-work parallel implementation of the algorithm is presented.

Tree Contraction. The tree contraction problem is to contract a tree into a single node (possibly combining node values), and again has many applications [335, 336, 243]. This chapter assumes that the tree is a rooted binary tree. The sequential algorithm that is considered iterates over the leaves of the tree in random order and, for each leaf, it splices the leaf and its parent out. This chapter shows that the dependence structure of this problem is shallow (logarithmic dependence length). Unfortunately, there seems to be no easy on-line way to determine when a step no longer depends on any other uncompleted steps. However, with some pre-processing, the dependencies can be identified. This leads to a linear-work parallelization of the algorithm.

Reducing Randomness for Random Permutation and List Contraction. Reducing the randomness required by algorithms is important, as randomness can be expensive. Straightforward implementations of the algorithms from this chapter require $O(n \log n)$ random bits. By making use of a pseudorandom generator for space-bounded computations by Nisan [354], we show that the algorithms for random permutation and list contraction require only a polylogarithmic number of random bits w.h.p. This result is based on leveraging the low depth of the algorithms to show that they can be simulated in polylogarithmic space.

Experiments. We have implemented all of our algorithms in the deterministic reservations framework (described in Section 3.4.3), and run experiments on shared-memory multicore machines. The implementations contain under a dozen to a few dozen lines of `C++` code. Experiments in this chapter show that achieving work-efficiency is indeed important for good performance, and more specifically show how the choice of prefix size affects total work performed, parallelism, and overall running time. With a careful choice of prefix

[2]The random order can be implemented by first randomly permuting the nodes, and then processing them in linear order.

size, the algorithms achieve good speedup and require only a modest number of cores to outperform optimized sequential implementations.

4.2 Analysis Tools

This chapter is concerned with the parallelism available in sequential iterative algorithms. Assume that an iterative algorithm takes n steps, where each ***step*** performs some computation, depending on the results or effects of a subset of previous steps. The goal is to run some of these steps in parallel. What can run safely in parallel will depend on both the algorithm and the input, which together will be referred to as a ***computation***. This chapter models the dependencies in the computation as a graph, where the steps $I = \{0, \ldots, n-1\}$ are vertices and dependencies between steps are directed edges, denoted by E.

Definition 3 (Iteration Dependence Graph)**.** *An* ***iteration dependence graph*** *for an iterative computation is a (directed acyclic) graph $G(I, E)$ such that if every step $i \in I$ runs after all predecessor steps in the graph complete, then every step will do the same computation as in the sequential order.*

The depth of an iteration dependence graph is referred to as the ***iteration depth***, $D(G)$. It should be clear that one can correctly simulate a computation with iteration dependence graph G in $D(G)$ rounds, each running a set of steps in parallel. However, it may not be clear how to efficiently determine for each step if all of its predecessors have completed. As we will see, and not surprisingly, the method for doing this check is algorithm-specific. We will say that a step can be ***efficiently checked*** if it can determine that all of its predecessors have completed in constant work/depth, and ***efficiently updated*** if the step itself takes constant work/depth.

The ***aggregate delay***, $A(G)$, of an iteration dependence graph G is defined to be the sum of the heights (one plus the longest directed path to a vertex) of the vertices in G. To understand why this is a useful measure, consider a process in which on every round all steps that have not yet completed check to see if their predecessors are complete, and if so they run and complete, otherwise they try again in the next round. Each round can be run in parallel, and each step is delayed by a number of rounds corresponding to its height in G. Assuming each non-completed step does constant work on each round, then the total work across all steps and all rounds will be bounded by $O(A(G))$.

4.3 Algorithmic Design Techniques

For MIS and maximal matching, this chapter will analyze the iteration depth of subsets of the elements to prove that the overall iteration depth of the algorithm is $O(\log^2 n)$ w.h.p. Linear-work algorithms for the two problems will also be presented. For random permutation, list contraction, and tree contraction, this chapter will show that the iteration depth of the *entire iteration dependence graph* is $O(\log n)$ depth w.h.p., and aggregate delay is $O(n)$ in expectation. These three problems have steps that can be checked and updated in constant time, although tree contraction requires a pre-processing step to allow for efficient checking.

For these all of these problems, one can easily obtain implementations from the iteration dependence graph. If steps in a computation can be efficiently checked and updated, then an algorithm for a problem with iteration depth $D(G)$ can be implemented with $O(nD(G))$ work and $O(D(G))$ depth simply by proceeding in rounds, where in each round all steps check if their predecessors in the iteration dependence graph have been processed, and proceed if so. As the goal is to obtain work-efficient (linear-work) algorithms, we prove the following lemma, which will be used to obtain linear-work algorithms for random permutation, list contraction, and tree contraction. The linear-work algorithms for MIS and maximal matching will require analysis specific to the problem and do not use this lemma.

Lemma 1. *If steps can be efficiently checked and updated, then an algorithm for a problem with iteration depth $D(G)$ can be implemented with $O(A(G))$ work and $O(D(G)\log n)$ depth without concurrent reads/writes or $O(D(G)\log^* n)$ depth with high probability with concurrent reads/writes.*

Proof. A step is defined to be ***ready*** if all of its predecessors in the iteration dependence graph have been processed. The algorithm proceeds in rounds, where in each round all remaining steps check if they are ready. If a step is ready, it proceeds in executing its computation. After processing the ready steps, consider them as having been removed from the iteration dependence graph, and hence the iteration depth of the remaining iteration dependence graph is 1 less than before. The initial iteration depth is $D(G)$, so $D(G)$ rounds suffice. In each round, the successful steps are packed out so that no additional work is done for them in later rounds. The pack requires linear work in the number of remaining steps. Since each round removes the leaves of the iteration dependence graph, and the steps can be efficiently checked and updated, the work done on each step is proportional to its height in the iteration dependence graph. The total work is proportional to the sum of the heights of all steps in the iteration dependence graph, which is the aggregate delay $A(G)$. The depth of the algorithm is $O(D(G)P(n))$, where $P(n)$ is the depth of the pack. A standard implementation of pack requires $O(\log n)$ depth. However, approximate compaction suffices for this purpose, and can be implemented work-efficiently in $O(\log^* n)$ depth w.h.p. using concurrent reads/writes [174]. This proves the lemma. $\square$

Algorithms developed using Lemma 1 can be mapped work-efficiently to the EREW PRAM with $O(D(G)\log n)$ depth (if they do not require concurrent reads/writes), to the

CRCW PRAM with $O(D(G)\log^* n)$ depth w.h.p., and to the scan PRAM with $O(D(G))$ depth (again, if they do not require concurrent reads/writes). The multiplicative factor in the depth only depends on how the pack is implemented, and processor allocation on each iteration can be done using the same packing algorithm.

Two techniques that are used to obtain algorithms for the problems are described below. The deterministic reservations method that checks all remaining steps in each round, executing the ones whose dependencies have all been satisfied, gives algorithms satisfying the bounds of Lemma 1. The activation-based approach directly activates a step when it is ready.

Deterministic Reservations. The deterministic reservations approach is discussed in Section 3.4.3. A fully parallel version of deterministic reservations which processes all remaining iterates in every round gives algorithms satisfying the bounds in Lemma 1, and this is the version used for analyzing linear-work implementations of random permutation, list contraction, and tree contraction. The linear-work MIS and maximal matching implementations require a careful choice of prefix size, and so Lemma 1 is not used.

Activation-based Approach. The activation-based approach directly "wakes-up" (activates) each step exactly when it is ready [55, 218, 427]. In particular, the predecessors in the iteration dependence graph are responsible for activating the step. At the beginning, the algorithm identifies all the steps that do not depend on any others (for the problems studied in this chapter, these can be determined easily). Then on each round, each active step executes its computation, and then detects whether it is the last predecessor of a successor; if so, it wakes up the successor. The approach is work-efficient since it only runs steps exactly when they are needed. As we will see, the implementations are problem-specific.

4.4 Maximal Independent Set

The sequential algorithm for computing the MIS of a graph is a simple greedy algorithm, shown in Algorithm 1 (refer to Section 2.4 for graph notation). In addition to a graph G, the algorithm takes an arbitrary total ordering on the vertices π. π is used to define priorities on the vertices. The algorithm adds the first remaining vertex v according to π to the MIS and then removes v and all of v's neighbors from the graph, repeating until the graph is empty. The MIS returned by this sequential algorithm is defined as the lexicographically first MIS for G according to π.

By allowing vertices to be added to the MIS as soon as they have no higher-priority neighbor, a parallel greedy algorithm is obtained (Algorithm 2). It is not difficult to see that this algorithm returns the same MIS as the sequential algorithm. A simple proof proceeds by induction on vertices in order. (A vertex v may only be resolved when all of its earlier neighbors have been classified. If its earlier neighbors match the sequential algorithm, then

Algorithm 1 Sequential greedy algorithm for MIS

```
1: procedure SequentialGreedyMIS($G = (V, E)$, $\pi$)
2:     if $|V| = 0$ then return $\emptyset$
3:     else
4:         let $v$ be the first vertex in $V$ by the ordering $\pi$
5:         $V' = V \setminus (v \cup N(v))$
6:         return $v \cup$ SequentialGreedyMIS($G[V']$, $\pi$)
```

Algorithm 2 Parallel greedy algorithm for MIS

```
1: procedure ParallelGreedyMIS($G = (V, E)$, $\pi$)
2:     if $|V| = 0$ then return $\emptyset$
3:     else
4:         let $W$ be the set of vertices in $V$ with no earlier neighbors (based on $\pi$)
5:         $V' = V \setminus (W \cup N(W))$
6:         return $W \cup$ ParallelGreedyMIS($G[V']$, $\pi$)
```

it does too.) Naturally, the parallel algorithm may (and should, if there is to be any parallel speedup) accept some vertices into the MIS at an earlier time than the sequential algorithm, but the final set produced is the same.

Note that if Algorithm 2 regenerates the ordering π randomly on each recursive call then the algorithm is effectively the same as Luby's Algorithm A [310]. It is the fact that a single permutation is used throughout that makes Algorithm 2 more difficult to analyze.

The iteration dependence graph. An iteration dependence graph for MIS can be constructed by taking the original graph and directing the edges from higher priority to lower priority endpoints based on π. Each iteration of Algorithm 2 can be viewed as adding all of the roots of the dependence graph to the MIS, and removing them and their children from the dependence graph. However, note that the iteration depth of the dependence graph is only an upper bound on the number of rounds the MIS algorithm takes to finish. Indeed for a complete graph, the longest directed path in the dependence graph is $\Omega(n)$, but the number of rounds is $O(1)$.

Therefore, instead of arguing that the number of rounds is polylogarithmic directly from the iteration depth of the entire graph, this section considers iteration dependence graphs induced by subsets of vertices and shows that these have small longest paths and hence small iteration depth. Aggregating across all subsets of vertices gives an upper bound on the total iteration depth.

Analysis via a modified parallel algorithm. Analyzing the depth of Algorithm 2 directly seems difficult as once some vertices are removed, the ordering among the set of remaining vertices may not be uniformly random. Rather than analyzing the algorithm directly, we preserve sufficient independence over priorities by adopting an analysis framework similar

Algorithm 3 Modified parallel greedy algorithm for MIS

```
1: procedure ModifiedParallelMIS($G = (V, E)$, $\pi$)
2:     if $|V| = 0$ then return $\emptyset$
3:     else
4:         choose prefix-size parameter $\delta$
5:         let $P = P(V, \pi, \delta)$ be the vertices in the prefix
6:         $W$ = ParallelGreedyMIS($G[P]$, $\pi$)
7:         $V' = V \setminus (P \cup N(W))$
8:         return $W \cup$ ModifiedParallelMIS($G[V']$, $\pi$)
```

to that of [111, 84]. Specifically, for the purpose of analysis, we consider a more restricted, less parallel algorithm given by Algorithm 3.

Algorithm 3 differs from Algorithm 2 in that it considers only a prefix of the remaining vertices rather than considering all vertices in parallel. This modification may cause some vertices to be processed later than they would in Algorithm 2, which can only *increase* the total number of iterations of the algorithm when the iterations are summed across all calls to Algorithm 2. We will show that Algorithm 3 has a polylogarithmic number of iterations, and hence Algorithm 2 does as well.

Each iteration (recursive call) of Algorithm 3 is referred to as a ***round***. For an ordered set V of vertices and fraction $0 < \delta \leq 1$, define the ***δ-prefix*** of V, denoted by $P(V, \pi, \delta)$, to be the subset of vertices corresponding to the $\delta\,|V|$ earliest in the ordering π. During each round, the algorithm selects the δ-prefix of remaining vertices for some value of δ to be discussed later. An MIS is then computed on the vertices in the prefix using Algorithm 2, ignoring the rest of the graph. When the call to Algorithm 2 finishes, all vertices in the prefix have been processed and either belong to the MIS or have a neighbor in the MIS. All neighbors of these newly discovered MIS vertices and their incident edges are removed from the graph to complete the round.

The advantage of analyzing Algorithm 3 instead of Algorithm 2 is that at the beginning of each round, the ordering among remaining vertices is still uniform, as the removal of a vertex outside of the prefix is independent of its position (priority) among vertices outside of the prefix. The goal of the analysis is then to argue that (a) the number of iterations in each parallel round is small, and (b) the number of rounds is small. The latter can be accomplished directly by selecting prefixes that are "large enough," and constructively using a small number of rounds. Larger prefixes increase the number of iterations within each round, however, so some care must be taken in tuning the prefix sizes.

The analysis assumes that the graph is arbitrary (i.e., adversarial), but that the ordering on vertices is random. In contrast, the previous analyses in this style [111, 84] assume that the underlying graph is random, a fact that is exploited to show that the number of iterations within each round is small. The analysis in this section, on the other hand, must cope with nonuniformity on the permutations of prefixes as the prefix is processed with Algorithm 2.

Reducing vertex degrees. A significant difficulty in analyzing the number of iterations of a single round of Algorithm 3 (i.e., the execution of Algorithm 2 on a prefix) is that the iterations of Algorithm 2 are not independent given a single random permutation that is not regenerated after each iteration. The dependence, however, arises partly due to vertices of drastically different degree, and can be bounded by considering only vertices of nearly the same degree during each round.

Let Δ be the *a priori* maximum degree in the graph. The algorithm will select prefix sizes so that after the i'th round, all remaining vertices have degree at most $\Delta/2^i$ with high probability. After $\log\Delta < \log n$ rounds, all vertices have degree 0, and thus can be removed in a single iteration. Bounding the number of iterations in each round by $O(\log n)$ then implies that Algorithm 3 has $O(\log^2 n)$ total iterations, and hence so does Algorithm 2.

The following lemma and corollary state that after processing the first $\Omega(n\log(n)/d)$ vertices, all remaining vertices have degree at most d.

Lemma 2. *Suppose that the ordering on vertices is uniformly random, and consider the (ℓ/d)-prefix for any positive ℓ and $d \leq n$. If a lexicographically first MIS of the prefix and all of its neighbors are removed from G, then all remaining vertices have degree at most d with probability at least $1 - n/e^{\ell}$.*

Proof. Consider the following sequential process, equivalent to the sequential Algorithm 1 (this proof refers to a recursive call of Algorithm 1 as a ***phase***). The process consists of $n\ell/d$ phases. Initially, all vertices are ***live***. Vertices become ***dead*** either when they are added to the MIS or when a neighbor is added to the MIS. During each phase, randomly select a vertex v, without replacement. The selected vertex may be live or dead. If v is live, it has no earlier neighbors in the MIS. Add v to the MIS, after which v and all of its neighbors become dead. If v is already dead, do nothing. Since vertices are selected in a random order, this process is equivalent to choosing a permutation first, and then processing the prefix.

Consider any vertex u not in the prefix. This proof will show that by the end of this sequential process, u is unlikely to have more than d live neighbors. (Specifically, during each phase that it has d neighbors, it is likely to become dead; thus, if it remains live, it is unlikely to have many neighbors.) Consider the i'th phase of the sequential process. If either u is dead or u has fewer than d live neighbors, then u alone cannot violate the property stated in the lemma. Suppose instead that u has at least d live neighbors. Then the probability that the i'th phase selects one of these neighbors is at least $d/(n-i) > d/n$. If the live neighbor is selected, that neighbor is added to the MIS and u becomes dead. The probability that u remains live during this phase is thus at most $1 - d/n$. Since each phase selects the next vertex uniformly at random, the probability that no phase selects any of the

d neighbors of u is at most $(1 - d/n)^{\delta n}$, where $\delta = \ell/d$. This failure probability is at most $((1 - d/n)^{n/d})^{\ell} < (1/e)^{\ell}$. Taking a union bound over all vertices completes the proof. $\square$

Corollary 1. *By setting $\delta = \Omega(2^i \log(n)/\Delta)$ for the i'th round of Algorithm 3, all remaining vertices after the i'th round have degree at most $\Delta/2^i$, with high probability.*

Proof. This follows from Lemma 2 with $\ell \geq c \ln n$ and $d = \Delta/2^i$ for any constant $c > 1$. The probability of success is at least $1 - 1/n^{c-1}$. $\square$

Bounding the number of iterations in each round. To bound the depth for each prefix in Algorithm 3, an upper bound on the iteration depth of the iteration dependence graph induced by the prefix is computed, as this path length provides an upper bound on the iteration depth.

The following lemma implies that as long as the prefix is not too large with respect to the maximum degree in the graph, then the longest path in the iteration dependence graph of the prefix has length $O(\log n)$.

Lemma 3. *Suppose that all vertices in a graph have degree at most d, and consider a randomly ordered δ-prefix. For any ℓ and r with $\ell \geq r \geq 1$, if $\delta < r/d$, then the longest path in the iteration dependence graph has length $O(\ell)$ with probability at least $1 - n(r/\ell)^{\ell}$.*

Proof. Consider an arbitrary set of k positions in the prefix—there are $\binom{\delta n}{k}$ of these, where n is the number of vertices in the graph.[3] Label these positions from lowest to highest $(x_1, \ldots, x_k)$. To have a directed path in these positions, there must be an edge between x_i and x_{i+1} for $1 \leq i < k$. Having the prefix be randomly ordered is equivalent to first selecting a random vertex for position x_1, then x_2, then x_3, and so on. The probability of an edge existing between x_1 and x_2 is at most $d/(n-1)$, as x_1 has at most d neighbors and there are $n-1$ other vertices remaining to sample from. The probability of an edge between x_2 and x_3 then becomes at most $d/(n-2)$. (In fact, the numerator should be $d-1$ as x_2 already has an edge to x_1, but rounding up here only weakens the bound.) In general, the probability of an edge existing between x_i and x_{i+1} is at most $d/(n-i)$, as x_i may have d other neighbors and $n-i$ vertices remain in the graph. The probability increases with each edge in the path since once $x_1, \ldots, x_i$ have been fixed, we may know, for example, that x_i has no edges to $x_1, \ldots, x_{i-2}$. Multiplying the k probabilities together gives the probability of a directed path from x_1 to x_k, which is rounded up to $(d/(n-k))^{k-1}$.

[3]The number of vertices n here refers to those that have not been processed yet. The bound holds whether or not this number accounts for the fact that some vertices may be "removed" from the graph out of order, as the n will cancel with another term that also has the same dependence.

Taking a union bound over all $\binom{\delta n}{k}$ sets of k positions (i.e., over all length-k paths through the prefix) gives a probability of at most

$$\begin{aligned}
\binom{\delta n}{k}\left(\frac{d}{n-k}\right)^{k-1} &\leq n\left(\frac{e\delta n}{k}\right)^{k}\left(\frac{d}{n-k}\right)^{k} \\
&= n\left(\frac{e\delta n d}{k(n-k)}\right)^{k} \\
&\leq n\left(\frac{2e\delta d}{k}\right)^{k}
\end{aligned}$$

where the last step holds for $k \leq n/2$. Setting $k = 4e\ell$ and $\delta < r/d$ gives a probability of at most $n(r/\ell)^{\ell}$ of having a path of length $4e\ell$ or longer. Note that if $4e\ell > n/2$, violating the assumption that $k \leq n/2$, then $n = O(\ell)$, and hence the claim holds trivially. $\square$

Corollary 2. *Suppose that all vertices in a graph have degree at most d, and consider a randomly ordered prefix. For an $O(\log(n)/d)$-prefix or smaller, the longest path in the iteration dependence graph has length $O(\log n)$ w.h.p. For a $(1/d)$-prefix or smaller, the longest path has length $O(\log n/\log\log n)$ w.h.p.*

Proof. For the first claim, applying Lemma 3 with $r = c\log n$ and $\ell = 4c\log n$ for a constant $c > 1/8$ gives a success probability of at least $1 - 1/n^{8c-1}$. For the second claim, using $r = 1$ and $\ell = c\ln n/\ln\ln n$ for a constant $c > 2$ gives a success probability of at least $1 - 1/n^{c-2}$ for large enough n. $\square$

The $\log n$ in this corollary should be treated as a constant across the execution of the algorithm, so that the bounds hold with high probability with respect to the original graph.

Parallel greedy MIS has low depth. The number $\log n$ of rounds is now combined with the $O(\log n)$ iterations per round to prove the following theorem on the number of iterations in Algorithm 2.

Theorem 2. *For a random ordering on vertices, where Δ is the maximum vertex degree, Algorithm 2 requires $O(\log\Delta\log n) = O(\log^2 n)$ iterations w.h.p.*

Proof. Let us first bound the number of rounds of Algorithm 3, choosing $\delta = c2^i\ln(n)/\Delta$ in the i'th round, for some constant c and constant $\ln n$ (i.e., n here means the original number of vertices). Corollary 1 states that with probability at least $1 - 1/n^{c-1}$, vertex degrees decrease in each round. Assuming this event occurs (i.e., vertex degree is $d < \Delta/2^i$), Corollary 2 states that with probability at least $1 - 1/n^{c-1}$, the number of iterations per round is at most $O(c\log n)$. Taking a union bound across any of these events failing says that every round decreases the degree sufficiently, and thus the number of rounds required is $\log \Delta$ with probability at least $1 - 1/n^{c-2}$. Multiplying the number of iterations in each round by the number of rounds gives the theorem bound with a success probability of at least $1 - 1/n^{c-3}$. Since Algorithm 3 only delays processing vertices as compared to Algorithm 2, it follows that this bound on iterations also applies to Algorithm 2. The constant in the big-O notation in the theorem statement is linear in c. $\square$

4.4.1 Linear-work MIS Algorithms

While Algorithm 2 has low depth, a naive implementation will require $O(m)$ work on each iteration to process all edges and vertices and therefore a total $O(m \log^2 n)$ work. This section describes two linear-work versions. The first follows the form of Algorithm 3, only processing prefixes of appropriate size. It has the advantage that it is particularly easy to implement, and is for the experiments. The second is an activation-based implementation of Algorithm 2 that directly traverses the iteration dependence graph of the entire graph only doing work on the roots and their neighbors on each iteration—and therefore every edge is only processed once. The algorithm therefore does linear work and has depth that is proportional to the number of iterations of the algorithm.

Prefix-based Implementation. The naive algorithm has high work because it processes every vertex and edge in every iteration. Intuitively, if small enough prefixes are processed (as in Algorithm 3) instead of the entire graph, there should be less wasted work. Indeed, a prefix of size 1 yields the sequential algorithm with $O(m)$ work but $\Omega(n)$ depth. There is some trade-off here—increasing the prefix size increases the work but also increases the parallelism. This section formalizes this intuition and describes a highly parallel algorithm that has linear work.

To bound the work, the number of edges operated on while considering a prefix is bounded. For any prefix $P \subseteq V$ with respect to permutation π, define the ***internal edges*** of P to be the edges in the sub-DAG induced by P, i.e., those edges that connect vertices in P. All other edges incident on P are referred to as ***external edges***. The internal edges may be processed multiple times, but external edges are processed only once.

The following lemma states that small prefixes have few internal edges. This lemma will be used to bound the work incurred by processing edges. The important feature to note is that for very small prefixes, i.e., $\delta < k/d$ with $k = o(1)$ and d denoting the maximum degree in the graph, the number of internal edges in the prefix is sub-linear in the size of the prefix, so the algorithm can afford to process those edges multiple times.

Lemma 4. *Suppose that all vertices in a graph have degree at most d, and consider a randomly ordered δ-prefix P. If $\delta < k/d$, then the expected number of internal edges in the prefix is at most $O(k\,|P|)$.*

Proof. Consider a vertex in P. Each of its neighbors joins the prefix with probability $< k/d$, so the expected number of neighbors is at most k. Summing over all vertices in P gives the bound. $\square$

The following related lemma states that for small prefixes, most vertices have no incoming edges and can be removed immediately. This lemma will be used to bound the work incurred by processing vertices, even those that may have already been added to the MIS or implicitly removed from the graph.

Lemma 5. *Suppose that all vertices in a graph have degree at most* d*, and consider a randomly ordered* δ*-prefix* P*. If* $\delta \leq k/d$*, then the expected number of vertices in* P *with at least 1 internal edge is at most* $O(k\,|P|)$*.*

Proof. Let X_E be the random variable denoting the number of internal edges in the prefix, and let X_V be the random variable denoting the number of vertices in the prefix with at least 1 internal edge. Since an edge touches (only) two vertices, this gives $X_V \leq 2X_E$. It follows that $E[X_V] \leq 2E[X_E]$, and hence $E[X_V] = O(k\,|P|)$ from Lemma 4. $\square$

The preceding lemmas indicate that small-enough prefixes are very sparse. Choosing $k = 1/\log n$, for example, the expected size of the subgraph induced by a prefix P is $O(|P|\,/\log n)$, and hence it can be processed $O(\log n)$ times without exceeding linear work. This fact suggests the following theorem. The implementation given in the theorem is relatively simple. The prefix sizes can be determined *a priori*, and the status of vertices can be updated lazily (i.e., when the vertex is processed). Moreover, each vertex and edge is only densely packed into a new array once, with other operations being done in place on the original vertex list.

Theorem 3. *Algorithm 3 can be implemented to run in expected* $O(n + m)$ *work and* $O(\log^4 n)$ *depth with high probability.*

Proof. This implementation updates a vertex's status (entering the MIS or removed due to a neighbor) only when that vertex is part of a prefix.

Let Δ be the *a priori* maximum vertex degree of the graph. As before, consider the rounds of Algorithm 3, with round i corresponding to an $O(\log(n)/d)$-prefix where $d = \Delta/2^i$. Corollary 1 states that each round reduces the maximum degree sufficiently, w.h.p. This prefix, however, may be too dense, so each round is divided into $\log^2 n$ *sub-rounds*, each operating on an $O(1/(d \log n))$-prefix P. To implement a sub-round, first process all external edges of P to remove those vertices with higher priority MIS neighbors. Then accept any remaining vertices with no internal edges into the MIS. These preceding steps are performed on the original vertex/edge lists, processing edges incident on the prefix a constant number of times. Let $P' \subseteq P$ be the set of prefix vertices that remain at this point. Use prefix sums to count the number of internal edges for each vertex (which can be determined by comparing priorities), and densely pack $G[P']$ into new arrays. This packing has $O(\log n)$ depth and linear work. Finally, process the induced subgraph $G[P']$ using a naive implementation of Algorithm 2, which has depth $O(D)$ and work equal to $O(|G[P']| \cdot D)$, where D is the iteration depth of P'. From Corollary 2, $D = O(\log n)$ with high probability. Combining this with an expected prefix size of $E[|G[P']|] = O(|P| / \log n)$ from Lemmas 4 and 5 yields expected $O(|P|)$ work for processing the prefix. Summing across all prefixes implies a total of $O(n)$ expected work for Algorithm 2 calls plus $O(m)$ work in the worst case for processing external edges. Multiplying the $O(\log n)$ prefix depth across all $O(\log^3 n)$ iterations ($O(\log n)$ iterations per of Algorithm 2 per sub-round) completes the proof for depth. Similar to the proof of Theorem 2, the success probability can be shown to be at least $1 - 1/n^{\alpha}$ for some large enough constant α, with the constant in the big-O notation linear in α. □

This result can be translated to a PRAM algorithm with the same bounds, as each round has $O(\log n)$ parallel slackness so processor allocation can be done with prefix sums.

Activation-based implementation. The idea of the linear-work implementation of Algorithm 2 is to explicitly keep on each iteration of the algorithm the set of roots of the remaining iteration dependence graph, e.g., as an array. With this set it is easy to identify the neighbors in parallel and remove them, but it is trickier to identify the new root set for the next iteration. One way to identify them would be to keep a count for each vertex of the number of neighbors with higher priorities (parents in the iteration dependence graph), decrement the counts whenever a parent is removed, and add a vertex to the root set when its count goes to zero. The decrement, however, needs to be done in parallel since many parents might be removed simultaneously. Such decrementing is hard to do work-efficiently when only some vertices are being decremented. Instead, note that the algorithm only needs to identify which vertices have at least one edge removed on the iteration and then check each of these to see if all their edges have been removed. Define a ***misCheck*** on a vertex as the operation of checking if it has any higher priority neighbors remaining. The implementation assumes that the neighbors of a vertex have been pre-partitioned into their parents (higher priorities) and children (lower priorities), and that edges are deleted lazily—i.e., deleting a vertex just marks it as deleted without removing it from the adjacency lists of its neighbors.

Lemma 6. *For a graph with m edges and n vertices where vertices are marked as deleted over time, any set of l misCheck operations can be done in $O(l + m)$ total work, and any set of misCheck operations in $O(\log n)$ depth.*

Proof. The pointers to parents are kept as an array (with a pointer to the start of the array). A vertex can be checked by examining the parents in order. If a parent is marked as deleted, the edge is removed by incrementing the pointer to the array start and the cost is charged to that edge. If it is not, the misCheck completes and the cost is charged to the check. Therefore the total charged across all operations is $l + m$, each of which does constant work. Processing the parents in order would require linear depth, so instead a doubling scheme is used: first examine one parent, then the next two, then the next four, etc. This completes once a parent that is not deleted is found and all work is charged to the previous ones that were deleted. The work can be at most twice the number of deleted edges thus guaranteeing linear work. The doubling scheme requires $O(\log n)$ steps each step requires $O(1)$ depth, hence the overall depth is $O(\log n)$. □

Lemma 7. *Algorithm 2 can be implemented in $O(m)$ total work and $O(\log^3 n)$ depth with high probability.*

Proof. The implementation works by keeping the roots in an array, and on each iteration marking the roots and its neighbors as deleted, and then using misCheck on the neighbors' neighbors to determine which ones belong in the root array for the next iteration. The total number of checks is at most m, so the total work spent on checks is $O(m)$. After the misCheck's all vertices with no previous vertex remaining are added to the root set for the next iteration. Some care needs to be taken to avoid duplicates in the root array since multiple neighbors might check the same vertex. Duplicates can be avoided, however, by having the neighbor write its identifier into the checked vertex using an arbitrary concurrent write, and whichever write succeeds is responsible for adding the vertex to the new root array. Each iteration can be implemented in $O(\log n)$ depth, required for the checks and for packing the successful checks into a new root set. Multiplying by the $O(\log^2 n)$ iterations gives an overall depth of $O(\log^3 n)$ w.h.p. Every vertex and its edges are visited once when removing them, and the total work on checks is $O(m)$, so the overall work is $O(m)$. □

Again, this result can be translated to a CRCW PRAM algorithm with the same work and depth bounds.

4.5 Maximal Matching

One way to implement maximal matching (MM) is to reduce it to MIS by replacing each edge with a vertex, and creating an edge between all adjacent edges in the original graph. An iteration dependence graph for MM is defined using this reduction. This reduction, however, can significantly increase the number of edges in the graph and therefore may not take work that is linear in the size of the original graph. Instead a standard greedy sequential algorithm is used to process the edges in an arbitrary order and include the edge in the MM if and only if no neighboring edge on either endpoint has already been added. As with the vertices in the greedy MIS algorithms, edges can be processed out of order

Algorithm 4 Parallel greedy algorithm for MM

```
1: procedure ParallelGreedyMM($G = (V, E)$, $\pi$)
2:     if $|E| = 0$ then return $\emptyset$
3:     else
4:         let $W$ be the set of edges in $E$ with no adjacent edges with higher priority by $\pi$
5:         $E' = E \setminus (W \cup N(W))$
6:         return $W \cup$ ParallelGreedyMM($G[E']$, $\pi$)
```

Algorithm 5 Modified parallel greedy algorithm for MM

```
1: procedure ModifiedParallelMM($G = (V, E)$, $\pi$)
2:     if $|V| = 0$ then return $\emptyset$
3:     else
4:         choose prefix-size parameter $\delta$
5:         let $P = P(E, \pi, \delta)$ be the edges in the prefix
6:         $W$ = ParallelGreedyMM($G[P]$, $\pi$)
7:         $E' = E \setminus (P \cup N(W))$
8:         return $W \cup$ ModifiedParallelMM($G[E']$, $\pi$)
```

when they do not have any earlier neighboring edges. This idea leads to Algorithm 4 where π is now an ordering of the edges.

Lemma 8. *For a random ordering on edges, the number of iterations of Algorithm 4 is $O(\log^2 m)$ with high probability.*

Proof. This follows directly from the reduction to MIS described above. In particular an edge is added or deleted in Algorithm 4 exactly on the same iteration it would be for the corresponding MIS graph in Algorithm 2. Therefore Lemma 2 applies. □

As done for MIS in the previous section, this section describes two linear-work algorithms for maximal matching, the first of which processes prefixes of the vertices in priority order and the second of which maintains the set of roots in the iteration dependence graph. The first algorithm is easier to implement and is the version used in the experiments.

Prefix-based implementation. Algorithm 5 is the prefix-based algorithm for maximal matching (the analogue of Algorithm 3). To obtain a linear-work maximal matching algorithm, Algorithm 5 is used with a prefix-size parameter $\delta = 1/d_e$, where d_e is the maximum number of neighboring edges any edge in G has. Each call to Algorithm 4 in Line 6 of Algorithm 5 proceeds in iterations. The algorithm assumes that the edges are pre-sorted by priority (for random priorities they can be sorted in linear work and within the depth bounds with bucket sorting [112]).

In each iteration, first every edge in the prefix does a priority write to its two endpoints (attempting to record its rank in the permutation), and after all writes are performed, every edge checks whether it won on (its value was written to) both endpoints. Since edges are sorted by priority, the highest priority edge incident on each vertex wins. If an edge wins on both sides, then it adds itself to the maximal matching and deletes all of its neighboring edges (by packing). Each edge does constant work per iteration for writing and checking. The packing takes work proportional to the remaining size of the prefix. It remains to show that the expected number of times an edge in the prefix is processed is constant.

Consider the iteration dependence graph on the δ-prefix of E, where a vertex in the iteration dependence graph corresponds to an edge in the original graph G, and a directed edge exists in the iteration dependence graph from E_i to E_j if and only if E_i is adjacent to E_j in G and E_i has a higher priority than E_j. Note that this iteration dependence graph is not explicitly constructed. Define the ***height*** of a vertex v_e in the iteration dependence graph to be the length of the longest incoming path to v_e. The height of v_e is an upper bound on the number of iterations of processing the iteration dependence graph required until v_e is either added to the MM or deleted.

Theorem 4. *For a $(1/d_e)$-prefix, the expected height of any vertex in the iteration dependence graph (corresponding to an edge in the original graph G) is $O(1)$.*

Proof. For a given vertex v_e, the expected length of a directed path ending at v_e is computed. For there to be a length k path to v_e, there must be k positions $p_1, \ldots, p_k$ (listed in priority order) before v_e's position, p_e, in the prefix such that there exists a directed edge from p_k to p_e and for all $1 < i < k$, a directed edge from p_i to p_{i+1}. Using an argument similar to the one used in the proof of Lemma 3, the probability of this particular path existing is at most $(d_e/(m-k))^k$. The number of positions appearing before p_e in the prefix is at most the size of the prefix itself. So summing over all possible choices of k positions implies that the probability of a directed path from the root to some vertex being length k is

$$\begin{aligned}\binom{\delta m}{k}\left(\frac{d_e}{m-k}\right)^k &\le \left(\frac{me}{kd_e}\right)^k\left(\frac{d_e}{m-k}\right)^k \\ &\le \left(\frac{me}{k(m-k)}\right)^k\end{aligned}$$

Now the expected length of a path from the root vertex is computed by summing over all possible lengths. This expectation is upper bounded by

$$\begin{aligned}\sum_{k-1}^{\delta m} k\left(\frac{me}{k(m-k)}\right)^k &\le \left[\sum_{k=0}^{m/2} k\left(\frac{me}{k(m-m/2)}\right)^k\right] + mPr(k > m/2) \\ &\le \left[\sum_{k=0}^{\infty} k\left(\frac{2e}{k}\right)^k\right] + o(1)\end{aligned}$$

$$= O(1)$$

To obtain the last inequality Lemma 3 is applied, giving $Pr(k > m/2) = O(1/m^c)$ for $c > 1$. The desired bound is obtained by using the formula $\sum_{k=0}^{\infty} k(x^k)/k! = xe^x$. □

Lemma 9. *Given a graph with m edges, n vertices, and a random permutation on the edges π, Algorithm 5 can be implemented in $O(m)$ total work in expectation and $O(\log^4 m/\log\log m)$ depth with high probability.*

Proof. Consider the rounds (recursive calls) of Algorithm 5. Each round operates on an $O(1/d_e)$-prefix, so after $O(\log m)$ rounds an $O(\log(m)/d_e)$-prefix is processed, and d_e decreases by a constant factor w.h.p. by Lemma 2. Therefore, a total of $O(\log^2 m)$ rounds are required until completion.

In each round, each iteration of Algorithm 4 processes the top level (root vertices) of the iteration dependence graph. Once an edge gets processed as a root of the iteration dependence graph or gets deleted by another edge, it will not be processed again in the algorithm. Since the expected height of an edge in the iteration dependence graph is $O(1)$, it will be processed a constant number of times in expectation (each time doing a constant amount of work), and contributes a constant amount of work to the packing cost. Hence the total work is linear in expectation.

For a given round, the packing per iteration requires $O(\log |P|)$ depth where $|P|$ is the remaining size of the prefix. By Corollary 2, there are at most $O(\log m/\log\log m)$ iterations w.h.p. Therefore, each round requires $O(\log^2 m/\log\log m)$ depth and the algorithm has an overall depth of $O(\log^4 m/\log\log m)$ w.h.p. As in the proof of Theorem 2, the success probability can be shown to be at least $1 - 1/n^\alpha$ for some large enough constant α, with the constant in the big-O notation linear in α. □

The algorithm can be implemented on a PRAM with the same complexity.

Activation-based Implementation. As with the algorithm used in Lemma 7, on each round an array of roots (edges that have no neighboring edges with higher priority) can be maintained and used to both delete edges and generate the root set for the next round. However, the algorithm cannot afford to look at all the neighbors' neighbors. Instead for each vertex an array of its incident edges sorted by priority is maintained. This list is maintained lazily such that deleting an edge only marks it as deleted and does not immediately remove it from its two incident vertices. Refer to an edge as ***ready*** if it has no remaining neighboring edges with higher priority. The algorithm uses an ***mmCheck*** procedure on a vertex to determine if any incident edge is ready and identifies the edge if so—a vertex can have at most one ready incident edge. The mmChecks do not happen in parallel with edge deletions.

Lemma 10. *For a graph with m edges and n vertices where edges are marked as deleted over time, any set of l mmCheck operations can be done in $O(l + m)$ total work, and any set of mmCheck operations in $O(\log m)$ depth.*

Proof. The mmCheck is partitioned into two phases. The first phase identifies the highest priority incident edge that remains, and the second phase checks if that edge is also the highest priority on its other endpoint and returns it if so. The first phase can be done by scanning the edges in priority order, removing those that have been deleted and stopping when the first non-deleted edge is found. As in Lemma 6 this can be done in parallel using doubling in $O(\log m)$ depth, and the work can be charged either to a deleted edge, which is removed, or the check itself. The total work is therefore $O(l + m)$. The second phase can similarly use doubling to see if the highest priority edge is also the highest priority on the other side. □

Lemma 11. *For a random ordering on the edges, Algorithm 4 can be implemented in $O(m)$ total work and $O(\log^3 m)$ depth with high probability.*

Proof. Since the edge priorities are selected at random, the initial sort to order the edges incident on each vertex can be done in $O(m)$ work and within the depth bounds w.h.p. using bucket sorting [112]. Initially the set of ready edges are selected by using an mmCheck on all edges. On each iteration of Algorithm 4, the set of ready edges and their neighbors are deleted (by marking them), and then all vertices incident on the far end of each of the deleted neighboring edges are checked. This returns the new set of ready edges in $O(\log m)$ depth. Redundant edges can easily be removed. Thus the depth per iteration is $O(\log m)$ and by Lemma 8 the total depth is $O(\log^3 m)$. Every edge is deleted once and the total number of checks is $O(m)$, so the total work is $O(m)$. □

This algorithm can be implemented on a CRCW PRAM with the same work and depth bounds.

4.6 Random Permutation

Durstenfeld [139] and Knuth [270] discuss a simple sequential algorithm for generating a random permutation which goes through the elements of an array from the end to the beginning (or the other way), and for each element swaps with a random position in the array earlier than or at the current position. This chapter assumes that the random integers used in the algorithm are generated beforehand, and stored in an array H—i.e., for $0 \leq i < n$, $H[i]$ is a (uniformly) random integer from 0 to i, inclusive. The pseudocode for Durstenfeld's sequential algorithm is given in Figure 4.1.

```
1: procedure SequentialRandPerm($A$, $H$)
2:     for $i = n - 1$ to $0$ do
3:         swap($A[H[i]]$, $A[i]$)
```

Figure 4.1: Sequential algorithm for random permutation.

4.6.1 Iteration Dependence Depth and Aggregate Delay

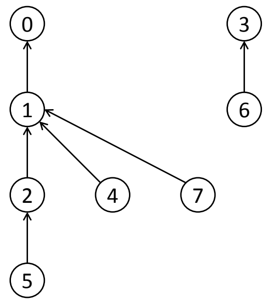
To analyze the iteration dependence depth of Durstenfeld's algorithm, the following definitions will be used. When performing a swap(x, y), x is the ***source*** of the swap and y is the ***target*** of the swap. For a given H, define i to ***dominate*** j if $H[i] = j$ and $i \neq j$. Define the ***dominance forest*** of H to be the directed graph formed on n nodes where node i points to node j if i dominates j. Since each node can dominate at most one other node, the graph is a forest. Note that the roots of the dominance forest are exactly the nodes where $H[i] = i$.

Define the ***dependence forest*** of H to be a modification of the dominance forest where the children of each node (from incoming edges) are chained together in decreasing order. In particular, for a node i with incoming edges from nodes $j_1 < \ldots < j_k$, add an edge from j_{l+1} to j_l for $1 \leq l < k$ (creating a chain) and delete the edges from j_l to i for $l > 1$. Note that the dependence forest is binary, since each node can have at most one incoming edge from the set of nodes pointing to it in the dominance forest, and since it can be part of at most one chain. See Figures 4.2(a) and 4.2(b) for an example of the dominance forest and dependence forest for a given H.

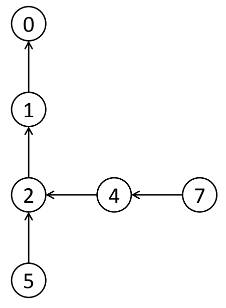
Lemma 12. *The dependence forest of H is an iteration dependence graph for* SequentialRandPerm.

Proof. Define a step to be ***ready*** if all of its descendants in the dependence forest have been processed. The proof will show that when a step is ready, its corresponding location in A will contain the same value as it would have when the sequential algorithm processes it. The proof uses induction on the iteration in which a step is processed in the sequential algorithm (i.e., step $n - 1$ is the first and step 0 is the last).

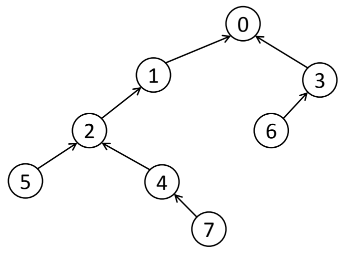
The base case is trivial as step $n - 1$ is ready at the start of any ordering (no node can point to $n - 1$ in the dependence forest) and has the correct value (location $n - 1$ cannot be the target of any swap with another element). Consider some step i. Suppose there are multiple steps $j_1, \ldots, j_k$, where $j_1 < j_2 < \ldots < j_k$, with location i as the target of a swap operation. Since $i < j_1 < \ldots < j_k$, by the inductive hypothesis we may assume that steps $j_1, \ldots, j_k$ had the correct value in their corresponding locations in A when they were ready. The sequential algorithm will perform the swaps in decreasing order of the steps (j_k down to j_1), and since $i < j_1$, in the sequential algorithm location i will not be the source of a swap until all of steps $j_1, \ldots, j_k$ have been processed. Any ordering respecting the dependence forest will also process steps $j_1, \ldots, j_k$ in decreasing order, since



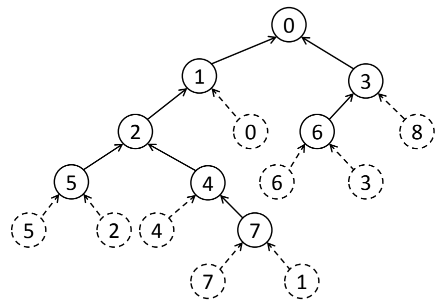
(a) Dominance forest



(b) Dependence forest



(c) Linked dependence tree



(d) Possible locations for $H[8]$

Figure 4.2: Dominance and dependence forests for $H = [0, 0, 1, 3, 1, 2, 3, 1]$ are shown in (a) and (b), respectively. (c) shows the linked dependence tree for H and (d) shows the possible locations for inserting the 9'th node; dashed circles correspond to the value of $H[8]$.

by definition the dependence forest contains a directed path from j_k to j_1. The fact that $j_1, \ldots, j_k$ have the same value as in the sequential algorithm when they are ready, and that they are processed in the same order as the sequential algorithm implies that the location corresponding to step i will also have the same value as in the sequential algorithm when it is ready (i.e., after all of its incoming steps have been processed). $\square$

The goal is to show that the dependence forest is shallow. To do this, we will actually add some additional edges to make a tree and then show that this tree has an identical distribution as random binary search trees, which are known to have $\Theta(\log n)$ depth with high probability. The standard definition of a ***random binary search tree*** will be used, i.e., the tree generated by inserting a random permutation of the integers $\{0, \ldots, n-1\}$ into a binary search tree. Define the ***linked dependence tree*** as the tree created by linking the roots of the dependence forest along the right spine of a tree with indices appearing in ascending order from the top of the spine to the bottom (see Figure 4.2(c) for an example of the linked dependence tree). The linked dependence tree is clearly also an iteration dependence graph since it only adds constraints.

Theorem 5. *Given a random H, the distribution of (unlabeled) linked dependence trees for H is identical to the distribution of (unlabeled) random binary search trees.*

Proof. This is proved by induction on the input size n. For the base case, $n = 1$, there is a single vertex and the claim is trivially true. For the inductive case, note that the linked dependence tree for the first $n-1$ locations is not affected by the last location since numbers at $H[i]$ point at or before i—i.e., the last location will end up as a leaf. By the inductive hypothesis, the distribution of trees on the first $n-1$ locations has the same distribution as random binary search trees of size $n-1$. Now we claim that, justified below, the n'th element can go into any leaf position. Since the n'th location is a uniformly random integer from 0 to $n-1$ and there are n possible leaf positions in a binary tree of size $n-1$, all leafs must be equally likely. Hence this is the same process as inserting randomly into a binary search tree.

To see that the n'th location can go into any leaf, first note that if it picks itself (index $n-1$), then it is at the bottom of the right spine of the tree, by definition. Otherwise if it picks $j < n-1$, and it will be placed at the bottom of the right spine of the left child of j. This allows for all possible tree positions—to be a left child of a node just pick the parent, and to be a right child follow the right spine up to the top, then pick its parent (e.g., see Figure 4.2(d)). □

Theorem 6. *For* SEQUENTIALRANDPERM *on a random H of length n, there is an iteration dependence graph G with $D(G) = \Theta(\log n)$ with high probability, and $A(G) = \Theta(n)$ in expectation.*

Proof. For the depth, it is a well-known fact that the height of a random binary search tree on n nodes is $\Theta(\log n)$ w.h.p. [136]. For example, to be exact, the height is bounded by $4e \log n$ with probability at least $1 - 1/n^{4e+1}$ (see Lemma 3.1 in [136]). Therefore, Theorem 5 implies that the longest path in the iteration dependence graph is $O(\log n)$ w.h.p. To show that this is tight, note that node 0 has $\Theta(\log n)$ incoming edges in the dominance forest w.h.p. This can be shown by applying Chernoff bounds [341] on the sum of indicator variables X_k (indicating whether $H[k] = 0$) from $k = 0, \ldots, n-1$, where $X_k = 1$ with probability $1/(k+1)$. With probability at least $1 - 1/n^{\delta^2/2}$, the sum is at least $(1-\delta)H_n$ where $H_n \approx \ln n$ is the n'th harmonic number and $0 < \delta < 1$. Hence the longest path to it in the iteration dependence graph is $\Omega(\log n)$ w.h.p.

```
H = swap targets
R = {−1, . . . , −1}
procedure RESERVE(i)
    writeMax(R[i], i)                ▷ reserve own location
    writeMax(R[H[i]], i)             ▷ reserve target location
    return 1
procedure COMMIT(i)
    if (R[i] = i and R[H[i]] = i) then
        swap(A[H[i]], A[i])          ▷ swap if reserved
        return 0
    else return 1
```

Figure 4.3: RESERVE and COMMIT functions and associated data for random permutation using deterministic reservations.

To analyze the aggregate delay, let us analyze the sum of heights of the nodes in a random binary search tree. Let $W(n)$ indicate the expected sum. The two children of the root of a random binary search tree are also random binary search trees of size i and $n-i-1$, respectively, for a randomly chosen i in $\{0, \dots, n-1\}$. This gives the recurrence $W(n) = \textit{Height}(n) + (1/n)\sum_{i=0}^{n-1}(W(i) + W(n-i-1))$, where $\textit{Height}(n) = \Theta(\log n)$ is the expected height of a random binary search tree with n nodes. This solves to $\Theta(n)$ and hence the theorem follows. □

4.6.2 Algorithms

This section describes parallel implementations of random permutation that return the same result as Durstenfeld's sequential algorithm.

Deterministic reservations-based implementation. To implement the random permutation algorithm using deterministic reservations, the RESERVE and COMMIT functions shown in Figure 4.3 are used. The implementation uses an array R, initialized to contain all -1, to store reservations. The implementation uses the function ***writeMax(l,i)***, a special case of the priority update described in Chapter 6 which writes value i to location l such that the maximum value written to l will end up in that location. The RESERVE function for index i simply calls writeMax to the two locations $R[i]$ and $R[H[i]]$ with value i and returns 1. The COMMIT function simply checks if both writeMax's were successful (i.e., both $R[i]$ and $R[H[i]]$ store the value i) and if so, swaps $A[H[i]]$ and $A[i]$ and returns 0; otherwise it returns 1. This process guarantees that a step will successfully commit (swap) if and only if its children in the dependence forest have finished in a previous round of deterministic reservations. This is because if any child were not finished, then it would have competed in the writeMax and won since it has a higher index. In particular, the left child as shown in Figure 4.2(b) will win on $R[i]$ and the right child in that figure will win on $R[H[i]]$.

Theorem 7. *For a random H, deterministic reservations using the* RESERVE *and* COMMIT *functions for random permutation runs in $O(n)$ expected work and $O(\log n \log^* n)$ depth with high probability using concurrent reads/writes.*

Proof. Apply Theorem 6 and Lemma 1. The RESERVE and COMMIT functions take constant work/depth, so the steps of the computation can be efficiently checked and updated. The writeMax requires concurrent reads/writes. □

This implementation can be mapped to the priority CRCW PRAM, as processor allocation on each round of deterministic reservations can be done in $O(\log^* n)$ depth w.h.p.

Activation-Based Implementation. A linear-work activation-based implementation of the parallel random permutation algorithm is now presented. The implementation keeps track of the nodes ready to be executed of the dependence graph, processes and deletes these nodes from the graph in each round, and identifies the new nodes that are ready for the next round. It relies on explicitly constructing the dependence forest, and the following lemma states that this can be done efficiently.

Lemma 13. *The dependence forest for a given H can be constructed in $O(n)$ expected work and $O(\log n)$ depth with high probability.*

Proof. Building the dependence forest of random permutation for a given H requires sorting all of the nodes which point to the same node in the forest. This can be done by (1) using a non-stable integer sort in the range $[0, \ldots, n-1]$ [388] to group all the nodes, and then (2) sorting the nodes within each group using a parallel comparison sort [243]. (1) can be done in $O(n)$ work and $O(\log n)$ depth (using concurrent reads/writes). The depth for (2) is $O(\log \log n)$ w.h.p. since the largest group is of size $O(\log n)$ w.h.p. The total work for (2) is $\sum_{i=0}^{n-1} cs_i \log s_i$ where s_i is the number of nodes pointing to node i and c_1 is a constant. To show that $\sum_{i=0}^{n-1} c_1 s_i \log s_i = O(n)$, a similar argument used in the analysis of perfect hash tables can be used [341]. Let $X_{ij} = 1$ if $H[i] = H[j]$ and $X_{ij} = 0$ otherwise.

$$\begin{aligned}
\sum_{i=0}^{n-1} c_1 s_i \log s_i &\leq \sum_{i=0}^{n-1} c_2 s_i^2 && \text{for some constant } c_2 \\
&= c_2 \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} X_{ij} \\
&= c_2 \left(n + 2 \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij} \right) && \text{consider } X_{ij} \text{ where } i < j \\
&\leq c_2 \left(n + 2 \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \frac{1}{i+1} \frac{1}{j+1} \right) && (*)
\end{aligned}$$

$$\begin{aligned}
&\leq c_2 \left(n + 2 \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \frac{1}{(i+1)^2} \right) \\
&\leq c_2 \left(n + 2n \sum_{i=1}^{n} \frac{1}{i^2} \right) \\
&< c_2 \left(n + 2n \cdot \frac{\pi^2}{6} \right) \\
&= O(n)
\end{aligned}$$

The line marked (*) follows because $H[i]$ and $H[j]$ are independent.

After sorting, creating the pointers in the dependence forest takes $O(n)$ work and $O(1)$ depth. □

The algorithm in Lemma 13 works on the CRCW PRAM as the integer sort requires concurrent reads and writes. The following theorem uses Lemma 13 to design an activation-based random permutation algorithm.

Theorem 8. *For a random H, an activation-based implementation of random permutation runs in $O(n)$ expected work and $O(\log n \log^* n)$ depth with high probability.*

Proof. The algorithm forms the dependence forest for a given H, which by Lemma 13 can be done in $O(n)$ expected work and $O(\log n)$ depth w.h.p.

The leaves of the dependence forest are first identified, and at each step the set of leaves is maintained (these are the steps that are ready to be processed). Then the algorithm repeatedly processes the leaf set, removes it and its edges from the graph, and identifies the new leaf set until the dependence forest has been completely processed. Since all dependencies in the dependence forest are satisfied, by Lemma 12, this guarantees correctness. The algorithm assumes that the neighbors of a node are represented in an array, and partitioned into incoming edges and outgoing edges. To identify the new leaf set at each step, nodes that are removed perform a check on its parent to see if it has any incoming edges remaining. The check can be done in $O(1)$ work and time per neighbor since each node has at most two incoming edges.

After all checks are completed, nodes with no incoming edges are added to the next leaf set. Duplicates can be eliminated by filtering in work linear in the size of the new leaf set since each node can be duplicated at most once (each node has at most 2 incoming edges). The new leaf set is packed with approximate compaction, requiring work linear in the leaf set size and $O(\log^* n)$ depth w.h.p. Each step is processed a constant number of times, so the total work is $O(n)$. Each round reduces the iteration depth of the iteration dependence graph on the remaining steps by 1, and since the initial iteration depth is $\Theta(\log n)$ w.h.p. by Theorem 6, the overall depth is $O(\log n \log^* n)$ w.h.p. □

The activation-based algorithm runs on the CRCW PRAM as processor allocation can be done with approximate compaction.

Adapting to the CRQW PRAM. The random permutation algorithms can be adapted to the concurrent-read queue-write (CRQW) PRAM [169, 171], which closely models cache coherence protocols in multicore machines. In this model, concurrent reads to a memory location are charged unit cost but concurrent writes to a memory location have a contention cost equal to the total number of concurrent writes to the location. In each step, the maximum contention over all locations is charged to the depth.

Lemma 13 also applies for the CRQW PRAM as integer sorting can be done in $O(n)$ work and $O(\log n)$ depth w.h.p. on the CRQW PRAM [169], and comparison sorting can be implemented on an EREW PRAM (a weaker model than the CRQW PRAM). Packing on the CRQW PRAM can be done in linear work and $O(\sqrt{\log n})$ depth w.h.p. [171], so an activation-based implementation of the sequential algorithm can be made to run in $O(n)$ expected work and $O(\log^{3/2} n)$ depth w.h.p.

The deterministic reservation-based implementation of random permutation can also be adapted to the CRQW PRAM, using prefix sums for packing. The only place in the algorithm that requires concurrent writes is the call to writeMax. However since the dominance forest has in-degree $O(\log n)$ w.h.p., there can be at most $O(\log n)$ concurrent calls to writeMax to a given location, leading to $O(\log n)$ contention. This requires $O(\log n)$ additional slackness (depth) per step. Using prefix sums for packing, each round already requires $O(\log n)$ depth, so this slackness does not affect the overall bounds. Therefore, the algorithm runs in linear work and $O(\log^2 n)$ depth w.h.p. on the CRQW PRAM.

Random Permutation via Rotations. The following describes another parallel implementation of the sequential algorithm, using the fact that the values at the locations of the nodes pointing to the same node in the dominance forest just get rotated. In particular, if $i_1, \ldots, i_k$ with $i_l < i_{l+1}$ point to j, then after all other dependencies to $i_1, \ldots, i_k$ are resolved, $A[j] = A[i_k], A[i_1] = A[j]$, and $A[i_{l+1}] = A[i_l]$ for $1 \leq l < k$. This algorithm builds the dominance forest using an integer sort to group the nodes and then a comparison sort within each group in $O(n)$ work and $O(\log n)$ depth w.h.p. by the same analysis as done in the proof of Lemma 13. Then it processes the forest level by level, starting with the leaves, and rotating the values of each group of leaves and the target node. The level numbers for the nodes can be computed using leaffix operations or Euler tours [243] in linear work and $O(\log n)$ depth. Rotating the values can be done in work proportional to the number of nodes processed, and $O(1)$ depth. As the height of the dominance forest is $\Theta(\log n)$ w.h.p., this gives an algorithm with $O(n)$ work and $O(\log n)$ depth w.h.p. The algorithm can be mapped to the CRCW PRAM or CRQW PRAM in the same bounds.

```
procedure SequentialListContract($L$)
    for $i = 0$ to $n - 1$ do
        if $L[i]$.prev $\neq$ null then
            $L[L[i]$.prev$]$.next $= L[i]$.next
        if $L[i]$.next $\neq$ null then
            $L[L[i]$.next$]$.prev $= L[i]$.prev
```

Figure 4.4: Sequential algorithm for list contraction.

4.7 List Contraction

List contraction, and the related list ranking, is one of the most canonical problems in the study of parallel algorithms. The problem has received considerable attention both because of its fundamental nature as a pointer-based algorithm, and also because it has many applications as a subroutine in other algorithms. A summary of the work can be found in a variety of books and surveys (see, e.g., [259, 243, 395]).

This section is concerned with analyzing a simple sequential algorithm for list contraction and showing that it has low iteration depth and aggregate delay. Assume the linked list is represented as an array L of nodes, where $L[i]$.prev stores the index of the predecessor of node i (null if none) and $L[i]$.next stores the index of the successor of node i (null if none). A natural sequential iterative algorithm works by splicing out the nodes in order of increasing index, as shown in Figure 4.4. Each list in L is contracted down to a single node. For simplicity the values stored on the nodes are not shown. If values are stored, then when a node is spliced out its value is combined with its predecessor's value using a combining function, and stored on its predecessor. To perform list ranking, the process is then reversed, adding the nodes back in with the appropriate values. Note that when the combining function is non-associative, then the result depends on the order in which the nodes are spliced out. In such a case, a parallel computation returns the same answer as the sequential algorithm if it satisfies the dependence structure of the sequential algorithm, which is defined next.

4.7.1 Iteration Dependence Depth and Aggregate Delay

The ***dependence forest*** for an input L is defined as follows. For a list, place the last position k in which any of its links appear at the root r of a tree. Now recursively for the sublists on each side of the node in position k, do the same and make the two roots the children of r. If either sublist is empty, r will not have a child on that side. This defines a tree for each list and a forest across multiple lists. As with the dependence forest for random permutation, the dependencies go up the tree—i.e., each parent depends on its children. An example list along with its dependence forest is shown in Figure 4.5.

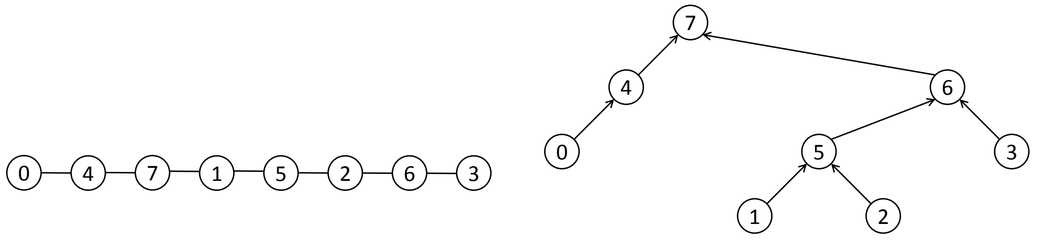


Figure 4.5: (a) An example list, where the numbers represent the position in the input array L, and (b) its dependence forest.

Lemma 14. *The dependence forest of* L *is an iteration dependence graph for* SEQUENTIALLISTCONTRACT(L).

Proof. For each step i, let j and k be the indices of prev and next nodes when i is spliced out in the sequential order. Clearly j and k must both be larger than i (or null) since they have not yet been spliced out. It suffices to show that for each i, once all of its descendants in the dependence forest are completed (spliced out), possibly not in the sequential order, it will point to j and k, and hence will do an identical splice as in the sequential order. By induction, this is assumed to be true for all indices less than i.

Consider the sublist between j and k (not inclusive). The index i must be the largest index on this list because if there were a larger index l, when i is contracted in the sequential order it cannot be linked with both j and k—l must be in the way. By construction of the dependence forest, and because i is the largest on the sublist, it is picked as the root of a tree containing the sublist. Therefore, when all descendants are completed (and by induction, they operated correctly) all other nodes on the sublist have been spliced out and i will point to j and k.

□

Lemma 15. *Assuming that the ordering of* L *has been randomized, for each list in* L *the distribution of (unlabeled) dependence trees is identical to the distribution of (unlabeled) random binary search trees of the same size.*

Proof. The root node of the dependence tree can appear in any position of the list with equal probability, since L is randomly ordered. This property also holds for each sublist of the list. Therefore in each subtree all nodes are equally likely to be the root, which is equivalent to the distribution for random binary search trees. □

The following theorem now follows from the same argument as in Theorem 6 since the iteration dependence graph (for each list) has the same distribution—a random binary search tree. There are no dependencies among different lists.

```
1:  $R = \{0, \ldots, 0\}$                                   ▷ boolean array
2:  procedure RESERVE($i$)
3:      if $i < L[i]$.prev and $i < L[i]$.next then
4:          $R[i] = 1$                                       ▷ reserve own location
5:      return 1
6:  procedure COMMIT($i$)
7:      if $(R[i] = 1)$ then
8:          if $L[i]$.prev $\neq$ null then
9:              $L[L[i]$.prev$]$.next $= L[i]$.next
10:         if $L[i]$.next $\neq$ null then
11:             $L[L[i]$.next$]$.prev $= L[i]$.prev
12:         return 0
13:     else return 1
```

Figure 4.6: RESERVE and COMMIT functions and associated data for list contraction using deterministic reservations.

Theorem 9. *For* SEQUENTIALLISTCONTRACT *on a randomly ordered L of length n, there is an iteration dependence graph G with $D(G) = O(\log n)$ with high probability, and $A(G) = \Theta(n)$ in expectation.*

4.7.2 Algorithms

This section describes parallel implementations of the list contraction that satisfy the dependencies of the sequential iterative algorithm.

Deterministic reservation-based implementation. The deterministic reservations implementation of list contraction (pseudocode shown in Figure 4.6) maintains a boolean array R initialized to all 0's. The RESERVE function for index i checks if $i < L[i]$.prev and $i < L[i]$.next, and if so, writes a value of 1 to $R[i]$. The COMMIT function for index i checks if $R[i]$ is equal to 1 and if so, splices out the node $L[i]$ and returns 0; otherwise it returns 1. These functions preserve the ordering imposed by the iteration dependence graph of L throughout its execution. To see this, note that if neither of its current neighbors in the list is lower-indexed, then step i will be a leaf in the iteration dependence graph by definition (both neighbors will be selected as roots before i in the dependence graph construction process, so i will have no descendants). Only in this case will $R[i]$ be set to 1 in the RESERVE phase, and the COMMIT phase of step i be executed. Otherwise, step i will not proceed. Therefore, by Lemma 14, it generates the same result as the sequential algorithm.

The RESERVE and COMMIT functions take constant work/depth, so the steps of the computation can be efficiently checked and updated. Applying Theorem 9 and 1 gives the following theorem for list contraction. List contraction can be implemented without concurrency because reads and writes of the neighbors inside the RESERVE and COMMIT steps can be separated into a constant number of phases such that there are no reads or writes to the same location in a phase.

Theorem 10. *For a random ordering of L, deterministic reservations using the* RESERVE *and* COMMIT *functions for list contraction runs in $O(n)$ expected work and $O(\log^2 n)$ depth w.h.p. without concurrent reads/writes or $O(\log n \log^* n)$ depth w.h.p. with concurrent reads/writes.*

Activation-Based Implementation.

Theorem 11. *For a random ordering of L, an activation-based implementation of list contraction runs in $O(n)$ work or $O(\log^2 n)$ depth w.h.p. without concurrent reads/writes, and $O(\log n \log^* n)$ depth w.h.p. using concurrent reads/writes.*

Proof. For each node, the algorithm stores a counter keeping track of the number of lower-indexed neighbors it has in the list. These counters can be initialized in linear work and constant depth. Then it identifies the "roots", which are the nodes whose counters are 0 (they have no lower-indexed neighbors). In each round, all roots are processed, and the counters of their neighbors are updated as follows. For a root v, let v_{next} be the successor node of v and v_{prev} be the predecessor node of v. Let us first analyze the case where $v_{next} > v_{prev}$. By definition of a root, $v_{prev} > v$. After splicing out v, v_{next} becomes a neighbor of v_{prev} so the algorithm decrements the counter of v_{prev}. If the counter of v_{prev} reaches 0, then v_{prev} is added to the next set of roots. The counter of v_{next} is left unchanged as its new neighbor is still a lower-indexed neighbor. In the case where $v_{prev} > v_{next}$, the algorithm decrements the counter of v_{next}, and checks whether it reaches 0. By splitting the reads and updates of neighbors into a constant number of phases, no concurrent reads or writes are required.

It can be seen that this algorithm satisfies the iteration dependence graph by noting that a node will only be spliced out if both of its neighbors in the list have higher indices, and appealing to the same argument made for the correctness of the deterministic reservations-based implementation of list contraction. Each round processes all leaves in the dependence graph, so by Theorem 9, $O(\log n)$ rounds are sufficient w.h.p. to process all of the nodes. On each round, $O(P(n))$ depth is required for packing the new roots into an array, leading to a total of $O(P(n) \log n)$ depth w.h.p. across all rounds. $P(n)$ is $O(\log n)$ if using prefix sums and $O(\log^* n)$ w.h.p. if using approximate compaction. The work spent on each node is constant, since its counter is decremented a constant number of times. The work for packing is linear in the number of nodes. Thus the total work is $O(n)$. $\square$

```
procedure SequentialTreeContract($T$)
    for $i = 0$ to $n - 1$ do
        $p = T[i]$.parent
        if $T[p]$.parent $\neq$ null then                ▷ $p$ is not root
            $s =$ sibling$(T, i)$
            $T[s]$.parent $= T[p]$.parent
            switchParentsChild$(T, p, s)$
        else switchParentsChild$(T, i,$ null$)$           ▷ $p$ is root
```

Figure 4.7: Sequential algorithm for tree contraction, where *sibling*(T, i) returns the sibling of i in T, and *switchParentsChild*(T, i, v) resets the appropriate child pointer of the parent of i to point to v instead of i.

It is straightforward to map the algorithms to the EREW PRAM or the CRCW PRAM in the same bounds as Theorem 11, and to the scan PRAM with linear work and $O(\log n)$ depth w.h.p.

4.8 Tree Contraction

As with list contraction, parallel algorithms for tree contraction have received considerable interest [335, 243, 395]. There are many variants of parallel tree contraction. This section assumes the contraction of rooted binary trees in which every internal node has exactly two children. To represent the tree, an array T of nodes is used, each node with a parent and two child pointers, with the first n nodes being leaves, and the next $n - 1$ being the internal nodes.

This section considers an iterative sequential algorithm for tree contraction that rakes the leaves of the tree one at a time, shown in Figure 4.7. To *rake* a leaf v, the algorithm splices it and its parent p out of the tree—i.e, sets v's sibling's parent pointer to be v's grandparent, and v's grandparent's child pointer to point to v's sibling instead of p. At the end, only the root node remains. As in list contraction, values can be stored on the nodes, and combined during contraction (e.g., for evaluating arithmetic expressions). This is left out of the pseudocode for simplicity. Again, if the combining function is non-associative, then the result depends on the order in which the leaves are raked, and a parallel computation returns the same result as the sequential algorithm if it satisfies the dependence structure of the sequential algorithm.

4.8.1 Iteration Dependence Depth and Aggregate Delay

This section defines the following labeling of internal nodes, and then defines a dependence structure based on it. Let $M(i)$ for each node i be the maximum index of any of the leaves in its subtree, and the *label* of each internal node be $L(i) = \min\{M(j), M(k)\}$, where j and k are the two children of i. The following fact about labels will be useful.

Lemma 16. *In* SEQUENTIALTREECONTRACT *on a tree T, the internal node with label i will be raked by the leaf with index i.*

Proof. The proof is by induction. The base case for a tree with a single leaf is trivial as there are no internal nodes. Now assume by induction that this holds for the internal nodes of two separate subtrees, joined together by a new root r. The highest-indexed leaf in each subtree will not appear as a label in the subtrees since the root takes the minimum of the two subtrees, and hence the highest-indexed leaf must be the leaf that remains when the tree is contracted (by induction). Thus, one of the two highest-indexed leaves in the two subtrees must be the node that rakes r. The smaller of these two leaves will be processed first, which is also the label on r by definition. This proves the lemma. □

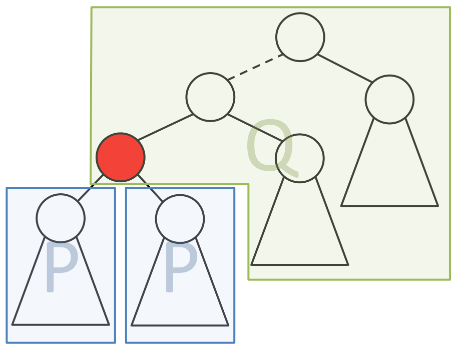
The ***dependence tree*** for a tree T is the tree created by taking the maximum label i and placing it at the root. The tree T is then partitioned by removing the internal node labeled with i, and this process is recursively applied to each subtree. The three resulting dependence trees become the children of i. This is repeated until a leaf is reached. Note that this process creates a tree over the leaf indices, since each label corresponds to a leaf index. Also note that this process is similar to how the dependence forest for the list contraction problem is generated, and hence the proof of the lemma below has a similar structure.

Lemma 17. *The dependence tree of T is an iteration dependence graph for* SEQUENTIALTREECONTRACT(T).

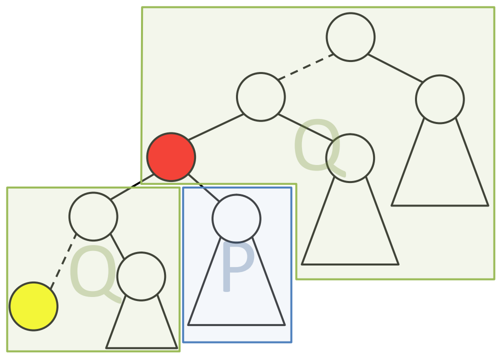
Proof. For each step i, let j and k be the labels of i's sibling and grandparent when it is raked in the sequential order. Assume leaves have null labels, so the sibling could be null. The labels j and k must both be larger than i (or null) since they have not yet been raked out. It suffices to show that for each i, once all of its descendants in the dependence tree are completed (raked out), it will have sibling j and grandparent k, and hence will do an identical rake as in the sequential order. By induction, assume that this is true for all indices less than i.

Consider the tree between j and k (not inclusive). The label i must be the largest label in this tree since if there were a larger label l, when i is contracted in the sequential order it cannot have both j as a sibling and k as a grandparent—the node with label l is not yet raked out and must be in the way. By construction of the dependence tree, and since i is the largest label in the subtree, it is picked as the root of a dependence tree containing the subtree. Therefore when all descendants are completed (and by induction we assumed they operated correctly), all other nodes on the subtree have been raked out and i will have j as a sibling and k as a grandparent. □

Let us now analyze the iteration depth and work of a dependence tree.



(a) Tree decomposition for P-state tree



(b) Tree decomposition for Q-state tree

Figure 4.8: P-state and Q-state trees used in the proof of Theorem 12. The red node is v_s, the interior node corresponding to the leaf with the second largest label. The yellow node is leaf l, the leaf with the largest label.

Theorem 12. *For* SEQUENTIALTREECONTRACT *on T with n randomly ordered leaves, there is an iteration dependence graph G with $D(G) = O(\log n)$ with high probability, and $A(G) = \Theta(n)$ in expectation.*

Proof. The dependence tree for T is based on recursively partitioning T into subtrees. To analyze the depth of the dependence tree, two types of subtrees, which have different properties, need to be considered. Define a (sub)tree to be in the ***P-state*** if the distribution of its leaves is uniformly random. Define a subtree to be in the ***Q-state*** if the location of its highest-indexed leaf is fixed. Without loss of generality, assume that a Q-state tree has its highest-indexed leaf on its left spine. Denote the leaf with the largest index in a subtree by l, the leaf with the second largest index by s, and the internal node with label s by v_s.

The initial tree is in the P-state since the ordering of the leaves is uniformly random. For a P-state tree, it is partitioned by v_s into three subtrees, where the two subtrees of the children of v_s are also in the P-state but the final tree is in the Q-state (see Figure 4.8(a)). This is because as v_s's children's subtrees are processed, there is no information about the location of the highest-indexed leaf. However, after both of the children's subtrees are processed, then leaf l will become a leaf in v_s's original position in (note that leaf l must be in v_s's subtree by definition), hence fixing the location of the highest-indexed leaf in the remaining subtree.

For a tree in the Q-state, it is partitioned by v_s into three subtrees (see Figure 4.8(b)), where v_s's left child subtree is in the Q-state (as leaf l was fixed to be on the left spine), v_s's right child subtree is in the P-state (there is no information about the location of the highest-index leaf in this subtree), and the remaining subtree is in the Q-state as after v_s's subtree is completely processed, leaf l will become a leaf in v_s's original position.

For a tree with n nodes in the P-state, the size of v_s's subtree is greater than $3n/4$ with probability at most $1/4$. This is because the location of leaf l is random and for v_s's subtree not to contain leaf l, it must appear in the rest of the tree, which has at most $1/4$ probability of occurring if v_s's subtree size is greater than $3n/4$. Hence, at least one of v_s's children's subtree has size greater than $3n/4$ with probability at most $1/4$. By a similar argument, the other subtree (of the Q-state) also has size greater than $3n/4$ with probability at most $1/4$.

For a tree with n nodes in Q-state, the size of v_s's left child's is greater than $3n/4$ with probability at most $1/4$. This is because the location of leaf s must appear in v_s's right subtree by definition, and the location of leaf s is uniformly random, so with at most $1/4$ probability it causes v_s to have a left child of size at least $3n/4$. For the subtree remaining after removing v_s's subtree, its size is greater than $3n/4$ with probability at most $1/4$ by a similar argument. Note that we have no bound on the size of v_s's right child subtree in the P-state. However, this is fine because once a tree transitions into P-state, it will be divided into small subtrees according to the analysis for P-state trees in the previous paragraph.

Consider paths from the root to each leaf in the dependence tree. Every two steps on such a path will shrink the size of the tree by a factor of $3/4$ with probability at least $1/4$ (by the arguments above). Therefore, using Markov's inequality, each path will have at most $2c\log_{16/3} n$ steps with probability at least $1 - 1/n^{c-1}$ for a constant $c > 2$. By a union bound (multiplying the failure probability by n), all path lengths and hence the tree depth will be $O(\log n)$ with probability at least $1 - 1/n^{c-2}$.

To show $A(G)$, note that a node in the dependence tree with a subtree of size k will have height $O(\log k)$ in expectation since it is true w.h.p. from the previous discussion. Let $W(n)$ indicate the expected sum of the heights of the nodes in the dependence tree. For a tree of size n, after two levels, with constant probability the largest remaining component will be $3/4n$. Assuming the worst case split is $3/4n$ and $1/4n$ when this is true, this gives the recurrence $W(n) \leq O(\log n) + p\left(W(\frac{3}{4}n) + W(\frac{1}{4}n)\right) + (1-p)W(n)$ for some constant $0 < p < 1$. By substitution, this gives $W(n) = O(n)$. $\square$

4.8.2 Algorithms

Enabling efficient checking of steps for tree contraction requires a pre-processing phase. The pre-processing phase labels each internal node with the highest-indexed leaf in its subtree. Then each internal node stores the smaller of the two computed labels of its children. Since the maximum operator does not have an inverse, the pre-processing must be done with tree contraction (using the maximum operator) in $O(n)$ work and $O(\log n)$ depth. Note that, however, maximum is associative, so the result of this pre-processing phase would be consistent with any tree contraction algorithm. After pre-processing, the parallel algorithms described in this section can be run with any operator (does not have to be associative), and give the same answer as the sequential algorithm (Algorithm 4.7).

```
1: $R = \{0, \ldots, 0\}$                                   ▷ boolean array
2: procedure RESERVE($i$)
3:     if $i < j,\ \forall j \in N(i)$ then
4:         $R[i] = 1$                                        ▷ reserve own location
5:     return 1
6: procedure COMMIT($i$)
7:     if $(R[i] = 1)$ then
8:         $p = T[i]$.parent
9:         if $T[p]$.parent $\neq$ null then                 ▷ $p$ is not root
10:            $s =$ sibling$(T, i)$
11:            $T[s]$.parent $= T[p]$.parent
12:            switchParentsChild$(T, p, s)$
13:        else                                              ▷ $p$ is root
14:            switchParentsChild$(T, i,$ null$)$
15:            return 0
16:    else return 1
```

Figure 4.9: RESERVE and COMMIT functions and associated data for tree contraction using deterministic reservations. *sibling*(T, i) returns the sibling of i in T, and *switchParentsChild*(T, i, v) resets the appropriate child pointer of the parent of i to point to v instead of i.

With the internal nodes labeled, the ***neighborhood*** of a leaf is defined as the leaves labeled on its parent and its grandparent nodes. Only when the labels on these two internal nodes are greater than or equal to the leaf's ID can the leaf proceed in raking.

Deterministic reservations-based implementation. Figure 4.9 defines the RESERVE and COMMIT functions and associated data required for deterministic reservations. $N(i)$ corresponds to the neighborhood of step i, which includes the leaf labeled on its parent (if it has one) and the leaf labeled on its grandparent (if it has one). These functions preserve the ordering imposed by the iteration dependence graph of T defined in this section throughout its execution because if the i'th leaf is spliced out, the RESERVE step guarantees that if $R[i]$ is set to 1, and guarantees that there are no lower-indexed leaves in the neighborhood of step i (i.e., step i has no children in the dependence forest). Only in this case does step i rake itself out in the COMMIT step (the procedure for raking is the same as in the sequential algorithm shown in Algorithm 4.7).

Again, the steps can be efficiently checked and updated because the RESERVE and COMMIT functions take constant work/depth. Applying Theorem 12 and Lemma 1 gives the following theorem for tree contraction. Again, concurrency can be avoided because reads and writes of the neighbors inside the RESERVE and COMMIT steps can be separated into a constant number of phases such that there are no reads or writes to the same location in a phase.

Theorem 13. *For a random ordering of T, deterministic reservations using the* RESERVE *and* COMMIT *functions for tree contraction runs in $O(n)$ expected work and $O(\log^2 n)$ depth w.h.p. without concurrent reads/writes or $O(\log n \log^* n)$ depth w.h.p. with concurrent reads/writes.*

The tree contraction used for pre-processing can be done deterministically in linear work and $O(\log n)$ depth (on the EREW PRAM), which is within the stated complexity bounds of Theorem 13.

Activation-based implementation.

Theorem 14. *An activation-based implementation of Algorithm 4.7 runs in $O(n)$ work and $O(\log^2 n)$ depth w.h.p. without concurrent reads/writes or $O(\log n \log^* n)$ depth w.h.p. with concurrent reads/writes.*

Proof. The activation-based implementation of list contraction described in Theorem 11 can be adapted for tree contraction. The "roots" are the steps with no lower labels on its parent and grandparent, which implies that it has no lower-indexed steps in its neighborhood. A root that is successfully processed potentially updates the counters of the steps in its neighborhood. The counter of each step is initialized to the number of lower-indexed steps that are in its neighborhood. Overall this takes linear work and constant depth. This algorithm satisfies the dependencies of the iteration dependence graph defined in this section because the roots are the steps that have no more dependencies. Again, the reads and updates can be split into a constant number of phases to avoid concurrency. Since the iteration depth is $O(\log n)$ w.h.p. by Theorem 12, and each round of the algorithm reduces the iteration depth of the remaining dependence graph by 1, $O(\log n)$ rounds are required w.h.p. Therefore, the total depth is $O(P(n) \log n)$ w.h.p., where $P(n)$ is $O(\log n)$ using prefix sums and $O(\log^* n)$ w.h.p. using approximate compaction (requiring concurrent reads/writes). The work is linear because each step is processed a constant number of times. □

Again, mapping the algorithms to the EREW PRAM, CRCW PRAM, or scan PRAM is straightforward.

4.9 Limited Randomness

The parallel algorithms described in this chapter use $O(\log n)$ random bits per input element, thus requiring $O(n \log n)$ bits of randomness in total.[4] This section describes how to reduce the amount of randomness to a polylogarithmic number of random bits while preserving the iteration dependence depth for random permutation and list contraction.

[4] $O(m \log m)$ bits of randomness for maximal matching.

To show that limited randomness suffices, this section employs Nisan's [354] pseudorandom generator for space-bounded computation, which uses $O(S \log n)$ truly random bits to generate pseudorandom bits that are capable of fooling an S-space machine. More accurately, the probability of failure event given the generated stream of pseudorandom bits differs by at most (an additive) ϵ from the failure probability given truly random bits, where the *bias* ϵ can be driven down to $O(1/n^c)$ for any constant c by increasing the number of truly random bits by a constant factor. Thus, a result that holds with high probability using truly random bits also holds with high probability using the pseudorandom bits, provided that the failure event can be tested by an S-space machine.

For the purposes of this section, it suffices to show that a space-S computation can verify the iteration depth of the dependence graph. As long as the low-space computation uses the same mapping from random bits to steps, the actual computation will have the same dependence graph. The challenge in designing these low-space verifiers and applying Nisan's theorem is that the verifier must consume the random bits as a one-pass stream of bits. By exhibiting such $O(\log n)$-space and $O(\log^2 n)$-space verifiers for the iteration depths of random permutation and list contraction, respectively, this section proves that $O(\log^2 n)$ random bits suffice for random permutation and $O(\log^3 n)$ random bits suffice for list contraction.

Theorem 15. *Using Nisan's generator with a seed of $O(\log^2 n)$ random bits, the iteration depth of the dependence graph for random permutation is $O(\log n)$ with high probability.*

Proof. Consider a single step i. Theorem 6 states that if each step chooses uniformly random numbers, then for any constant c the probability of step i exceeding depth $O(c \log n)$ is $O(1/n^c)$. Assuming that the depth bound for step i can be verified in $O(\log n)$ space, Nisan's theorem states that the probability of exceeding the depth bound using the generated pseudorandom bits is at most $O(1/n^c) + \epsilon = O(1/n^c)$. Taking a union bound over all steps, the probability of choosing a seed that causes any step to have high depth is $O(1/n^{c-1})$.

The following is an $O(\log n)$-space procedure for calculating the depth of step i, using a single pass through the stream of random bits. Scan from step i down to step $H[i]$ in the input array, counting the number of intervening steps k such that $H[k] = H[i]$. These steps form a chain in the dependence forest directed from i to $H[i]$. Repeat this process starting from $i' = H[i]$ down to $H[i']$, until reaching the root of this tree (i.e., the starting node i' has $H[i'] = i'$). The sum of the lengths is equal to the depth of i in the dependence forest. This process requires $O(\log n)$ space to maintain a few pointers and the sum.

One additional detail is that the permutation algorithm expects random values in the range $[0, \ldots, i]$, but what we have access to is a stream of (pseudo)random bits. Without loss of generality, assume n is a power of 2. To generate a number in the range $[0, \ldots, i]$, for any constant c first generate a number x in the range $[0, \ldots, n^c - 1]$. For values

$x < (i+1)\lfloor n^c/(i+1)\rfloor$, use $H[i] = x/(\lfloor n^c/(i+1)\rfloor)$. If any larger value is generated, the algorithm fails. The probability of failure for a particular value is at most $n/n^c = 1/n^{c-1}$, and using a union bound over all values, the failure probability becomes $O(1/n^{c-2})$. $\square$

Note that the random permutation produced using limited randomness is not truly random.

For list contraction, assume that each node is assigned a random number, called a ***priority***, from the random bits of Nisan's generator. The random ordering of the list L can be viewed as the ordering in which the priorities are sorted in increasing order. By choosing random numbers from the range $[0, \ldots, n^c - 1]$ for constant $c > 1$, the priorities are distinct w.h.p. and Theorem 9 applies.

Theorem 16. *Using Nisan's generator with a seed of $O(\log^3 n)$ random bits to assign each node a (pseudo)random priority, the iteration depth of the dependence graph for list contraction is $O(\log n)$ with high probability.*

Proof. As in the proof of Theorem 15, this proof will exhibit an algorithm that can verify the depth of a node/step in the dependence tree using a single pass through the random priorities. Since the probability of the depth bound being exceeded is polynomially small, a union bound over all steps completes the proof.

To verify the depth of node x in the dependence forest, the verifier simulates the incremental insertion of nodes, in input order, into the dependence forest. After each step, the structure of the dependence tree containing x is identical to a treap using the same priorities and node comparisons respecting list-order. The simulation begins by inserting the node x, assuming pessimistically that it has minimum priority (which only increases its depth). Throughout the process, the root-to-leaf path down to x is maintained. When inserting a new node z, the idea is to simulate the treap insertion process with respect to the path down to x. To insert z, step down the path until finding the first (highest) node y such that either x and z are in different subtrees of y, or $y = x$. If z has lower priority than y, then the path to x is unchanged. Otherwise, splice in z to be the parent of y, and repeatedly rotate z and its parent until z has lower priority than its parent. This rotation process may result in the path shortening and/or the ancestors being rearranged, depending on the list-order comparisons among nodes.

List-order comparisons can be performed in $O(\log n)$ space using a constant number of pointers and traversing the list. As long as the depth of a node never exceeds $O(\log n)$, then the space used by the simulation is $O(\log^2 n)$. If the depth ever exceeds $O(\log n)$, then the simulation stops and reports a high-depth node. By Theorem 9, this is a low probability event. $\square$

The work and depth required to generate the random numbers from Nisan's pseudorandom generator will be analyzed next. The generator uses $O(\log n)$ independent hash functions $h_1, \ldots, h_S$, each requiring $O(S)$ random bits, and a seed x with $O(S)$ random bits [354]. Define $G_0(x) = x$ and $G_t(x) = (G_{t-1}(x), h_t(G_{t-1}(x)))$ for $t \geq 1$. The output of the generator is $G_{t'}(x)$, where $t' = O(\log(n \log(n)/S))$, which has $O(n \log n)$ bits.

Lemma 18. *The output of Nisan's pseudorandom generator can be computed in $O(nS/\log n)$ work and $O(\log n \log(1 + S/\log n))$ depth.*

Proof. Construct $G_{t'}(x)$ recursively using the definition above. Level t of the recursion requires $O(2^t(S/\log n)^2)$ work and $O(\log(1 + S/\log n))$ depth, as the hash functions can be evaluated in $O((S/\log n)^2)$ work and $O(\log(1 + S/\log n))$ depth using naive multiplication ($O(\log n)$ bits can be evaluated with one unit of work). Generating $O(n \log n)$ pseudorandom bits requires $O(\log(n \log(n)/S))$ levels of recursion. The total work is $\sum_{t=0}^{\log(n\log(n)/S)} O(2^t(S/\log n)^2) = O(nS/\log n)$ and depth is $O(\log n \log(1 + S/\log n))$. □

Plugging in the space bounds for random permutation and list contraction into Lemma 18 gives the following corollary.

Corollary 3. *The random bits of Nisan's pseudorandom generator for the random permutation and list contraction algorithms can be computed in $O(n)$ work and $O(\log n)$ depth, and $O(n \log n)$ work and $O(\log n \log \log n)$ depth, respectively.*

4.10 Experiments

This section describes experimental results for the deterministic reservations-based implementations of the problems studied in this chapter. The experiments are done using varying prefix sizes, to show how prefix size affects work, parallelism, and overall running time. The parallel codes are compared to their corresponding sequential implementations.

4.10.1 MIS and Maximal Matching

Experimental Setup. The experiments are run on the 32-core Intel machine described in Section 2.7. The parallel programs were compiled using the `cilk++` compiler, and sequential programs were compiled using `g++`. For each prefix size, thread count, and input, the reported time is the median time over three trials.

Inputs. The input graphs and their sizes are listed in Table 4.1. The random local graph (rg) was generated such that probability of an edge existing between two vertices is inversely proportional to their distance in the vertex array. The rMat graph has a power-law distribution of degrees and was generated according to the procedure described in [87], with

Input Graph	Size
Random local graph (**rg**)	$n = 10^7$, $m = 5 \times 10^7$
rMat graph (**rMat**)	$n = 2^{24}$, $m = 5 \times 10^7$
3D grid (**3D**)	$n = 10^7$, $m = 2 \times 10^7$

Table 4.1: Input Graphs for maximal independent set and maximal matching.

```
enum FlType {IN, OUT, LIVE};

struct MISStep {
  FlType flag; vertex *V;
  MISStep(char* _F, vertex* _V) : flag(_F), V(_V) {}

  bool reserve(int i) {
    int d = V[i].degree;
    flag = IN;
    for (int j = 0; j < d; j++) {
      int ngh = V[i].Neighbors[j];
      if (ngh < i) {                                                  //earlier neighbor
        if (Fl[ngh] == IN) { flag = OUT; return 1;}       //drop out if neighbor is in MIS
        else if (Fl[ngh] == LIVE) flag = LIVE; } }  //undecided if neighbor is still live
    return 1; }

  bool commit(int i) { return (Fl[i] = flag) != LIVE;}                     //write status
};

void MIS(FlType* Fl, vertex* V, int n, int psize)
  speculative_for(MISStep(Fl, V), 0, n, psize);       //deterministic reservations driver
}
```

Figure 4.10: C++ code for maximal independent set using deterministic reservations.

parameters $a = 0.5,\ b = 0.1,\ c = 0.1$ and $d = 0.3$. The 3D grid graph consists of vertices on a grid in a 3-dimensional space, where each vertex has edges to its 6 nearest neighbors (2 in each dimension).

Implementation. The implementation of the prefix-based MIS and MM algorithms differ slightly from the ones with good theoretical guarantees described in the previous sections, but we found that these implementations work better in practice. Firstly, the prefix size is fixed throughout the algorithm. Secondly, the algorithm does not process each prefix to completion but instead process each particular prefix only once, and moves the iterates which still need to be processed into the next prefix (the number of new iterates in the next prefix is equal to the difference between the prefix size and the number of iterates which still need to be processed from the current prefix).

For MIS, each time a prefix is processed, there are 3 possible outcomes for each vertex in the prefix: 1) the vertex joins the MIS and is deleted because it has the highest priority among all of its neighbors; 2) the vertex is deleted because at least one of its neighbors is already in the MIS; or 3) the vertex is undecided and is moved to the next prefix. The C++ code based on the deterministic reservations interface from Chapter 3 is given in

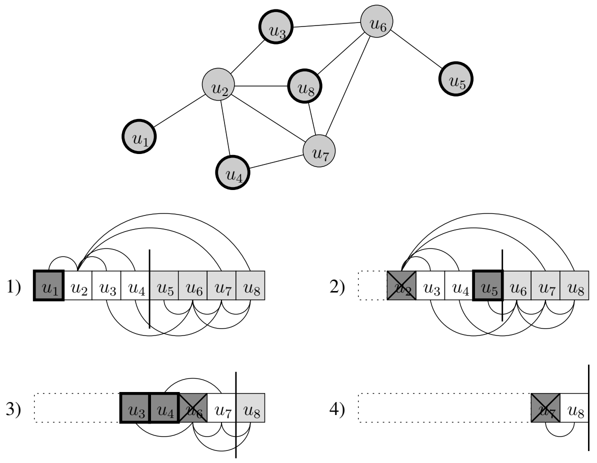
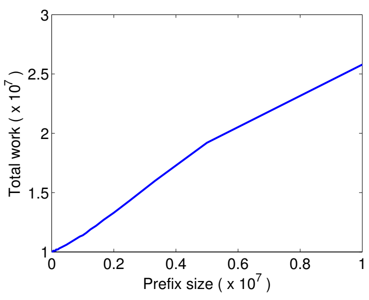


Figure 4.11: An example graph and an execution of deterministic reservations for finding a maximal independent set. Here, the subscript of a vertex corresponds to its priority in the deterministic reservations. The prefix size is chosen to be 4. (1) shows the initial graph in priority order, and (2)–(4) show subsequent rounds of the algorithm. The vertical line indicates the end of the current prefix. Dark-gray vertices are those that become IN or OUT during that round: vertices with a thick border are IN and accepted into the MIS, and vertices with an "X" are OUT as they have a neighbor already in the MIS. For example, u_1 is the only vertex accepted into the MIS during the first round. Similarly, u_2 becomes OUT in the second round as it has a neighbor already in the MIS (namely, u_1). White vertices are those belonging to the current prefix that remain LIVE. For example, in the first round u_2, u_3, and u_4 all have a higher priority neighbor in the same prefix and remain live. Only vertices that survive the previous round (LIVE vertices) are displayed in the array and part of the current prefix, so u_5 is skipped in (3). Vertices in the MIS are also shown with thick border in the graph.

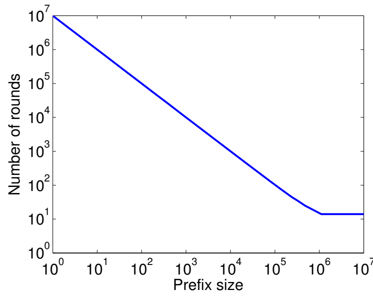
Figure 4.10 and an example of how the algorithm proceeds is shown in Figure 4.11. The `struct MISStep` defines the code for the reserve and commit components for each loop iteration. The array `V` stores for each of the `n` vertices its degree and a pointer to an array of neighbors. The array `Fl` keeps track of the status of each vertex—`IN` indicates it is done and in the set (corresponding to the first outcome), `OUT` indicates it is done and not in the set (a neighbor is in the set; this corresponds to the second outcome), and `LIVE` indicates it is still live (corresponding to the third outcome). The reserve phase for each iteration `i` loops over the neighbors of `V[i]` and sets a local variable `flag` as follows:

$$\texttt{flag} = \begin{cases} \texttt{OUT} & \text{any earlier neighbor is } \texttt{IN} \\ \texttt{LIVE} & \text{any earlier neighbor is } \texttt{LIVE} \\ \texttt{IN} & \text{otherwise} \end{cases}$$

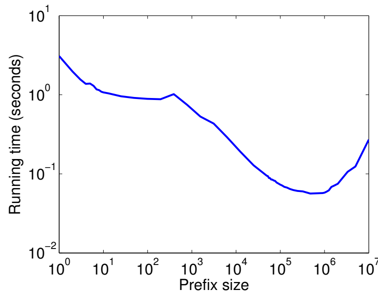
The second case corresponds to a conflict since for an earlier neighbor it is not yet known if it is IN or OUT. The commit phase for iteration `i` simply copies the local `flag` to `Fl[i]`. Since `Fl` is only read in the reserve phase and only written (to location `i`) in the commit



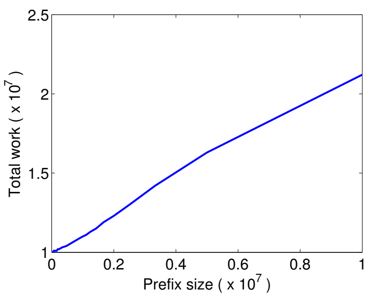
(a) Total work done vs. prefix size on **rg**



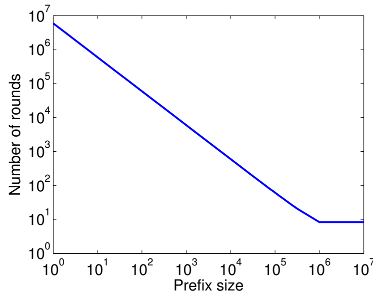
(b) Number of rounds vs. prefix size on **rg** in log-log scale



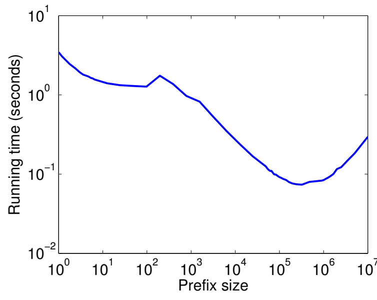
(c) Running time (32 cores) vs. prefix size on **rg** in log-log scale



(d) Total work done vs. prefix size on **rMat**



(e) Number of rounds vs. prefix size on **rMat** in log-log scale



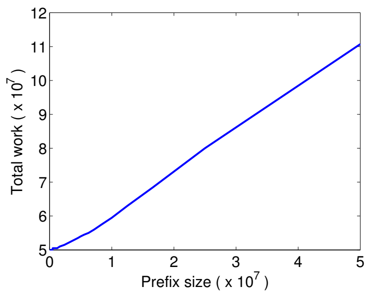
(f) Running time (32 cores) vs. prefix size on **rMat** in log-log scale

Figure 4.12: Plots showing the trade-off between various properties and the prefix size in maximal independent set.

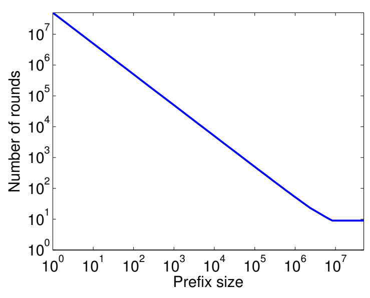
phase, all operations commute. Note that surprisingly, this implementation does not even require any priority writes. Also, note that the reserve phase for each vertex is implemented sequentially, which allows the loop to break early when possible (an earlier neighbor is in the MIS). While this loop could be parallelized, we did not find a performance improvement by doing so for the inputs considered, due to the extra overheads involved.

For MM, each time a prefix is processed, there are 2 phases: In the first phase, each edge in the prefix checks whether or not either of its endpoints have been matched, and if not, the edge does a priority write to each of its two endpoints; in the second phase, each edge checks whether its priority writes were successful on both of its endpoints, and if so joins the MM and marks its endpoints as matched. Successful edges from the second phase and edges which discovered during the first phase that it had an endpoint already matched are deleted.

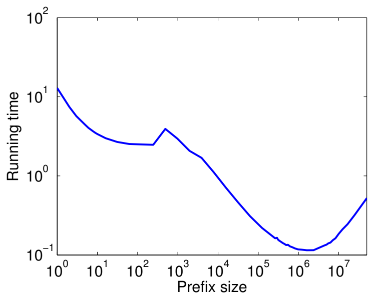
Results. The first set of experiments analyze the work, parallelism, and running time of the MIS and MM implementations as a function of the prefix size on the random local and rMat graphs. The results are plotted in Figures 4.12 and 4.13.



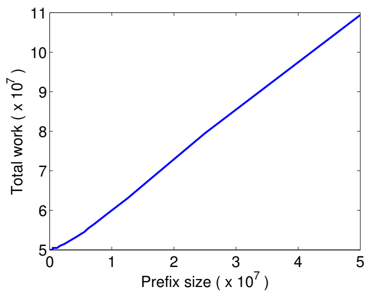
(a) Total work done vs. prefix size on **rg**



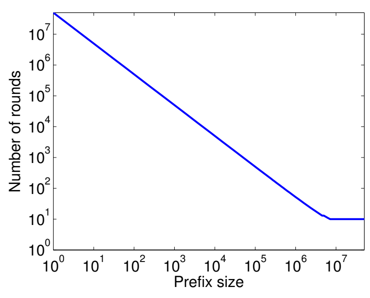
(b) Number of rounds vs. prefix size on **rg** in log-log scale



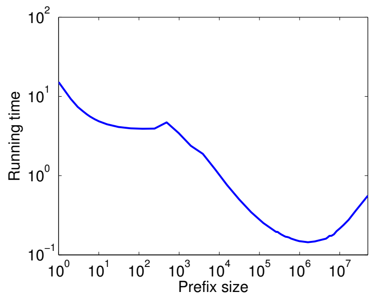
(c) Running time (32 cores) vs. prefix size on **rg** in log-log scale



(d) Total work done vs. prefix size on **rMat**



(e) Number of rounds vs. prefix size on **rMat** in log-log scale



(f) Running time (32 cores) vs. prefix size on **rMat** in log-log scale

Figure 4.13: Plots showing the trade-off between various properties and the prefix size in maximal matching.

For both MIS and MM, the reader can observe that, as expected, increasing the prefix size increases both the total work performed (Figures 4.12(a), 4.12(d), 4.13(a), and 4.13(d)) and the parallelism, which is estimated by the number of rounds of the outer loop (selecting prefixes) the algorithm takes to complete (Figures 4.12(b), 4.12(e), 4.13(b), and 4.13(e)). As expected, the total work performed and the number of rounds taken by a sequential implementation are both equal to the input size. By examining the graphs of running time versus prefix size (Figures 4.12(c), 4.12(f), 4.13(c), and 4.13(f)), we see that there is some optimal prefix size between 1 (fully sequential) and the input size (fully parallel). In the running time versus prefix size graphs, there is a small bump when the prefix-to-input size ratio is between 10^{-6} and 10^{-4} corresponding to the point when the for-loop in the implementation transitions from sequential to parallel (the implementation uses a grain size of 256).

The single-thread and 32-core parallel times on the input graphs for MIS and MM using the optimal prefix size (refer to Figures 4.12(c), 4.12(f), 4.13(c), and 4.13(f)) are reported in Tables 4.2 and 4.3, respectively. The experiments also compare the prefix-based algorithms to optimized sequential implementations, and additionally for MIS compare with our

Input Graph	Serial MIS	Prefix-based MIS	Prefix-based MIS	Luby	Luby
	(1)	(1)	(32h)	(1)	(32h)
rg	0.455	0.57	0.059	6.49	0.245
rMat	0.677	0.939	0.073	8.33	0.313
3D	0.393	0.519	0.051	4.18	0.161

Table 4.2: Running times (in seconds) of the various MIS algorithms on different input graphs on a 32-core machine with hyper-threading using one thread (1) and all threads (32h).

Input Graph	Serial MM	Prefix-based MM	Prefix-based MM
	(1)	(1)	(32h)
rg	1.04	2.24	0.135
rMat	1.41	3.51	0.155
3D	0.792	1.8	0.11

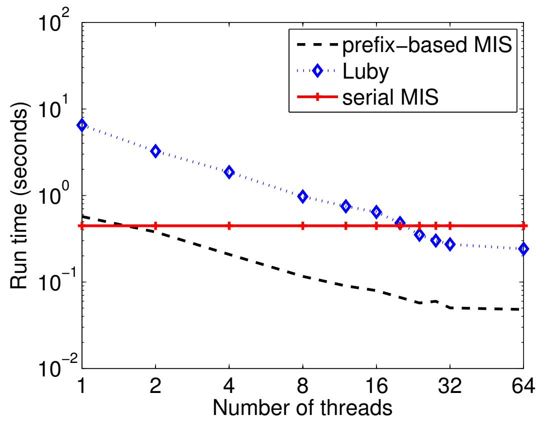
Table 4.3: Running times (in seconds) of the various MM algorithms on different input graphs on a 32-core machine with hyper-threading using one thread (1) and all threads (32h).

optimized implementation of Luby's algorithm. We implemented several versions of Luby's algorithm and report the times for the fastest one. The prefix-based MIS implementation is 3–8 times faster than Luby's algorithm (shown in Figures 4.14(a) and 4.14(b)) which processes the entire remaining graph (and generates new priorities) in each round. This improvement demonstrates that the prefix-based approach, although sacrificing some parallelism, leads to less overall work and lower running time. When using more than 2 threads, the prefix-based implementation of MIS outperforms the serial version, while the implementation of Luby's algorithm requires 16 or more threads to outperform the serial version. The prefix-based algorithm achieves 9–13x speedup on 32 cores. For MM, the prefix-based algorithm outperforms the corresponding serial implementation with 4 or more threads and achieves 16–23x speedup on 32 cores (Figures 4.15(a) and 4.15(b)). Note that since the serial MIS and MM algorithms are so simple, it is not easy for a parallel implementation to outperform the corresponding serial implementation.

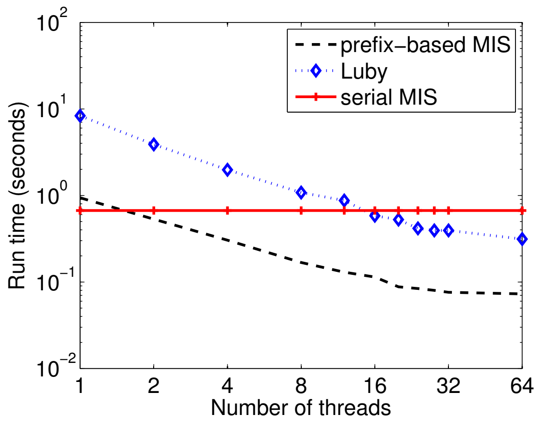
4.10.2 Random Permutation, List Contraction, and Tree Contraction

Experimental Setup. The implementations of random permutation, list contraction, and tree contraction use Cilk Plus, and are compiled using g++. The experiments are performed on the 40-core Intel machine with two-way hyper-threading, described in Section 2.7. The times that are reported are based on a median of three trials.

Inputs. The number of elements for random permutation, number of nodes for list contraction, and number of leaves for tree contraction is 10^9. For random permutation, the data array A stores 32-bit integers and the swap targets (the H array) are randomly generated. For list contraction, to generate the input, a random permutation was first generated, giving a collection of cycles on the nodes, and then one edge on each cycle was deleted, giving

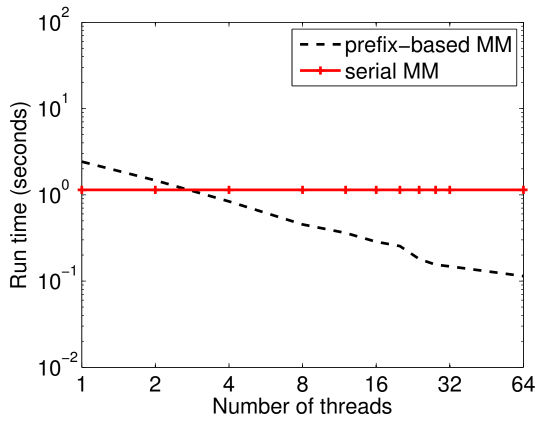


(a) Running time vs. number of threads on **rg** in log-log scale

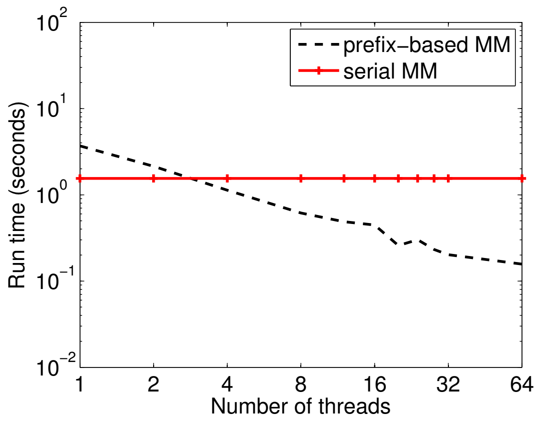


(b) Running time vs. number of threads on **rMat** in log-log scale

Figure 4.14: Plots showing the running time vs. number of threads for the different MIS algorithms on a 32-core machine (with hyper-threading). For the prefix-based algorithm, a prefix size of $n/50$ was used.



(a) Running time vs. number of threads on **rg** in log-log scale



(b) Running time vs. number of threads on **rMat** in log-log scale

Figure 4.15: Plots showing the running time vs. number of threads for the different MM algorithms on a 32-core machine (with hyper-threading). For the prefix-based algorithm, a prefix size of $m/50$ was used.

a collection of linked lists. For tree contraction, the input was a random binary tree with 10^9 randomly-indexed leaves, giving a total of $2 \times 10^9 - 1$ nodes. Often, list and tree contraction are used as a part of a larger algorithm, so the pre-processing step of randomly permuting the elements only needs to be applied once. The experiments do not store values on the nodes for list contraction and tree contraction.

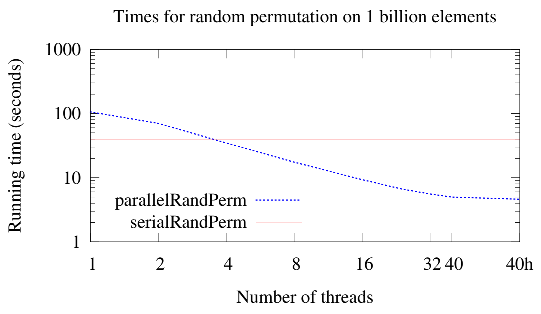
Algorithm	(1)	(40h)	(seq)
Random permutation	92.1	4.62	38.8
List contraction	160	3.97	46
Tree contraction	350	10.0	172

Table 4.4: Times (seconds) for $n = 10^9$ on 40 cores with hyper-threading. (1) indicates 1 thread, (40h) indicates 80 hyper-threads, and (seq) is the sequential iterative implementation.

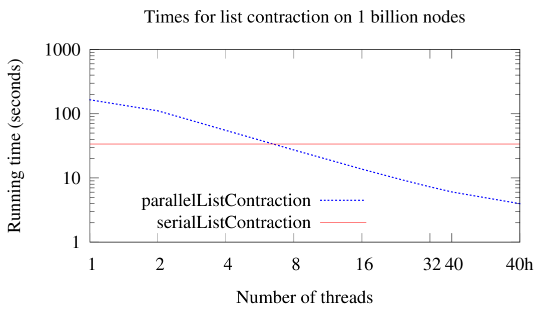
Implementation. We implement the deterministic parallel algorithms for random permutation, list contraction, and tree contraction. The writeMax operation used in random permutation is a case of priority update (discussed in Chapter 6). For tree contraction, we use a version that does not do a pre-processing step, and each leaf simply checks its nearby leaves to see if there are any conflicts. This version does not return the same answer as the sequential algorithm (but is still deterministic), and it is more efficient as it does not require a pre-processing step. All of the parallel implementations use the prefix-based version of deterministic reservations, which performs better in practice than the version used in the analysis that processes all remaining steps in each round. Proofs of the complexity bounds of the prefix-based algorithms can be found in the Appendix of [427]. As in the implementations of maximal independent set and maximal matching, each prefix is processed once, and the unsuccessful steps are moved to the next prefix. For random permutation, the implementation uses a prefix size of $n_i/50$ where n_i is the number of remaining steps. For list contraction, the implementation uses a fixed prefix size of $n/100$, and for tree contraction the implementation uses a fixed prefix size of $n/50$. These were experimentally determined to give the best performance. The implementations are all very simple—the random permutation and list contraction implementations use under a dozen lines of `C++` code and the tree contraction implementation uses a few dozen lines. For comparison, we also implement the corresponding sequential iterative algorithms for the three problems.

Results. A summary of the timings for each of the three algorithms are shown in Table 4.4. Plots of running time versus number of threads in log-log scale for each of the three algorithms are shown in Figure 4.16. Observe that the parallel implementations all get good speedup, and outperform the corresponding sequential implementation with a modest number of threads.

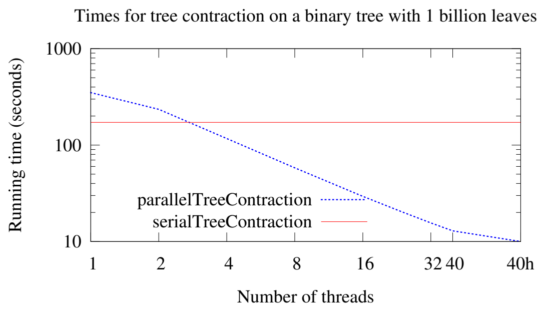
For random permutation, the parallel implementation outperforms the standard simple sequential implementation [270] with 4 or more threads. We also compared it to a sorting-based random permutation algorithm that we implemented, which creates pairs $(A[i], r_i)$ where each r_i is a random number drawn from $[1, \ldots, n^2]$, and sorts on the second value of the pair. Note that this does not give the same permutation as the sequential algorithm. The implementation uses a parallel sample sort, which is part of the Problem Based Benchmark Suite. On 80 hyper-threads the sorting-based algorithm took 5.38 seconds, and on a single thread it took 204 seconds. Both of these timings are inferior to the times



(a) random permutation



(b) list contraction



(c) tree contraction

Figure 4.16: Running time vs. number of threads for $n = 10^9$ on 40 cores with hyper-threading (log-log scale). "40h" indicates 80 hyper-threads.

reported in Table 4.4 for the random permutation algorithm implemented with deterministic reservations.

An experimental study of other parallel random permutation algorithms has recently been conducted by Cong and Bader [109], which compares algorithms based on sorting [388], dart-throwing [335, 169, 173], and an adaptation of Sander's distributed algorithm [406]. None of these algorithms generate the same permutation as the sequential algorithm. It is difficult to directly compare with their reported numbers because their numbers include the cost for generating random numbers, while the numbers reported in this section do not, their input sizes are much smaller (the largest size was 20 million elements), and the machine specifications are different.

For list contraction, the parallel implementation outperforms the serial implementation with 8 or more threads. The experiments also compare to a parallel implementation of list contraction where the random numbers are regenerated in each round. In this strawman implementation, the prefix processing idea cannot be directly applied because the priorities of the nodes are not fixed. Therefore all remaining nodes are processed in each iteration.

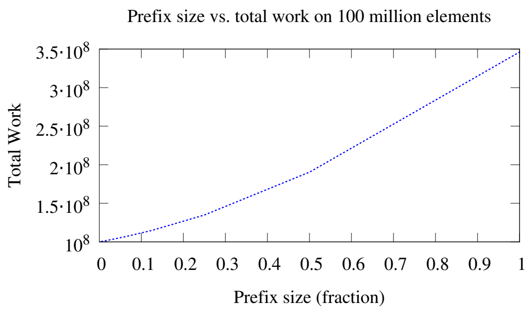
On 80 hyper-threads, the implementation took 6.46 seconds to finish. This is slower than the prefix-based parallel implementation reported in Table 4.4, which took 3.97 seconds on the same input. The reason is that there is more wasted work in processing all of the nodes on each iteration, and also an added cost of regenerating random numbers on each iteration. In addition, this implementation does not return the same answer as the sequential implementation.

List ranking algorithms have been studied experimentally in the literature [392, 430, 370, 128, 221, 222, 20, 391]. None of these implementations return the same answer as a sequential ordering of processing the nodes would. The most recent experimental work on list ranking for multicores is by Bader et al. [20]. However since they used a much older machine, and they are solving list ranking instead of list contraction, it is hard to compare.

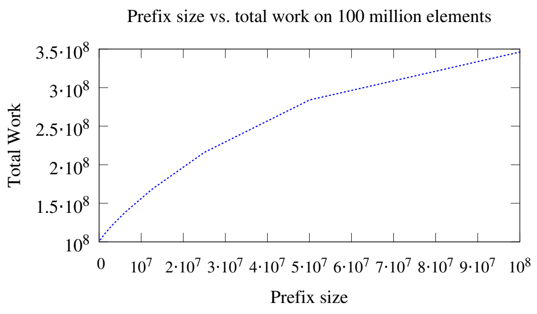
Finally, for tree contraction the parallel implementation outperforms the sequential implementation with 4 or more threads. Again, the experiments compare it with a parallel strawman version that processes all remaining leaves and regenerates the random numbers on each iteration. On 80 hyper-threads this implementation took 23.3 seconds, compared to 10 seconds for the prefix-based parallel implementation reported in Table 4.4. As in list contraction, this is due to the wasted work of processing all leaves on each iteration and the added cost of regenerating the random numbers.

The most recent experimental work on tree contraction on multicores is by Bader et al. [25]. They present an implementation of tree contraction based on the standard algorithm that only rakes leaves [243]. The algorithm is more complicated than the one described in this chapter as it involves using Euler tours and list ranking to label the leaves to allow non-conflicting leaves to be raked in parallel. Furthermore, it does not return the same answer as a natural sequential algorithm. Again, because they use a much older machine and they solve the more expensive arithmetic expression computation, it is hard to compare.

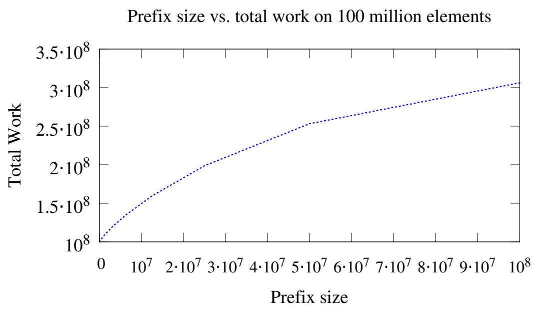
Figure 4.17 plots the total work performed by the three algorithms as a function of the prefix size for $n = 10^8$. Since the prefix size is a constant fraction for random permutation, in the plots, the x-axis shows the fraction used. For list contraction and tree contraction, the prefix size is fixed across rounds, so the x-axis shows the actual size of the prefix. Similar to the case of maximal independent set and maximal matching, the work goes up as the prefix size is increased as there is more wasted work due to failed steps. Note that a prefix size of 1 corresponds to the work performed by the sequential algorithm. Figure 4.18 plots the number of rounds of deterministic reservations as a function of prefix size in log-log scale. The opposite effect is observed here—a larger prefix size leads to fewer rounds because there is more parallelism. These plots show the trade-off between work and parallelism. Finally, Figure 4.19 plots the parallel running time as a function of the prefix size in log-log scale, showing that the best running times use a prefix size somewhere in between 1 and n.



(a) random permutation

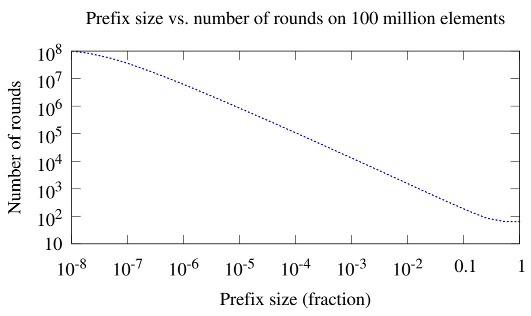


(b) list contraction

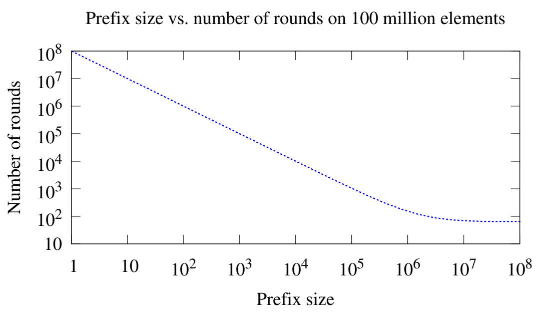


(c) tree contraction

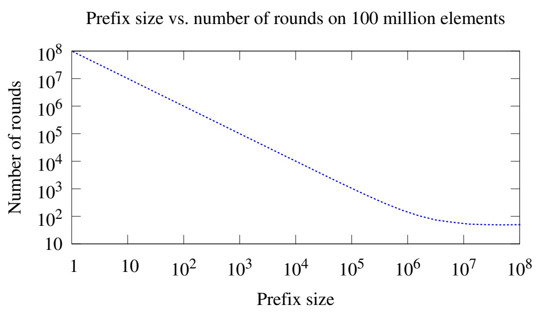
Figure 4.17: Total work vs. prefix size for $n = 10^8$ for random permutation, list contraction, and tree contraction.



(a) random permutation

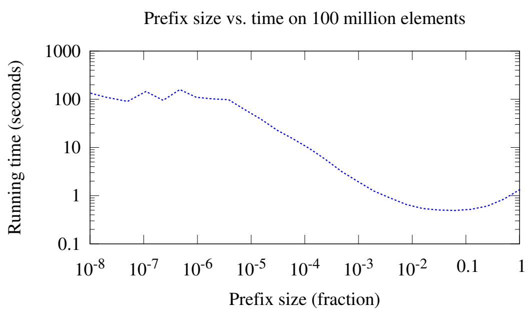


(b) list contraction

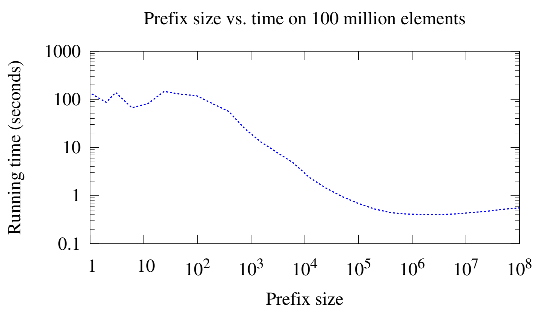


(c) tree contraction

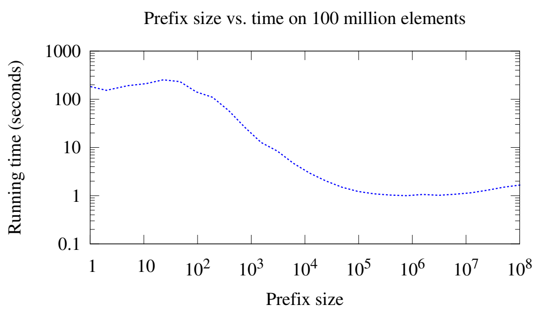
Figure 4.18: Number of rounds vs. prefix size for $n = 10^8$ (log-log scale) for random permutation, list contraction, and tree contraction.



(a) random permutation



(b) list contraction



(c) tree contraction

Figure 4.19: Running time vs. prefix size for $n = 10^8$ on 40 cores with hyper-threading (log-log scale) for random permutation, list contraction, and tree contraction.

Chapter 5

A Deterministic Phase-Concurrent Parallel Hash Table

5.1 Introduction

The importance of internal determinism in developing and debugging parallel programs has been argued in Chapter 3. In the context of concurrent access, a data structure is internally deterministic if even when operations are applied concurrently the final observable state depends uniquely on the set of operations applied, but not on their order. This property is equivalent to saying the operations commute with respect to the final observable state of the structure [459, 436]. However, for certain data structures, the operations naturally do not commute. For example, in a hash table, mixing insertions and deletions in time would inherently depend on ordering since inserting and deleting the same element do not commute, but insertions commute with each other and deletions commute with each other, independently of value. The same is true for searching mixed with either insertion or deletion. For a data structure in which certain operations commute but others do not, it is useful to group the operations into phases such that the concurrent operations within a phase commute. This chapter defines a data structure to be ***phase-concurrent*** if subsets of operations can proceed (safely) concurrently. If the operations within a phase also commute, then the data structure is deterministic. Note that phase-concurrency can have other uses besides determinism, such as giving more efficient data structures. It is the programmer's responsibility to separate concurrent operations into phases, with synchronization in between, which for most nested parallel programs is easy and natural to do.

This chapter focuses on the hash table data structure. We develop a deterministic phase-concurrent hash table and prove its correctness. This hash table is part of the Problem Based

Benchmark Suite, and is also what is used to implement the dynamic map in Section 3.3. The data structure builds upon a sequential history-independent hash table [58] and allows concurrent insertions, concurrent deletions, concurrent searches, and reporting the contents. It does not allow different types of operations to be mixed in time, because commutativity (and hence determinism) would be violated in general. This chapter shows that using one type of operation at a time is still very useful for many applications. The hash table uses open addressing with a prioritized variant of linear probing and guarantees that in a quiescent state (when there are no operations ongoing) the exact content of the array is independent of the ordering of previous updates. This allows, for example, quickly returning the contents of the hash table in a deterministic order simply by packing out the empty cells, which is useful in many applications. Returning the contents could be done deterministically by sorting, but this is more expensive. The hash table can store key-value pairs either directly or via a pointer.

The experimental section in this chapter (Section 5.6) presents timings for insertions, deletions, finds, and returning the contents into an array on a 40-core machine. These timings are compared with the timings of several other implementations of concurrent and phase-concurrent hash tables, including the fastest concurrent open addressing [226] and closed addressing [293] hash tables that we could find, and two of our nondeterministic phase-concurrent implementations (based on linear probing and cuckoo hashing). The experiments also compare the implementations to standard sequential linear probing, and to the sequential history-independent hash table. The experiments show that the deterministic hash table developed in this chapter significantly outperforms the existing concurrent (nondeterministic) versions on updates by a factor of 1.3–4.1. Furthermore, it gets up to a 52$\times$ speedup over the (standard) nondeterministic sequential version on 40 cores with two-way hyper-threading. The experiments compare insertions to simply writing into an array at random locations (a scatter). On 40 cores, and for a load factor of $1/3$, insertions into the deterministic hash table is only about $1.3\times$ the cost of random writes. This is because most insertions only involve a single cache miss, as does a random write, and that is the dominant cost.

Such a deterministic hash table is useful in many applications. For example, Delaunay refinement iteratively adds triangles to a triangulation until all triangles satisfy some criteria (see Section 5.5). "Bad triangles" which do not satisfy the criteria are broken up into smaller triangles, possibly creating new bad triangles. The result of Delaunay refinement depends on the order in which bad triangles are added. Chapter 3 showed that using deterministic reservations, triangles can be added in parallel in a deterministic order on each iteration. However, for the algorithm to be deterministic, the list of new bad triangles returned in each iteration must also be deterministic. Since each bad triangle does not know how many new bad triangles will be created, the most natural and efficient way to accomplish this is to add the bad triangles to a deterministic hash table and return the contents of the table at the end of each iteration. Without a hash table, one would either have to first mark the bad triangles and then look through all the triangles identifying the bad ones, which is inefficient, or use a fetch-and-add to a vector storing bad triangles (nondeterministic), leading to high contention, or possibly use a lock-free queue (nondeterministic), again leading to high contention. By using a deterministic hash table in conjunction with deterministic reservations, the order of the bad triangles is deterministic, giving a deterministic implementation of parallel Delaunay refinement.

This chapter presents six applications which use hash tables in a phase-concurrent manner, and shows that the deterministic phase-concurrent hash table can be used both for efficiency and for determinism. For four of these applications—remove duplicates, Delaunay refinement, suffix trees and edge contraction—we believe the most natural and/or efficient way to write an implementation is to use a hash table. Experiments shows that for these applications, using the deterministic hash table is only slightly slower than using a nondeterministic one based on linear probing, and is faster than using cuckoo hashing or chained hashing (which are also nondeterministic). For two other applications—breadth-first search and spanning tree—this chapter presents simpler implementations using hash tables, compared to array-based versions directly addressing memory. Experiments show that the implementations using hash tables are not much slower than the array-based implementations, and again using our deterministic hash table is only slightly slower than using our nondeterministic linear probing hash table and faster than using the other hash tables.

Contributions. The contributions of this chapter are as follows. First, the notion of phase-concurrency is formalized. Second, this chapter shows that phase-concurrency can be applied to hash tables to obtain both determinism and efficiency. Proofs of correctness and termination of the deterministic phase-concurrent hash table are given. Third, a comprehensive experimental evaluation of our hash tables with the fastest existing parallel hash tables is presented. The experiments compare our deterministic and nondeterministic phase-concurrent linear probing hash tables, our phase-concurrent implementation of cuckoo hashing, hopscotch hashing, which is the fastest existing concurrent open addressing hash table at the time of this work, and an optimized implementation of concurrent chained hashing. Finally, the chapter describes several applications of the deterministic hash table, and presents experimental results comparing the running times of using different hash tables in these applications.

5.2 Related Work

A data structure is defined to be ***history-independent*** if its layout depends only on its current contents, and not the ordering of the operations that created it [217, 344]. For sequential data structures, history-independence is motivated by security concerns, and in particular ensures that examining a structure after its creation does not reveal anything about its history. This chapter extends a sequential history-independent hash table based on open addressing [58] to work phase-concurrently. The motivation is to design a data structure which is deterministic independent of the order of updates. Although this work is not concerned with the exact memory layout, it is important to be able to return the contents of the hash table very quickly and in an order that is independent of when the updates arrived. For a history-independent open addressing table, this can be done easily by packing the non-empty elements into a contiguous array, which just involves a parallel prefix sum and cache-friendly writes.

Several concurrent hash tables have been developed over the years. There has been significant work on concurrent closed addressing hash tables using separate chaining [231, 143, 282, 333, 412, 196, 443, 293, 225, 304]. It would not be hard to make one of these deterministic when reporting the contents of the buckets since each list could be sorted by a priority at that time. However, such hash tables are expensive relative to open address hashing because they involve more cache misses, and also because they need memory management to allocate and de-allocate the cells for the links. The fastest closed addressing hash that we know of is Lea's `ConcurrentHashMap` from the Java Concurrency Package [293], and the experiments in this chapter compare with a `C++` implementation of it, obtained from Herlihy et al. [226].

Martin and Davis [323], Purcell and Harris [386], and Gao et al. [162] describe lock-free hash tables with open addressing. For deletions, Gao et al.'s version marks the locations with a special "deleted" value, commonly known as tombstones, and insertions and finds simply skip over the tombstones (an insertion is not allowed to fill a tombstone). This means that the only way to remove deleted elements is to copy the whole hash table. All of these hash tables are nondeterministic and quite complex. The experiments in this chapter use an implementation of nondeterministic linear probing similar to that of Gao et al. (see Section 5.6).

Herlihy et al. [226] describe and implement an open addressing concurrent hash table called hopscotch hashing, which is based on cuckoo hashing [363] and linear probing. Their hash table guarantees that an element is within K locations of the location it hashed to (where K could be set to the machine word size), so that finds will touch few cache lines. To maintain this property, insertions which find an empty location more than K locations away from the location h that it hashed to will repeatedly displace elements closer to h until it finds an empty slot within K locations of h (or resizes if no empty slot is found). A deletion will recursively bring in elements later in the probe sequence to the empty slot created. Their hash table requires locks and its layout is nondeterministic even if only one type of operation is performed concurrently. Hopscotch hashing is the fastest concurrent hash table available at the time of this work, and is used for comparison in Section 5.6.

Kim and Kim [267] recently present several implementations of parallel hash tables, though our experiments showed that the code developed in this chapter and the hopscotch hashing code of [226] are much faster. Van der Vegt and Laarman describe a concurrent hash table using a variant of linear probing called bidirectional linear probing [452, 453], however it requires a monotonic hash function, which may be too restrictive for many applications. Their hash table is nondeterministic and requires locks. Alcantara et al. describe a parallel hashing algorithm using GPUs [7], which involves a synchronized form of cuckoo hashing, and is nondeterministic because collisions are resolved nondeterministically. Concurrent cuckoo hashing has also been discussed by Fan et al. [145], and very recently by Li et al. [299]. The hash table of Fan et al. supports concurrent access by multiple readers and a single writer, but do not support concurrent writers. Li et al. extends this work by supporting concurrent writers as well. Subsequent to the publication of the results in this chapter [421], Nguyen and Tsigas describe a lock-free implementation of cuckoo hashing [353].

Phase-concurrency has been previously explored in the work on room synchronizations by Blelloch et al. [51]. They describe phase-concurrent implementations of stacks and queues. However, they were concerned only about efficiency, and their data structures are not deterministic even within a single phase.

5.3 Preliminaries

Let us now review the sequential history-independent hash table of Blelloch and Golovin [58]. The algorithm is similar to that of standard linear probing. It assumes a total order on the keys used as priorities. For insertion, the only difference is that if during the probe sequence a key currently in the location has lower priority than the key being inserted, then the two keys are swapped. An insertion probes the exact same number of elements as in standard linear probing. For finds, the only difference is that since the keys are ordered by priority, it means that a find for a key k can stop once it finds a location i with a lower priority key. This means that searching for keys not in the table can actually be faster than in standard linear probing. One common method for handling deletions in linear probing is to simply mark the location as "deleted" (a tombstone), and modify the insert and search accordingly. However, this would not be history-independent. Instead, for deletions in the history-independent hash table, the location where the key is deleted is filled with the next lower priority element in the probe sequence that hashed to or after that location (or the empty element if it is at the end of the probe sequence). This process is done recursively until the element that gets swapped in is the empty element.

This chapter defines phase-concurrency as follows:

Definition 4 (Phase-Concurrency). *A data structure with operations O and operation subsets S is phase-concurrent if $\forall s \in S$, we have $s \subseteq O$ and all operations in s can proceed concurrently and are linearizable.*

5.4 Deterministic Phase-Concurrent Hash Table

The deterministic phase-concurrent hash table developed in this chapter extends the sequential history-independent hash table to allow for concurrent inserts, concurrent deletes, and concurrent finds. The contents can also be extracted (referred to as the ***elements*** operation) easily by simply packing the non-empty cells. Using the notation of Definition 4, the hash table is phase-concurrent with:

- $O = \{$insert, delete, find, elements$\}$, and
- $S = \big\{\{$insert$\}, \{$delete$\}, \{$find, elements$\}\big\}$

The code for insertion, deletion, and find is shown in Figure 5.1, and assumes that the table is not full and that different keys have different priorities (total ordering). For simplicity, the code assumes there is no data associated with the key, although it could easily be modified for key-value pairs. Note that the code works for arbitrary key-value sizes as for structure sizes larger that what a compare-and-swap can operate on, a pointer (which fits in a word) to the structure can be stored in the hash table instead. The code assumes a hash function h that maps keys into the range $[0, \ldots, |M| - 1]$, and that the keys have a total priority ordering that can be compared with the function $<_p$. By convention, assume that the empty element ($\perp$) has lower priority than all other elements. The code uses NEXTINDEX(i) and PREVINDEX(i) to increment and decrement the index modulo the table size. Note that neither INSERT nor DELETE have return values, so the implementation only needs to ensure that a set of inserts (or deletes) are commutative with respect to the resulting configuration of the table.

For a given element v, INSERT loops until it finds a location with $\perp$ (Line 3) or it finds that v is already in the hash table (Line 5), at which point it terminates. If during the insert, it finds a location that stores a lower priority value (Line 8), it attempts to replace the value there with v using a CAS, and if successful the lower priority key is temporarily removed from the table and INSERT is now responsible for inserting the replaced element later in the probe sequence, i.e. the replaced element is set to v (Line 9).

For a given element v, DELETE first finds v or an element after v in the probe sequence at location k (Lines 27–29) since v may either not be in the table or its position has been shifted back due to concurrent deletions. If v is not at location k, then DELETE decrements the location (Lines 30–32) until either v is found (Line 33) or the location becomes less than $h(v)$ (Line 30), in which case v is not in the table. After finding v, DELETE finds the

```
procedure INSERT($v$)
    $i = h(v)$
    while ($v \neq \perp$)
        $c = M[i]$
        if ($c = v$) return
        elseif ($c >_p v$) then
            $i =$ NEXTINDEX($i$)
        elseif (CAS($\&M[i], c, v$)) then
            $v = c$
            $i =$ NEXTINDEX($i$)
procedure FINDREPLACEMENT($i$)
    $j = i$
    do
        $j =$ NEXTINDEX($j$)
        $v = M[j]$
    while ($v \neq \perp$ and $h(v) > i$)
    $k =$ PREVINDEX($j$)
    while ($k > i$)
        $v' = M[k]$
        if ($v' = \perp$ or $h(v') \leq i$) then
            $v = v'$
            $j = k$
        $k =$ PREVINDEX($k$)
    return ($j, v$)
procedure DELETE($v$)
    $i = h(v)$
    $k = i$
    while ($M[k] \neq \perp$ and $v <_p M[k]$)
        $k =$ NEXTINDEX($k$)
    while ($k \geq i$)
        if ($v = \perp$ or $v \neq_p M[k]$)
            $k =$ PREVINDEX($k$)
        else
            $(j, v') =$ FINDREPLACEMENT($k$)
            if (CAS($\&M[k], v, v'$)) then
                if ($v' \neq \perp$) then
                    $v = v'$
                    $k = j$
                    $i = h(v)$
                else return
            else $k =$ PREVINDEX($k$)
procedure FIND($v$)
    $i = h(v)$
    while ($M[i] \neq \perp$ and $v <_p M[i]$)
        $i =$ NEXTINDEX($i$)
    return ($M[i] = v$)
```

Figure 5.1: Pseudocode for the phase-concurrent deterministic hashing with linear probing.

replacement element for v by calling FINDREPLACEMENT (Line 34). FINDREPLACEMENT first increments the location until finding a replacement element that is either $\perp$ or a lower priority element that hashes after v (Lines 13–16). The resulting location will be one past the replacement element, so it is decremented on Line 17. Then because the replacement element could have shifted, it decrements the location until finding the replacement element (Lines 18–23). DELETE then attempts to swap in the replacement element v' on Line 35, and if successful, and $v' \neq \perp$ (Line 36), there is now an additional copy of v' in the table so DELETE is responsible for deleting v' (Lines 37–39). Otherwise, if the CAS was unsuccessful, either v has already been deleted or used as a replacement element so possibly appears at some earlier location. DELETE decrements the location and continues looping (Line 41).

To FIND an element v, the algorithm starts at $h(v)$ and loops upward until finding either an empty location or a location with a key with equal or lower priority (Lines 43–45). Then it returns the result of the comparison of v with that key (Line 46). Since there is a total priority ordering on the keys, $M[i]$ will contain v if and only if v is in the table.

Note that for INSERT, DELETE, and FIND, it is crucial that the hash table is not full, otherwise the operations may not terminate. Throughout the discussion, we assume wraparound with modulo arithmetic. Since the table is not full, every cluster has a beginning, and when comparing the positions of two elements within a cluster, the "higher" position is the one further from the beginning of the cluster in the forward direction with wraparound. The goal is to show that when starting with an empty hash table, the phase-concurrent hash table maintains the following invariant:

Definition 5 (Ordering Invariant). *If a key v hashes to location i and is stored in location j in the hash table, then for all $k, i \leq k < j$ it must be that $M[k] \geq_p v$.*

As long as the keys are totally ordered by their priorities, the ordering invariant guarantees a unique representation for a given set of keys [58]. This invariant was shown to hold in the sequential history-independent hash table [58].

The concurrent versions of insert and delete work similarly to the sequential versions, but need to be careful about concurrent modifications. What this section shows is that the union of the keys being inserted and the current content always equals the union of all initial keys and all insertions that started. A key property to make it work is that since only insertions are occurring, the priority of the keys at a given location can only increase. It should be clear from the implementation that is not safe to run inserts concurrently with finds, since an unrelated key can be temporarily removed and invisible to a find.

The deletion routine is somewhat trickier. It allows for multiple copies of a key to appear in the table during deletions. In fact, with p concurrent threads it is possible that up to $p + 1$ copies of a single key appear in the table at a given time. This might seem counterintuitive since the goal is to delete keys. Recall, however, that when a key v is deleted, a replacement v' needs to be found to fill its slot. When v' is copied into the slot occupied by v, there will temporarily be two copies of v', but the delete operation is now responsible for deleting one of them. The sequential code deletes the second copy, but in the concurrent version since there might be concurrent deletes aimed at the same key, the delete might end up deleting the version it copied into, another thread's copy, or it might end up not finding a copy and quitting. The important invariant is that for a value v the number of copies minus the number of outstanding deletes does not change (when a copy is made, the number of copies is increased but so is the number of outstanding deletes). A key property that makes deletions work is that since only deletions are occurring, the priority of the keys at a given location can only decrease, and hence a key can only move to locations with a lower index.

The rest of this section proves important properties of the hash table. M_v is used to indicate the set of (non-empty) values contained in the hash table, I_v to indicate the set of values in a collection of insertion operations I, and $|M|$ to indicate the size of the table.

Theorem 17. *Starting with a table M that satisfies the ordering invariant and with no operations in progress, after any collection of concurrent insertions I complete (and none are in progress) with $|M_v \cup I_v| < |M|$, M will satisfy the following properties:*

- *M contains the union of the keys initially in the table and all values in I, and*
- *M satisfies the ordering invariant.*

Furthermore, all insertion operations are non-blocking and terminate in a finite number of steps.

Proof. The proof assumes all instructions are linearizable and considers the linearized sequential ordering of operations. A ***step*** is used to refer to a position in this sequential ordering. At a given step, I_v is used to indicate the set of values for which an INSERT has started. Between when an INSERT starts and finishes, it is said to be *active* with some value. At its start, an INSERT(v) is active with the value v, but whenever it performs a successful CAS$(\&M[i], v, c)$ on Line 8, the INSERT becomes active with the value c on the next step (Line 9)—it is now responsible for inserting c instead of v. When it does a successful CAS$(\&M[i], v, \perp)$ an INSERT is no longer active—it will terminate as soon as it gets to the next start of the while loop and do nothing to the shared state in the meantime. An INSERT is also no longer active when it reads a value c on Line 4 that is equal to v—it will terminate on Line 5.

A_v is used to indicate the union of values of all INSERT's that are active. M_v is used to indicate the values contained in M on a given step, and M_s to be the initial values contained in M. We will prove that the following invariants are maintained on every step:

1. $M_v \cup A_v = M_s \cup I_v$, and

2. the table M satisfies the ordering invariant.

Since at the end $A_v = \emptyset$, these invariants imply the two properties of the theorem.

Invariant 1 is true at the start since A_v and I_v are both empty and $M_s = M_v$ by definition. The invariant is maintained since (1) when an INSERT starts, its value is added to both A_v and I_v and therefore the invariant is unchanged, (2) when an INSERT terminates it reads a $M[i] = v$, so a v is removed from A_v but it exists in M_v so the union is unaffected, (3) every CAS with $c = \perp$ removes a v from A_v but inserts it into M_v, maintaining the union, and (4) every CAS with $c \neq \perp$ swaps an element in M_v with an element in A_v, again maintaining the union. In the code, whenever a CAS succeeds, c is placed in the location where v was (by the definition of CAS) and immediately afterward v is set to c (Line 9).

Invariant 2 is true at the start by assumption. The invariant is maintained since whenever a CAS$(\&M[i], v, c)$ succeeds it must be the case after the CAS that (1) all locations from $h(v)$ up to i have equal or higher priority than v, and (2) all keys that hash to or before i but appear after i have lower priority than v. These properties imply that the ordering invariant is maintained. The first case is true since the only time i is incremented for v is when $c = M[i]$ has a equal or higher priority (Lines 6–7) and since the code only swaps higher priority values with lower priority ones ($v >_p c$ for all CAS's), once a cell has an equal or larger priority than v, it always will. Also when the code has a successful CAS, swaps v and c, and increments i, it must be the case that all locations in the probe sequence for the new v and before the new i have priority higher than the new v. This is because it was true before the swap and the only thing changed by the swap was putting the old v into the table, which we know has a higher priority than the new v. The second case of invariant 2 is true since whenever a CAS is performed, the priority of the value at that location only increases.

The termination condition is true since when the hash table of size $|M|$ is not full, an INSERT can call NEXTINDEX at most $|M|$ times before finding an empty location. Therefore for p parallel INSERT's, there can be at most $p|M|$ calls to NEXTINDEX. Furthermore, any CAS failure of an INSERT is associated with a CAS success of another INSERT. A CAS success corresponds to either a call to NEXTINDEX (Line 7) or termination of the insertion. Therefore, for a set of p parallel INSERT's, there can be at most $p - 1$ CAS failures for any one CAS success and call to NEXTINDEX. So after $p^2|M|$ CAS attempts, all INSERT's have terminated. It is non-blocking because an INSERT can only fail on a CAS attempt if another INSERT succeeds and thus makes progress. □

Theorem 18. *Starting with a table M with $|M_v| < |M|$ that satisfies the ordering invariant and with no operations in progress, after any collection of concurrent deletes D complete (and none are in progress), the table will satisfy the following properties:*

- *M contains the difference of the keys initially in the table and all values in D, and*
- *M satisfies the ordering invariant.*

Furthermore, all delete operations are non-blocking and terminate in a finite number of steps.

Proof. Similar to insertions, from when a DELETE starts until it ends, it is active with some value: initially it is active with the v it was called with, and after a successful $\text{CAS}(\&M[k], v, v')$ for $v' \neq \perp$ it becomes active with v' (Lines 35–37). A DELETE finishes on $\text{CAS}(\&M[k], v, \perp)$ or when the condition of the while loop on Line 30 no longer holds (in this case, it finishes because v is not in the table).

During deletions, the table M can contain multiple copies of a key. The definition of the ordering invariant is still valid with multiple copies of a key, and for a fixed multiplicity the layout remains unique. Unlike insertions, analyzing deletions requires keeping track of multiplicities.

The proof uses D_v to indicate the set of values in D, and M_s the initial contents of M. $A(v)$ is used to indicate the number of active DELETE's with value v, and $M(v)$ to indicate the number of copies of v in M. We will prove that the following invariants are maintained at every step:

1. $\forall v \in M_s$, if $v \in M_s \setminus D_v$ then $M(v) - A(v) = 1$, and otherwise $M(v) - A(v) < 1$,
2. the table M satisfies the ordering invariant allowing for repeated keys, and
3. on Line 30, the index k of a DELETE of v must point to or past the last copy of v (the "rightmost" copy with respect to the cluster).

Since at the end $A(v) = 0$ for all v, these invariants prove the properties of the theorem.

Invariant 1 is true at the start since D_v is empty and $\forall v \in M_s$, $A(v) = 0$. To show that the invariant is maintained, consider all events that can change $M(v)$, $A(v)$, or D_v. These are: (1) when a DELETE on v starts, then $A(v)$ is incremented making $M(v) - A(v)$ less than 1 (since it can be at most 1 before the start) and v is added to D_v so v is not in $M_s \setminus D_v$, (2) when a $\text{CAS}(\&M[k], v, \perp)$ succeeds, $A(v)$ and $M(v)$ are both decremented, therefore canceling out, (3) when a $\text{CAS}(\&M[k], v, v')$ for $v' \neq \perp$ succeeds, then by Lines 35–37, $A(v)$ and $M(v)$ are both decremented, canceling out, and $A(v')$ and $M(v')$ are both incremented, again canceling out, and (4) when a DELETE finishes due to the condition not holding on Line 30, the value v cannot be in the table because of invariant 3, so $A(v)$ is decremented, but $M(v) - A(v)$ is less than 1 both before and after since $M(v) = 0$.

Invariant 2 is true at the start by assumption. The only way it could become violated is if as a result of a $\text{CAS}(\&M[k], v, v')$, the value v' falls out of order with respect to values after location j (i.e., there is some key that hashes at or before j, is located after j, and has a higher priority than v'). This cannot happen since the replacement element found is the closest key to j that hashes after j and has lower priority than v. The loop in Lines 13–16 scans upward to find an element that hashes after v in the probe sequence, and the while loop at Lines 18–23 scans downward in case the desired replacement element was shifted down in the meantime by another thread. It is important that this loop runs backwards and is the reason that there are two redundant looking loops, one going up and one going back down.

Invariant 3 is true since the initial find (Lines 27–29) locates an index of an element with priority lower that v, which must be past v, and FINDREPLACEMENT returns an index at or past the replacement v'. k is only decremented on a failed CAS, which in this case means that v can only be at an index lower than k.

To prove termination, let us bound the number of index increments and decrements a single DELETE operation can perform while executing in parallel with other deletes. For a hash table of size $|M|$, the while loop on Lines 30–41 can execute at most $|M|$ times before i changes, and i will only increase since the replacement element must have a higher index than the deleted element. i can increase at most $|M|$ times before $v' = \perp$, so the number of calls to FINDREPLACEMENT is at most $|M|^2$. The number of decrements and assignments to k in the while loop on Lines 30–41 is at most $|M|$ per iteration of the while loop (for a total of $|M|^2$). FINDREPLACEMENT contains a loop incrementing j, which eventually finishes because the condition on Line 16 will be true for a location containing $\perp$, and a loop decrementing j, which eventually finishes due to the condition on Line 18. So the total number of increments and decrements is at most $2|M|$ per call to FINDREPLACEMENT. The initial find on Lines 27–29 involves at most $|M|$ increments. Therefore, a DELETE operation terminates after at most $|M|+|M|^2+2|M|^3$ increments/decrements, independent of the result of the CAS on Line 35. A collection of p DELETE's terminates in at most $p(|M|+|M|^2+2|M|^3)$ increments/decrements. Increments, decrements, and all instructions in between are non-blocking and thus finish in a finite amount of time. Therefore, concurrent deletions are non-blocking. □

Combining. For a deterministic hash table that stores key-value pairs, if there are duplicate keys, the implementation must decide how to combine the values of these keys deterministically. This can be done by passing a commutative combining function that is applied to the values of pairs with equal keys and updating the location (using a double-word CAS) with a pair containing the key with the combined values. The experiments in Section 5.6 use $\min$ or $+$ as the combining function.

Resizing. Using well-known techniques it is relatively easy to extend the hash table with resizing [225]. Here we outline an approach for growing a table based on incrementally copying the old contents to a new table when the load factor in the table is too high. An INSERT can detect that a table is overfull when a probe sequence is too long. In particular, theoretically a probe sequence should not be longer than $k \log n$ with high probability for some constant k that depends on the allowable load factor. Once a process detects that the table is overfull, it allocates a new table of twice the size and (atomically) places a link to the new table accessible to all users. A lock can be used to avoid multiple processes allocating simultaneously. This would mean that an insertion will have to wait between when the lock is taken and the new table is available, but this should be a short time, and only on rare occasions.

Once the link is set, new INSERT's are placed in the new table. Furthermore, as long as the old table is not empty, every INSERT is responsible for copying at least two elements from the old table to the new one. The thread responsible for creating the new table allocates the elements to copy to other threads, and thereafter some form of work-stealing [65] is used to guarantee that a thread has elements to copy when there are still uncopied elements. As long as a constant number of keys are copied for every one that is inserted, the old table will be emptied before the new one is filled. This way only two tables are active at any time. There is an extra cost of indirection on every INSERT since the table has to be checked to find if it has been relocated. However, most of the time this pointer will be in a local cache in shared mode (loaded by any previous table access) and therefore the cost is very cheap. When there are two active tables, FIND's and DELETE's would look in both tables.

5.5 Applications

This section describes applications which use the deterministic hash table. For these applications, using a hash table is either the most natural and/or efficient way to implement an algorithm, or it simplifies the implementation compared to directly addressing the memory locations. The hash table implementation contains a function ELEMENTS() which packs the contents of the table into an array and returns it. It is important that ELEMENTS() is deterministic to guarantee determinism for the algorithms that use it.

Delaunay refinement and breadth-first search use the WRITEMIN function for determinism, which is an instantiation of the priority update operation that will be described in Section 6.2. It takes two arguments–a memory location *loc* and a value *val* and stores *val* at *loc* if and only if *val* is less than the value at *loc*. It returns *true* if it updates the value at *loc* and *false* otherwise.

5.5.1 Remove Duplicates

This is a simple application which can be implemented using a hash table by simply inserting all of the elements into the table and returning the result of ELEMENTS(), as described in Section 3.4.4. For determinism, the sequence returned by ELEMENTS() should contain the elements in the same order every time, which is guaranteed by a deterministic hash table. This is an example of an application where the most natural and efficient implementation uses hashing (one could remove duplicates by sorting and removing consecutive equal-valued elements, but it would be less efficient).

5.5.2 Delaunay Refinement

Recall from Section 2.6.4 that the Delaunay refinement problem takes as input a Delaunay triangulation and an angle α, and adds new points to the triangulation such that no triangle has an angle less than α. A triangle with an angle less than α is referred to as a ***bad triangle***. This section elaborates on the Delaunay refinement implementation used in Section 3.4.4.

Initially all of the bad triangles of the input triangulation are computed and stored into a hash table. On each iteration of Delaunay refinement, the contents of the hash table are obtained via a call to ELEMENTS(). The next step of an iteration follows that of the deterministic reservations-based implementation of Delaunay triangulation described in Section 3.4.4. Using deterministic reservations, the bad triangles mark (using a WRITEMIN with their index in the sequence) all of the triangles that would be affected if they were to be inserted. Bad triangles whose affected triangles all contain their mark are "active" and can proceed to modify the triangulation by adding their center point. This method guarantees there are no conflicts, as any triangle in the triangulation is affected by at most one active bad triangle. During each iteration of the refinement, new triangles with angles less than α are generated and they are inserted into the hash table as they are discovered. This process is repeated until either a specified number of new points are added or the triangulation contains no more bad triangles. For determinism, it is important that the call to ELEMENTS() is deterministic, as this makes the indices/priorities of the bad triangles, and hence the resulting triangulation deterministic.

This is an example of an application where using a hash table significantly simplifies the implementation. Prior to inserting a point, it is hard to efficiently determine how many new bad triangles it will create, and pre-allocate an array of the correct size to allow for storing the new bad triangles in parallel.

5.5.3 Suffix Tree

Recall from Section 2.6.3 that a suffix tree stores all suffixes of a string S in a trie where internal nodes with a single child are contracted. A suffix tree allows for efficient searches for patterns in S, and also has many other applications in string analysis and computational biology. To allow for expected constant time look-ups, a hash table is used to store the children of each internal node. The phase-concurrent hash table allows for parallel insertions of nodes into a suffix tree and parallel searches on the suffix tree. This is an example of an application where hash tables are used for efficiency, and where the inserts and finds are naturally split into two phases. The suffix tree implementation is discussed in more detail in Chapter 11.

5.5.4 Edge Contraction

The ***edge contraction*** problem takes as input a sequence of edges (possibly with weights) and a label array R, which specifies that vertex v should be relabeled with the value $R[v]$. It returns a sequence of unique edges relabeled according to R. Edge contraction is used in recursive graph algorithms where certain vertices are merged into "supervertices" and the endpoints of edges need to be relabeled to the IDs of these supervertices. Duplicate edges are processed differently depending on the algorithm.

To implement edge contraction, the edges are inserted into a hash table using the two new vertex IDs as the key, and any data on the edge as the value. A commutative combining function can be supplied for combining data on duplicate edges. For example, the edge with minimum weight might be kept for a minimum spanning tree algorithm, or the edge weights added together for a graph partitioning algorithm [261]. To obtain the relabeled edges for the next iteration, a call to ELEMENTS() is made. To guarantee determinism in the algorithm, the hash table must be deterministic.

The edge contraction idea described here is used to combine duplicate edges in the parallel graph reordering algorithm described in Chapter 8, and to remove duplicate edges in the contraction-based parallel connected components implementation described in Chapter 9.

5.5.5 Breadth-First Search

Recall that the standard parallel breadth-first search (BFS) implementation proceeds by visiting each frontier of the search in parallel, and generates a BFS tree. This can be made deterministic using a priority write (WRITEMIN), as discussed in Section 3.4.4. The approach discussed in Section 3.4.4, however, requires first creating an array large enough to contain all unvisited neighbors of all vertices in the current frontier (since at this point parents have not been assigned yet), assign segments of the array to each vertex in the frontier, and have each frontier vertex copy unvisited neighbors that it is a parent of into the array. This array is then packed down with a prefix sums and assigned to the next frontier.

An alternative solution is to use a concurrent hash table and insert unvisited neighbors into the table. Obtaining the next frontier simply involves a call to ELEMENTS(). With this method, duplicates are removed automatically, and the packing is hidden from the user. This leads to a much cleaner solution. If one wants to look at or store the frontiers or

```
procedure BFS($G$, $r$)                                          ▷ $r$ is the root
    Parents = $\{\infty, \ldots, \infty\}$             ▷ initialized to all $\infty$ (unvisited)
    Parents$[r] = r$
    Frontier $= \{r\}$
    while (Frontier $\neq \{\}$) do
        Create hash table $T$
        parfor $v \in$ Frontier do                      ▷ loop over frontier vertices
            parfor ngh $\in N(v)$ do                          ▷ loop over neighbors
                if (WRITEMIN(&Parents[ngh], $v$)) then
                    $T$.INSERT(ngh)
        Frontier $= T$.ELEMENTS()                             ▷ get contents of $T$
        parfor $v \in$ Frontier do
            Parents[v] $= -$Parents[v]                  ▷ negative indicates visited
    return Parents
```

Figure 5.2: Hash table-based implementation of breadth-first search.

simply generate a level ordering of the vertices, then it is important that ELEMENTS() is deterministic. The pseudocode for this algorithm is shown in Figure 5.2. This method gives a deterministic BFS tree. Section 5.6 shows that using the deterministic phase-concurrent hash table does not slow down the BFS code by much compared to the best previous deterministic BFS code (from Chapter 3), which uses memory directly as described in the first method above.

5.5.6 Spanning Forest

Recall that a spanning forest algorithm can be implemented using the deterministic reservations approach as described in Section 3.4.4. If the vertex IDs are integers from the range $[0, \ldots, n-1]$, then an array of size n can be used to store the reservations. However, if the IDs are much larger integers or strings, it may be more convenient to use a hash table to perform the reservations to avoid vertex relabeling. Determinism is maintained if the hash table is deterministic. For the reservation phase, edges insert into a hash table each of its vertices (as the key), with value equal to the edge priority. For a deterministic hash table, if duplicate vertices are inserted, the one with the value with the highest priority remains in the hash table. In the commit phase, each edge performs a hash table find on the vertex it inserted and if it contain the edge's priority value, then it proceeds with linking its two components together. The experiments in Section 5.6 show that the implementation of spanning forest using a hash table is only slightly slower than the array-based version from Section 3.4.4.

5.6 Experiments

This section experimentally analyzes the performance of the concurrent deterministic history-independent hash table (***linearHash-D***) on its own, and also when used in the applications described in Section 5.5.

The experiments compare it with two nondeterministic phase-concurrent hash tables that my co-author and I implement, and with the best existing concurrent hash tables that we know of (hopscotchHash and chainedHash). ***linearHash-ND*** is a concurrent version of linear probing that we implement, which places values in the first empty location and hence depends on history (nondeterministic). It is based on the implementation of Gao et al. [162], except that for deletions it shifts elements back instead of using tombstones, and does not support resizing. In linearHash-ND, insertions and finds can proceed concurrently (although they are still separated in the experiments), since inserted elements are not displaced. ***cuckooHash*** is a concurrent version of cuckoo hashing that we implement, which locks two locations for an element insertion, places the element in one of the locations, and recursively inserts any evicted elements. To prevent deadlocks, it acquires the locks in increasing order of location. It is nondeterministic because an element can be placed in either of its two locations based on the order of insertions. For key-value pairs, on encountering duplicate keys linearHash-D uses a priority function [423] on the values to deterministically decide which pair to keep, while the nondeterministic hash tables do not replace on duplicate keys.

hopscotchHash is a fully-concurrent open-addressing hash table by Herlihy et al. [226], which is based on a combination of linear probing and cuckoo hashing. It uses locks on segments of the hash table during insertions and deletions. We noticed that there is a time-stamp field in the code which is not needed if operations of different types are not performed concurrently. We modified the code accordingly and call this phase-concurrent version ***hopscotchHash-PC***. ***chainedHash*** is a widely-used fully-concurrent closed-addressing hash table by Lea [293] which places elements in linked lists. It was originally implemented in Java, but we were able to obtain a `C++` version from the authors of [226]. We also tried the chained hash map (`concurrent_hash_map`) implemented as part of Intel Threading Building Blocks, but found it to be slower than chainedHash. We implement the ELEMENTS() routine for both hopscotch hashing and chained hashing, as the original implementations did not come with this routine. For hopscotch hashing, we simply pack out the empty locations. For chained hashing, we first count the number of elements per bucket by traversing the linked lists, compute each bucket's offset into an array using a parallel prefix sum, and then traverse the linked lists per bucket copying elements into the array (each bucket can proceed in parallel). The original implementation of chainedHash acquires a lock at the beginning of an insertion and deletion. This leads to high lock contention for distributions with many repeated keys. We optimized the chained hash table such that insertion only acquires a lock after an initial find operation does not find the key, and deletion only acquires a lock after an initial find operation successfully finds the key. This contention-reducing version is referred to as ***chainedHash-CR***.

The experiments also include timings for a serial implementation of the history-independent hash table using linear probing (***serialHash-HI***) and a serial implementation using standard linear probing (***serialHash-HD***).

For the applications, the experiments compare their performance using the phase-concurrent hash tables that we implement and the chained hash table.[1] For breadth-first search and spanning tree, the experiments also compare with implementations that directly address memory and show that the additional cost of using hash tables is small.

All of the implementations developed in this chapter use Cilk Plus, and are compiled using g++. The experiments were run on the 40-core Intel machine with two-way hyper-threading, described in Section 2.7. The experiments use six input distributions from the Problem Based Benchmark Suite. ***randomSeq-int*** is a sequence of n random integer keys in the range $[1, \dots, n]$ drawn from a uniform distribution. ***randomSeq-pairInt*** is a sequence of n key-value pairs of random integers in the range $[1, \dots, n]$ drawn from a uniform distribution. ***trigramSeq*** is a sequence of n string keys generated from trigram probabilities of English text (there are many duplicate keys in this input). ***trigramSeq-pairInt*** has the same keys as trigramSeq, but each key maintains a corresponding random integer value. For this input, the key-value pairs are stored as a pointer to a structure with a pointer to a string, and therefore involves an extra level of indirection. ***exptSeq-int*** is a sequence of n random integer keys drawn from an exponential distribution—this input is also used to test high collision rates in the hash table. ***exptSeq-pairInt*** contains keys from the same distribution, but with an additional integer value per key. For all distributions, the input size was set to $n = 10^8$. For the open addressing hash tables, the experiments initialized a table of size 2^{28}.

Figures 5.3(a) and 5.3(b) compare the hash tables for several operations on randomSeq-int and trigramSeq-pairInt, respectively. For **Insert**, a random set of keys from the distribution is inserted starting from an empty table. For **Find Random** and **Delete Random**, n elements are first inserted (not included in the time) and then the operations are performed for a random set of keys from the distribution. **Elements** is the time for returning the contents of the hash table in a packed array. Table 5.1 lists the parallel and serial running times (seconds) for insertions, finds, deletions, and returning the elements for the various hash tables on different input sequences. For **Find** and **Delete**, n elements are first inserted (not

[1]The source code for hopscotch hashing that we obtained online sometimes does not work correctly on our Intel machine (it was originally designed for a Sun UltraSPARC machine), so it is not used it in the applications.

(a) **Insert**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	3.94	–	4.76	–	5.42	–	8.58	–	3.01	–	3.58	–
serialHash-HD	3.89	–	4.43	–	4.99	–	7.71	–	2.91	–	3.04	–
linearHash-D	4.53	0.171	5.45	0.216	5.53	0.115	8.66	0.204	3.08	0.119	3.71	0.141
linearHash-ND	4.52	0.17	4.77	0.213	5.02	0.108	8.2	0.174	2.96	0.109	3.12	0.119
cuckooHash	7.91	0.364	14.0	0.43	8.3	0.177	12.0	0.242	4.7	0.184	7.23	0.208
chainedHash	13.3	0.774	15.3	0.784	9.54	9.78	14.0	18.4	7.9	2.57	8.48	5.25
chainedHash-CR	14.4	0.708	16.8	0.71	9.1	0.324	13.7	0.438	7.19	0.35	7.56	0.401
hopscotchHash	9.19	0.349	9.21	0.363	7.04	1.54	9.63	2.36	6.15	1.97	6.0	2.02
hopscotchHash-PC	9.18	0.345	9.21	0.365	7.03	1.55	9.59	2.45	6.16	1.94	5.99	2.09

(b) **Find Random**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	3.97	–	4.17	–	6.11	–	10.9	–	3.38	–	3.12	–
serialHash-HD	4.03	–	4.36	–	5.95	–	9.42	–	2.77	–	2.91	–
linearHash-D	4.23	0.114	4.19	0.149	6.17	0.12	10.6	0.219	3.16	0.069	3.11	0.07
linearHash-ND	4.02	0.119	4.35	0.144	5.89	0.117	10.1	0.19	2.79	0.067	2.91	0.078
cuckooHash	6.64	0.21	8.13	0.255	7.7	0.174	12.4	0.24	5.1	0.127	6.1	0.14
chainedHash	9.04	0.356	9.06	0.3	9.84	0.247	15.0	0.364	5.0	0.189	6.01	0.17
chainedHash-CR	9.06	0.359	9.05	0.301	9.74	0.245	15.0	0.365	5.9	0.188	5.99	0.168
hopscotchHash	5.2	0.173	5.02	0.169	6.8	0.167	10.2	0.236	3.51	0.094	3.49	0.091
hopscotchHash-PC	4.76	0.151	4.72	0.15	6.84	0.167	9.7	0.241	3.42	0.088	3.43	0.088

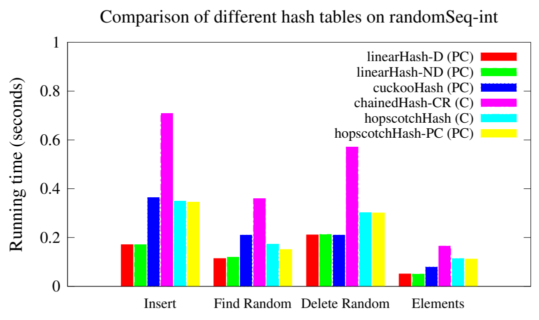
(c) **Find Inserted**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	3.36	–	3.59	–	5.78	–	10.3	–	2.8	–	2.78	–
serialHash-HD	3.22	–	3.45	–	5.6	–	8.66	–	2.48	–	2.62	–
linearHash-D	3.36	0.109	3.6	0.142	5.73	0.114	9.94	0.204	2.6	0.067	2.6	0.068
linearHash-ND	3.22	0.106	3.44	0.125	5.5	0.11	9.55	0.195	2.48	0.064	2.61	0.073
cuckooHash	6.03	0.205	7.34	0.228	7.88	0.165	11.6	0.222	4.66	0.12	5.59	0.13
chainedHash	7.83	0.403	7.91	0.327	9.47	0.253	14.5	0.367	5.68	0.214	5.73	0.191
chainedHash-CR	7.87	0.406	7.89	0.327	9.36	0.249	14.5	0.366	5.69	0.213	5.7	0.188
hopscotchHash	4.67	0.168	4.67	0.166	6.44	0.157	9.31	0.22	3.22	0.09	3.22	0.09
hopscotchHash-PC	4.45	0.154	4.46	0.15	6.48	0.157	9.25	0.24	3.14	0.083	3.16	0.084

(d) **Delete Random**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	4.89	–	5.8	–	3.69	–	4.17	–	2.82	–	3.13	–
serialHash-HD	4.87	–	5.85	–	3.09	–	3.77	–	2.83	–	3.14	–
linearHash-D	5.84	0.211	7.27	0.229	3.79	0.071	4.6	0.109	2.95	0.0968	3.7	0.099
linearHash-ND	5.9	0.213	7.43	0.235	3.85	0.071	4.64	0.109	3.02	0.0936	3.76	0.107
cuckooHash	6.16	0.21	7.16	0.266	5.57	0.15	8.01	0.166	4.25	0.109	4.69	0.142
chainedHash	16.2	0.63	16.4	0.597	4.79	2.38	6.02	2.7	7.16	2.79	7.28	7.01
chainedHash-CR	15.0	0.571	14.9	0.512	4.33	0.11	5.19	0.137	6.04	0.204	6.03	0.358
hopscotchHash	7.19	0.302	7.1	0.316	4.16	1.32	4.89	1.29	4.36	1.32	4.31	1.25
hopscotchHash-PC	7.07	0.301	7.06	0.32	4.15	1.33	4.95	1.34	4.36	1.31	4.28	1.24

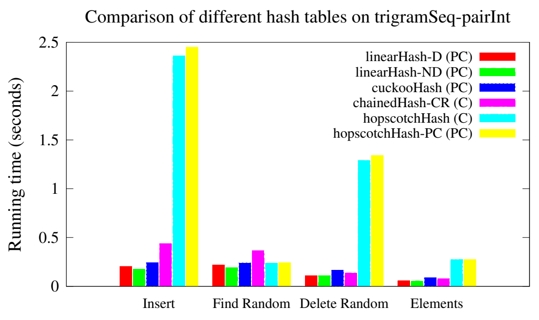
(e) **Delete Inserted**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	5.05	–	6.1	–	3.51	–	4.36	–	3.11	–	3.5	–
serialHash-HD	5.15	–	6.37	–	3.48	–	4.01	–	3.13	–	3.5	–
linearHash-D	6.13	0.24	7.98	0.264	3.73	0.068	4.59	0.102	3.33	0.115	4.18	0.126
linearHash-ND	6.36	0.242	8.38	0.269	3.8	0.07	4.34	0.102	3.35	0.11	4.23	0.119
cuckooHash	6.16	0.217	7.41	0.272	5.74	0.143	7.72	0.16	4.41	0.114	4.99	0.147
chainedHash	15.7	0.737	16.6	0.69	4.22	2.2	5.15	2.65	6.8	2.59	6.92	4.58
chainedHash-CR	14.9	0.714	14.9	0.624	3.77	0.126	4.62	0.153	5.64	0.372	5.65	0.45
hopscotchHash	7.2	0.33	7.8	0.343	3.96	1.32	4.89	1.28	4.69	1.38	4.54	1.29
hopscotchHash-PC	7.06	0.319	7.75	0.347	3.93	1.31	4.85	1.36	4.68	1.38	4.52	1.27

(f) **Elements**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	0.974	–	1.1	–	0.758	–	0.753	–	0.603	–	0.821	–
serialHash-HD	0.986	–	1.08	–	0.759	–	0.761	–	0.554	–	0.814	–
linearHash-D	1.55	0.0511	2.25	0.0875	1.41	0.0575	1.43	0.056	1.05	0.0468	1.7	0.0514
linearHash-ND	1.55	0.0504	2.21	0.0857	1.42	0.0576	1.46	0.0554	1.06	0.0477	1.69	0.0794
cuckooHash	1.91	0.0791	2.54	0.115	2.45	0.0856	2.4	0.0866	1.64	0.0733	2.23	0.101
chainedHash	6.3	0.159	6.47	0.132	1.96	0.0782	1.97	0.0789	3.36	0.0934	3.38	0.0963
chainedHash-CR	6.33	0.165	6.44	0.131	1.97	0.0784	1.96	0.0785	3.38	0.091	3.37	0.0938
hopscotchHash	2.25	0.114	2.7	0.15	2.1	0.228	2.16	0.275	2.14	0.103	2.6	0.127
hopscotchHash-PC	2.26	0.112	2.73	0.147	2.09	0.229	2.16	0.274	2.14	0.1	2.61	0.128

Table 5.1: Times (seconds) for hash table operations with $n = 10^8$. (40h) indicates 40 cores with hyper-threading, and (1) indicates one thread.



(a) Times (seconds) for 10^8 operations on randomSeq-int



(b) Times (seconds) for 10^8 operations on trigramSeq-pairInt

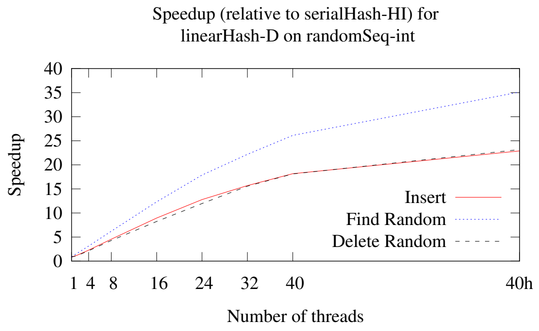
Figure 5.3: Times (seconds) for 10^8 operations for the hash tables on 40 cores (with hyper-threading). (PC) indicates a phase-concurrent implementation and (C) indicates a concurrent implementation.

included in the time) and then operations are performed either on the same keys (**Inserted**) or for a random set of keys from the distribution (**Random**).

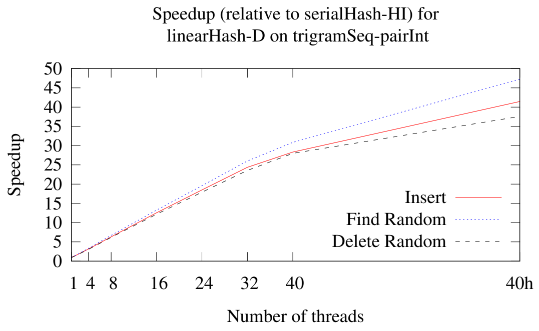
As Figure 5.3 and Table 5.1 indicate, insertion, finds, and deletions into the deterministic (history-independent) hash table are slightly more expensive than into the history-dependent linear probing version. This is due to the overhead of swapping and checking priorities. Elements just involves packing the contents of the hash table into a contiguous array, and since for a given input, the locations occupied in the hash table are the same in the linear probing tables, the times are roughly the same (within noise) between the two serial versions and the two parallel version. On a single thread, the serial versions are cheaper since they do not use a prefix sum.

Overall, linearHash-D and linearHash-ND are faster than cuckooHash, since cuckooHash involves more cache misses on average (it has to check two random locations). Elements is also slower for cuckooHash because each hash table entry includes a lock, which increases the memory footprint. For random integer keys, the linear probing hash tables are 2.3–4.1$\times$ faster than chainedHash and chainedHash-CR, as chained hashing incurs more cache misses. As expected, in parallel chainedHash performs very poorly under the sequences with many duplicates (trigramSeq, trigramSeq-pairInt, exptSeq and exptSeq-pairInt) due to high lock contention, while chainedHash-CR performs better.

Compared to hopscotch hashing, which is the fastest concurrent open addressing hash table that we are aware of, both of our phase-concurrent versions of linear probing are faster. For random integer keys, the deterministic version is about 2$\times$ faster than hopscotch hashing for inserts, and 1.3$\times$ faster for finds and deletes. For elements, the deterministic hash table is also faster because it stores less information per hash table entry. Hopscotch hashing does not get good speedup for insertions and deletions for the sequences with many repeats (i.e., the trigram and exponential sequences) due to lock contention. Compared to



(a) Speedup on randomSeq-int



(b) Speedup on trigramSeq-pairInt

Figure 5.4: Speedup relative to serialHash-HI for linearHash-D versus number of threads. “40h” indicates 80 hyper-threads.

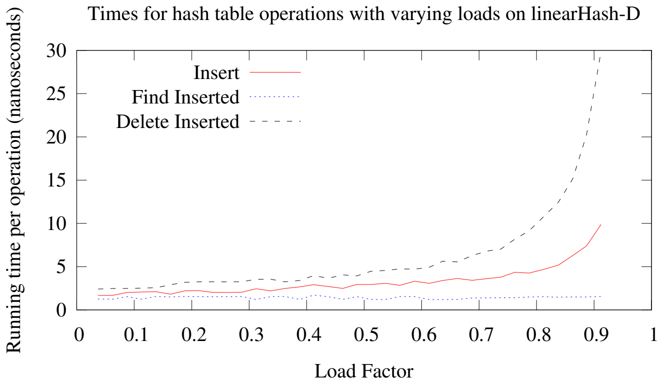


Figure 5.5: Times (nanoseconds) per operation with varying loads for linearHash-D on 40 cores (with hyper-threading). Values on the x-axis indicate the load factor (fraction of the table that is full).

cuckooHash, on the lower-contention random integer sequence hopscotch hashing is faster for finds and inserts but slower for deletes and elements (it stores more data).

Figures 5.4(a) and 5.4(b) show the speedup of linearHash-D relative to serialHash-HI on varying number of threads on randomSeq-int and trigramSeq-pairInt, respectively. The experiments use a hash table of size 2^{28} and apply 10^8 operations of each type. Observe that all of the operations get good speedup as the number of threads increases.

Figure 5.5 shows the per operation running times on linearHash-D with varying loads. For this experiment, a hash table of size 2^{27} was used, and the table is first filled to the specified load before timing the operations. Observe that inserts and deletes become more expensive as load increases, with a rapid increase as the load approaches 1.

The experiments compare the performance of hash table inserts to doing random writes (times for 10^8 writes are shown in Table 5.2). For a uniformly random sequence (randomSeq-int), parallel insertion into the deterministic hash table with a load of $1/3$ is

Memory Operation	(1)	(40h)
Random write	1.62	0.129
Conditional random write	1.82	0.131
Hash table insertion	4.53	0.171

Table 5.2: Times (seconds) for 10^8 random writes (scatter)

1.3× slower than parallel random writes. The experiments also compare with a conditional random write, which only writes to the location if it is empty, and the parallel running time is about the same as for random writes.

Very recently, Li et al. [299] describe a concurrent cuckoo hash table that achieves up to 40 million inserts per second for filling up a hash table to 95% load using 16 cores and integer key-value pairs, where the integers are 8 bytes each. On 16 cores, our linearHash-ND performs 75 million inserts per second and linearHash-D performs 65 million inserts per second filling the table up to 95% load and using integer key-value pairs with 8-byte integers. As the performance of linear probing degrades significantly at high loads, for smaller loads our hash table is faster than theirs by a larger factor. However, the hash table of Li et al. is fully-concurrent, and optimizations can probably be made for a phase-concurrent setting. Subsequent to the publication of the results of this chapter [421], Nguyen and Tsigas describe a lock-free cuckoo hash table that is fully-concurrent [353]. The experiments in their paper are not for a phase-concurrent workload, and we leave a comparison with their hash table on phase-concurrent workloads for future work.

Applications. Experiments were performed to compare implementations of the applications using different versions of the hash tables. For the open addressing hash tables, a larger table size decreases the load and usually leads to faster insertions, deletions and finds, but the algorithms require either returning the elements of the hash table or mapping over the elements, which takes time proportional to the size of the hash table. Due to this trade-off, we chose table sizes that gave the best overall performance per application. For chained hashing, only the times for chainedHash-CR are presented, as we tried both chainedHash and chainedHash-CR and found that the timings were within 5% of each other since the inputs do not exhibit high contention. The experiments on applications did not use hopscotch hashing as the implementation that we obtained did not always work correctly.

The experiments for remove duplicates use the same input distributions as in the previous set of experiments ($n = 10^8$). Removing duplicates involves a phase of insertions, which is more efficient with a larger table in open addressing, and a call to ELEMENTS(), which is more efficient with a smaller table in open addressing. Setting the table size to 2^{27} for the open addressing hash tables gave the best overall performance. The times for using linearHash-D, linearHash-ND, cuckooHash, and chainedHash to remove duplicates on several input distributions are shown in Table 5.3. The results show that our deterministic version of linear probing is 7–23% slower than our nondeterministic version on the key-

Remove Duplicates	randomSeq-int		trigramSeq-pairInt		exptSeq-int	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
linearHash-D	6.36	0.212	10.4	0.242	3.72	0.139
linearHash-ND	6.33	0.212	9.64	0.213	3.63	0.116
cuckooHash	11.0	0.417	12.9	0.3	5.76	0.185
chainedHash-CR	19.9	1.32	15.6	0.586	9.67	0.541

Table 5.3: Times (seconds) for remove duplicates

Delaunay Refinement	2DinCube		2Dkuzmin	
	(1)	(40h)	(1)	(40h)
linearHash-D	1.01	0.033	0.986	0.033
linearHash-ND	0.95	0.031	0.956	0.032
cuckooHash	1.62	0.051	1.56	0.054
chainedHash-CR	1.89	0.079	1.95	0.099

Table 5.4: Times (seconds) for Delaunay refinement

value inputs with many duplicates because the deterministic version may perform a swap on duplicate keys, whereas the nondeterministic version does not. Both linear probing tables outperform the cuckoo and chained hash tables.

The Delaunay refinement experiments use as input the Delaunay triangulation of the 2D-cube and 2D-kuzmin geometry data from the PBBS, each of which contain 5 million points. The times for the hash table portion of one iteration of Delaunay refinement, which involves a call to ELEMENTS() and hash table insertions, are shown in Table 5.4. For the open addressing hash tables, a table size of twice the number of bad triangles rounded up to the nearest power of 2 is used. LinearHash-D performs slightly slower than linearHash-ND, but allows for a deterministic implementation of Delaunay refinement. Both of the linear probing hash tables outperform the cuckoo hash table and chained hash tables for this application.

The experiments for suffix trees use three real-world texts from `http://people.unipmn.it/manzini/lightweight/corpus/`. ***etext99*** (105 MB) and ***rctail96*** (115 MB) are taken from real English texts, and ***sprot34.dat*** (110 MB) is taken from a protein sequence. The experiments measure the times for the portion of the code which inserts the nodes into the suffix tree (represented with a hash table), and also the times for searching one million random strings in the suffix tree (which uses hash table finds). The searches use strings with lengths distributed uniformly between 1 and 50. Half of the search strings are random sub-strings of the text, which should all be found, and the other half are random strings, most of which will not be found. The open addressing hash tables use a size of twice the number of nodes in the suffix tree rounded up to the nearest power of 2. The times are shown in Table 5.5. Again the deterministic linear probing hash table is only slightly slower than the nondeterministic one, and both of them outperform the cuckoo

(a) **Suffix Tree Insert (Size)**	etext99 (105 MB)		rctial96 (115 MB)		sprot34.dat (110 MB)	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
linearHash-D	4.84	0.12	4.96	0.117	4.77	0.115
linearHash-ND	4.6	0.114	4.74	0.112	4.57	0.109
cuckooHash	9.11	0.184	8.85	0.177	8.6	0.172
chainedHash-CR	7.72	0.256	7.65	0.238	7.39	0.235

(b) **Suffix Tree Search**	etext99		rctial96		sprot34.dat	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
linearHash-D	1.08	0.023	0.728	0.015	0.803	0.017
linearHash-ND	1.07	0.023	0.713	0.015	0.787	0.017
cuckooHash	1.22	0.026	0.826	0.017	0.911	0.019
chainedHash-CR	1.35	0.03	0.91	0.02	1.01	0.023

Table 5.5: Times (seconds) for suffix tree operations

hash table and chained hash tables.

The experiments for edge contraction, breadth-first search, and spanning forest use three undirected graphs from the PBBS. ***3D-grid*** is a grid graph in 3-dimensional space where every vertex has six edges, each connecting it to its 2 neighbors in each dimension. It has a total of 10^7 vertices and 3×10^7 edges. ***random*** is a random graph where every vertex has five edges to neighbors chosen randomly. It has a total of 10^7 vertices and 5×10^7 edges. The ***rMat*** graph [87] has a power-law degree distribution. It has a total of 2^{24} vertices and 5×10^7 edges.

The experiments time one round of edge contraction when used as a part of a graph separator program. A maximal matching is first computed on the input graph to generate the vertex relabelings (not timed) and then edges with their relabeled endpoints are inserted into a hash table if the endpoints are different (timed). Duplicate edges between the same vertices after relabeling have their weights added together using a fetch-and-add. Since in linearHash-D, the edges may shift around during insertions, it requires using compare-and-swap on the entire edge. On the other hand, in linearHash-ND, once an element is inserted it no longer moves, so when encountering duplicate edges, it only needs to add the weight of the duplicate edge to the inserted edge and can use the faster `xadd` atomic hardware primitive to do this. The linear probing hash table sizes are set to $4/3$ times the number of edges, rounded up to the nearest power of 2. The times are shown in Table 5.6. The deterministic version of linear probing is about 15% slower than the nondeterministic version, but guarantees a deterministic ordering of the edges and hence a deterministic graph partition when used in a graph partitioning algorithm. Again, both linear probing hash tables outperform cuckoo hashing and chained hashing.

Each iteration of BFS uses a hash table with size equal to the sum of the degrees of the frontier vertices rounded up to the nearest power of 2 for linear probing and twice that size

Edge Contraction	3D-grid		random		rMat	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
linearHash-D	6.03	0.154	10.9	0.265	10.8	0.272
linearHash-ND	5.4	0.136	9.09	0.229	9.18	0.235
cuckooHash	9.31	0.269	16.8	0.447	16.7	0.455
chainedHash-CR	11.6	0.55	20.1	0.907	20.0	0.917

Table 5.6: Times (seconds) for edge contraction

Breadth-First	3D-grid		random		rMat	
Search	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial	2.3	–	2.89	–	3.33	–
array	3.57	0.271	4.89	0.169	6.81	0.225
linearHash-D	3.2	0.367	5.44	0.211	6.25	0.262
linearHash-ND	3.21	0.362	5.43	0.204	6.24	0.256
cuckooHash	4.56	0.454	7.3	0.292	9.1	0.373
chainedHash-CR	5.08	1.14	8.11	0.343	9.78	0.439

Table 5.7: Times (seconds) for breadth-first search

Spanning Forest	3D-grid		random		rMat	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial	1.42	–	1.87	–	2.35	–
array	3.54	0.186	4.68	0.226	6.13	0.289
linearHash-D	4.73	0.212	5.87	0.286	7.31	0.346
linearHash-ND	4.8	0.215	5.86	0.282	7.36	0.344
cuckooHash	5.86	0.251	7.08	0.341	9.08	0.387
chainedHash-CR	6.04	0.408	7.46	0.544	9.73	0.662

Table 5.8: Times (seconds) for spanning forest

for cuckoo hashing. Table 5.7 gives the running times for various BFS implementations where ***serial*** is the serial implementation, and ***array*** is the implementation which uses a temporary array to compute new frontiers as described in Section 5.5. LinearHash-D is slightly slower than linearHash-ND, and both linear probing tables outperform cuckooHash and chainedHash-CR. In parallel, the deterministic hash table-based BFS is 16–35% slower than the array-based BFS. On a single thread, linearHash-D is faster on two of the inputs, however it does not get as good speedup. We observed that in parallel, the linear probing hash table-based BFS implementations spend 70-80% of the time performing hash table insertions, and sequentially they spend 80-90% of the time on insertions.

For spanning forest, the experiments compare versions using hash tables with a serial version and the array-based version from Section 3.4.4. For the versions using open addressing tables, a table of size twice the number of vertices rounded up to the nearest power of 2 was used. The timings are shown in Table 5.8. LinearHash-D and linearHash-ND perform similarly, and they both outperform the cuckoo and chained hash tables. The deterministic hash table-based version is 14–26% slower than the array-based version, but avoids vertex relabeling when the vertex IDs are integers from a large range or are not integers.

For BFS and spanning forest, the experiments show that hash tables can replace directly addressing memory, while incurring only a small performance penalty.

Chapter 6

Priority Updates: A Contention-Reducing Primitive for Deterministic Programming

6.1 Introduction

When programming algorithms and applications on shared memory machines, contention in accessing shared data structures is often a major source of performance problems. The problems can be particularly severe when there is a high degree of sharing of data among threads. With naive data structures the performance issues are typically due to contention over locks. Lock-free data structures alleviate the contention, but such solutions only partially solve issues of contention because even the simplest lock-free shared write access to a single memory location can create severe performance problems. For example, simply having all threads write to a small set of shared locations can lead to orders of magnitude loss in performance relative to writing to distinct locations. The problem is caused by coherence protocols that require each thread to acquire the cache line in exclusive mode to update a location; this cycling of the cache line through the caches incurs significant overhead—far greater than even the cost of having a single thread perform all of the writes. The performance is even worse when using operations such as a compare-and-swap to atomically update shared locations.

To avoid these issues, researchers have suggested a variety of approaches to reduce the cost of memory contention. One approach is to use *contention-aware schedulers* [474, 150] that seek to avoid co-scheduling threads likely to contend for resources. For many algorithms, however, high degrees of sharing cannot be avoided via scheduling choices. A second approach is to use *hardware combining*, in which concurrent associative operations on the same memory location can be "combined" on their way through the memory system [189, 188, 146, 56]. Multiple writes to a location, for example, can be combined by dropping all but one write. No current machines, however, support hardware combining. A third approach is to use *software combining* based on techniques such as combining funnels [414] or diffracting trees [413, 130]. These approaches tend to be complicated and have significant overhead, because a single operation is implemented by multiple accesses that traverse the shared combining structure. In cases where the contending operations are (atomic) updates to a shared data structure, more recent work has shown that having a single combiner thread perform the updates greatly reduces the overheads [223, 149]. This approach, however, does not scale in general. A fourth approach *partitions* the memory among the threads such that each location (more specifically, each cache line) can be written by only a single thread. This avoids the cycling-of-cache-lines problem: Each cache line alternates between the designated writer and a set of parallel readers. Such partitioning, however, severely limits the sorts of algorithms that can be used. Finally, the *test and test-and-set* operation can be used to significantly reduce contention in some settings [400, 328, 330, 329]. While contention can still arise from multiple threads attempting to initially set the location, any subsequent thread will see the location set during its "test" and drop out without performing a test-and-set. This operation has limited applicability, however, so the aim of this chapter is to identify a more generally applicable operation with the same contention-reducing benefits.

Throughout the chapter, the term *sharing* will be used to indicate that a location is shared among many parallel operations, and *contention* to indicate a performance problem due to such sharing.

Priority Update. This chapter studies a generalization of the test-and-set operation, which we call *priority update*. A *priority update* takes as arguments a memory location, a new value, and a $>_p$ function that enforces a partial order over values. The operation atomically compares the new value with the current value in the memory location, and writes the new value only if it has *higher priority* according to $>_p$. At any (quiescent) time a location will contain the highest priority value written to it so far. A test-and-set is a special case of priority update over two values—the location initially holds 0, the new value to be written is 1, and 1 has a higher priority than 0. The priority write operation discussed in Section 3.3 is also a special case of priority update, where the maximum (or minimum) value written has priority. The priority update, however, can also be used when values do not fit in a hardware "word". For example the values could be character strings represented as pointers to the string stored in a memory word, or complex structures where a subfield is compared. The operation is therefore more general than what could be reasonably expected to be implemented in hardware.

This chapter provides evidence that the priority update operation serves as a good

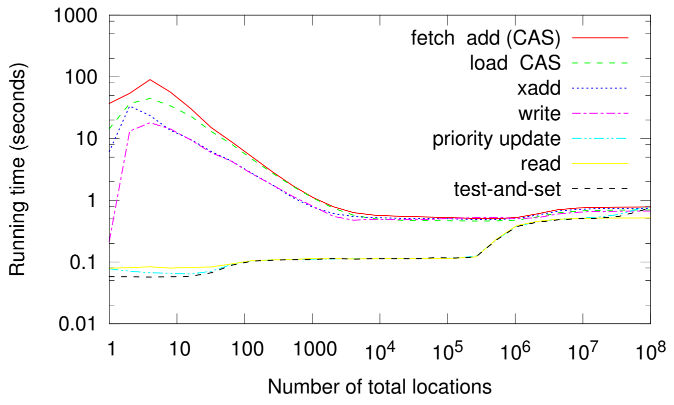


Figure 6.1: Impact of sharing on a variety of operations. Times are for 5 runs of 100 million operations to varying number of memory locations on a 40-core Intel Nehalem with hyper-threading (log-log scale). Since the number of operations is fixed, fewer locations implies more operations sharing those locations.

abstraction for programmers of shared memory machines because it is useful in many applications on shared data (often in a way that is deterministic, guarantees progress, and avoids serial bottlenecks), and when implemented appropriately performs reasonably well under any degree of sharing. This latter point is illustrated in Figure 6.1. Each data point represents the time for 5 runs of 10^8 operations each on a 40-core machine. The x-axis gives the number of distinct locations being operated on—hence the leftmost point is when all operations are on the same location and at the right the graph approaches no sharing. (More details on the setup and further experimental comparisons are described in Section 6.3.1.) As can be seen, when there is a high degree of sharing (e.g., only 8 locations) the read, the test-and-set, and the priority update (with random values) are all over two orders of magnitude faster than the other operations. One would expect the read to do well because the cache lines can be shared. Similarly the test-and-set does well because it can be implemented using a test and test-and-set (as described above) so that under a high degree of sharing only the early operations will attempt to set a location, and the rest will access the already set location in shared mode.

The priority update can be implemented in software with a read, a local comparison, and a compare-and-swap. The compare-and-swap is needed only when the value being written is smaller than the existing value. Thus, when applied with random values (or in a random order) most invocations of priority update only *read* shared data, which is why the running time nearly matches the read curve, and is effectively the same as the test-and-set curve. The curve shows that the high sharing case is actually the best case for a priority update. This implies the user need not worry about contention, although, as with reads, the user might still need to worry about the memory footprint and whether it fits in cache—the steps in the curve arise each time the number of locations no longer fits within a cache at a particular level.

Applications of Priority Updates. Priority updates have many applications. Here we outline several such applications and go into significantly more detail in Sections 6.4 and 6.5. The operation can be used directly within an algorithm to take the minimum or maximum of a set of values, but it also has several other important properties. Due to the fact that the operation is commutative [459, 436] (order does not matter) in the common case when $>_p$ is a total order, it can often be used to avoid nondeterminism when sharing data. By assigning threads unique priorities it can also be used to guarantee (good) progress by making sure at least the highest priority thread for each location succeeds in a protocol.

Priority updates are used in the deterministic reservations framework, introduced in Section 3.4.3, to guarantee the same order of execution of the iterates in a for-loop as the sequential order every time. Furthermore, it guarantees progress since at least the earliest iterate will always succeed (and often many iterates succeed in parallel, if different locations are used). It is also used in BFS, as described in Section 3.4.4.

Priority updates on locations can also be used to efficiently implement a more general dictionary-based priority update where the “locations” are based on keys. Each insert consists of a key-value pair, and updates the data associated with the key if either the key does not appear in the dictionary or the new value has higher priority. The deterministic hash table described in Chapter 5 uses priority updates for determinism.

This chapter describes algorithms for several important problems using priority updates. The chapter studies the performance of several of these algorithms including BFS, Kruskal’s minimum spanning forest algorithm, a maximal matching algorithm, and a dictionary-based remove duplicates algorithm. Timing results for inputs with high sharing are presented, and for BFS and remove duplicates, the experiments compare with versions that use writes instead of priority updates and show that the versions using priority update are significantly faster under high sharing.

Contributions. In summary, the main contributions of this chapter are as follows. First, this chapter generalizes and unifies special cases of priority update operations from the literature, and is the first to call out priority update as a key primitive in ensuring that having many threads updating a few locations does not result in cache/memory system performance problems. Second, the first comprehensive experimental study of priority update versus other widely-used operations under varying degrees of sharing is presented, demonstrating up to orders of magnitude differences on modern multicores from both Intel and AMD. The first analytic justification for priority update’s good performance is also given. Third, several examples of algorithms for a number of important problems that demonstrate a variety of ways to benefit from priority updates are presented. Finally, this chapter presents the first experimental study demonstrating the (good) performance of priority update algorithms on inputs that result in a high degree of write sharing, extending

```
procedure PRIORITYUPDATE($addr$, $newval$, $>_p$)
    $oldval \leftarrow *addr$
    while ($newval >_p oldval$) do
        if CAS($addr$, $oldval$, $newval$) then
            return
        else
            $oldval \leftarrow *addr$
```

Figure 6.2: Priority update implementation.

the experimental studies in Chapters 3–5 by considering a wider range of degrees of sharing, running on more cores, and providing a comparison to implementations using alternative primitives.

6.2 Priority Updates

A ***priority update*** takes as arguments a memory location containing a value of type T, a new value of type T to write, and a binary comparison function $>_p : T \times T \rightarrow bool$ that enforces a partial order over values. The priority update atomically compares the two values and replaces the current value with the new value if the new value has higher priority according to $>_p$. It does not return a value. In the simplest form, called a ***write-with-min*** (or write-with-max), T is a number type, and the comparison function is standard numeric less-than (or greater-than). The implementation in this chapter, however, allows T to be an arbitrary type with an arbitrary comparison function. When $>_p$ defines a total order over T, priority updates commute—i.e., the value ending up in the location will be the same independent of the ordering of the updates.

A priority update can be implemented as shown in Figure 6.2 using a compare-and-swap (CAS). Because CAS (on a single word, or sometimes a double length word) is provided as a hardware atomic on modern machines, no new hardware primitives are required. If the value does not "fit" in a word, one can use a pointer to the actual data being compared (pointers certainly fit in a word), so the implementation can easily be applied to a variety of types (e.g., structures with one of the fields being compared, variable-length character strings with lexicographic comparison, or even more complex structures). One should distinguish the comparison function $>_p$ defining the partial order over the values from the "compare" in compare-and-swap, which is a comparison for equality and is applied to the indirect representation of the value (e.g., the bits in the pointer) and not the abstract type. The object is assumed to not be mutated during the operation so that equality of the indirect representation (pointer) implies equality of the abstract value.

In the best case, the given implementation of priority update completes immediately after a single application of the comparison function, determining that the value already stored in the location has higher priority than the new value. Otherwise an ***update attempt*** occurs with the compare-and-swap operation. (Because the implementation uses CAS to attempt an update, we will also refer to this as a ***CAS attempt***.) If successful, we say that an ***update*** occurs. If not, the priority update retries, completing only when the value currently stored has an equal or higher priority than the new value, or when a successful update occurs.

As noted earlier, a test-and-set is a special case of priority update over two values. A ***write-once*** operation is another special case of a priority update where the contents of a location starts in an "empty" state and once one value is written to the location, making it "full", no future values will overwrite it. As with test-and-set there are just two priorities—empty and full. A third special case is the priority write from the PRAM literature [243]—a synchronous concurrent write from the cores that resolves writes to a common location by taking the value from the highest (or lowest) numbered core. This can be implemented by using pointers to (core number, value) pairs: *addr* contains a pointer to the current pair, *newval* is a pointer to a new pair, and $>_p$ chases the two pointers and compares the core numbers. Note that both test-and-sets and PRAM-style priority writes commute because the values form a total order, but that write-once operations do not because there are many values with equal priority and the first one that arrives is written.

Although the version of priority update described does not return a value, it is easy to extend it to return the old value stored in the location. Indeed in one of the applications in this chapter makes use of this feature.

6.3 Contention in Shared Memory Operations

This chapter distinguishes between sharing and contention. ***Sharing*** means operations that share the same memory location (or possibly some other resource)—for example, a set of instructions reading a single location, and ***contention*** means some form of sequential access to a resource that causes a bottleneck. Contention can be a major source of performance problems on parallel systems while sharing need not be. A key motivation for the priority update operation is to reduce contention under a high degree of sharing.

Although contention can be a problem in any system with sequential access to a shared resource, the problem is amplified for memory updates on cache coherent shared memory machines because of the need to acquire a cache line in exclusive mode. In the widely used MESI (Modified, Exclusive, Shared, Invalid) protocol [367] and its variants, a read can acquire a cache line in shared mode and any number of other caches can simultaneously acquire the line. Concurrent reads to shared locations therefore tend to be reasonably efficient. In fact since most machines support some form of snooping, reading a value that is in another cache can be faster than reading from memory.

On the other hand, in the MESI protocol (and other similar protocols implemented on current multicores) concurrent writes can be very inefficient. In particular, the protocol requires that a cache line be acquired in exclusive mode before making an update to a memory location. This involves invalidating all copies in other caches and waiting for the invalidates to complete. If a set of caches simultaneously make an update request for a location (or even different locations within a line) then the cache line will need to be acquired in exclusive mode by the caches one at a time, doing a dance around the machine. The cost of each acquisition is high because it involves communicating with the cache that has the line in exclusive or modified state, waiting for it to complete its operation, getting a copy of the newly updated line, and updating any tables that keep track of ownership. If the cores make a sequence of requests to a small set of locations then all requests could be rotating through the caches. Because of the cost of the protocol, this can be much more expensive than simply having one core do all the writes. On a system with just 8 cores this can be a serious performance bottleneck, and on one with 40 cores it can be crippling, as the experiments later in this section demonstrate.

If there are a mix of read and write requests to a shared location then the efficiency will fall in between the all-read and all-write cases, depending on the ratio of reads to writes as well as more specifics about how the protocol is implemented. The experiments in this section show that for this case there is actually a significant difference in performance between the protocols implemented on the AMD Opteron and the Intel Nehalem multicores.

This section studies the cost of write sharing among caches (cores) on modern multicores. Along with other operations, we study the cost of a priority update and give both experimental evidence (Section 6.3.1) and theoretical justification (Section 6.3.2) of its efficiency.

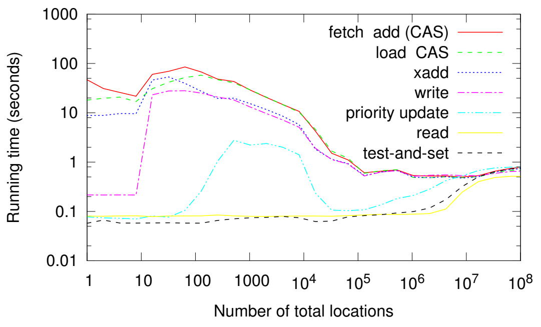
6.3.1 Experimental Measurements of Contention

This section experimentally studies the cost of contention under varying degrees of sharing on two contemporary shared memory multicores (from Intel and AMD) for a variety of memory operations—priority update (using write-with-min), test-and-set, fetch-and-add using CAS, fetch-and-add using the x86 assembly instruction `xadd`, load-and-CAS, (plain) write, and read.[1] The experiments compare the performance of priority update (write-with-min) when values are random versus when values arrive in a decreasing order (the worst case). The experiments also study the performance of priority update where the comparison is on character strings.

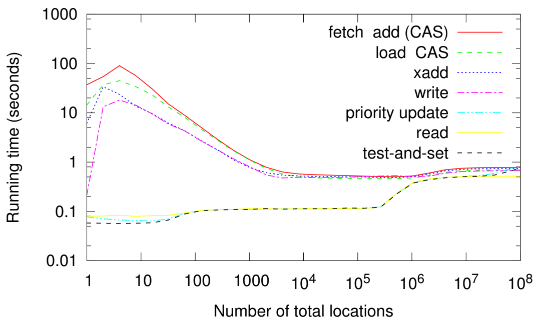
The experiments are performed on the 40-core (with two-way hyper-threading) Intel machine and the 64-core AMD machine described in Section 2.7. The programs were written in Cilk Plus, and compiled with the `icpc` compiler on the Intel machine and the `g++` compiler on the AMD machine.

In the experiments, 10^8 operations are performed on a varying number of random

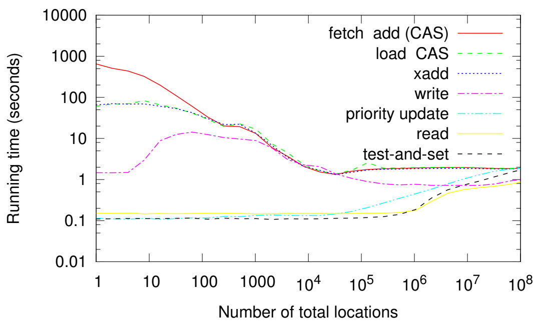
[1]The read includes a write to local memory to get around compiler optimizations.



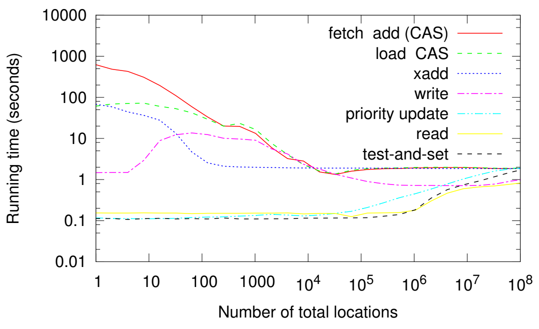
(a) High false sharing on 40-core Intel machine



(b) Low false sharing on 40-core Intel machine



(c) High false sharing on 64-core AMD machine



(d) Low false sharing on 64-core AMD machine

Figure 6.3: Impact of sharing. Times are for 5 runs of 100 million operations to varying number of memory locations on Intel and AMD machines under high and low degrees of false sharing (log-log scale). Since the number of operations is fixed, fewer locations implies more operations sharing those locations.

locations. Two sets of experiments were performed on each machine. The first set of experiments choose the locations randomly in $[0, x)$ where x is the total number of locations written to and locations 0 through x appear contiguously in memory. The second set of experiments choose the locations randomly from $\{h(i) : i \in [0, x)\}$ where $h(i)$ is a hash function that maps i to an integer in $[0, 10^8)$. In the first set of experiments, there will be high false sharing due to concurrent writing to locations on the same cache line. The second set is supposed to represent a more common usage of priority update, which is a set of writes to a potentially large set of locations but for which there is heavy load at a few locations. There is significantly less effect of false sharing in the second set since the heavily loaded locations are unlikely to be on the same cache line.

Figure 6.3(a) shows that with high sharing (low number of total locations) and high false sharing, priority update outperforms plain write, both versions of fetch-and-add, and load-and-CAS by orders of magnitude. Due to an Intel anomaly (described in [423]), there is a spike in the running time for priority update between 256 and 8192 locations, but

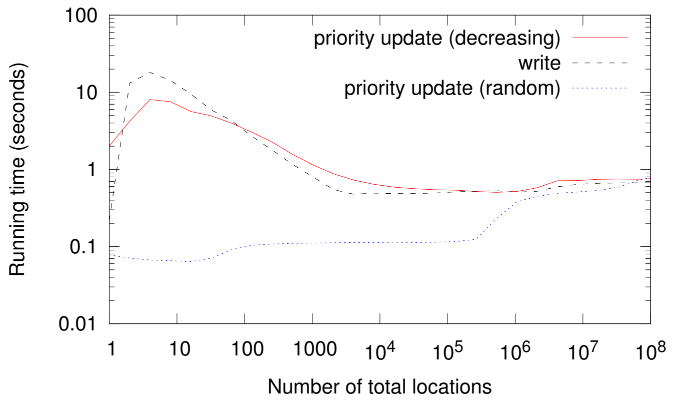


Figure 6.4: Comparing priority update (write-with-min) on random values vs. decreasing values. Times are for 5 runs of 100 million operations to varying number of memory locations with low false sharing on the 40-core Intel machine with hyper-threading (log-log scale).

even with this anomaly, priority update still outperforms plain write, fetch-and-add, and load-and-CAS by an order of magnitude. This anomaly disappears when the false sharing effect is reduced, as shown in Figure 6.3(b). Figure 6.3(b), which is a repeat of Figure 6.1, also shows that the performance of priority update is very close to the performance of both test-and-set and read. For writing to 10^8 locations (the lowest degree of sharing), priority update is slightly slower than fetch-and-add, and test-and-set is slightly slower than write (even though intuitively fetch-and-add does more work than priority update and write does more work than test-and-set). We conjecture this behavior to be due to the branch in both priority update and test-and-set obstructing speculation on the hardware compare-and-swap instruction. Note that `xadd` is consistently faster than implementing a fetch-and-add with a CAS, because the CAS could fail. Also, we noticed that `xadd` performs about the same as a CAS without a load. Preliminary experiments on a 32-core Intel Sandy Bridge machine yielded results that were qualitatively similar to Figures 6.3(a) and 6.3(b).

Figures 6.3(c) and 6.3(d) show the same two experiments on the AMD machine. Note that even with high false sharing, the anomaly for the priority update operation observed for the Intel machine does not appear for the AMD machine. Except for this anomaly, the performance on the Intel machine is better than the performance on the AMD machine.

Note that for priority update, the relative order of values over time greatly impacts the number of update attempts and hence the cost. In the above experiments, the priority update uses random values, which is also the setting that will be studied in the theoretical analysis in Section 6.3.2. The worst case is when the values have increasing priorities over time, as this incurs the most update attempts. With write-with-min, for example, this case arises when values occur in decreasing order. Figure 6.4 shows that the performance of this case (labeled “priority update (decreasing)”) is much worse than the random case.

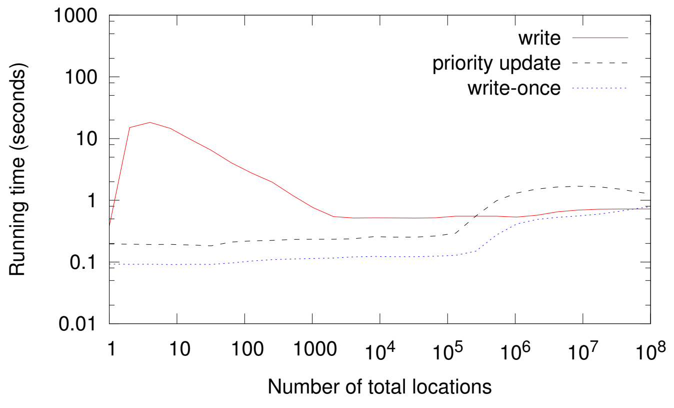


Figure 6.5: Priority update on character strings based on trigram distribution of the English language. Times are for 5 runs of 100 million operations to varying number of memory locations with low false sharing on the 40-core Intel machine with hyper-threading (log-log scale).

Figure 6.5 shows the performance of priority update, where the comparison is on character strings based on the trigram distribution of the English language (the *trigrams* input in Section 6.5). This uses the more general form of priority update as the comparison function requires dereferencing the pointers to the strings. The experiment also compares the performance to using plain writes and write-once to update the values at the shared locations. Note that no pointer dereferencing needs to be done in these versions—plain write just overwrites the pointer at the location, and write-once writes the pointer to the location only if it is empty. Similar to the performance on integer values shown in Figure 6.3, the performance of the version using plain writes is an order of magnitude worse than the priority update and write-once versions. The write-once version is faster than the priority update version, and the gap is more significant here (compared to priority update vs. test-and-set in Figure 6.3) due to the cost of pointer dereferencing in the priority update.

6.3.2 Priority Update Performance Guarantees

As discussed in Section 6.2, the priority update is a further generalization of the test-and-set and write-once operations. Unlike those operations, in a priority update a value can change multiple times instead of just once. However, if the ordering of operations is randomized, then the analysis in this section shows that the number of updates is small, with most invocations only reading the shared data. This section begins with a straightforward analysis of sequential updates and then extends the analysis to a collection of parallel priority updates. There are two main challenges in the parallel analysis: developing a cost model that reasonably captures the read/write asymmetry in the coherence protocol, and coping with the fact that different access delays cause operations to fall out of sync.

This section considers priority update operations where $>_p$ defines a total order over the value domain T. Values can be repeated, so that the number of operations n can be much larger than the number of priorities or the size of T. A collection of priority update operations is said to have ϕ ***occurring priorities*** if the values in those operations fall into exactly ϕ distinct priorities according to $>_p$.

Let us begin with the simplest case of a sequence of priority updates, performed in random order. Here, all update attempts succeed as there are no concurrent CAS operations. This simple lemma shows that the value stored in the location is updated very few times.

Lemma 19. *Consider a random sequential ordering on a collection of priority update operations to a single location, with ϕ occurring priorities. Then H_ϕ updates occur in expectation and $O(\ln \phi)$ updates occur with high probability (in ϕ), where $H_i \approx \ln i$ is the i'th harmonic number.*

Proof. Let S be the subsequence of priority updates that are the first occurrences in the original sequence of a distinct priority—these are the only operations that could possibly perform an update. Let X_k be an indicator for the event that the k'th operation in S performs an update. Then $X_k = 1$ with probability $1/k$, as it updates only if its priority is the highest among the first k operations in S. The expected number of updates is then given by $E[X_1 + \cdots + X_\phi] = E[X_1] + \cdots + E[X_\phi] = \sum_{k=1}^{\phi} 1/k = H_\phi$. Applying a Chernoff bound [341] implies that the probability that more than αH_ϕ updates occur is at most $(e^{\alpha-1}/\alpha^\alpha)^{\ln \phi} < 1/\phi^\alpha$ for a large enough constant α. $\square$

Lemma 19 can be generalized to provide bounds on the running time when performing priority updates in parallel under two models. In either model, assume that if multiple concurrent CAS'es are executing an update attempt, the one that "wins" and successfully updates the value is independent of the data being written. The analysis also assumes that the comparison function $>_p$ takes constant time, although it can be easily extended to non-constant time comparison functions.

Assume that a collection of n priority updates are ordered[2] and have values corresponding to a random permutation of the set $\{1, \ldots, n\}$, with 1 being the highest priority and each location initialized to a special lowest-priority value ∞. This is equivalent to randomly ordering the priority updates and then assigning each value to its relative rank in the total order. While the analysis assumes that the values are distinct, the bounds can be readily sharpened to take into account the actual number of occurring priorities, as in Lemma 19. Note that the actual values of the priority updates do not matter, as long as the order of the priority updates is randomized.

The models in this section are based around a simplified cache-coherence protocol, where a cache line can be in invalid, shared, or exclusive mode. A core performing a CAS

[2]Cores have disjoint subsequences of this ordering, determined at runtime by the scheduler.

requests the relevant cache line in exclusive mode, thereby invalidating the line in all other caches, and performs the CAS.[3] When reading a cache line that is invalid in the local cache, the core first requests the line in shared mode then performs the read. A constant time of c is charged for acquiring the line in either mode, but some acquisitions may serialize due to conflicts depending on which model is adopted.

In the ***fair model***, outstanding cache-line requests to a particular memory location are viewed as ordered in a queue. New requests are added to the end of the queue. When a CAS (exclusive request) is serviced, no other operations may proceed. When a read is processed, all other reads before the next CAS in the queue may be serviced in parallel, and if the cache line is modified, c time is charged for acquiring the line (the first reader puts the line in shared mode).

In the ***adversarial model***, operations are not queued. Instead, an adversary may arbitrarily order any outstanding CAS and read operations (e.g., based on the locations being written), but without considering the values being written.

Bounds for the Fair Model. To analyze priority updates to a single location in the fair model, operations are viewed as being processed in rounds induced by the queue ordering. Each round processes p operations, one per core, which may be either of the two steps of a priority update: a read or a CAS.[4] More precisely, let v_j denote the value stored at the start of round j. For any core performing the read step, the analysis pessimistically assumes that it observes the value v_j. The core then compares its value to v_j, and commits to either performing a CAS in round $j + 1$ or skipping the CAS attempt step and proceeding to the next operation (i.e., issuing another read). Since a CAS in round $j + 1$ is based on the value observed in round j, there is at most 1 successful CAS per round. All reads between consecutive CAS attempts complete in c time, so those reads can be charged against the preceding CAS attempt. The goal is to bound the number of unsuccessful CAS attempts.

Initially, $v_1 = \infty$. Every core issues a read in round 1, compares its value against ∞, and then issues a CAS in round 2 comparing against $v_1 = \infty$. Because the CAS attempts are serialized, the time to complete round 2 is $\Theta(cp)$. Exactly one core (the first one in the queue) succeeds in round 2, so the value v_3 observed at the start of round 3 is one drawn uniformly at random from $\{1, \ldots, n\}$.

Lemma 20. *The expected total time for performing n randomly ordered priority updates to a single location using p cores under the fair model is $O((n/p) + c \ln n + cp)$.*

[3]To clarify, once a core is granted exclusive mode, the model assumes that the CAS completes immediately. A priority update, however, consists of two steps—a read and a CAS—and while the line could be invalidated in between those two steps, the experiments in this chapter on both Intel Nehalem and AMD Opteron multicores support assuming it is not.

[4]Here, the analysis assumes that the type fits in a word. The analysis readily extends to the more general case where $>_p$ must chase pointers.

Proof. By Lemma 19, there are $O(\ln n)$ successful updates, so the goal is to bound the number of unsuccessful CAS attempts. The analysis starts by bounding the number of priority updates that include at least one failed CAS.

An unsuccessful CAS occurs only if a successful CAS is made in the same or preceding round (which is bounded by $O(\ln n)$ in Lemma 19). Define ***phase*** i to be the set of rounds during which (a) the value stored in the location falls between $n/2^{i-1}$ and $n/2^i$ (recall that the values are assumed to be the relative ranks), and (b) a successful CAS occurs. The goal is to bound the number of new priority updates during these rounds that perform a (failed) CAS attempt. First, observe that phase i consists of $O(1)$ rounds in expectation, as each successful update has probability $1/2$ of reducing the value below the threshold of $n/2^i$. Moreover, in each of these rounds, each core has probability at most $1/2^{i-1}$ of performing a priority update of a value below $n/2^{i-1}$. Summing across all cores and all rounds in the phase, the expected number of (failed) priority updates during phase i is at most $O(p/2^{i-1})$. Summing across all phases, the total number of such failed priority updates is $O(p)$.

A failed priority update may retry several times, but a random failed update has probability $1/2$ of retrying through each subsequent phase because the value stored at the location is halved. Thus, there are an expected $O(1)$ retries per priority update that make any CAS attempt. Combining with the above gives a total of $O(p)$ unsuccessful CAS attempts.

Each of the $O(\ln n)$ successful and $O(p)$ unsuccessful CAS'es take c time. As for the reads, any of the reads that must reacquire a cache line (taking c time) can be charged to the preceding CAS attempt, only doubling the time. The first read takes c time, and the remaining reads and all local computation take $O(n/p)$ time, completing the proof. □

The above results are for performing priority updates to a single location. Let us now analyze the time for *multiple locations* where cores apply operations to locations chosen uniformly at random from $\{1, \ldots, m\}$, where m is the number of locations. Let n_i be the number of operations at the i'th location. Here, the analysis assumes that all locations can fit simultaneously in cache and that there are no false-sharing effects. The difficulty here is that the round analysis only applies to each individual location—the model has a separate queue for each location, and simply multiplying the CAS-components of the bound by m is too pessimistic.

Theorem 19. *The expected total time for performing n randomly ordered priority updates to m randomly chosen locations under the fair model is $O((n/p) + cm\ln(n/m) + (cp)^2)$.*

Proof. According to the analysis of Lemma 20, there are at most $O(\ln(n_i) + p)$ CAS attempts when p cores perform $O(n_i)$ updates to location i. Increasing the number of locations only decreases the number of CAS failures, since not all cores choose the same location. So a bound of $O((n/p) + cm\ln(n/m) + cpm)$ follows by maximizing the logarithmic term (setting $n_i = n/m$ for all i) and multiplying by m locations. This bound is pessimistic, so the analysis will improve it for $m > p$. The $O(cm \ln(n/m))$ term seems inherent because each update invalidates the line in all other caches, so the time to reload those lines later is $O(cpm \ln(n/m))$ (which is divided across p cores). The goal is to reduce the $O(cpm)$ term.

Consider the round analysis as in Lemma 20 applied to a single location. The main question is how many (unsuccessful) CAS'es are launched on this location during a round containing a successful CAS. The maximum duration of a round is $O(cp)$ if every core performs a CAS attempt. Each core may thus sample up to $O(cp)$ locations within a round (each sample is independent from the rest), giving a probability of $O(cp/m)$ of choosing this location in any of those attempts. Summing across all cores, the expected number of priority updates to this location per round is $O(cp^2/m)$, only some of which may actually perform a CAS attempt. As in Lemma 20, the likelihood of performing a CAS attempt decreases geometrically per phase, so the total number of failed CAS'es on this location is $O(cp^2/m)$. Summing across all locations gives $O(cp^2)$ failed attempts, each taking c time. $\square$

Bounds for the Adversarial Model. Let us now analyze priority updates under the adversarial model. Recall that in the adversarial model, an adversary may order any outstanding CAS and read operations arbitrarily (e.g., based on the locations being written), but without considering the actual values being written.

Lemma 21. *The total time for performing n randomly ordered priority updates to a single location using p cores under the adversarial model is $O((n/p) + cp \ln n)$ with high probability.*

Proof. By Lemma 19, the number of random updates is $O(\ln n)$ with high probability. The analysis now shows that the number of attempts is at most $O(p \ln n)$, which implies the lemma. A CAS is said to fail due to the i'th update if the old value conditioned on in the CAS is that of the $(i-1)$'st update. There can be at most 1 CAS failure due to the i'th update on each core, as any subsequent priority update on the same core would read the i'th update and hence only fail due to a later update. There can thus be at most $p-1$ CAS failures per update, for a total of $O(p \ln n)$ CAS attempts. The high probability in this lemma is the same as in Lemma 19. $\square$

In the adversarial model, the bound of Lemma 21 generalizes to $O((n/p)+cpm \ln(n/m))$—for n operations the time for reads is still $O(n/p)$; now each location i can take $O(cp \ln(n_i))$ time, leading to a total contribution of $O(\sum_{i=1}^{m} cp \ln(n_i))$ which is maximized when $n_i = n/m$ for all i.

Theorem 20. *The total time for performing n randomly ordered priority updates to m randomly chosen locations under the adversarial model is $O((n/p) + cpm\ln(n/m))$ with high probability.*

For reasonably sized n, the bounds in this section (under both models) are much better than the bounds for operations that always have to access a cache line in exclusive mode. Such operations will run in $O(cn)$ at best assuming either the fair or adversarial model—all accesses will be sequentialized and will involve a cache miss.

6.4 Applications of Priority Update

Priority updates are well-suited to a widely applicable two-phase programming style, which we call *update-and-read* in its general form, and *reserve-and-commit* in a special case. An ***update-and-read*** program alternates two types of phases. During an *update* phase, multiple update attempts occur on some collection of objects, using either a priority update, a plain write, or another write primitive. During the subsequent *read* phase, the value that was successfully recorded is read. Using priority updates or write-once operations during the update phase is desirable to achieve better performance (see Section 6.5). Moreover, the commutative nature of priority updates implies that the values stored at completion of the read phase are deterministic.

When operating on a collection of interacting objects (e.g., vertices of a graph), where each object seeks to update a "neighborhood" of objects, a ***reserve-and-commit*** style is more appropriate. In the *reserve* (update) phase, each object in parallel attempts to reserve the neighborhood of objects that it would read from or write to. In the *commit* (read) phase, each object in parallel checks whether it holds a reservation on its neighborhood, and if so, performs the desired operations. There should be a synchronization point between the reserve and commit phases, guaranteeing that commits and reserves cannot occur concurrently with each other. Since reservations are exclusive (indeed reservations are acting as mutual-exclusion locks), this approach guarantees that each commit behaves atomically. As with the generic update-and-read, the reservations can be implemented using either a priority update, write-once or plain write. The priority update is more desirable both for performance and to guarantee forward progress when multiple objects are reserved. The technique of *deterministic reservations*, described in Section 3.4.3, extends this reserve-and-commit abstraction to an entire parallel loop.

If used correctly and employing a priority update, this reserve-and-commit style can be thought of as a special case of transactional programming, but one in which forward progress guarantees are possible. The reserve phase essentially speculatively attempts a "transaction," and the commit phase commits transactions that do not interfere. By using priority updates, there is a total order over reservations, guaranteeing that at least one reserver (i.e., the one with the highest priority) is able to commit. This forward-progress guarantee does not apply when using a plain write or a write-once, as it is possible that no reserver "wins" on all of its neighbors.

Note that because the highest priority update succeeds for *each* location, priority updates often enable considerable *parallel* progress in each update-and-read phase, yielding good parallel speed-ups (see Section 6.5). For example, with deterministic reservations, often $\Omega(p)$ iterates succeed in parallel.

The remainder of this section describes several algorithms that use priority update, most of which employ some form of update-and-read. An exception is connected components, where a priority update is used to asynchronously update values. The definitions of the problems are described in Section 2.6. In some of these cases (e.g., breadth-first-search and maximal matching), several write primitives maintain correctness of the algorithms and priority updates are just desirable for performance. In others (e.g., connected components, minimum spanning forest, and single-source shortest paths), the priority update is necessary for correctness of the given algorithm.

6.4.1 Breadth-First Search (BFS)

Recall the parallel BFS algorithm discussed in Section 3.4.4 that proceeds in rounds, during which all vertices on the frontier (initialized to contain only the source vertex) attempt to place all of their neighbors on the next frontier. To guarantee that each vertex is added only once, each round is implemented with an update-and-read style. During the update phase, a frontier vertex writes its ID to its neighbors. During the read phase, each frontier vertex checks to see if it successfully reserved its neighbor, and if so it adds the neighbor to the next frontier. Since only one frontier vertex will successfully reserve a neighbor, there will be no duplicates on the next frontier.

This BFS algorithm may be correctly implemented by using priority updates (write-with-min), write-once, or plain writes, with plain writes being less efficient (see Section 6.5) and priority updates guaranteeing a deterministic BFS-tree output (this is the version described in Section 3.4.4).

This chapter also uses a version of deterministic BFS that has only one phase per round and returns the same BFS tree as a sequential implementation. This version uses a priority update on pairs $(index, parent)$, where $index$ is a vertex's parent's order in a sequential BFS traversal, and $parent$ is the vertex's parent's ID. The priority update does a min-comparison only on the $index$ field of the pair. All frontier vertices perform priority updates to neighbors and if it successfully updates the neighbor's location, it adds the neighbor to the next frontier in the same phase. Since this implementation only has a single phase, it allows for duplicate vertices on the frontier (multiple priority updates may succeed on the same neighbor). The form of priority update used here is more general than write-with-min.

6.4.2 Maximal Matching

The maximal matching (MM) problem can be solved with deterministic reservations using a priority update (write-with-min), as discussed in Section 4.10. The algorithm can also be implemented using write-once or plain writes, but forward progress is not guaranteed because it is possible that no edge succeeds in reserving both of its endpoints in an iteration.

6.4.3 Connected Components

A simple vertex-based algorithm for connected components assigns each vertex a unique ID at the start, and in each iteration every vertex sets its ID to the minimum ID of all its neighbors. The algorithm terminates when no vertex's ID changes in an iteration. In each iteration, each vertex performs a priority update (write-with-min) to all of its neighbors' IDs. This is an example of using priority update to guarantee the correctness of an algorithm, and where the priority update yields a remarkably simple solution. The Ligra graph processing framework that will be described in Chapter 7 uses this algorithm.

6.4.4 Minimum Spanning Forest

Most minimum spanning forest (MSF) algorithms begin with an empty spanning forest and grow the spanning forest incrementally by adding "safe" edges (those with minimum weight crossing a cut) [112]. Kruskal's algorithm considers edges in sorted order by weight and iteratively adds edges that connect two different components, using a union-find data structure to query the components. This algorithm can be parallelized by accepting an edge into the MSF if no earlier edge in the sorted order is connected to the same component. As described in Section 3.4.4, this can be implemented using deterministic reservations with a priority update (write-with-min) on the edge weight (and breaking ties by edge ID) if it joins separate components. As with connected components, the priority update is required for correctness here, otherwise the edge added may not be a safe edge. Boruvka's algorithm is similar to Kruskal's except that Kruskal's sorts all edges initially and employs a union-find data structure over connected components, whereas Boruvka's algorithm uses contraction to reduce connected components.

6.4.5 Hash-based Dictionary

By using priority updates to a single location, it is possible to implement a dictionary that supports insertions of $(key, value)$-pairs such that the values of multiple insertions of the same key will be combined with a priority update. This can be thought of as a generalization of priority updates in which the "locations" are not memory addresses or positions in an array, but instead are indexed by arbitrary (hashable) keys. Applications of such key-based

Input	Number of Vertices	Number of Directed Edges	Sharing Level
3D-grid	10^7	6×10^7	Low
random-local	10^7	10^8	Low
rMat	2^{24}	10^8	Medium
4-comb	2.5×10^7	10^8	High
exponential	5×10^6	1.1×10^8	High
4-star	5×10^7	10^8	High

Table 6.1: Inputs for graph applications.

priority updates include making reservations in a dictionary instead of locations in memory as discussed in Section 5.5. Another application is to remove duplicates in a prioritized and/or deterministic way, as discussed in Sections 3.4.4 and 5.5.

6.4.6 Other Applications

Priority updates are applicable to other problems whose solutions are implemented using deterministic reservations (see Chapters 3 and 4). In most of these cases (as with maximal matching), write-once and plain write implementations are correct, but because multiple reservations are required to commit, priority updates are necessary to guarantee forward progress. Moreover, the priority update version guarantees a consistent, deterministic output once the random numbers are fixed. A priority update (write-with-min) can be naturally applied to a single-source shortest paths implementation to asynchronously update potentially shorter paths to vertices (this is the implementation used in Ligra [420], and described in Section 7.4.6). A write-once or plain write implementation would not be correct here, since the shortest path to each vertex must be stored. Priority updates are also useful in other parallel algorithms that, like deterministic reservations, impose a random priority order among elements [63].

6.5 Experiment Study: Applications

The experiments on applications use the Intel Nehalem machine setup described in Section 6.3.1. Sequential programs were compiled using the `g++` compiler with the `-O2` flag. For the breadth-first search, maximal matching, minimum spanning forest, and remove duplicates applications, experiments were run on inputs that exhibit varying degrees of sharing. The experimental setup for each of applications is described in more detail below. All times reported are based on the median of three trials.

The inputs used for the graph algorithms are shown in Table 6.1. Because in the algorithms a vertex can only be simultaneously processed by its neighbors, graphs with low degree overall exhibit low sharing while graphs containing some vertices of high degree can exhibit high sharing (depending on the application). ***3D-grid*** is a grid graph in 3-dimensional space. Every vertex has six edges, each connecting it to its two neighbors

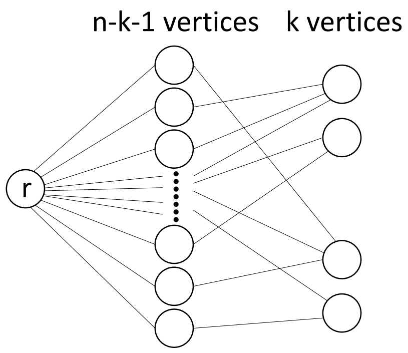


Figure 6.6: k-comb graph (used for BFS experiments to measure varying degrees of sharing).

in each dimension, and thus is a low-sharing graph. ***random-local*** is another low-sharing graph in which every vertex has five undirected edges to neighbors chosen randomly where the probability of an edge between two vertices is inversely correlated with their distance in the vertex array (vertices tend to have edges to other vertices that are close in memory). The ***rMat*** graph is a graph with a power-law distribution of degrees generated using the algorithm described in [87] with parameters $a = 0.5, b = c = 0.1, d = 0.3$. The ***k-comb*** graph is a three layered graph (see Figure 6.6) with the first layer containing only the source vertex r, second layer containing $n - k - 1$ vertices and third layer containing k vertices. The source vertex has an edge to all vertices in the second layer, and each vertex in the second layer has an edge to a randomly chosen vertex in the third layer. There are a total of $4(n-k-1)$ directed edges in this graph. The experiments use varying values of k to model concurrent operations to k random locations. The ***exponential*** graph has an exponential distribution in vertex degrees, and given a degree, incident edges from each vertex are chosen uniformly at random. The ***4-star*** graph is a graph with four "center" vertices and each of the $n - 4$ remaining vertices is connected to a randomly chosen center vertex (total of $2(n - 4)$ directed edges).

In BFS, because many vertices may compete to become the parent of the same neighbor, there can be high sharing. The k-comb graph illustrates this: In the first round the source vertex r explores the $n - k - 1$ vertices in the second level, without sharing; in the second round all of the second level vertices contend on vertices in the third level (see Figure 6.6). The experiments models sharing on k-comb graphs with different k values in order to observe the effect of write sharing that was discussed in Section 6.3. The experiments use four versions of parallel BFS which deal with reserving neighbors and placing them onto the frontier differently. The first version uses a priority update with the minimum function (***priorityUpdate-BFS***) in a two-phase update-and-read style; the second uses a priority update in a single phase, produces the sequential BFS tree but allows for duplicate vertices

Breadth-First Search	3D-grid		random-local		rMat		4-comb		exponential		4-star	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-BFS	2.03	–	2.77	–	3.13	–	0.555	–	1.19	–	0.317	–
priorityUpdate-BFS	4.03	0.307	7.02	0.247	8.37	0.306	1.38	0.08	3.18	0.199	0.885	0.066
seqOrder-BFS	3.12	0.339	5.42	0.258	6.28	0.365	1.54	0.081	3.05	0.285	0.849	0.064
writeOnce-BFS (nd)	2.66	0.25	4.8	0.16	5.45	0.211	1.14	0.066	2.17	0.097	0.664	0.055
write-BFS (nd)	4.3	0.28	6.13	0.246	7.74	0.298	1.2	0.954	3.18	0.224	0.888	0.063

Maximal Matching	3D-grid		random-local		rMat		exponential		4-star	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-Matching	0.527	–	0.764	–	1.0	–	0.674	–	0.823	–
priorityUpdate-Matching	1.41	0.091	1.8	0.113	2.82	0.142	1.27	0.082	0.641	0.062

Minimum Spanning Forest	3D-grid		random-local		rMat		exponential		4-star	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-MSF	5.3	–	7.29	–	9.54	–	7.45	–	13.3	–
priorityUpdate-MSF	10.7	0.455	14.1	0.614	19.0	0.816	12.2	0.53	29.4	1.04

Remove Duplicates Algorithm	allDiff		$\sqrt{n}$-unique		trigrams		allEqual	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-RemDups	3.25	–	0.364	–	0.975	–	0.255	–
priority-UpdateRemDups	3.31	0.078	0.442	0.021	1.07	0.033	0.318	0.02
writeOnce-RemDups (nd)	2.16	0.072	0.433	0.021	1.03	0.035	0.312	0.021
write-RemDups (nd)	3.3	0.083	0.471	0.028	1.05	0.291	0.386	3.19

Table 6.2: Running times (seconds) of algorithms over various inputs. (40h) indicates the running time on 40 cores with hyper-threading and (1) indicates the running time on 1 thread. “40h” corresponds to 80 hyper-threads. (nd) indicates a nondeterministic implementation.

on the frontier (***seqOrder-BFS***); the third uses a test-and-set (***writeOnce-BFS***); and the fourth uses a plain write (***write-BFS***) (see Section 6.4 for details). Figure 6.7 compares the four BFS implementations and the sequential BFS implementation (***serial-BFS***) as a function of number of cores on the 4-comb graph. Table 6.2 shows the running times for each of the BFS implementations on all of the graphs. The (nondeterministic) test-and-set implementation is the fastest because only one actual write is done per vertex. However, the priority update implementations do not do much worse even on the high-sharing comb graph while the plain-write implementation does poorly on it (even worse than serial-BFS). The two-phase and one-phase priority update implementations are comparable in performance. Figure 6.8 shows the 40-core running times of the different BFS implementations using a family of k-comb graphs with varying k. A lower value of k corresponds to higher sharing. Observe that for values of k up to around 10000, priorityUpdate-BFS and seqOrder-BFS outperform write-BFS, by nearly an order of magnitude for small k, and is almost as fast as

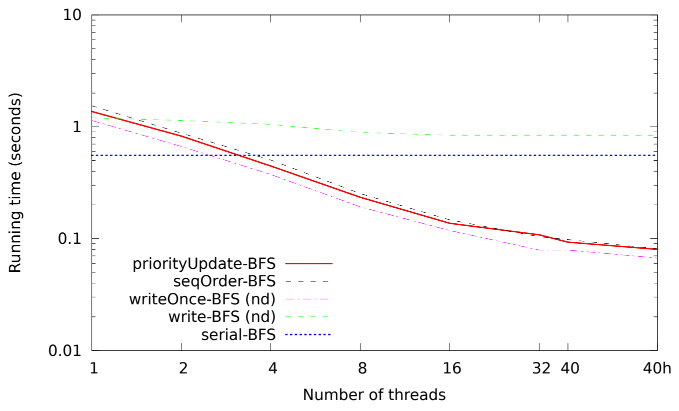


Figure 6.7: BFS times vs. number of cores on the 4-comb graph (log-log scale). (nd) indicates a nondeterministic implementation.

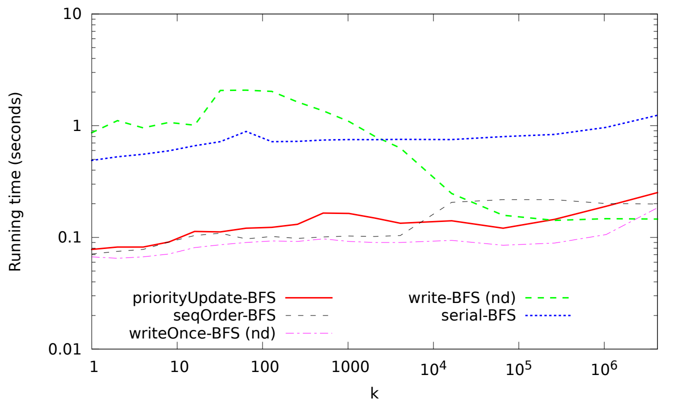


Figure 6.8: BFS times on different k-comb graphs with $n = 2.5 \times 10^7$ on 40 cores with hyper-threading (log-log scale). Lower k means higher sharing. (nd) indicates a nondeterministic implementation.

writeOnce-BFS. For higher values of k where there is little sharing, priorityUpdate-BFS and seqOrder-BFS are slower than writeBFS due to the overhead of the test and compare-and-swap, however they have the benefit of being deterministic. For values of k less than 2000 (high sharing), write-BFS is worse than even the sequential implementation.

For maximal matching and minimum spanning forest, the 4-star and exponential graphs exhibit high sharing. Table 6.2 shows the times for implementations using priority updates and also serial implementations on the various graphs. Observe that even for the high-sharing graphs the implementations performs well (less than 3 times worse than the lower-sharing inputs on 80 hyper-threads).

The input to the remove duplicates problem is a sequence of $(key, value)$ pairs, and the return value is a sequence containing a subset of the input pairs that contains only one

Input	Size	Sharing Level
allDiff	10^7	Low
$\sqrt{n}$-unique	10^7	Medium
trigrams	10^7	Medium
allEqual	10^7	High

Table 6.3: Inputs for remove duplicates.

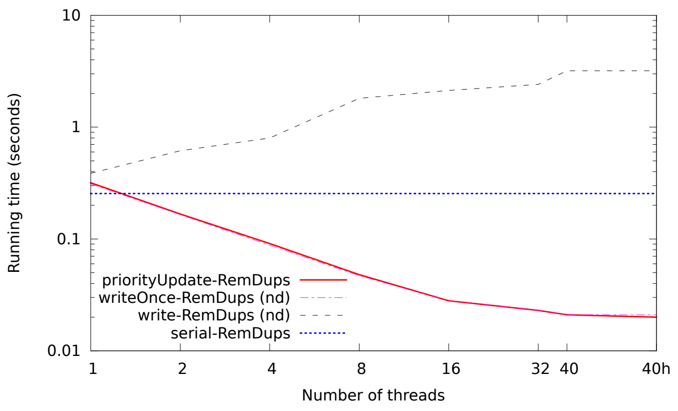


Figure 6.9: Remove duplicates times on the allEqual sequence on 40 cores with hyper-threading (log-log scale). “40h” corresponds to 80 hyper-threads. (nd) indicates a nondeterministic implementation.

element of any given key from the input. The experiments use the hash-based dictionary (and modifications of it) described in Chapter 5. For pairs with equal keys, the pair that is kept is determined based on the $value$ of the keys. The sequence inputs are shown in Table 6.3. The ***allDiff*** sequence contains pairs all with different keys. The $\sqrt{n}$***-unique*** sequence contains $\sqrt{n}$ copies of each of $\sqrt{n}$ unique keys. The ***allEqual*** sequence contains pairs with all the same key. Finally, the ***trigrams*** sequence contains string keys based on the trigram distribution of the English language. The values of the pairs are random integers. The level of sharing at a location in the hash table is a function of the number of equal keys inserted at the location, hence sequences with many equal keys will exhibit high sharing, whereas sequences with few equal keys will have low sharing. Experiments are performed for three versions of the parallel hash table which deal with insertions of duplicate keys differently. The first version, ***write-RemDups***, always performs a write of the value to the location when encountering a key that has already been inserted; the second version, ***writeOnce-RemDups***, does not do anything when encountering an already inserted key (this is the nondeterministic hash table described in Section 5.6); and the last version, ***priorityUpdate-RemDups***, uses a priority update with the minimum function on the values associated with the keys when encountering duplicate keys (this is the deterministic hash table from Chapter 5).

Figure 6.9 compares the performance of the various parallel implementations, along with a serial implementation (***serial-RemDups***) on the sequence of all equal keys, which exhibits the highest sharing. The priority update and write-once implementations scale gracefully with an increasing number of threads, while on a large number of threads, the plain write implementation performs an order of magnitude worse. The priority update and write-once implementations of remove duplicates have similar performance, but the former also has the advantage that it is deterministic. The timings for all of the inputs are shown in Table 6.2.

Part II

Large-Scale Shared-Memory Graph Analytics

Introduction

Chapter 7 introduces Ligra, a lightweight graph processing framework for shared-memory multicore machines, which makes graph traversal algorithms easy to write. The framework has a simple data structure for representing a subset of vertices, and two very simple routines, one for mapping over edges and one for mapping over vertices. The algorithms expressed in Ligra are extremely simple and concise. Furthermore, they get impressive parallel speedups on a modern multicore machine and are significantly more efficient than previously reported results using graph frameworks on machines with many more cores. Ligra is able to process the largest publicly-available real-world graphs on just a single multicore machine. Chapter 8 integrates graph compression techniques into Ligra. The resulting system, called Ligra+, reduces space usage, and surprisingly also improves parallel performance compared to the original Ligra system. Ligra+ increases the sizes of graphs that can be processed for a given memory budget, and also enables even larger graphs to be processed on a single shared-memory machine.

The results in this part of the thesis have appeared in the following publications:

- Julian Shun and Guy Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 135–146, 2013.

- Julian Shun, Laxman Dhulipala and Guy Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. *Proceedings of the IEEE Data Compression Conference (DCC)*, pp. 403–412, 2015.

Chapter 7

Ligra: A Lightweight Graph Processing Framework for Shared Memory

7.1 Introduction

There has been significant recent interest in processing large graphs due to their applicability in studying social networks, the Web graph, networks in biology, and unstructured meshes in scientific simulation. Prior to the work in this thesis, several packages were developed for processing such large graphs on parallel machines including the parallel Boost graph library (PBGL) [197], Pregel [318], Pegasus [250], GraphLab [306, 307], PowerGraph [186], the Knowledge Discovery Toolkit [78, 311], GPS [405], Giraph [176], and Grace [382]. Motivated by the need to process very large graphs, most of these systems (with the exception of the original GraphLab [306] and Grace) have been designed to work on distributed-memory parallel machines.

This chapter studies Ligra, a lightweight interface for graph algorithms that is particularly well-suited for graph traversal problems. Such problems visit possibly small subsets of the vertices on each step. The interface is lightweight in that it supplies only a few functions, the implementation is simple, and it is fast.

Ligra is motivated in part by Beamer et al.'s recent work on a very fast BFS for shared-memory machines [31, 32]. They use a hybrid BFS which uses a sparse representation of the vertices when the frontier is small and a dense representation when it is large. The Ligra interface supports hybrid graph traversal algorithms and for BFS, it achieves close to the same efficiency (time and space) as the optimized BFS of Beamer et al., and the Ligra code is much simpler than theirs. In addition, this chapter applies the interface to many other applications including betweenness centrality, graph eccentricity estimation, graph connectivity, PageRank, and single-source shortest paths.

Ligra is designed for shared-memory machines. As discussed in Chapter 1, compared to distributed-memory systems, communication costs are much cheaper in shared-memory systems, leading to performance benefits. Although shared-memory machines cannot scale to the same size as distributed-memory clusters, current commodity single unit servers can easily fit graphs with well over a hundred billion edges in memory, large enough for any of the graphs reported in the papers mentioned above.[1] Shared-memory along with the existing support for parallel code (e.g., Cilk Plus [294]) on multicores allows for a lightweight implementation. Furthermore, these multicore servers have sufficient memory bandwidth to get quite good speedups over sequential codes (up to 39 fold on 40 cores in our experiments). Shared-memory algorithms tend to be simpler than their distributed counterparts. Unlike in distributed-memory, race conditions can occur in shared-memory, but as this chapter later shows, this can be dealt with in Ligra with appropriate uses of the atomic compare-and-swap instruction. Compared to the distributed-memory systems mentioned above, Ligra is over an order of magnitude faster on a per-core basis for the benchmarks we could compare with, and typically faster even on absolute terms to the largest systems run, which sometimes have two orders of magnitude more cores. Finally, commodity shared-memory servers are quite reliable, often running for up to months or possibly years without a failure.

Ligra supports two data types, one representing a graph $G = (V, E)$ with vertices V and edges E, and another for representing subsets of the vertices V, which is referred to as *vertexSubset*. Other than constructors and size queries, the interface supplies only two functions, one for mapping over vertices (VERTEXMAP) and the other for mapping over edges (EDGEMAP). Since a vertexSubset is a subset of V, the VERTEXMAP can be used to map over any subset of the original vertices, and hence its utility in traversal algorithms—or more generally in any algorithm in which only (possibly small) subsets of the graph are processed on each round. The EDGEMAP also processes a subset of the edges, which is specified using a vertexSubset to indicate the valid sources, and a boolean function to indicate the valid targets of each edge. Abstractly, a vertexSubset is simply a set of integer labels for the included vertices and the VERTEXMAP simply applies the user supplied function to each integer. It is up to the user to maintain any vertex-based data. The implementation switches between a sparse and dense representation of the integers depending on the size of the vertexSubset. In the Ligra interface, multiple vertexSubsets can be maintained and furthermore, a vertexSubset can be used for multiple graphs with different edge sets, as long as the number of vertices in the graphs are the same.

With this interface a breadth-first search (BFS), for example, can be implemented as

[1]The largest graph in the papers cited is a synthetic 127 billion edges in the Pregel paper [318]. The rest of the papers do not use any graphs larger than 20 billion edges. The largest real-world graph described is the Yahoo! Web graph with 6.6 billion directed edges [466].

```
1: Parents = $\{-1, \ldots, -1\}$                    ▷ initialized to all -1's, indicating unexplored
2: procedure UPDATE($s$, $d$)
3:     return (CAS(&Parents[$d$], $-1$ , $s$ ))              ▷ atomically explore vertex
4: procedure COND($i$)
5:     return (Parents[$i$] $==$ $-1$)                        ▷ check if unexplored
6: procedure BFS($G$, $r$)                 ▷ $G$ is the graph and $r$ is the source vertex
7:     Parents[$r$] $= r$
8:     Frontier $= \{r\}$                  ▷ vertexSubset initialized to contain only $r$
9:     while (SIZE(Frontier) $\neq 0$) do
10:        Frontier $=$ EDGEMAP($G$, Frontier, UPDATE, COND)         ▷ visit next frontier
```

Figure 7.1: Pseudocode for breadth-first search in Ligra.

shown in Figure 7.1. This version of BFS uses a Parents array (initialized all to -1, except for the root r where Parents$[r] = r$) in which each vertex will point to its parent in a BFS tree. As with standard parallel versions of BFS [424, 294], on each step i (starting at 0) the algorithm maintains a frontier of all vertices reachable from the root r in i steps. Initially a vertexSubset containing just the root vertex is created to represent the frontier (Line 8). Using EDGEMAP, each step checks the neighbors of the frontier to see which have not been visited, updates those to point to their parent in the frontier, and adds them to the next frontier (Line 10). The user supplied function UPDATE (Lines 2–3) atomically checks to see if a vertex has been visited using a compare and swap (CAS) and returns true if not previously visited (Parents$[i] == -1$). The COND function (Lines 4–5) tells EDGEMAP to consider only target vertices which have not been visited (here, this is not needed for correctness, but is used for efficiency). The EDGEMAP function returns a new vertex set containing the target vertices for which UPDATE returns true, i.e., all the vertices in the next frontier (Line 10). The BFS completes when the frontier is empty and hence no more vertices are reachable.

The interface is designed to allow the edges to be processed in different orders depending on the particular situation. This is different from many of the interfaces mentioned in the first paragraph of this section (e.g. Pregel, GraphLab, GPS, and Giraph) which are vertex-based and have the user hardcode how to loop over the out-edges or in-edges. The Ligra implementation supports a few different ways to traverse the edges. One way is to loop over each vertex in a sparse representation of the active source vertices applying the function to each out-edge (this is basically the order Pregel, GPS, and Giraph supports). This loop over the out-edges can either be parallel or sequential depending on the degree of the vertex (Pregel and the others do not support parallel looping over out-edges, although the most recent version of GraphLab does [186]). A dense representation of the set of source vertices could also be used. Another way to map over the edges is to loop over all destination vertices sequentially or in parallel, and for each in-edge check if the source is in the source vertex set and apply the edge function if so. Finally, a flat map can be simply be applied over all edges, checking which need to be processed.

This chapter applies the Ligra framework to a collection of problems: breadth-first search, betweenness centrality, graph eccentricity estimation, graph-connectivity, PageRank, and Bellman-Ford single-source shortest paths. All of these applications have the property that they work in rounds and each round potentially processes only a subset of the vertices. In the case of BFS, each vertex is only processed once, and in the others they can be processed multiple times. For example, in the shortest paths algorithm a vertex only needs to be added to the active vertex set if its distance has changed. Similarly in a variant of PageRank, a vertex needs to be processed only if its PageRank value has changed by more than some delta since it was last processed.

Betweenness centrality, a technique for measuring the "importance" of vertices in a graph, is basically a version of BFS that accumulates statistics along the way and propagates first in the forward direction and then backward direction. In betweenness centrality, one needs to keep around the frontiers during the forward traversal to facilitate the backward traversal. In Ligra, this is easily done by storing the vertexSubsets in each iteration during the forward traversal. In contrast, this cannot be easily expressed in Pregel and GraphLab, because although vertices can be made inactive in Pregel and GraphLab, the state is associated with the vertices as opposed to being separate.

The contributions of this chapter are as follows:

1. An abstraction based on edgeMaps, vertexMaps and vertexSubsets for programming a class of parallel graph algorithms.

2. An efficient and lightweight implementation of the framework, and applications using the framework.

3. An experimental evaluation of using the framework and timing results of different applications on various input graphs, including the largest publicly-available real-world graph.

7.2 Related Work

7.2.1 Hybrid Breadth-first Search

Beamer et al. [31, 32] recently developed a very fast BFS for shared-memory machines.They use a hybrid BFS consisting of the conventional top-down approach, where each vertex on the current frontier explores all of its neighbors and adds unvisited neighbors to the next frontier (write-based), and a bottom-up approach, where each unvisited vertex in the graph tries to find any parent (visited vertex) among its neighbors (read-based). While the neighbor visits in the top-down approach will mostly be to unvisited vertices when the frontier is small, for large frontiers many of the edges will be to neighbors already visited. The edges to visited neighbors can be avoided in the bottom-up approach because an unvisited vertex can stop checking once it has found a parent; this makes it more efficient than the top-down approach for large frontiers. The disadvantage of the bottom-up approach is that it processes all of the vertices, so is more expensive than the top-down approach for small frontiers. Beamer et al.'s hybrid BFS switches between the two approaches based on the size of the frontier, and the representation of the active set of vertices also switches between sparse and dense accordingly. They show that for small-world and scale-free graphs, the hybrid BFS achieves a significant speedup over previous BFS implementations based on the top-down approach. Ligra uses this same idea in a more general setting.

There has been additional work on hybrid breadth-first search algorithms [467, 468] since the publication of this work [420].

7.2.2 Graph Processing Systems

Pegasus [250] and the Knowledge Discovery Toolbox (KDT) [311, 175] process graphs by using sparse matrix operations with generalized matrix operations. Each row/column corresponds to a vertex and each non-zero in the matrix represents an edge. Pegasus uses the Hadoop implementation of MapReduce in the distributed-computing setting, and includes implementations for PageRank, random walk with restart, graph diameter/eccentricity, and connected components. It does not allow a sparse representation of the vertices and therefore is inefficient when only a small subset of vertices are active. Also, because it is built on top of MapReduce, it is hard to make it perform well. KDT provides a set of generalized matrix-vector building blocks for graph computations. It is built on top of the Combinatorial BLAS [78], a lower-level generalized sparse matrix library for the distributed setting. Using the building blocks, the KDT developers implement algorithms for breadth-first search, betweenness centrality, PageRank, belief propagation, and Markov clustering. Since the abstraction allows for sparse vectors as well as sparse matrices, it is suited for the case when only a small number of vertices are active. However, it does not switch representations of the vertex sets based on its density. Section 7.5 gives some performance comparisons with both systems.

Pregel is an API for processing large graphs in the distributed setting [318]. It is a vertex-centric framework, where vertices can loop over their edges and send messages to all their out-neighbors. These messages are then collected at the target vertex, possibly using associative combining. The system is bulk-synchronous so the received value is not seen until the next round. The reported performance of Pregel is relatively slow, likely due to the overhead of the framework and the use of a distributed memory machine. The GPS [405] and Giraph [176] systems are public source implementations of the Pregel interface with some additional features. The GPS system allows for graph partitioning and reallocation during the computation. This improves performance over Pregel, but only marginally.

GraphLab is a framework for asynchronous parallel graph computations in machine learning. It works in both shared-memory and distributed-memory architectures [306, 307]. It differs from Pregel in that it does not work in bulk-synchronous steps, but rather allows the vertices to be processed asynchronously based on a scheduler. The vertex functions can run at any time as long as specified consistency rules are obeyed. It is therefore well-suited for the machine learning types of applications for which it is defined, where each vertex accumulates information from its neighbors states and updates its state, possibly asynchronously. The recent PowerGraph framework combines the shared-memory and asynchronous properties of GraphLab with the associative combining concept of Pregel [186]. In contrast to Ligra's vertexSubset data type, both Pregel and GraphLab assume a single graph, and do not allow for multiple vertex sets, since state is associated with the vertices.

Grace is a graph management system for shared-memory [382]. It uses graph partitioning techniques and batched updates to exploit locality. Updates to the graph are done transactionally. Their reported times are slower than that of Ligra for applications like BFS and PageRank, after accounting for differences in input size and machine specifications.

GraphChi is a system for handling graph computations using just a PC [289]. It uses a novel parallel sliding windows method for processing graphs from disk. Although their running times are slower than Ligra, their system is designed for processing graphs out of memory, whereas Ligra assumes that the graphs fit in memory.

Galois is a graph system for shared-memory based on set iterators [379]. Unlike Ligra's EDGEMAP and VERTEXMAP functions, their set iterator does not abstract the internal details of the loop from the user. Their sets of active elements for each iteration must be generated directly by the user, unlike our EDGEMAP that generates a vertexSubset which can be used for the next iteration.

Green-Marl is a domain-specific language for writing graph algorithms for shared-memory [229]. Graph traversal algorithms using Green-Marl are written using built-in breadth-first search (BFS) and depth-first search (DFS) primitives whose implementations are built into the compiler. Their language does not support operations over arbitrary sets of vertices on each iteration of the traversal, and instead the user must explicitly filter out the vertices to skip. This makes it less flexible than our framework, which can operate on arbitrary vertexSubsets. In Green-Marl, for traversal algorithms which cannot be expressed using a BFS or DFS (e.g., eccentricity estimation and Bellman-Ford shortest paths), the user has to write the for-loops themselves. On the other hand, such algorithms are naturally expressed in the Ligra framework.

TOTEM [166] is a programming model for designing graph algorithms that run on both CPUs and GPUs. The framework executes algorithms iteratively, where each iteration consists of a computation, communication, and synchronization phase. However, the user has to write the for-loops within each phase, and furthermore has deal with the complexity of GPU programming, which is much more complicated than programming for CPUs.

Other high-performance libraries for parallel graph computations include the Parallel Boost Graph Library (PBGL) [197] and the Multithreaded Graph Library (MTGL) [39]. The former is developed for the distributed-memory setting and the latter is developed for massively multithreaded architectures. These libraries provide few higher-level abstractions beyond the graphs themselves.

Since the publication of Ligra [420], there have been many other graph processing systems developed. Shared-memory multicore systems include X-Stream (an edge-centric system) [399], Prism (a deterministic graph processing framework) [247] and Polymer (a NUMA-aware version of Ligra) [471]. Galois has also been extended to include other programming abstractions [351]. There have also been graph processing systems developed for GPUs [472, 265, 159, 457, 409].

7.3 Framework

The following notation will be used in this chapter. A variable *var* with type *type* is denoted as *var* : *type*. A function f is denoted by $f : X \mapsto Y$ if each $x \in X$ has a unique value $y \in Y$ such that $f(x) = y$. The Cartesian product of sets A and B is denoted by $A \times B$ where $A \times B = \{(a, b) : a \in A \wedge b \in B\}$. The boolean value set *bool* is defined to be the set $\{0, 1\}$ (equivalently $\{$*false*,*true*$\}$). Unweighted graphs have type *graph*, vertices have type *vertex* and edges have type *vertex* $\times$ *vertex*, where the first vertex is the source of the edge and the second the target. For a weighted graph $G = (V, E, w)$, w is a function which maps an edge to a real value (w : *vertex* $\times$ *vertex* $\mapsto \mathbb{R}$).

7.3.1 Interface

For an unweighted graph $G = (V, E)$ or weighted graph $G = (V, E, w)$, Ligra provides a ***vertexSubset*** type, which represents a subset of vertices $U \subseteq V$. Note that V, and hence U, may be shared among graphs with different edge sets. Except for some constructor functions and some optional arguments described in Section 7.3.4, the following describes the entire Ligra interface.

1. SIZE(U: *vertexSubset*) : $\mathbb{N}$.

   Returns $|U|$.

2. EDGEMAP(G : *graph*, U : *vertexSubset*, F : (*vertex* $\times$ *vertex*) $\mapsto$ *bool*, C : *vertex* $\mapsto$ *bool*) : *vertexSubset*.

For an unweighted graph $G = (V, E)$ EDGEMAP applies the function F to all edges with source vertex in U and target vertex satisfying C. More precisely, for an active edge set

$$E_a = \{(u, v) \in E \mid u \in U \wedge C(v) = \textit{true}\},$$

F is applied to each element in E_a, and the return value of EDGEMAP is a vertexSubset:

$$\text{Out} = \{v \mid (u, v) \in E_a \wedge F(u, v) = \textit{true}\}.$$

In this framework, F can run in parallel, so the user must ensure parallel correctness. F is allowed to side effect any data that it is associated with (and does so when used in the graph algorithms we discuss later), so F, C, E_a, and *Out* can depend on order. The function C is useful in algorithms where a value associated with a vertex only needs to be updated once (i.e. breadth-first search). If the user does not need the this functionality, a default function $C_{\textit{true}}$ which always returns true may be supplied.

For weighted graphs, F takes the edge weight as an additional argument.

3. **VERTEXMAP**(U : *vertexSubset*, F : *vertex* $\mapsto$ *bool*) : *vertexSubset*.

   Applies F to every vertex in U. Its returns a vertexSubset:

   $$\text{Out} = \{u \in U \mid F(u) = \textit{true}\}$$

   As with EDGEMAP, the function F can run in parallel.

7.3.2 Implementation

The framework indexes the vertices V of a graph from 0 to $|V| = n - 1$. A vertexSubset $U \subseteq V$ is therefore a set of integers in the range $0, \ldots, n - 1$. In the implementation this set is either represented sparsely as an array of $|U|$ integers (not necessarily sorted) or as a boolean array of length n, *true* in location i if and only if $i \in U$. For example, for a graph with 8 vertices the sparse representation of a vertex subset $\{0, 2, 3\}$ could be $[0, 2, 3]$ or $[3, 0, 2]$ and the corresponding dense representation would be $[1, 0, 1, 1, 0, 0, 0, 0]$. The implementation of vertexSubset contains routines for converting its sparse representation to a dense representation and vice versa. The following pseudocode assumes unweighted graphs, but it can easily be extended to weighted graphs. Also we will overload notation and use U and Out both to denote subsets of vertices and also to denote the vertexSubsets representing them.

For a given graph $G = (V, E)$, a vertexSubset representing a set of vertices $U \subseteq V$ and functions F and C, the EDGEMAP function (pseudocode shown in Figure 7.2) calls

```
procedure EDGEMAP($G$, $U$, $F$, $C$)
    if ($|U|$ + sum of out-degrees of $U >$ threshold) then
        return EDGEMAPDENSE($G$, $U$, $F$, $C$)
    else return EDGEMAPSPARSE($G$, $U$, $F$, $C$)
```

Figure 7.2: Ligra EDGEMAP implementation.

```
procedure EDGEMAPSPARSE($G$, $U$, $F$, $C$)
    Out = {}
    parfor each $v \in U$ do
        parfor ngh $\in N^+(v)$ do
            if ($C$(ngh) == 1 and $F(v, \text{ngh})$ == 1) then
                Add ngh to Out
    Remove duplicates from Out
    return Out
```

Figure 7.3: Ligra EDGEMAPSPARSE implementation.

```
procedure EDGEMAPDENSE($G$, $U$, $F$, $C$)
    Out = {}
    parfor $i \in \{0, \ldots, n-1\}$ do
        if ($C(i)$ == 1) then
            for ngh $\in N^-(i)$ do
                if (ngh $\in U$ and $F(\text{ngh}, i)$ == 1) then
                    Add i to Out
                if ($C(i)$ == 0) then break
    return Out
```

Figure 7.4: Ligra EDGEMAPDENSE implementation.

one of **EDGEMAPSPARSE** (Figure 7.3) and **EDGEMAPDENSE** (Figure 7.4) based on $|U|$ and the number of outgoing edges of U (if this quantity is greater than some threshold, it calls EDGEMAPDENSE, and otherwise it calls EDGEMAPSPARSE). EDGEMAPSPARSE loops through all vertices present in U in parallel and for a given $u \in U$ applies $F(u, \text{ngh})$ to all of u's neighbors ngh in G in parallel. It returns a vertexSubset that is represented sparsely. The work performed by EDGEMAPSPARSE is proportional to $|U|$ plus the sum of the out-degrees of U. On the other hand, EDGEMAPDENSE loops through all vertices in V in parallel and for each vertex $v \in V$ it sequentially applies the function $F(\text{ngh}, v)$ for each of v's neighbors ngh that are in U, until $C(u)$ returns *false*. It returns a dense representation of a vertexSubset. For EDGEMAPSPARSE, since a sparse representation of a vertexSubset is returned, duplicate vertex IDs in the output vertexSubset must be removed. Intuitively EDGEMAPSPARSE should be more efficient than EDGEMAPDENSE for small vertexSubsets, while for larger vertexSubsets EDGEMAPDENSE should be faster. The default threshold of when to use EDGEMAPSPARSE versus EDGEMAPDENSE is set to $m/20$, which was found to work well across all of our applications.

```
1: procedure VERTEXMAP($U$, $F$)
2:     Out = {}
3:     parfor $u \in U$ do
4:         if ($F(u) == 1$) then Add $u$ to Out
5:     return Out
```

Figure 7.5: Ligra VERTEXMAP implementation.

The VERTEXMAP function (Figure 7.5) takes as inputs a vertexSubset representing the vertices U and a Boolean function F, and applies F to all vertices in U. It returns a vertexSubset representing subset Out $\subseteq U$ containing vertices u such that $F(u)$ returns *true*.

7.3.3 Graph Representation

Ligra represents in-edges and out-edges as arrays. In particular, the in-edges for all vertices are kept in one array partitioned by their target vertex and storing the source vertices. Similarly, the out-edges are in an array partitioned by the source vertices and storing the target vertices. Each vertex points to the start of their in-edge and out-edge partitions and also maintains their in-degree and out-degree. Note that EDGEMAPSPARSE only uses the out-edges and EDGEMAPDENSE only uses the in-edges. To transpose a graph (i.e., switch the direction of all edges), which is needed in betweenness centrality, the roles of the in-edges and out-edges are swapped. When a graph is symmetric (or undirected), the in-neighbors and out-neighbors are the same so only one copy needs to be stored. For weighted graphs, the weights are interleaved with the edge targets in the edge array for cache efficiency.

7.3.4 Optimizations

This section describes several optimizations to the interface and implementation. These optimizations affect only performance and not correctness.

Note that EDGEMAPSPARSE applies F in parallel to target vertices (second argument), while EDGEMAPDENSE applies F sequentially given a target vertex. Therefore the F in EDGEMAPDENSE does not need to be atomic with respect to the target vertex. An optimization is for EDGEMAP to accept two version of its function F, the first of which must be correct when run in parallel with respect to both arguments, and the second of which must be correct when run in parallel only with respect to the first argument (source vertex). Both functions should behave exactly the same if EDGEMAP were run sequentially. If this optimization is used, then EDGEMAPSPARSE uses the first version of F as before, but EDGEMAPDENSE uses the second version of F (which we found to be slightly faster for some applications).

The default threshold of when to use EDGEMAPSPARSE versus EDGEMAPDENSE is

```
1: procedure EDGEMAPDENSE-WRITE($G$, $U$, $F$, $C$)
2:     Out = {}
3:     parfor $i \in \{0, \ldots, n-1\}$ do
4:         if ($i \in U$) then
5:             parfor ngh $\in N^+(i)$ do
6:                 if ($C(\text{ngh}) == 1$ and $F(i, \text{ngh}) == 1$) then
7:                     Add ngh to Out
8:     return Out
```

Figure 7.6: Ligra EDGEMAPDENSE-WRITE implementation.

$m/20$, but if the user discovers a better threshold, it can be passed as an optional argument to EDGEMAP.

If the user is careful in defining the F and C functions passed to EDGEMAP to guarantee that no duplicate vertices will appear in the output vertexSubset of EDGEMAP, then the remove-duplicates stage of EDGEMAPSPARSE can be bypassed. The EDGEMAP function takes a flag indicating whether duplicate vertices need to be removed.

For EDGEMAPDENSE, the inner for-loop is sequential (see Figure 7.4) because the behavior of C may allow it to break early (e.g., in BFS, breaking after the first valid parent is found). If instead the user wants to run the inner for-loop in parallel and give up the option of breaking early, a flag can be passed to EDGEMAP to indicate this.

Since EDGEMAPDENSE is read-based, Ligra also provides a write-based version of EDGEMAPDENSE called **EDGEMAPDENSE-WRITE** (shown in Figure 7.6). This write-based version loops through all vertices in V in parallel and for vertices contained in U it applies F (now required to correct when run in parallel with respect to both arguments) to all of its neighbors in parallel, as in EDGEMAPSPARSE. It returns a dense representation of a vertexSubset. We experimentally found EDGEMAPDENSE-WRITE to be more efficient than EDGEMAPDENSE only for two of the applications—PageRank and Bellman-Ford shortest paths. In the framework, the user may pass a flag to EDGEMAP specifying whether to use EDGEMAPDENSE (default) or EDGEMAPDENSE-WRITE when the vertexSubset is dense. The user would need to figure out experimentally which version is more efficient.

For VERTEXMAP, if the user knows that the input and output vertexSubsets are the same, an optimized version of VERTEXMAP that avoids creating a new vertexSubset can be used.

7.4 Applications

This section describes six applications of the Ligra framework. In the following discussions, the "frontiers" of the algorithms are represented as vertexSubsets.

7.4.1 Breadth-First Search

A simple parallel algorithm processes each level of the BFS in parallel. The number of iterations required is equal to the (unweighted) distance of the furthest vertex reachable from the starting vertex, and the algorithm processes each edge at most once. In Ligra, a breadth-first search implementation is very simple as described in Section 7.1. To make the computation more efficient for dense frontiers for which EDGEMAPDENSE is used, one can also provide a version of UPDATE, which is not atomic with respect to d and does not use a CAS. The code for BFS is shown in Figure 7.1.

7.4.2 Betweenness Centrality

Centrality indices for graphs have been widely studied in social network analysis because they are useful indicators of the relative importance of vertices in a graph. One such index is the betweenness centrality index [156].

To precisely define the betweenness centrality index, let us first introduce some additional definitions. For a graph $G = (V, E)$ and some $s, t \in V$, let σ_{st} be the number of shortest paths from s to t in G. For vertices $s, t, v \in V$, define $\sigma_{st}(v)$ to be the number of shortest paths from s to t that pass through v. Define $\delta_{st}(v) = \sigma_{st}(v)/\sigma_{st}$ to be the ***pair-dependency*** of s, t on v. The ***betweenness centrality*** of a vertex v, denoted by $C_B(v)$ is equal to $\sum_{s \neq v \neq t \in V} \delta_{st}(v)$. A naive method to compute the betweenness centrality scores is to perform a BFS starting at each vertex to compute the pair-dependencies, and then sum the pair-dependencies for each $v \in V$. There are $O(n^2)$ pair-dependency terms associated with each vertex, hence this method requires $O(n^3)$ work.

Brandes [72] presents an algorithm which avoids the explicit summation of pair-dependencies and runs in $(nm + n^2 \log n)$ work for weighted graphs and $O(nm + n^2)$ work for unweighted graphs. Brandes defines the ***dependency*** of a vertex r on a vertex v as follows:

$$\delta_{r\bullet}(v) = \sum_{t \in V} \delta_{rt}(v) \tag{7.1}$$

For any given r, Brandes' algorithm computes $\delta_{r\bullet}(v)$ for all v in linear work for unweighted graphs, by using the following two equations, where $P_r(v)$ is defined to contain all immediate parents of v in the BFS tree rooted at r:

$$\sigma_{rv} = \sum_{u \in P_r(v)} \sigma_{ru} \tag{7.2}$$

$$\delta_{r\bullet}(v) = \sum_{w : v \in P_r(w)} \frac{\sigma_{rv}}{\sigma_{rw}} \times (1 + \delta_{r\bullet}(w)) \tag{7.3}$$

The algorithm works in two phases: the first phase of the algorithm computes the number of shortest paths from r to each vertex using Equation 7.2, and the second phase computes the dependency scores via Equation 7.3. The first phase is similar to a forward BFS from vertex r and the second phase works backwards from the last frontier of the BFS. This algorithm can be parallelized in two way—(1) for each vertex, the traversal can be done in parallel, and (2) each vertex can perform their individual computations independently in parallel with other vertices' computations. Although much more efficient than the naive algorithm, Brandes' algorithm still requires at least quadratic time, and is thus prohibitive for large graphs. To address this problem, there has been work on computing approximate betweenness centrality scores based on using the pair-dependency contributions from just a sample of the vertices of the vertices and scaling the betweenness centrality scores appropriately [23, 164]. The KDT package provides a parallel implementation of batched computation of betweenness centrality scores by running multiple individual computations independently in parallel [311].

This section describes the Ligra implementation of betweenness centrality computation from a single root vertex—these computations can be run independently in parallel for any sample of the vertices. The computation here is different from the BFS described in Section 7.4.1 in that instead of finding a parent, each vertex v needs to maintain a count of the number of shortest paths passing through it. This means the number of updates to v is equal to its number of parents in the BFS tree, instead of just one update as in BFS.

The psuedocode for the Ligra implementation is shown in Algorithm 6. The frontier is initialized to contain just r. For the first phase, the code uses an array of integers *NumPaths*, which is initialized to all 0's except for the root vertex which has NumPaths$[r]$ set to 1. By traversing the graph in a breadth-first manner and updating the NumPaths value for each v that is traversed, this gives the number of shortest paths passing through each v from r (NumPaths$[v]$ will remain 0 if v is unreachable from r). The PATHSUPDATE function passed to EDGEMAP is shown in Lines 11–16. As there can be multiple updates to some NumPaths$[v]$ in parallel, the update attempt is repeated with a compare-and-swap until successful. Line 18 guarantees that a vertex is placed on the frontier only once, since the old NumPaths value will be 0 for at most one update. Each frontier of the search is stored in a *Levels* array for use in the second phase.

To keep track of vertices that have been visited (and avoid having to remove duplicates in EDGEMAPSPARSE), the code also maintain a boolean array *Visited*. Visited is initialized to all 0's (except for the root vertex whose entry is set to 1), and a vertex's entry in Visited is set to 1 after it is first visited in the computation. To do this, a VERTEXMAP is used with the VISIT function shown in Lines 8–10 of Algorithm 6 to VERTEXMAP. The COND function in Lines 23–24 makes EDGEMAP only consider unvisited target vertices. The psuedocode for the first phase starting at a root vertex is shown in Lines 27–31.

For the second phase, a new array *Dependencies* (initialized to all 0.0) is used and the Visited array (reinitialized to all 0) is reused. Also the graph is transposed (Line 34),

Algorithm 6 Betweenness Centrality

```
1: NumPaths = $\{0, \dots, 0\}$                          ▷ initialized to all 0
2: Visited = $\{0, \dots, 0\}$                           ▷ initialized to all 0
3: NumPaths[$r$] = 1
4: Visited[$r$] = 1
5: currLevel = 0
6: Levels = [ ]
7: Dependencies = $\{0.0, \dots, 0.0\}$                  ▷ initialized to all 0.0
8: procedure VISIT($i$)
9:     Visited[$i$] = 1
10:    return 1
11: procedure PATHSUPDATE($s$, $d$)
12:    repeat
13:        oldV = NumPaths[$d$]
14:        newV = oldV + NumPaths[$s$]
15:    until (CAS(&NumPaths[$d$], oldV, newV) == 1)
16:    return (oldV == 0)
17: procedure DEPUPDATE($s$, $d$)
18:    repeat
19:        oldV = Dependencies[$d$]
20:        newV = oldV + $\frac{\text{NumPaths}[d]}{\text{NumPaths}[s]} \times (1 + \text{Dependencies}[s])$
21:    until (CAS(&Dependencies[$d$], oldV, newV) == 1)
22:    return (oldV == 0.0)
23: procedure COND($i$)
24:    return (Visited[$i$] == 0)
25: procedure BC($G$, $r$)
26:    Frontier = $\{r\}$                    ▷ vertexSubset initialized to contain only $r$
27:    while (SIZE(Frontier) $\neq 0$) do                        ▷ Phase 1
28:        Frontier = EDGEMAP($G$, Frontier, PATHSUPDATE, COND)
29:        Levels[currLevel] = Frontier
30:        Frontier = VERTEXMAP(Frontier, VISIT)
31:        currLevel = currLevel + 1
32:    Visited = $\{0, \dots, 0\}$                           ▷ reinitialize to all 0
33:    currLevel = currLevel − 1
34:    TRANSPOSE($G$)                                        ▷ transpose graph
35:    while (currLevel $\geq 0$) do                          ▷ Phase 2
36:        Frontier = Levels[currLevel]
37:        VERTEXMAP(Frontier, VISIT)
38:        EDGEMAP($G$, Frontier, DEPUPDATE, COND)
39:        currLevel = currLevel − 1
40:    return Dependencies
```

since edges now need to point in the reverse direction. The algorithm operates on the vertexSubsets in the Levels array returned from the first phase in reverse order, uses the same VISIT and COND functions as in the first phase, and passes the DEPUPDATE function shown in Lines 17–22 of Algorithm 6 to EDGEMAP. Psuedocode for the second phase of the betweenness-centrality computation is shown in Lines 35–40.

7.4.3 Graph Eccentricity Estimation and Multiple BFS

For a graph $G = (V, E)$, the ***eccentricity*** of a vertex $v \in V$ is defined to be the shortest distance to the furthest reachable vertex of v. The ***diameter*** of the graph is defined to be the maximum eccentricity over all $v \in V$. For unweighted graphs, one simple method for computing the eccentricity of all vertices (and hence the diameter of the graph) is to run n BFS's, one starting at each vertex. However, for large graphs this method is impractical as each BFS requires $O(n + m)$ work, leading to a total of $O(n^2 + nm)$ work (see [112]). This approach can be parallelized by running the BFS's independently in parallel, and also by parallelizing each individual BFS, but currently this is still impractical for large graphs.

There has been work on techniques to estimate the diameter of a graph. Magnien et al. [315] describe several techniques for computing upper and lower bounds on the diameter of a graph, using BFS's and spanning subgraphs. They describe a method called the *double sweep lower bound*, which works by first running a BFS from some vertex v and then a second BFS from the furthest vertex from v (call it w). The radius of w is then taken to be a lower bound on the diameter of the graph. Their method can be repeated by picking more vertices to run BFS's from. Ferrez et al. [152] perform experiments with parallel implementations of some of these methods. Another approach based on counting neighborhood sizes was described by Palmer et al. [366]. Their algorithm approximates the neighborhood function for each vertex in a graph, which is more general than computing graph eccentricities. Kang et al. [249] parallelize this algorithm using MapReduce. Cohen [103] describes an algorithm for approximating neighborhood sizes, which requires $O(m \log n)$ expected work for undirected graphs.

Ligra implements the simple method for estimating graph eccentricities by performing BFS's from a sample of K vertices. Its accuracy can be improved by using the double sweep method [113, 315]. Instead of running the BFS's in parallel independently, the Ligra implementation runs multiple BFS's together. In the ***multiple-BFS*** algorithm, each vertex maintains a bit-vector of length K. Initially K vertices are chosen randomly to act as "source" vertices and each of these K vertices has exactly one unique bit in their bit-vector set to 1; all other vertices have their bit-vectors initialized to all 0's. The K sampled vertices are placed on the initial frontier of the multiple-BFS search. In each iteration, each frontier vertex bitwise-ORs its vector into each of its neighbors' vectors. Vertices whose bit-vectors changed in an iteration are placed on the frontier for the next iteration. The algorithm

Algorithm 7 Eccentricity Estimation

```
1:  Visited = $\{0, \ldots, 0\}$                                          ▷ initialized to all 0
2:  NextVisited = $\{0, \ldots, 0\}$                                      ▷ initialized to all 0
3:  Ecc = $\{\infty, \ldots, \infty\}$                                    ▷ initialized to all $\infty$
4:  round = 0
5:  procedure EccUpdate($s$, $d$)
6:      if (Visited[$d$] $\neq$ Visited[$s$]) then
7:          atomicOR(&NextVisited[$d$], Visited[$d$] | Visited[$s$])
8:          oldEcc = Ecc[$d$]
9:          if (Ecc[$d$] $\neq$ round) then
10:             return CAS(&Ecc[$d$], oldEcc, round)
11:     return 0
12: procedure ORcopy($i$)
13:     NextVisited[$i$] = NextVisited[$i$] | Visited[$i$]
14:     return 1
15: procedure Ecc($G$)
16:     Sample K vertices and for each one set a unique bit in Visited to 1
17:     Initialize Frontier to contain the K sampled vertices
18:     Set the Ecc entries of the sampled vertices to 0
19:     while (size(Frontier) $\neq$ 0) do
20:         round = round + 1
21:         Frontier = edgeMap($G$, Frontier, EccUpdate, $C_{true}$)
22:         vertexMap(Frontier, ORcopy)
23:         swap(Visited, NextVisited)                           ▷ switch roles of bit-vectors
24:     return Ecc
```

iterates until none of the bit-vectors change.

For a sample of size K this algorithm simulates running K BFS's in parallel, but without computing the BFS tree (which is not needed for the eccentricity computation). Storing the iteration number in which a vertex v's bit-vector last changed is a lower-bound on the radius of v since at least one of the K sampled vertices took this many rounds to reach v. If K is set to be the number of bits in a word (32 or 64) this algorithm is more efficient than naively performing K individual BFS's in two ways: (1) the frontiers of the K BFS's could overlap in any given iteration and this algorithm stores the union of these frontiers usually leading to fewer edges traversed per iteration and (2) performing a bitwise-OR on bit-vectors can pass information from more than one of the K BFS's while only requiring one arithmetic operation. Note that this algorithm only estimates the diameter of the connected components of the graph which contain at least one of the K sampled vertices; if there are multiple connected components in the graph, one would first compute in parallel the components of the graph and then run the multiple-BFS algorithm in parallel on each component.

To implement the multiple-BFS algorithm in Ligra (pseudocode shown in Algorithm 7), the code maintains two bit-vectors, *Visited* and *NextVisited*, which are initialized to all 0's, except for the K sampled vertices each of which has a unique bit in their Visited bit-vector set to 1. An array *Ecc* is also maintained, which for each vertex stores the iteration number in which the bit-vector of the vertex last changed. It is initialized to all ∞ except for the K sampled vertices which have a Ecc entry of 0. At the end of the algorithm, Ecc contains the estimated (lower-bound) radius of each vertex, the maximum of which is a lower-bound on the graph diameter. In the pseudocode, "|" is used to denote the bitwise-OR operation. The initial frontier contains the K sampled vertices. The update function ECCUPDATE passed to EDGEMAP is shown in Lines 5–11 of Algorithm 7. ATOMICOR(x, y) performs a bitwise-OR of y with the value stored at x and atomically updates x with this new value. It is implemented using a compare-and-swap. The reason that the code has both Visited and NextVisited is so that new bits that a vertex receives in an iteration do not get propagated to its neighbors in the same round, otherwise the values in Ecc would be incorrect. The compare-and-swap on Line 10 guarantees that any Ecc entry is updated at most once (and returns *true*) per iteration. Therefore any vertex will be placed at most once on the next frontier, eliminating the need for removing duplicates. As in the other implementations, the implementation can provide a version of ECCUPDATE non-atomic with respect to d to EDGEMAP.

The COPY function (Lines 12–14) passed to VERTEXMAP simply copies Visited$[i]$ into NextVisited$[i]$ for each vertex i. This is used because the roles of NextVisited and Visited are switched between iterations. The while loop in Lines 19–23 is executed until the entries of the Ecc array do not change (or equivalently, none of the bit-vectors change).

A detailed study of the performance and accuracy of different parallel eccentricity algorithms, including the one described in this section, has recently been conducted in [417].

7.4.4 Connected Components

Recall the definition of the connected components problem from Section 2.6. One method of computing the connected components of a graph is to maintain an array *IDs* of size $|V|$ initialized such that IDs$[i] = i$, and iteratively have every vertex update its IDs entry to be the minimum IDs entry of all of its neighbors in G. This method is known as *label propagation*, and the total work performed by this algorithm is $O(d(n + m))$ where d is the diameter of G. For high-diameter graphs, this algorithm can perform much worse than other parallel algorithms that require less work (see Chapter 9), but for low-diameter graphs it runs reasonably well. This section describes the label propagation algorithm as a simple application of Ligra.

The pseudocode for the Ligra implementation is shown in Algorithm 8. The initial frontier contains all vertices in V. In addition to the IDs array, the code maintains a second array *prevIDs* (used to check whether a vertex has been placed on the frontier in a given

Algorithm 8 Connected Components

```
1: IDs = $\{0, \ldots, n-1\}$                    ▷ initialized such that IDs[$i$] = $i$
2: prevIDs = $\{0, \ldots, n-1\}$                ▷ initialized such that prevIDs[$i$] = $i$
3: procedure CCUPDATE($s$, $d$)
4:     origID = IDs[$d$]
5:     if (WRITEMIN(&IDs[$d$], IDs[$s$])) then
6:         return (origID == prevIDs[$d$])
7:     return 0
8: procedure COPY($i$)
9:     prevIDs[$i$] = IDs[$i$]
10:    return 1
11: procedure CC($G$)
12:    Frontier = $\{0, \ldots, n-1\}$           ▷ vertexSubset initialized to $V$
13:    while (SIZE(Frontier) $\neq 0$) do
14:        VERTEXMAP(Frontier, COPY)
15:        Frontier = EDGEMAP($G$, Frontier, CCUPDATE, $C_{true}$)
16:    return IDs
```

iteration yet), and passes the CCUPDATE function shown in Lines 3–7 of Algorithm 8 to EDGEMAP. WRITEMIN(x, y) is an instantiation of the priority update operation from Chapter 6—it atomically updates the value at location x to be the minimum of x's old value and y, returning *true* if the value at location x was changed, and *false* otherwise. Line 6 places a vertex on the next frontier if and only if its ID changed in the iteration. To synchronize the values of prevIDs and IDs after every iteration, the COPY function is passed to VERTEXMAP. The while loop in Lines 13–15 is executed until IDs remains the same as prevIDs. When the algorithm terminates, all vertices in the same component will have the same value stored in their IDs entry.

7.4.5 PageRank

PageRank is an algorithm that was first used by Google to compute the relative importance of webpages [74]. It takes as input a graph $G = (V, E)$, a damping factor $0 \leq \gamma \leq 1$ and a convergence constant ϵ. It initializes a PageRank vector *PR* of length n to have all entries set to $1/n$, and iteratively applies the following equation[2] for all indices v, until the sum of the differences of PR values between iterations drops to below ϵ:

$$\text{PR}[v] = \frac{1-\gamma}{n} + \gamma \sum_{u \in N^-(v)} \frac{\text{PR}[u]}{d^+(u)} \tag{7.4}$$

[2]This equation assumes $d^+(u) > 0$ for all u. If the graph has any vertices with an out-degree of 0 (dangling vertices), the PageRank entries will not sum to 1. This can be fixed by adding outgoing edges from dangling vertices to all vertices in the graph.

Algorithm 9 PageRank

```
 1: $p_{curr} = \{1/n, \ldots, 1/n\}$                                  ▷ initialized to all $\frac{1}{n}$
 2: $p_{next} = \{0.0, \ldots, 0.0\}$                                 ▷ initialized to all 0.0
 3: diff $= \{\}$                                                     ▷ array to store differences
 4: procedure PRUPDATE($s$, $d$)
 5:     ATOMICINCREMENT($\&p_{next}[d]$, $\frac{p_{curr}[s]}{d^+(s)}$)
 6:     return 1
 7: procedure PRLOCALCOMPUTE($i$)
 8:     $p_{next}[i] = (\gamma \times p_{next}[i]) + (1-\gamma)/n$
 9:     diff$[i] = |p_{next}[i] - p_{curr}[i]|$
10:     $p_{curr}[i] = 0.0$
11:     return 1
12: procedure PAGERANK($G$, $\gamma$, $\epsilon$)
13:     Frontier $= \{0, \ldots, n-1\}$                              ▷ vertexSubset initialized to $V$
14:     error $= \infty$
15:     while (error $> \epsilon$) do
16:         Frontier = EDGEMAP($G$, Frontier, PRUPDATE, $C_{true}$)
17:         Frontier = VERTEXMAP(Frontier, PRLOCALCOMPUTE)
18:         error = sum of diff entries
19:         SWAP($p_{curr}$, $p_{next}$)
20:     return $p_{curr}$
```

This leads to a very simple implementation in Ligra. This section also describes a variant of PageRank (PageRank-Delta) which applies Equation (7.4) to only a subset of V in an iteration. By choosing the subset to contain only vertices whose PageRank entry that changed by more than a certain amount, the computation can be sped up.

The pseudocode for the Ligra implementation of PageRank is shown in Algorithm 9. In every iteration, the frontier contains all vertices. The implementation maintains two arrays p_{curr} and p_{next} each of length n. p_{curr} is initialized to $1/n$ for each entry and p_{next} is initialized to all 0.0's. The PRUPDATE function passed to EDGEMAP is shown in Lines 4–6. ATOMICINCREMENT(x, y) atomically adds y to the value at location x and stores the result in location x; it can be implemented with a compare-and-swap. Each iteration of the while loop (Lines 15–19) applies an EDGEMAP, uses a VERTEXMAP to process the result of the EDGEMAP, computes the error for the iteration and switches the roles of p_{next} and p_{curr}. The PRLOCALCOMPUTE function (Lines 7–11) passed to VERTEXMAP normalizes the result of the EDGEMAP by γ, adds a constant, computes the absolute difference between p_{next} and p_{curr}, and resets p_{curr} to 0.0 for the next iteration (since the roles of p_{next} and p_{curr} become switched). The while loop is executed until the error drops below ϵ.

PageRank-Delta is a variant of PageRank in which vertices are active in an iteration only if they have accumulated enough change in their PR value. This idea is described in [326] and used in GraphLab for computing PageRank [307]. In the Ligra framework, in each EDGEMAP vertices pass their changes (deltas) in PR value to their neighbors, and all vertices accumulate a sum of delta contributions from their neighbors. Each VERTEXMAP only updates and returns vertices whose accumulated delta contributions from neighbors is more than an α-fraction of its PR value since the last time it was active. Such an implementation allows for vertices which do not influence the PR values much to stay inactive, thereby shrinking the frontier. PageRank-Delta can be implemented in Ligra by modifying the function passed to EDGEMAP to pass the deltas instead of the PR values, and modifying the function passed to VERTEXMAP to only perform updates and return true for the vertices whose accumulated delta contributions from neighbors since it was last active is more than an α-fraction of its PR value.

7.4.6 Bellman-Ford Shortest Paths

This section studies the single-source shortest paths problem (recall the definition from Section 2.6). If the edge weights are all non-negative, then the single-source shortest paths problem can be solved with Dijkstra's algorithm [112]. Parallel variants of Dijkstra's algorithm have been studied [332], and have been shown to work well on real-world graphs [314]. However, Dijkstra's algorithm does not work with negative edge weights, and the Bellman-Ford algorithm can be used instead in this case. Although in the worst case the Bellman-Ford algorithm requires $O(nm)$ work, in contrast to the $O(m + n \log n)$ worst-case work of Dijkstra's algorithm, in practice it can require many fewer than the worst case since on every step only some of the vertices might change distances. It is therefore important to take advantage of this fact and only process vertices when their distance actually changes.

This section first describes the standard Bellman-Ford algorithm [112] and then shows how it can be implemented in Ligra. The algorithm initializes the shortest paths array *SP* to all ∞ except for the root vertex which has an entry of 0. A **RELAX** procedure is repeatedly invoked by Bellman-Ford. RELAX takes G as an input and checks for each edge (u, v) if SP$[u] + w(u, v) <$ SP$[v]$; if so, it sets SP$[v]$ to SP$[u] + w(u, v)$. If a call to RELAX does not change any SP values then the algorithm terminates. If RELAX is called n or more times, then there is a negative cycle in G and the Bellman-Ford algorithm reports the existence of one.

To implement the Bellman-Ford algorithm in Ligra (pseudocode shown in Algorithm 10), a *Visited* array is maintained in addition to the SP array. Since only vertices whose SP value has changed in an iteration need to propagate its SP value to its neighbors, the Visited array (initialized to all 0's) keeps track of which vertices had their SP value changed in an iteration. The update function passed to EDGEMAP is shown in Lines 3–6 of Algorithm 10 (note that since this algorithm works on weighted graphs, the update function has the edge weight as an additional argument). It uses WRITEMIN (as described

Algorithm 10 Bellman-Ford

```
1: SP = $\{\infty, \ldots, \infty\}$                                    ▷ initialized to all $\infty$
2: Visited = $\{0, \ldots, 0\}$                                          ▷ initialized to all 0
3: **procedure** BFUPDATE($s$, $d$, edgeWeight)
4:     **if** (WRITEMIN(&SP[$d$], SP[$s$] + edgeWeight)) **then**
5:         **return** CAS(&Visited[$d$], 0, 1)
6:     **else return** 0
7: **procedure** BFRESET($i$)
8:     Visited[$i$] = 0
9:     **return** 1
10: **procedure** BELLMAN-FORD($G$, $r$)
11:     SP[$r$] = 0
12:     Frontier = $\{r\}$                           ▷ vertexSubset initialized to contain just $r$
13:     round = 0
14:     **while** (SIZE(Frontier) $\neq$ 0 and round $< n$) **do**
15:         Frontier = EDGEMAP($G$, Frontier, BF-UPDATE, $C_{true}$)
16:         VERTEXMAP(Frontier, BF-RESET)
17:         round = round + 1
18:     **if** (round == $n$) **then return** "negative-weight cycle"
19:     **else return** SP
```

in Section 7.4.4) to possibly update SP with a smaller path length. The compare-and-swap on Line 5 guarantees that a vertex is placed on the frontier at most once per iteration. The initial frontier contains just the root vertex r. Each iteration of the while loop in Lines 14–17 applies the EDGEMAP, which outputs a vertexSubset containing the vertices whose SP value changed. In order to reset the Visited array after an EDGEMAP, the BFRESET function (Lines 7–9) is passed to VERTEXMAP. The algorithm either runs until no SP values change or runs for n iterations and reports the existence of a negative-weight cycle. An iteration here differs from the RELAX procedure in that RELAX processes all vertices each time.

7.5 Experiments

All of the experiments presented in this section are performed on the 40-core (with two-way hyper-threading) Intel machine described in Section 2.7. The programs are written in Cilk Plus and compiled with Intel's `icpc` compiler. Experiments were also performed on a 64-core AMD Opteron machine, but the results were slower than the ones from the Intel machine so only the Intel results are reported.

The input graphs used in the experiments are shown in Table 7.1. ***3D-grid*** is a grid graph in 3-dimensional space in which every vertex has six edges—one connecting it to each of its two neighbors in each dimension. ***randLocal*** is a synthetic graph in which every vertex has edges to five randomly chosen neighbors, where the probability of an edge between

Input	Number of Vertices	Number of Directed Edges
3D-grid	10^7	6×10^7
randLocal	10^7	9.8×10^7
rMat24	1.68×10^7	9.9×10^7
rMat27	1.34×10^8	2.12×10^9
Twitter	4.17×10^7	1.47×10^9
Yahoo!*	1.4×10^9	12.9×10^9

Table 7.1: Graph inputs for Ligra experiments. *The original asymmetric graph has 6.6×10^9 edges.

two vertices is inversely correlated with their distance in the vertex array (vertices tend to have edges to other vertices that are close in memory). The ***rMat*** graphs are synthetic graphs with a power-law distribution of degrees [87]. ***rMat24*** (scale 24) contains 1.68×10^7 vertices and was generated with parameters $a = 0.5, b = c = 0.1, d = 0.3$. ***rMat27*** (scale 27) is one of the Graph500 benchmark graphs [190], and was generated with parameters $a = 0.57, b = c = 0.19, d = 0.05$. ***Twitter*** is a real-world graph of the Twitter social network containing 41.7 million vertices and 1.47 billion directed edges [288]. ***Yahoo!*** is a real-world graph of the Web containing 1.4 billion vertices and 6.6 billion directed edges (12.9 billion after symmetrizing and removing duplicates) [466]. With the exception of Pregel, the Yahoo! graph is the largest real-world graph reported by other graph processing systems.

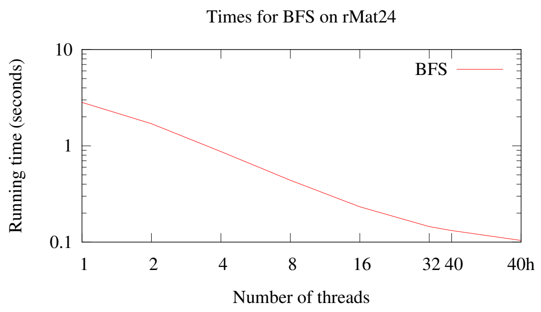
The number of edges reported is the number of directed edges in the graph with duplicate edges removed. The synthetic graphs are all symmetric, and the Yahoo! graph was symmetrized to created a larger graph for the experiments. The original asymmetric Twitter graph was used. For the synthetic weighted graphs, the edge weights were generated randomly and were verified to contain no negative cycles. The experiments used unit weights on the Twitter and Yahoo! graphs for the Bellman-Ford experiments.

Table 7.2 shows the running times for our implementations on each of the input graphs using a single thread and 40 cores with hyper-threading. All of the implementations used EDGEMAPDENSE for the dense iterations with the exception of Bellman-Ford, PageRank, and PageRank-Delta, which used EDGEMAPDENSE-WRITE, an optimization described in Section 7.3.4 (it was found to be more efficient in these cases). Figure 7.7 shows that all of the Ligra implementations scale well with the number of threads. Each application is discussed in more detail below, and compared with the fastest graph processing system that also supports a high-level programming abstraction available at the time Ligra [420] was published.

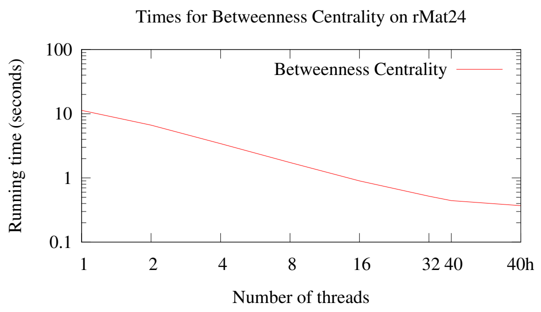
For BFS, Ligra achieves a 10–28 fold parallel speedup on 40 cores. Ligra integrates the ideas of [31] to give a simple implementation of BFS, which is almost as fast as their highly-optimized implementation while being much simpler. The Ligra running times are better than those reported in [294, 424, 5], which do not take advantage of changes in the

Application	3D-grid			randLocal			rMat24			rMat27			Twitter			Yahoo!		
	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)
Breadth-First Search	2.9	0.28	10.4	2.11	0.073	28.9	2.83	0.104	27.2	11.8	0.423	27.9	6.92	0.321	21.6	173	8.58	20.2
Betweenness Centrality	9.15	0.765	12.0	8.53	0.265	32.2	11.3	0.37	30.5	113	4.07	27.8	47.8	2.64	18.1	634	23.1	27.4
Graph Eccentricity	351	10.0	35.1	25.6	0.734	34.9	39.7	1.21	32.8	337	12.0	28.1	171	7.39	23.1	1280	39.6	32.3
Connected Components	51.5	1.71	30.1	14.8	0.399	37.1	14.1	0.527	26.8	204	10.2	20.0	78.7	3.86	20.4	609	29.7	20.5
PageRank (1 iteration)	4.29	0.145	29.6	6.55	0.224	29.2	8.93	0.25	35.7	243	6.13	39.6	72.9	2.91	25.1	465	15.2	30.6
Bellman-Ford	63.4	2.39	26.5	18.8	0.677	27.8	17.8	0.694	25.6	116	4.03	28.8	75.1	2.66	28.2	255	14.2	18.0

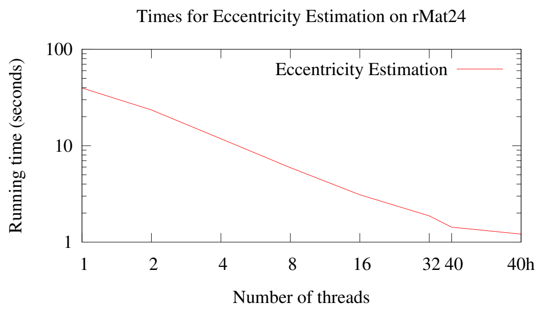
Table 7.2: Running times (in seconds) of algorithms over various inputs on a 40-core machine (with hyper-threading). (SU) indicates the speedup of the application (single-thread time divided by 40-core time).



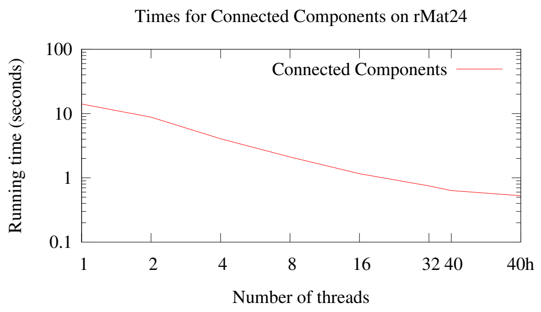
(a) BFS



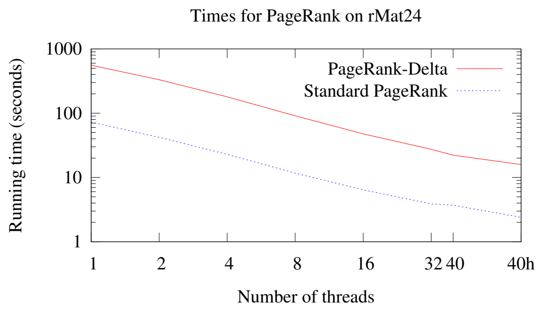
(b) Betweenness Centrality



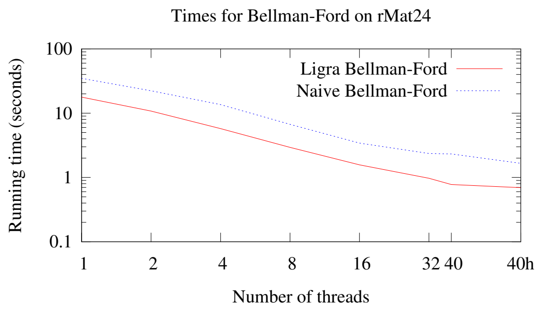
(c) Eccentricity Estimation



(d) Connected Components



(e) PageRank



(f) Bellman-Ford

Figure 7.7: Log-log plots of running times on rMat24 on a 40-core machine with two-way hyper-threading. "40h" corresponds to 80 hyper-threads.

frontier density. Compared to the sequential BFS implementation in the Problem Based Benchmark Suite, Ligra is faster on two or more threads.

For betweenness centrality (performing the two-phase computation for a single source), Ligra achieves a 12–32 fold speedup on 40 cores. The KDT system [311] reports that on

256 cores (2.1 GHz AMD Opteron) their batched implementation of betweenness centrality (performs the two-phase computation for multiple sources in parallel) traverses almost 125 million edges per second on an rMat graph with 2^{18} vertices and 16×2^{18} edges. On rMat27, the Ligra implementation traverses 526 million edges per second using 40 cores on the Intel Nehalem machine, but it is difficult to directly compare because the machine used for experiments is different and Ligra does not do a batched computation. For the Twitter graph, since the graph is transposed for the second phase, the in-degree of some of the vertices increases dramatically, so we found that using a parallel inner loop in EDGEMAPDENSE, an optimization described in Section 7.3.4, was more efficient.

The graph eccentricity estimation implementation was run using a 64-bit vector for each vertex ($K = 64$) and it achieves a 23–35x speedup on 40 cores. Kang et al. [249] implement a slightly different algorithm for estimating the eccentricity distribution using MapReduce, and run experiments on the Yahoo! M45 Hadoop cluster (480 machines with 2 quad-core Intel Xeon 1.86 GHz processors per machine). Using 90 machines their reported running time for 3 iterations on a 2 billion-edge graph is almost 30 minutes. Using a 40-core machine, the Ligra code is able to process the rMat27 graph of similar size until completion (9 iterations) in 12 seconds.

The Ligra connected components implementation achieves a 20–37 fold speedup on 40 cores. The Pegasus library [250] also has a connected components algorithm implemented for the MapReduce framework. For a graph with 59,000 vertices and 282 million edges, and using 90 machines of the Yahoo! M45 cluster, they report a runtime of over 10 minutes for 6 iterations. In contrast, for the much larger rMat27 graph (also requiring 6 iterations) the Ligra algorithm completes in about 10 seconds on the 40-core machine.

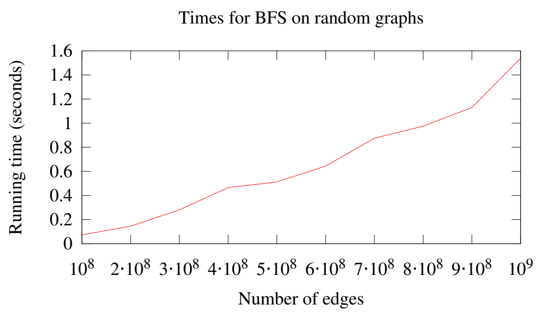
For a single iteration, Ligra's PageRank implementation achieves a 29–39 fold speedup on 40 cores. GPS [405] reports a running time of 144 minutes for 100 iterations (1.44 minutes per iteration) of PageRank on a web graph with 3.7 billion directed edges on an Amazon EC2 cluster using 30 large instances, each with 4 virtual cores and 7.5GB of memory. In contrast, Ligra's PageRank implementation takes less than 20 seconds per iteration on the larger Yahoo! graph. For PageRank on the Twitter graph [288], the Ligra system is slightly faster per iteration (2.91 seconds vs. 3.6 seconds) on 40 cores than PowerGraph [186] on 8×64 cores (processors are 2.933 GHz Intel Xeon X5570 with a 3200 MHz bus). The experiments also compared the Ligra implementations of PageRank and PageRank-Delta, run to convergence with a damping factor of $\gamma = 0.85$ and parameters $\epsilon = 10^{-7}$ and $\alpha = 10^{-2}$. Figure 7.7(e) shows that PageRank-Delta is faster (by more than a factor of 6 on rMat24) because in any given iteration it processes only vertices whose accumulated change is above a α-fraction of its PageRank value at the time it was previously active. The error (which depends on α) of the PageRank-Delta implementation is not analyzed in this work—the purpose of this experiment is to show that Ligra also works well for problems other than standard graph traversals.

Ligra's parallel implementation of Bellman-Ford achieves a 18–28$\times$ speedup on 40 cores. Figure 7.7(f) compares this implementation with a naive one which visits all vertices and edges in each iteration, and Ligra's more efficient version is almost twice as fast. The single-source shortest paths algorithm of Pregel [318] for a binary tree with 1 billion vertices takes almost 20 seconds on a cluster of 300 multicore commodity PCs. Ligra's Bellman-Ford algorithm on a larger binary tree with $2^{27} (\approx 1.68 \times 10^7)$ vertices completed in under 2 seconds (time not shown in Table 7.2). Compared to the implementation of the standard sequential algorithm described in [112], Ligra's parallel implementation is faster on a single thread.

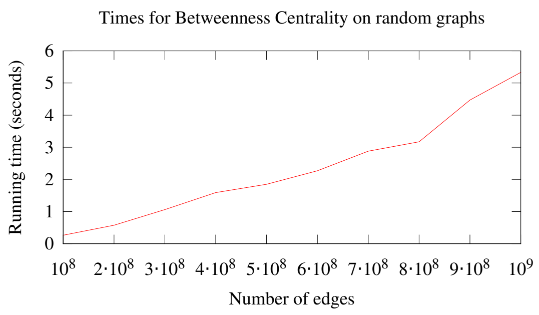
Since the Yahoo! graph is highly disconnected, we computed the number of vertices and directed edges traversed for BFS and betweenness centrality and found it to be 701 million and 12.8 billion, respectively (this is the largest connected component of the graph). The number of vertex and edge traversals for the graph eccentricity algorithm ($K = 64$) on the Yahoo! graph were 2.7 billion and 50 billion, respectively. Note that doing 64 individual BFS's to compute the same thing would require many more vertex and edge traversal; the Ligra implementation of eccentricity estimation (multiple-BFS) reduces the number of traversals (and hence the running time) by combining the operations of multiple BFS's into fewer operations.

Figure 7.8 shows scalability plots for the various applications. The experiments were performed on random graphs of varying size with the number of directed edges being ten times the number of vertices. The reader can observe that the implementations scale quite well with increasing graph size, with some noise due to the variability in the structures of the different random graphs.

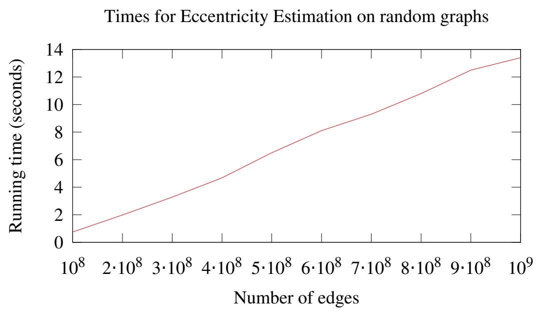
Figure 7.9 shows plots of the size of the frontier plus the number of outgoing edges for each iteration and each application on rMat24. The rMat24 graph is a scale-free graph and hence able to take advantage of the hybrid BFS idea of Beamer et al. [32]. The y-axes are shown in log-scale. The figures also plot the threshold, above which EDGEMAP uses the dense implementation and below which EDGEMAP uses the sparse implementation. For BFS, betweenness centrality (same frontier plot as that of BFS), eccentricity estimation, and Bellman-Ford, the frontier is initially sparse, switches to dense after a few iterations and then switches back to sparse later. For connected components and PageRank-Delta, the frontier starts off as dense (the vertexSubset contains all vertices), and becomes sparser as the algorithm continues. See [32] for a more detailed analysis of frontier plots for BFS.



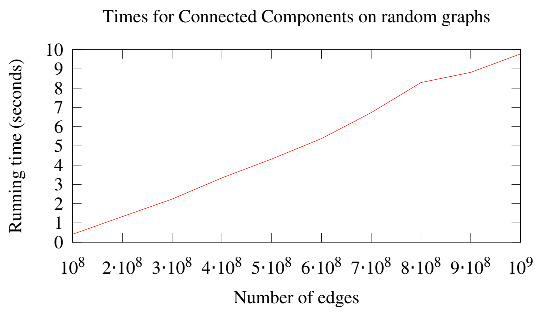
(a) BFS



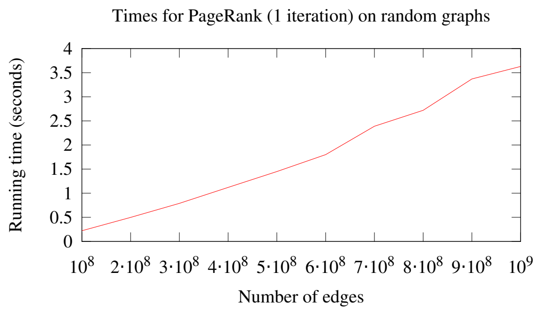
(b) Betweenness Centrality



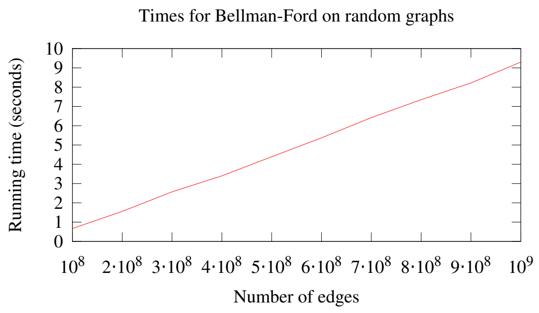
(c) Eccentricity Estimation



(d) Connected Components

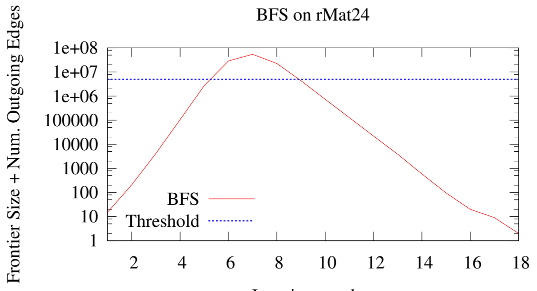


(e) PageRank (1 iteration)

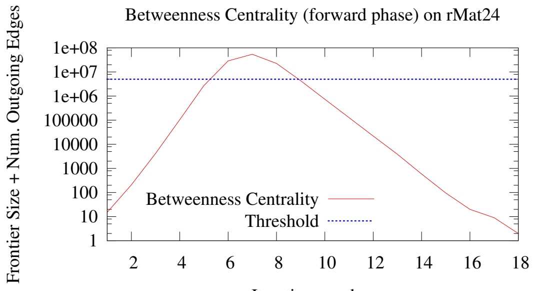


(f) Bellman-Ford

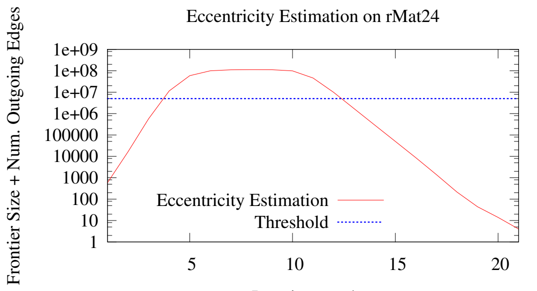
Figure 7.8: Plots of running times versus edge counts in random graphs on a 40-core machine (with hyper-threading).



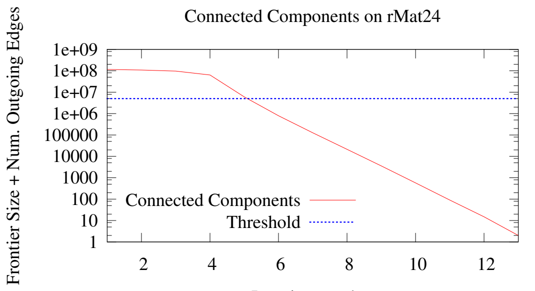
(a) BFS



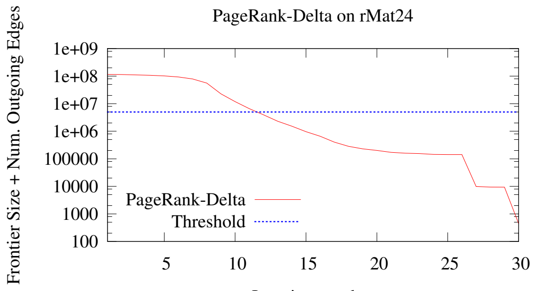
(b) Betweenness Centrality



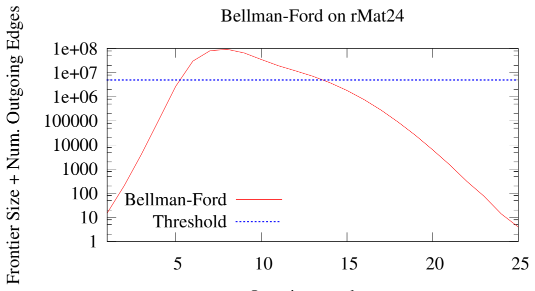
(c) Eccentricity Estimation



(d) Connected Components



(e) PageRank-Delta



(f) Bellman-Ford

Figure 7.9: Plots of frontier size plus number of outgoing edges (y-axis in log scale) versus iteration number for rMat24.

Chapter 8

Ligra+: Adding Compression to Ligra

8.1 Introduction

The previous chapter showed the simplicity, expressiveness, and efficiency of Ligra for shared-memory graph processing. This chapter describes graph compression techniques that can be used to reduce Ligra's memory usage. While the largest real-world graphs can fit on a single shared-memory server, reducing memory usage allows one to use machines with less memory for graph processing. This leads to reduced costs, whether one is purchasing the machines or renting machines in the cloud. Additionally, it is interesting to know if using compression can speed up parallel graph algorithms.

This chapter parallelizes and integrates various compression and decoding techniques from the graph compression literature as well as from the sparse matrix-vector multiplication literature into Ligra. The extended framework, called Ligra+, uses less space than Ligra, while providing comparable or improved performance. Ligra+ is able to represent a variety of synthetic and real-world graphs using 49–56% of its original size on average, depending on the compression scheme. The performance of the graph algorithms in Ligra+ ranges from 2.2x faster to 1.1x slower than the original Ligra system. In many cases, Ligra+ outperforms Ligra due to its smaller memory footprint, and is about 14% faster on average when using the fastest compression scheme. Using compression, Ligra+ is able to process graphs using less memory, and fit larger graphs in memory, while performing just as well as or better than Ligra. As the compression techniques are part of a graph processing framework, users can easily work with compressed graphs without worrying about the implementation details. Applications written in Ligra can also be used in Ligra+, as the interfaces are identical.

8.2 Previous Work

There has been a large amount of work on compressing graphs, especially planar graphs and graphs with constant genus (see, e.g., [45] and the references within). Blandford et al. [45, 46] experiment with a variety of graph compression techniques, and study the performance of graph algorithms on compressed graphs, which requires on-the-fly decoding. Their techniques can reduce space usage by up to a factor of 3–6 compared to normal adjacency arrays. However, they only study the techniques in a sequential setting, and for only three specific algorithms—depth-first search, PageRank, and bipartite matching. They show that the algorithms on compressed graphs are about 25% slower. This chapter parallelizes their compression techniques, and studies the performance of a broad class of parallel graph algorithms on much larger graphs than used in [45, 46]. The experiments show that sequentially, the algorithms on compressed graphs are indeed often slower, however in parallel they become competitive with or faster than the algorithms on uncompressed graphs. This is because graph algorithms are memory-bound, and memory is a larger bottleneck in parallel due to multiple cores competing for bandwidth—therefore reducing the memory footprint is more important, while at the same time decoding becomes less of an overhead as it has better parallel scalability relative to the rest of the computation.

Recent work [248, 301] has used compression to reduce graph sizes in the MapReduce setting. Their focus is on reducing the storage size on disk because a large portion of the running time of MapReduce is from disk I/O's, and they show performance improvements for MapReduce graph algorithms. However, the techniques are not used to reduce the in-memory space usage. In contrast, this work focuses on reducing the in-memory space usage while maintaining or improving performance, so it becomes necessary to efficiently decode on-the-fly.

Other work has focused mainly on compressing Web and social network graphs (see, e.g., [3, 71, 93]). Most of these works have not been used to improve the performance of general graph algorithms. The techniques that have been applied to graph algorithms are particular to the algorithm and compression scheme [390, 214, 77, 251, 75], and not used in a general framework. The algorithms are also studied in the sequential setting.

Running algorithms on compressed inputs has been previously explored in the setting of sparse matrix-vector (spMV) multiplication [463, 276, 61, 277, 79]. Like graph algorithms, spMV is also a memory-bound computation, and so better improvements are observed in parallel. These papers show promising results, but only study the specific spMV computation. In contrast, this chapter studies the impact of compression in a broad class of parallel graph algorithms.

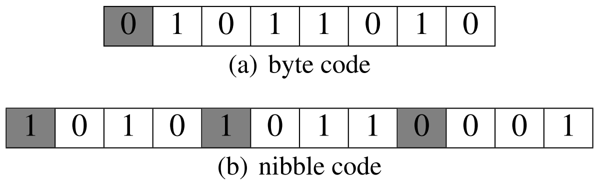


Figure 8.1: Encoding the value “90” with a byte (8-bit) code and a nibble (4-bit) code. The continue bits are shaded in gray. In this case, the nibble code uses more space.

8.3 Ligra+ Implementation

This section first describes preliminaries and then presents the implementation of Ligra+ to support processing of compressed graphs.

8.3.1 Preliminaries

The ***difference encoding*** scheme takes a vertex v's adjacency list, $\{v_0, v_1, \ldots, v_{deg(v)-1}\}$, given in increasing order and encodes the differences, $\{v_0 - v, v_1 - v_0, \ldots, v_{deg(v)-1} - v_{deg(v)-2}\}$. The graph compression scheme of Blandford et. al [46] uses a class of variable-length codes, known as k-bit codes, which encode (compress) an integer x as a series of k-bit blocks. Each block uses one bit as a ***continue bit***, which indicates if the following block is also a part of x's encoded representation. To encode x, we first check if $x < 2^{k-1}$. If this is the case, we simply write the binary representation of x into a single block, and set the continue bit to 0. Otherwise, we write the binary code for $x \mod 2^{k-1}$ in the block, set the continue bit to 1, and then encode $\lfloor x/2^{k-1} \rfloor$ in the subsequent blocks. Decoding works by examining blocks until a block with a continue bit of 0 is found. The decoded value in the i'th examined block is multiplied by $2^{i(k-1)}$, and added to the result. For values that can be negative (e.g., the first edge of a vertex), an extra bit in the first block is used as the sign bit, and decoding is modified accordingly.

8.3.2 Encoding

Ligra+ uses two types of k-bit codes—byte codes and nibble codes, which correspond to 8-bit and 4-bit codes, respectively. Byte codes are fast to decode, as compressed blocks lie on byte-aligned boundaries. Nibble codes lie on 4-bit boundaries, and are slower to decode due to the extra bit arithmetic required. Blandford et al. [46] show that 2-bit codes and gamma codes (effectively 1-bit codes) do not provide much additional space savings compared to nibble codes, while being more expensive to decode, so Ligra+ does not use them.

For byte codes, Ligra+ also uses an idea from the sparse matrix-vector multiplication (spMV) compression literature [276] which reduces the decoding time. Instead of storing a variable-length code for each value, it finds consecutive groups of elements that require the same number of bytes (1 to 4 bytes) to store. For each group it stores an 8-bit header indicating the number of bytes each element requires (2 bits of the header) and the size of the group (6 bits of the header, which allows for groups of up to size 64). This technique slightly increases the space usage, but decreases the decoding time as there is no longer a continue bit which needs to be checked to figure out when to stop decoding. This allows the decoding loop for an element to be unrolled, as the number of bytes it requires is known beforehand, and branch mispredictions are reduced. This chapter refers to this scheme as ***run-length encoded byte codes***. Note that with this scheme, each byte can use all 8 bits for data, as it no longer needs to store a continue bit.

Ligra+ implements an encoder program that generates a binary file representing a compressed graph using one of the coding schemes. The encoding is parallelized over the vertices. For each vertex in the graph, the edges are first sorted in non-decreasing order, and then the edge set of each vertex is compressed by encoding the differences between consecutive edges. For the first edge of each vertex, the difference between the source and the target vertex (which can be negative) is encoded, with an additional sign bit in the first block. The run-length encoded byte codes do this as well for the first edge of each vertex (the discussion in the previous paragraph is only applied to the remaining edges). The implementation maintains a single array of compressed edge values, and stores the vertex offsets into the array. To process the vertices in parallel in the applications, vertex degrees must be known before decoding so that appropriate offsets into shared arrays can be computed. In Ligra+, the vertex degrees are not implicit from the offsets, while in Ligra they are. Hence the vertex degrees are stored as well. Vertex offsets and degrees are not compressed, since for many real-world graphs the number of edges is much larger than the number of vertices, so the space savings are low. For asymmetric input graphs, the in-edges for each vertex are also generated and encoded.

8.3.3 Decoding

The vertexSubset and VERTEXMAP implementations in Ligra+ are the same as in Ligra, since vertices are not compressed. The implementations of EDGEMAP are modified, so that the neighbors of a vertex are decoded using a special function. In particular, Lines 4–6 of the EDGEMAPSPARSE pseudocode in Figure 7.3 and Lines 5–7 of the EDGEMAPDENSE-WRITE pseudocode in Figure 7.6 are replaced by a call to DECODESPARSE, and Lines 5–8 of the EDGEMAPDENSE pseudocode in Figure 7.4 are replaced by a call to DECODEDENSE.

This section describes the sequential implementations of DECODESPARSE and DECODEDENSE for the variable-length codes, and Section 8.3.4 will describe how to parallelize them. The implementations use two decoding functions ***FirstEdge*** and ***NextEdge***. FirstEdge takes as input a pointer into the compressed edge array, and decodes one value representing the difference between the edge and source vertex (which can be negative).

```
 1: procedure DECODESPARSE($v$, $d$, outEdges, $F$, $C$, Out)
 2:     prevEdge $= -1$
 3:     for $j = 0$ to $d - 1$ do                                  ▷ Loop over out-neighbors
 4:         if $j == 0$ then
 5:             ngh = FirstEdge(outEdges) + $v$
 6:         else
 7:             ngh = NextEdge(outEdges) + prevEdge
 8:         prevEdge = ngh
 9:         if ($C(\text{ngh}) == 1$ and $F(v, \text{ngh}) == 1$) then
10:             Add ngh to Out
```

Figure 8.2: Ligra+ DECODESPARSE implementation

It then modifies the pointer to point to the start of the next value in the compressed edge array. NextEdge takes as input a pointer into the compressed edge array, decodes one value representing the difference between consecutive edges (which can only be positive), and modifies the pointer to point to the start of the next value in the compressed edge array. The decoding functions decode byte codes or nibble codes following the procedure described in Section 8.3.1.

The pseudocode for DECODESPARSE is shown in Figure 8.2. It takes as input the source vertex v, its degree d, a pointer to the start of its out-neighbors in the compressed array of out-edges (outEdges), the functions F and C, and a pointer to the output vertexSubset of EDGEMAPSPARSE (Out). It decodes the first neighbor by calling the function FirstEdge, which returns the difference between the source and target vertex, and then adds the value of the source vertex v to the result to obtain the value of the neighbor (Lines 4–5). The result is assigned to the variable prevEdge to allow for decoding of subsequent edges (Line 8). In later iterations, the difference between the previous edge and current edge is obtained by calling the function NextEdge; the edge value is obtained by adding the difference to the value of prevEdge (Lines 6–7), and then subsequently assigned to prevEdge (Line 8). As in EDGEMAPSPARSE, the function C is applied to ngh, and if it returns *true*, F is applied to (v, ngh); if F returns *true* then the neighbor is added to the output vertexSubset (Lines 9–10).

The pseudocode for DECODEDENSE is shown in Figure 8.3. It takes the same arguments as DECODESPARSE, except that the compressed edge array is for the in-edges (inEdges) instead of the out-edges, and it also takes the input vertexSubset U to EDGEMAPDENSE. Decoding the edges is done in the same way as in DECODESPARSE. When DECODEDENSE is called with vertex v, it is assumed that $C(v)$ is *true*. As in the original EDGEMAPDENSE, it checks if an in-neighbor ngh is in the input vertexSubset U, and if so applies F to (v, ngh); if F returns *true* then v is added to the resulting vertexSubset (Lines 9–10). The optimization of breaking early is done on Line 11.

For run-length encoded byte codes, the decoding procedures are modified to process

```
procedure DECODEDENSE($v$, $d$, inEdges, $F$, $C$, Out, $U$)
    prevEdge $= -1$
    for $j = 0$ to $d - 1$ do                                ▷ Loop over in-neighbors
        if $j == 0$ then
            ngh = FirstEdge(inEdges) + $v$
        else
            ngh = NextEdge(inEdges) + prevEdge
        prevEdge = ngh
        if (ngh $\in U$ and $F(\text{ngh}, v) == 1$) then
            Add $v$ to Out
        if $(C(v) == 0)$ then break
```

Figure 8.3: Ligra+ DECODEDENSE implementation

groups of edges after reading the header. Lines 9–10 of DECODESPARSE and Lines 9–11 of DECODEDENSE are applied immediately after each neighbor ID is decoded.

8.3.4 Parallel Decoding

Although for most graphs, decoding the edges sequentially for each vertex gives performance competitive with Ligra, my co-authors and I found that for some applications on certain graphs, it was up to 2 times slower. In these cases, parallelizing over the vertices was not sufficient due to the highly skewed distributions of degrees. Therefore we designed a parallel decoding scheme, in which vertices with degree greater than some threshold T split their edges into chunks each containing T edges (except for possibly the last chunk), and the first edge of each chunk is difference encoded with respect to the source vertex. For each vertex, offsets into its chunks of edges are stored, except for the first chunk. Thus, for vertices with only one chunk (degree at most T), no extra storage is required. For each vertex, the different chunks of the edge array are decoded in parallel, as the offset to the start of each chunk is known. For DECODEDENSE, the optimization of breaking early is applied inside each chunk. The threshold T represents a trade-off between parallelism and space overhead. The experiments used a threshold $T = 1000$, which was found to work best overall, although the performance was similar across a wide range of T (from 100 to 10,000). The storage required for the additional offsets is minimal for this range of T, as there are at most $m/T - n$ offsets needed for the graph. As an example, Figure 8.4 shows the parallel BFS running time of Ligra+ using run-length encoded byte codes on the Twitter graph and its space as a function of T. The Twitter graph [288] (see Section 8.4 for its size) is a graph with a very skewed degree distribution, and thus benefits from parallel decoding. The rightmost point ($T =$ maximum degree) of each plot corresponds to not chunking the edges at all. Observe that the running time is similar for T in the range 100 to 10,000, but increases if T is too small or too large. The space usage for $T \geq 1000$ is about the same as not using chunking at all, but can be significantly higher if T is too small. A similar trend

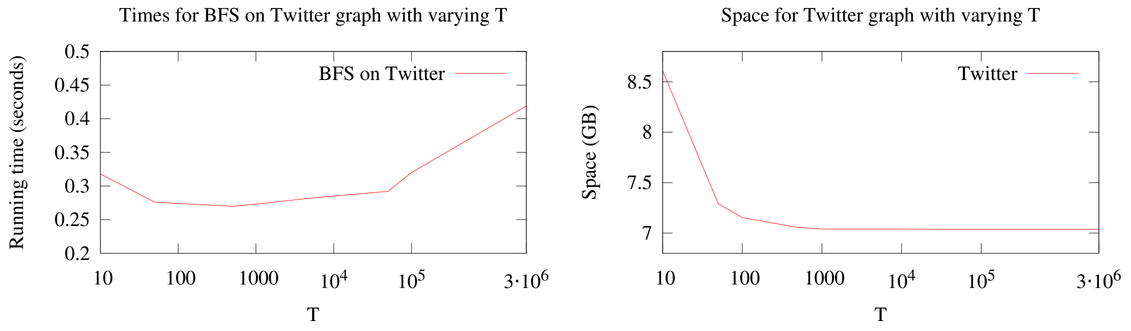


Figure 8.4: BFS running time of Ligra+ using run-length encoded byte codes on Twitter on 40 cores with hyper-threading versus T (**left**), and space of Twitter versus T (**right**).

was observed in other applications and other graphs with skewed degree distributions. It is worth noting that the papers describing compression in parallel spMV do not perform parallel decoding within rows of the matrix (analogously, the edges of a vertex).

8.3.5 Graph Storage

Two arrays for edges are used—one for the compressed in-edges and one for the compressed out-edges. Vertex offsets into the edge arrays and their degrees are stored in a separate array, uncompressed. For symmetric graphs, only one edge array is required, and for asymmetric graphs, both the in-edges and out-edges are required.

8.3.6 Weighted Graphs

For weighted graphs, the edge weights are encoded using difference encoding with respect to the value 0, and a bit in the first block is used as the sign bit. Decoding is done in the same manner as decoding the first edge of a vertex, but relative to the value 0. The edge targets and weights are interleaved to improve cache locality. The FirstEdge and NextEdge functions are modified to decode the target of an edge along with its weight.

For run-length encoded byte codes, the encoder finds groups of edges that require at most x bytes for the difference with the previous edge and y bytes for the weight, where $x \in \{1, 2, 3, 4\}$ and $y \in \{1, 4\}$. The header byte uses 3 bits to store the (x, y) combination and 5 bits for the size of the group (allowing for groups of up to size 32). The decoding functions are modified accordingly to decode groups of edge targets/weights after reading the header.

8.3.7 Comparison to Ligra

The user interface to Ligra+ is the same as in Ligra, so applications developed using Ligra are compatible with Ligra+. Only the graph representation and the implementations of

Input Graph	Number of Vertices	Number of Directed Edges	Ligra	Ligra+ (byte)	Ligra+ (byte-RLE)	Ligra+ (nibble)
randLocal	10,000,000	98,201,048	433 MB	228 MB	246 MB	221 MB
3D-grid	9,938,375	59,630,250	278 MB	219 MB	209 MB	209 MB
soc-LJ	4,847,571	85,702,474	362 MB	188 MB	204 MB	178 MB
cit-Patents	6,009,555	33,037,894	156 MB	107 MB	117 MB	105 MB
com-LJ	4,036,538	69,362,378	294 MB	152 MB	166 MB	143 MB
com-Orkut	3,072,627	234,370,166	950 MB	440 MB	466 MB	421 MB
nlpkkt240	27,993,601	746,478,752	3.1 GB	1.06 GB	1.16 GB	815 MB
Twitter	41,652,231	1,468,365,182	12.08 GB	6.17 GB	6.46 GB	5.95 GB
uk-union	133,633,041	5,507,679,822	45.9 GB	15.5 GB	16.2 GB	10.9 GB
Yahoo!	1,413,511,391	12,869,122,070	62.8 GB	37.9 GB	39.3 GB	34.4 GB

Table 8.1: Graph input sizes and storage sizes, including both vertices and edges.

EDGEMAP have changed, and the user is not exposed to this.

8.4 Experiments

This section analyzes the effect of compression on the space usage and running time using a collection of large-scale graphs. The experiments are done on the six graph applications described in Section 7.4: breadth-first search (***BFS***), betweenness centrality computation from a source vertex (***BC***), graph eccentricity estimation (***Eccentricity***), connected components (***Components***), ***PageRank*** (one iteration) and ***Bellman-Ford*** shortest-paths. This section only compares Ligra+ with Ligra, as the goal of the experimental study is to observe the impact of graph compression on running time and space usage, while keeping other factors the same. The code is written in Cilk Plus and compiled with the `icpc` compiler, and the experiments are run on the 40-core Intel machine described in Section 2.7.

Input Graphs. The experiments use a set of synthetic and real-world graphs, whose sizes are shown in Table 8.1. The ***randLocal***, ***3D-grid***, ***Twitter*** and ***Yahoo!*** graphs are as described in Section 7.5. The experiments also use the ***soc-LJ***, ***cit-Patents***, ***com-LJ*** and ***com-Orkut*** graphs from the Stanford Network Analysis Project [298], which we symmetrized. ***nlpkkt240*** is a graph from an optimization problem obtained from [124]. ***uk-union*** is a graph generated from snapshots of a subset of the UK web network [69]. Twitter and uk-union are asymmetric, and the rest of the graphs are symmetric. All self and duplicate edges are removed from the graphs.

Compression Quality. My co-authors and I experimented with several graph reordering schemes (e.g., [45, 261]) to improve the locality (i.e., renumber vertices such that the IDs of vertices and their neighbors are close), and hence compression of the graphs. Detailed experiments on various reordering schemes are presented in Section 8.4.1. While for most graphs, applying the best reordering algorithm improves compression, the locality of our real-world graphs is already quite good without reordering. The experiments use the best

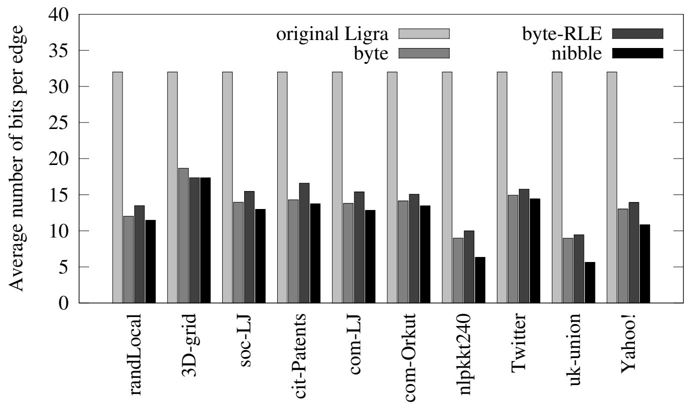


Figure 8.5: Average number of bits per edge required for the different coding schemes in Ligra+.

ordering for each graph, but we confirmed that reordering is not always necessary to obtain good compression.

Figure 8.5 compares the average bits per edge required for byte coding (***byte***), run-length encoded byte coding (***byte-RLE***), and nibble coding (***nibble***) using the best reordering algorithm for each graph. For reference, the figure also shows that the uncompressed graph in Ligra requires 32 bits per edge. For the input graphs, all three coding schemes use many fewer bits per edge than in Ligra (at most 19 bits per edge). Among the three coding schemes, nibble codes require the least space, followed by byte codes, and finally byte-RLE codes.

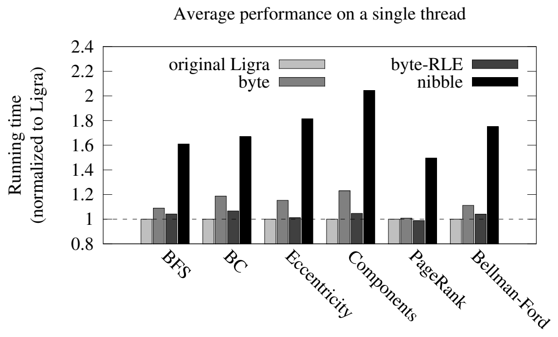
Table 8.1 reports the size required to store each graph in Ligra, Ligra+ with byte coding, run-length encoded byte coding, and nibble coding. This includes the edges, vertex offsets, and vertex degrees (for Ligra+). For graphs that have a high vertex-to-edge ratio (e.g., 3D-grid) the space savings of Ligra+ compared to Ligra are smaller, since Ligra+ does not compress vertices. However, for graphs with good compression and/or low vertex-to-edge ratio, such as nlpkkt240 and uk-union, the space savings are up to 3x for byte and byte-RLE coding and 4x for nibble coding. *On average, byte codes, byte-RLE codes and nibble codes reduce the space to about 53%, 56% and 49% of the uncompressed size, respectively.*

Running Time. Table 8.2 reports the times using a single-thread (T_1) and times using 40 cores with hyper-threading (T_{40h}) for each application on each input graph. The time for encoding graphs is not included in the running times, as this process only needs to be done once per graph and is hence the cost is amortized across all subsequent computations on the graph. The encoding step is quite efficient as it essentially amounts to a scan over each vertex's edges, and is done in parallel. Figure 8.6 plots the average performance per application of Ligra+ with each encoding scheme relative to Ligra. The reader can observe that sequentially, Ligra+ is slower on average than Ligra for all of the applications except

Input Graph	BFS								BC								Eccentricity							
	orig.		byte		byte-RLE		nibble		orig.		byte		byte-RLE		nibble		orig.		byte		byte-RLE		nibble	
	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})
randLocal	1.46	0.055	1.93	0.056	1.8	0.054	3.23	0.08	4.82	0.152	6.36	0.167	5.33	0.159	7.95	0.228	9.22	0.286	10.7	0.295	9.82	0.284	17.1	0.433
3D-grid	1.47	0.214	1.3	0.216	1.26	0.214	1.66	0.233	4.75	0.559	4.64	0.572	5.36	0.558	5.78	0.588	135	5.23	173	5.57	139	5.25	244	7.73
soc-LJ	0.634	0.028	0.677	0.027	0.676	0.026	0.902	0.031	2.6	0.093	3.37	0.108	3.11	0.1	5.08	0.139	8.06	0.22	10.6	0.233	10.6	0.218	15	0.363
cit-Patents	0.639	0.029	0.758	0.03	0.752	0.03	1.08	0.037	2.1	0.086	2.59	0.091	2.49	0.091	3.62	0.115	4.75	0.149	6.91	0.16	5.81	0.157	8.61	0.224
com-LJ	0.523	0.023	0.539	0.023	0.54	0.023	0.708	0.026	2.12	0.082	2.89	0.091	2.57	0.087	4.14	0.121	7.75	0.212	8.73	0.222	8.23	0.213	14	0.343
com-Orkut	0.663	0.029	0.899	0.031	0.789	0.029	1.65	0.049	4.38	0.14	5.98	0.163	4.94	0.142	9.59	0.268	13.2	0.355	14.4	0.367	12.3	0.323	26.1	0.645
nlpkkt240	10.3	0.489	9.41	0.463	8.74	0.466	14.2	0.517	33.3	1.34	33.3	1.28	28.1	1.23	35.7	1.39	897	22.6	1120	24.4	906	21.1	1820	39.1
Twitter	6.91	0.27	8.79	0.274	8.33	0.268	13	0.347	40.1	4.62	47.4	3.16	44.9	3.53	75.2	3.78	172	7.46	193	7.26	172	7.13	392	10.2
uk-union	48.5	2.29	45.9	1.48	37.6	1.34	60.4	1.99	128	5.4	131	4.05	101	3.46	177	5.47	664	32	462	16.7	383	14.5	718	25.6
Yahoo!	124	4.68	113	3.98	128	3.8	161	4.81	458	13.8	510	13.6	438	12.4	767	19	1390	36.4	1440	35.2	1250	32.7	2280	53.7

	Components								PageRank								Bellman-Ford							
	orig.		byte		byte-RLE		nibble		orig.		byte		byte-RLE		nibble		orig.		byte		byte-RLE		nibble	
	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})	(T_1)	(T_{40h})
randLocal	2.03	0.074	2.87	0.08	2.51	0.074	4.94	0.116	1.74	0.062	1.87	0.062	1.79	0.062	3	0.082	9.64	0.329	10.1	0.325	10.1	0.326	15.6	0.432
3D-grid	1.02	0.635	1.36	0.772	1.06	0.68	2	1.21	0.871	0.04	0.799	0.039	0.823	0.036	1.2	0.048	24.7	1.36	23.4	1.14	23.2	1.16	31.5	1.31
soc-LJ	2.37	0.074	3.32	0.083	2.84	0.075	5.5	0.132	1.85	0.059	1.88	0.061	1.79	0.055	3.11	0.088	3.63	0.139	4.87	0.141	4.38	0.138	6.99	0.203
cit-Patents	1.16	0.044	1.66	0.05	1.52	0.046	2.61	0.069	0.884	0.032	0.857	0.034	0.849	0.032	1.33	0.042	3.28	0.14	3.97	0.145	4.05	0.145	6.02	0.179
com-LJ	1.87	0.061	2.63	0.067	2.26	0.062	4.47	0.108	1.51	0.049	1.5	0.045	1.42	0.041	2.46	0.068	4.02	0.124	3.59	0.127	3.62	0.128	6.65	0.181
com-Orkut	3.7	0.108	4.31	0.119	3.74	0.094	8.52	0.223	4.53	0.158	4.16	0.146	3.98	0.144	7.44	0.236	5.55	0.241	6.26	0.246	5.78	0.228	13.2	0.427
nlpkkt240	10.6	0.547	14	0.596	10.5	0.49	23	0.927	7.36	0.24	6.89	0.224	8.09	0.226	9.2	0.269	142	4.88	144	4.46	141	4.43	265	6.91
Twitter	76.4	3.35	82.2	2.42	72.3	2.27	147	3.83	48.6	2.68	73.5	2.74	70.5	2.66	95.2	3.15	41.3	1.14	50.2	1.11	34.9	1.06	65.7	1.68
uk-union	71.1	5.57	53.2	2.73	45.7	2.61	76.1	3.9	74.4	4.89	56.9	2.26	52.3	2.24	64.8	2.65	42.9	2.9	45.1	1.74	42.8	1.53	63.8	2.34
Yahoo!	307	12.1	309	10.7	271	9.84	500	15.8	263	8.2	258	7.73	238	7.39	347	9.79	176	6.28	225	6.54	210	6.11	331	8.92

Table 8.2: Sequential (T_1) and parallel (T_{40h}) times (seconds) on a 40-core machine with hyper-threading on different applications for the original Ligra (orig.), Ligra+ using byte coding (byte), byte coding with run-length encoding (byte-RLE), and nibble coding (nibble).



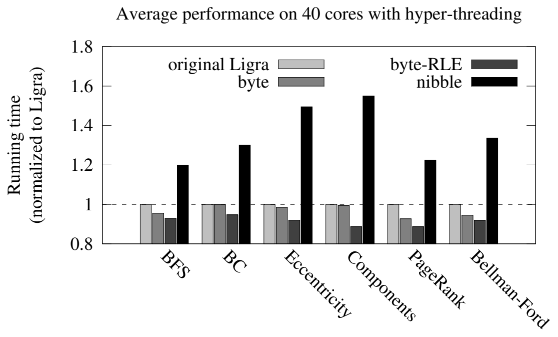


Figure 8.6: Average performance of Ligra+ relative to Ligra for each application on a single-thread (**left**) and on 40 cores with hyper-threading (**right**).

PageRank, but in parallel, Ligra+ with byte-RLE or byte codes is faster on all applications. In parallel, Ligra+ using nibble codes is still generally slower than Ligra due to the high overhead of decoding, but not by as much as on a single thread (see Figure 8.6). Decoding nibble codes is slower than decoding byte and byte-RLE codes because the operations are not on byte-aligned memory addresses. Ligra+ with byte-RLE codes is generally faster than with byte codes because there is a lower decoding overhead.

Graph algorithms are memory-bound, and the reason for the improvement in the parallel setting is because memory is more of a bottleneck in parallel than in the sequential case, and so the reduced memory footprint of Ligra+ is important in reducing the effect of the memory bottleneck. In addition, the decoding overhead is lower in parallel than sequentially because it gets better parallel speedup relative to the rest of the computation.

Overall, Ligra+ is at most 1.1x slower and up to 2.2x faster than Ligra on 40 cores with hyper-threading. *On average, over all applications and inputs, Ligra+ using byte-RLE codes is about 14% faster than Ligra in parallel and about 8% faster using byte codes.* In parallel, Ligra+ using nibble codes is about 35% slower than Ligra on average. The graphs with better compression (e.g., nlpkkt240 and uk-union) tend to have better performance in Ligra+. For the larger graphs, Ligra+ outperforms Ligra in most cases because vertices tend to have higher degrees and neighbors no longer fit on a cache line, making the reduced memory footprint a more significant benefit. Sequentially, Ligra+ is slower than Ligra by about 3%, 13%, and 73% on average when using byte-RLE, byte, and nibble codes, respectively.

Figure 8.7 plots the average parallel self-relative speedups (T_1/T_{40h}) over all inputs for each of the coding schemes per application. Both Ligra and Ligra+ achieve good speedups on the applications—at least a factor of 20 for Ligra and 25 for Ligra+. The three compression schemes all achieve better speedup than Ligra. Again, this is because compression alleviates the memory bottleneck which is a bigger issue in parallel, and the

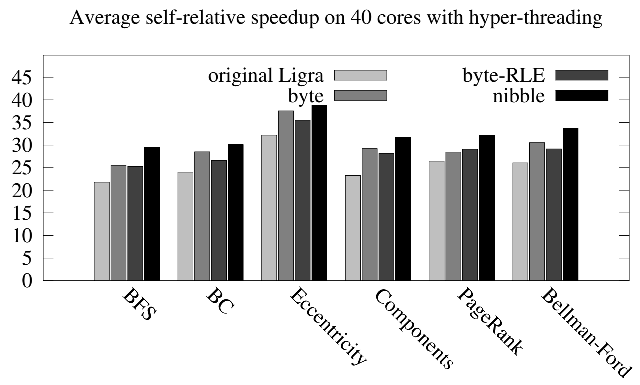


Figure 8.7: Average self-relative speedup over all inputs for each application on 40 cores with hyper-threading of Ligra and Ligra+.

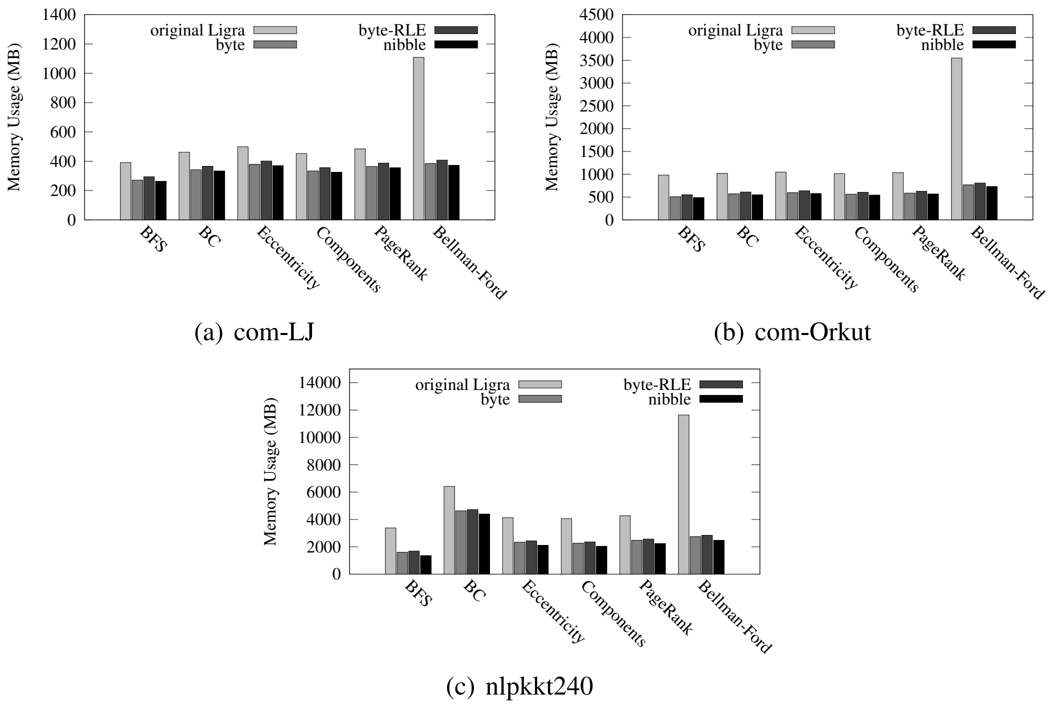


Figure 8.8: Peak memory usage of graph algorithms on com-LJ, com-Orkut and nlpkkt240 in Ligra and Ligra+.

overhead of decoding is lower because it has better parallel scalability relative to the rest of the computation.

Memory Usage. Figure 8.8 plots the peak memory usage of the applications using Ligra and Ligra+ for several graphs. For all graphs, Ligra+ has a lower peak memory usage than Ligra. Since the applications use auxiliary data structures of size proportional to the number of vertices, for graphs with a low vertex-to-edge ratio (e.g., com-Orkut and nlpkkt240), there is a significant saving in memory usage with Ligra+ compared to Ligra, and for other graphs the saving is lower.

8.4.1 Experimental Analysis of Graph Reordering Algorithms

As discussed by Blandford et al. [45], vertex ordering in graphs can affect compression quality and cache performance of graph algorithms. Graph orderings have also been studied in the context of sparse matrix computations to reduce the number of arithmetic operations and memory usage (fill-in). It is also used in the spMV algorithm of [61] to improve compression. This section discusses several graph reordering algorithms that we experimented with. There are also other reordering techniques not discussed in this section, which are mostly designed specifically for Web and social network graphs (see, e.g., [70, 68, 390, 93, 403]).

Depth-first search (dfs). The numbering of the vertices is determined by the depth-first traversal of the vertices starting from an arbitrary vertex. Pre-order, in-order and post-order traversals can all be used, and we found the compression quality to be very similar in all three cases.

Breadth-first search (bfs). The numbering of the vertices is determined by the breadth-first traversal of the vertices starting from an arbitrary vertex. This technique was suggested by Apostolico and Dronvandi [12] for Web graphs.

Hybrid depth-first/breadth-first search (hybrid). BFS tends to label children of vertices close together, leading to good difference compression between neighbors of a vertex, while DFS tends to label subtrees of vertices close together, leading to good difference compression between source and target vertex. We tried a hybrid approach using properties of both BFS and DFS. In particular, the vertices are visited in DFS order, but the children of each vertex are labeled with consecutive IDs before recursively calling DFS on the children. The motivation here is to exploit properties of both BFS and DFS to obtain small differences between source and target vertices, and neighbors of the same vertex.

Recursive breadth-first search (bfs-r). This approach is described by Blandford et al. [45]. It first performs a BFS from an arbitrary vertex, finds the furthest vertex from that starting vertex, and performs a BFS from the furthest vertex until half of the vertices are visited. This partitions the vertices into two halves, and the algorithm assigns a consecutive range of the indices to each half and recurses on each half.

METIS. We use the ordering program in the graph partitioning software METIS [261]. The method is based on multi-level graph partitioning, and recursively partitions a graph using a separator algorithm, assigning consecutive IDs to each half, and recursing on each half. We also tried the reordering program in Scotch [374], and found the quality to be very close to that of METIS.

A Parallel Separator-Based Reordering Algorithm (p-sep). Blandford et al. [45] describe a separator-based method in their paper, and we develop a parallel version of it in this section. Let us first review the sequential algorithm of Blandford et al. [45].

The algorithm of Blandford et al. [45] repeatedly coarsens a graph by contracting edges until a single vertex remains, building a separator tree, in which every two vertices contracted together become the children of the new vertex. To choose which edges to contract at each step, it uses the metric $w(E_{AB})/(s(A)s(B))$ where $w(E_{AB})$ is the weight of the edge between vertices A and B, and $s(A)$ and $s(B)$ are the weights of A and B, respectively. Initially all vertices and edges have a weight of 1. When contracting two vertices A and B, the resulting vertex is assigned a weight of $s(A) + s(B)$, and when there are multiple edges between two vertices after contracting, a single edge with weight equal to the sum of the edges is kept. The new ordering is then generated by an in-order traversal of the leaves of the separator tree. Blandford et al. apply a child-flipping optimization, in which the children of two siblings in the separator tree are rearranged if it leads to a better ordering.

A parallel version of the Blandford et al. algorithm that we develop is described next. The contraction of edges at each step can be done in parallel as long as any single vertex only participates in at most one contraction. To guarantee this, the parallel algorithm first selects an edge for each vertex which maximizes the metric used by Blandford et al., creating a sub-graph (which is a forest). A parallel maximal matching algorithm optimized for forests (based on the maximal matching algorithm developed in Chapter 4) is then executed on the sub-graph. The resulting maximal matching determines the edges which will be contracted in a phase. The algorithm then contracts the edges, relabels the vertices, and relabels the remaining edges with their new endpoints. Contracted vertices have their weights added together. Duplicate edges between vertices after contraction have their weights summed together. This is done by inserting the edges into a parallel hash table (the algorithm uses the hash table developed in Chapter 5), and if an edge already exists in the table an atomic fetch-and-add is used to add the weight to the existing edge in the table. The maximum number of levels of recursion can be controlled (10,000 in the experiments), as for graphs with skewed degree distribution, very few vertices are contracted per level of recursion. The parallel algorithm does not apply the child-flipping optimization of Blandford et al. [45]. As in the sequential algorithm, an in-order traversal of the leaves of the separator tree generated gives the vertex ordering.

Measures of Locality/Compression. Two useful statistics of the degree of locality, which correlate with the compression quality of a graph, are the average log cost and average log gap cost. The ***log cost*** of an edge (v, u) is defined to be the logarithm (base 2) of the absolute difference between u and v, i.e. $\log_2 |u - v|$. The ***average log cost*** of a graph is average log cost over all edges in the graph, i.e. $(1/m)\sum_{(u,v)\in E}\log_2|u-v|$. If the adjacency list $\{v_0,\ldots,v_{deg(v)-1}\}$ of each vertex v are sorted in ascending order, then the ***log gap cost*** of an edge (v,v_i) is defined to be $\log_2|v_i-v|$ if $i=0$ and $\log_2|v_i-v_{i-1}|$ otherwise. The ***average log gap cost*** of a graph is the average log gap cost over all edges in the graph, i.e. $(1/m)\sum_{v\in V'}(\log_2|v_0-v|+\sum_{i=1}^{deg(v)-1}\log_2|v_i-v_{i-1}|)$, where V' contains all vertices in V with non-zero degree.

Compression Statistics. Table 8.3 shows the average log gap cost (***gap***) and average log cost (***log***) of the reordered graphs using each algorithm described in this section, with the lowest average log gap cost per graph shown in bold. The average log gap cost is a more accurate indicator of compression performance in Ligra+ since it uses difference encoding between consecutive edges. The reordering algorithms were applied on the graphs with the original ordering. Also shown in Table 8.3 are compression statistics for the original ordering (***orig.***) and a random ordering of the vertices (***rand.***).

For the randLocal and 3D-grid graphs, the graph generator generates an ordering with good locality already, so the reordering algorithms are not applied. Due to the high memory requirements of the parallel separator code and METIS, and the high running time of bfs-r, we were unable to obtain compression statistics for these reordering algorithms on the large uk-union and Yahoo! graphs. For the Twitter graph, none of the reordering algorithms gave a better average log gap cost than the original ordering. The timing experiments in Table 8.2 and Figures 8.6 and 8.7 use the ordering which give the best average log gap cost as this also corresponded to the fewest bits per edge. Note that the compression rates shown in Figure 8.5 are higher than the average log gap cost because the compression schemes requires each edge to be byte- or nibble-aligned, therefore possibly wasting some bits per edge.

Orderings with low average log gap and average log costs have more locality (i.e., the IDs of vertices and their neighbor are close to each other), which lead to improvements in performance even without using compression, due to incurring fewer cache misses. In other words, reordering the graphs improves performance for the uncompressed graphs using Ligra as well. Furthermore, Ligra+ still reduces the space usage even without applying graph reordering, as our experiments confirmed that the average bits per edge for the original ordering using the various compression schemes is still much lower than 32. Therefore, while graph reordering can help with compression, it is not necessary to obtain reduced space usage. This section experiments with a broad set of graph reordering algorithms, but there are certainly other algorithms and variants that can be experimented with, possibly giving even better compression statistics. A further study is left for future work.

Input Graph	gap	log	gap	log	gap	log	gap	log	gap	log	gap	log	gap	log	gap	log
Ordering	orig.	orig.	rand.	rand.	p-sep	p-sep	dfs	dfs	bfs	bfs	hybrid	hybrid	bfs-r	bfs-r	METIS	METIS
randLocal	**6.88**	6.74	–	–	–	–	–	–	–	–	–	–	–	–	–	–
3D-grid	**10.6**	8.12	–	–	–	–	–	–	–	–	–	–	–	–	–	–
soc-LJ	10.6	16.97	15.71	20.05	**8.08**	12.18	9.86	16.16	10.67	16.96	9.64	15.3	10.36	16.48	9.39	15.2
cit-Patents	16.43	19.48	17.97	20.35	**8.57**	10.1	11.7	16.37	12.3	17.53	11.66	15.09	13.0	16.39	10.25	13.98
com-LJ	10.28	16.13	15.65	19.78	**7.95**	11.84	9.71	15.83	10.84	16.93	9.52	14.91	10.34	16.19	9.33	14.93
com-Orkut	10.42	17.5	13.61	19.39	**8.58**	14.53	10.09	17.7	10.35	17.85	9.87	17.26	10.16	17.74	10.03	16.85
nlpkkt240	4.49	23.74	19.28	22.57	4.13	8.18	5.1	14.27	4.02	17.44	3.81	11.17	**3.15**	8.56	3.87	10.61
Twitter	**9.23**	18.76	15.22	23.14	12.12	20.64	12.16	22.17	10.6	22.15	11.59	21.69	10.74	21.01	11.01	20.97
uk-union	3.14	11.44	17.08	24.83	–	–	3.0	13.39	3.01	18.62	**2.31**	14.41	–	–	–	–
Yahoo!	7.6	24.56	21.33	28.22	–	–	6.56	18.09	7.14	23.34	**6.22**	17.66	–	–	–	–

Table 8.3: Average log cost and average log gap cost of graphs using various reordering algorithms. The lowest average log gap cost per graph is shown in bold.

Part III

Parallel Graph Algorithms

Introduction

Chapter 9 presents the first linear-work (work-efficient) and polylogarithmic-depth parallel algorithm for graph connectivity that is also practical. The chapter describes several implementation variants of the algorithm, and shows experimentally that the fastest implementation is competitive with the fastest existing parallel connectivity implementations (which are not theoretically linear-work and polylogarithmic-depth) and does not have "worst-case" inputs due to its theoretical guarantees. Chapter 10 presents the design and implementation of simple, fast, and cache-efficient shared-memory algorithms for exact, as well as approximate, triangle counting and other triangle computations. In addition, the chapter proves strong asymptotic bounds on the work, depth, and cache complexity of the solutions. A comprehensive experimental evaluation shows that the implementations scale to the largest publicly available real-world graphs, obtain excellent parallel scalability on multicore machines, and are significantly faster than previous parallel solutions for the same problem.

The results in this part of the thesis have appeared in the following publications:

- Julian Shun, Laxman Dhulipala and Guy Blelloch. A Simple and Practical Linear-Work Parallel Algorithm for Connectivity. *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 143–153, 2014.
- Julian Shun and Kanat Tangwongsan. Multicore Triangle Computations Without Tuning. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 149–160, 2015.

Chapter 9

Linear-Work Parallel Graph Connectivity

9.1 Introduction

Finding the connected components of a graph is a fundamental problem in computer science that has been well-studied (see Section 2.6 for the definition), having many important applications such as in VLSI design and image analysis for computer vision. Sequentially, connectivity can be easily implemented in linear work using breadth-first search (BFS) or depth-first search, or nearly linear work with union-find. On the other hand, computing connected components and spanning forests[1] in parallel has been a long studied problem [6, 18, 94, 97, 106, 213, 228, 241, 246, 252, 275, 279, 300, 345, 355, 375, 376, 394, 415, 454]. Some of the parallel algorithms developed are relatively simple, but require super-linear work. The algorithms of Shiloach and Vishkin [415] and Awerbuch and Shiloach [18] work by combining the vertices into trees such that at the end of the algorithm vertices in the same component will belong to the same tree. These algorithms guarantee that the number of trees decreases by a constant factor in each iteration, but do not guarantee that a constant fraction of the edges are removed, and thus require $O(m \log n)$ work. The random mate algorithms of Reif [394] and Phillips [376] work by contracting vertices in the same component together and guarantee that a constant fraction of the vertices decrease in expectation per iteration, but again do not guarantee that a constant fraction of the edges are removed. Therefore, these algorithms also require $O(m \log n)$ expected work and are not work-efficient.

Work-efficient polylogarithmic-depth parallel connectivity algorithms have been designed in theory [105, 163, 208, 209, 375, 380]. These algorithms are based on random

[1]A spanning forest algorithm can be used to compute connected components.

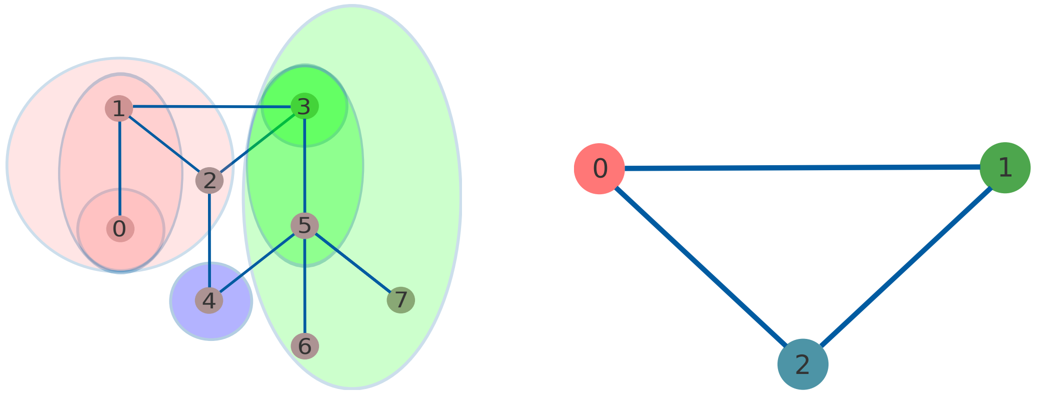
edge sampling [163, 208, 209] or linear-work minimum spanning forest algorithms, which also involve sampling and filtering edges [105, 375, 380]. However, these algorithms are complicated and unlikely to be practical (there are no implementations of these algorithms available).

There has also been significant experimental work on parallel connectivity algorithms in the past. Hambrusch and TeWinkel [211] implement connected component algorithms on the Massively Parallel Processor (MPP). Greiner [198] implements and compares parallel connectivity algorithms using NESL [49]. Goddard et al. [177], Hsu et al. [232], Bader et al. [24, 20], Patwary et al. [372], Shun et al. [424], Slota et al. [434], and the Galois system [351] implement algorithms for shared-memory CPUs. Bus and Tvrdik [82], Krishnamurthy et al. [278], Bader and JaJa [21] and Caceres et al. [83] implement connected components algorithms for distributed-memory machines. There has been some recent work on designing connectivity algorithms for GPUs [220, 435, 27]. There have also been connectivity algorithms that require time proportional to the diameter of the graph in recent graph processing packages [250, 289, 290, 420]. None of the previous parallel algorithms implemented are theoretically work-efficient.

Note that a parallel BFS can be performed to visit the components of the graph one-by-one. While this approach is linear-work, the depth is proportional to the sum of the diameters of the connected components. Therefore this approach is not efficient as a general-purpose parallel connectivity algorithm, although it works well for low-diameter graphs with few connected components.

This chapter introduces a simple linear-work algorithm for connectivity requiring polylogarithmic depth, and experimentally show that it rivals the best existing parallel implementations for connectivity. The algorithm is the first work-efficient parallel graph connectivity algorithm with an implementation, and furthermore the implementation also performs well in practice.

The algorithm is based on a simple parallel algorithm for generating low-diameter decompositions of graphs by Miller et al. [334], which is an improvement of an algorithm by Blelloch et al. [60]. A low-diameter decomposition of a graph partitions the vertices, such that the diameter of each partition is small, and the number of edges between partitions is small [302]. Such decompositions have many uses in computer science, including in linear system solvers [60] and in metric embeddings [29]. The algorithm of Miller et al. partitions a graph such that the diameter of each partition is $O(\log(n)/\beta)$ and the number of edges between components is $O(\beta m)$ for $0 < \beta < 1$. It runs in linear work and $O(\log^2(n)/\beta)$ depth with high probability. Their algorithm is based on performing breadth-first searches from different starting vertices in parallel, with start times drawn from an exponential distribution. Due to properties of the exponential distribution, the algorithm only needs to run the multiple breadth-first searches for at most $O(\log(n)/\beta)$



(a) graph decomposition (b) contracted graph

Figure 9.1: Illustration of the decomposition-based connectivity algorithm. (a) At $t = 0$, vertex 0 starts a BFS (red ball), and at $t = 1$, vertices 3 (green ball) and 4 (blue ball) start BFS's. In this illustration, when there are ties (multiple BFS's visiting the same unvisited neighbor), the BFS center with the lowest ID wins. The balls represent the resulting partitions and the rings around the balls represent each level of the corresponding BFS. (b) Each ball is contracted into a single vertex, and the decomposition is applied recursively.

iterations before visiting all vertices.

My co-authors and I observe that this decomposition algorithm can be used to generate the connected components labeling of a graph. Our algorithm simply calls the decomposition algorithm recursively with β set to a constant fraction, and after each call contracts each partition into a single vertex, and relabels the vertices and edges between partitions. Since the number of edges decreases by a constant fraction in expectation in each recursive call, the algorithm terminates after $O(\log n)$ calls with high probability. This results in an algorithm for connected components labeling that runs in linear work and $O(\log^3 n)$ depth with high probability. An illustration of this algorithm is shown in Figure 9.1. Our implementation is based on parallel breadth-first searches and some simple parallel routines.

We also present a slight modification of the decomposition algorithm of Miller et al., which relaxes the relative ordering among vertices due to different breadth-first search start times. We show that this modification does not affect the asymptotic complexity of the decomposition algorithm, while leading to a simpler and faster implementation. We use this decomposition algorithm for connectivity and apply various optimizations to our implementations.

This chapter experimentally compares the decomposition-based connectivity algorithm against the fastest existing parallel connectivity implementations (which are not theoretically linear-work and polylogarithmic-depth) [424, 372, 420, 434] on a variety of input graphs and shows that the decomposition-based algorithm is competitive. On 40 cores, the parallel implementations achieve 18–39 times speedup over the same implementation run on a single thread, and achieve good speedups over the sequential implementations on many graphs. Experiments show that on most graphs, the number of edges decreases by significantly more than predicted by the theoretical bounds due to duplicate edges between components. In addition, the chapter presents experiments that study how the performance of the connectivity algorithms varies with different settings of β in the decomposition algorithms.

Contributions. The main contributions of this chapter are as follows. Firstly, a simple linear-work and polylogarithmic-depth parallel algorithm for connectivity is presented. This is the first practical parallel connectivity algorithm with a linear-work guarantee. Secondly, the chapter describes a (modest) variation of the parallel decomposition algorithm by Miller et al. that leads to a faster implementation and proves that it has the same theoretical guarantees as the original algorithm. Next, optimized implementations of the connectivity algorithm are presented. Finally, an experimental evaluation is performed, showing that the algorithm is competitive with the best previously available parallel implementations of graph connectivity, which are not linear-work and polylogarithmic-depth.

9.2 Linear-Work Low-Diameter Decomposition

The ***exponential distribution*** with parameter λ is defined by the probability density function:

$$f(x, \lambda) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The mean of the exponential distribution is $1/\lambda$.

A (β, d)***-decomposition*** $(0 < \beta < 1)$ of an undirected graph $G = (V, E)$ is a partition of V into subsets $V_1, \ldots, V_k$ such that (1) the shortest path between any two vertices in each V_i using only vertices in V_i is at most d, and (2) the number of edges $(u, v) \in E$ such that $u \in V_i$, $v \in V_j$, $i \neq j$ is at most βm.

Miller et al. present a parallel decomposition algorithm based on parallel BFS's [334], which this chapter refers to as DECOMP. They prove that for a value β, DECOMP generates a $(\beta, O(\log(n)/\beta))$ decomposition in $O(m)$ work and $O(\log^2(n)/\beta)$ depth with high probability on a CRCW PRAM. The algorithm works by assigning each vertex v a *shift value* δ_v drawn from an exponential distribution with parameter β (mean $1/\beta$). Miller et al. show that the maximum shift value is $O(\log(n)/\beta)$ w.h.p. Each vertex v is then assigned to the partition S_u that minimizes the *shifted distance* $\text{dist}_{-\delta}(u, v) = \text{dist}(u, v) - \delta_u$. This can be implemented by performing multiple BFS's in parallel. Each iteration of the implementation explores one level of each BFS and at iteration t (starting with $t = 0$) breadth-first searches are started from the unvisited vertices v such that $\delta_v \in [t, t + 1)$. If multiple

Algorithm 11 Parallel decomposition-based algorithm for connected components labeling

```
1: $\beta$ = some constant fraction in $(0, 1)$
2: procedure CC($G(V, E)$)
3:     $L$ = DECOMP($G(V, E)$, $\beta$)          ▷ $L$ contains the labels returned by DECOMP
4:     $G'(V', E')$ = CONTRACT($G(V, E)$, $L$)
5:     if $|E'| = 0$ then
6:         return $L$
7:     else
8:         $L'$ = CC($G'(V', E')$)
9:         $L''$ = RELABELUP($L$, $L'$)
10:        return $L''$
```

BFS's reach the same unvisited vertex w in the same time step, then w is assigned to the partition corresponding to the origin of the BFS with the smaller fractional portion of the shift value (equivalently, w is assigned to the partition whose origin has the smallest shifted distance to w). Since the maximum shift value is $O(\log(n)/\beta)$, the algorithm terminates in $O(\log(n)/\beta)$ iterations. Each iteration requires $O(\log n)$ depth for packing the frontiers of the BFS's, leading to an overall depth of $O(\log^2(n)/\beta)$ w.h.p. The BFS's are work-efficient, so the total work is $O(m)$.

9.3 Linear-Work Connectivity

This section introduces a simple linear-work parallel algorithm for connectivity. As a subroutine, it uses the parallel decomposition algorithm DECOMP described in Section 9.2. By the definition of a decomposition, the number of inter-component edges remaining after a call to DECOMP starting with m edges is at most βm in expectation. The algorithm contracts each component into a single vertex and recurses on the remaining graph, whose edge count has decreased by at least a constant factor in expectation. This leads to a linear-work parallel connectivity algorithm, assuming that the contraction and relabeling can be done efficiently.

The pseudocode for this connected components algorithm (CC) is shown in Algorithm 11. The input to DECOMP is a graph $G(V, E)$ and a value β, and the output is a labeling L of the vertices in V, such that vertices in the same partition will have the same label. CONTRACT takes a graph $G(V, E)$ and a labeling L as input, and returns a new graph $G'(V', E')$ such that vertices with the same label in V according to L are contracted into a single vertex, forming the vertex set V', and the inter-component edges in E are relabeled according to L and form the edge set E'. RELABELUP takes as input labelings L and L' and returns a new labeling L'' such that $L''[i] = L'[L[i]]$. RELABELUP is necessary because the original labels L must be updated with the labels L' returned by the recursive call to CC.

Theorem 21. *Algorithm 11 runs in $O(m)$ expected work and $O(\log^3 n)$ depth with high probability.*

Proof. The algorithm sets β to a constant between 0 and 1. Since the number of edges decreases to at most βm in expectation after each recursive call, and the rate of reduction is independent across iterations, the total number of calls is $O(\alpha \log_{1/\beta} m)$ with probability at least $1 - 1/m^{\alpha-1}$ for some constant $\alpha > 1$. Each recursive call requires $O(\gamma \log^2(n)/\beta)$ depth with probability at least $1 - 1/n^{\gamma-1}$ for some constant $\gamma > 1$ and $O(m')$ work where m' is the number of remaining edges for DECOMP [334]. Hence the total contribution of DECOMP to the depth of CC is $O(\delta \log_{1/\beta} m \log^2(n)/\beta)$ = $O(\log^3 n)$ with probability at least $1 - 1/m^{\delta}$ for some large enough constant δ (depending on α amd γ), and the total contribution to the work of CC is upper bounded by $\sum_{i=0}^{\infty} \beta^i cm$ for some constant c, which is $O(m)$ in expectation.

Let us now walk through an implementation of DECOMP that allows contraction and relabeling to be done within the same complexity bounds. Recall that DECOMP performs multiple breadth-first searches in parallel, with each BFS corresponding to one of the components (partitions) of the graph. All BFS's can be maintained using a single frontier array, where vertices belonging to the same component are in consecutive positions in the frontier. On each iteration, vertices that need to start their own BFS are added to the end of this frontier array in parallel. The algorithm stores all of the frontiers created throughout one call to DECOMP, and there are $O(\log(n)/\beta)$ such frontiers w.h.p. Each individual BFS stores the starting and ending position of its component's vertices on each frontier, as well as the total number of edges for these vertices. Using this information, the algorithm can compute appropriate offsets into shared arrays for each component using prefix sums over all the $O(\log(n)/\beta)$ frontiers for each BFS. For each iteration of CC, the work for computing offsets is $O(m')$ where m' is the number of edges at the beginning of the iteration, and the depth is $O(\log(n)/\beta)$.

As a vertex visits other vertices during the BFS's, if it encounters an edge to a vertex belonging to the same component (an intra-component edge), it will mark that edge as deleted (using some special value). These edges will be packed out at the end of DECOMP, which can be done in $O(m')$ total work and $O(\log m')$ depth, where m' is the number of edges at the beginning of the iteration. The rest of the edges will be inter-component edges and hence need to be kept for the next iteration. Each component will become a single vertex in the next iteration, with all of the edges of the component vertices merged. The algorithm then creates a new edge array and the original vertices copy their edges into the new array (each vertex's offset into this array can be computed with a prefix sum), and since the vertices of each component are stored consecutively on the frontiers, this guarantees that the resulting array will store each component's edges consecutively. The algorithm then removes duplicate edges within the complexity bounds of an iteration using parallel hashing [324, 174], although the number of edges decreases by a constant factor in expectation even if duplicates are not removed.

To relabel the new vertices, the algorithm first computes the total number of components k and assigns each original label with a new label in the range $[0, \ldots, k-1]$, which can be done using prefix sums. Singleton vertices are then removed, but their labels are kept. For the k' non-singleton vertices remaining, the algorithm relabels them to the range $[0, \ldots, k'-1]$ and recursively call CC. After the recursive call, the original labels are relabeled according to the result of CC. This can all be done using prefix sums in linear work in the number of remaining vertices and $O(\log n)$ depth per iteration.

Let us summarize the proof of this theorem. For a constant fraction β, there are $O(\log n)$ calls to DECOMP w.h.p., each of which does $O(\log n)$ iterations of BFS. Each iteration of BFS requires $O(\log n)$ depth for packing. The depth for contraction and relabeling is absorbed by the depth of DECOMP. This gives an overall depth of $O(\log^3 n)$ with high probability. DECOMP, contraction and relabeling can be done work-efficiently, and each call to DECOMP decreases the number of edges by a constant fraction in expectation, leading to $O(m)$ expected work overall. □

This algorithm can be implemented on the CRCW PRAM as there is $O(\log n)$ parallel slackness per iteration, which is enough to do processor allocation with prefix sums. Theoretically the depth of DECOMP could be improved to $O(\log n \log^* n)$ by using approximate compaction [174] (which is linear-work) for packing the frontiers of the BFS's, as well as processor allocation on the CRCW PRAM. The depth of the connectivity algorithm can further be improved by running just $O(\log \log n)$ iterations of the algorithm, at which point there are $O(m/\log n)$ edges remaining, and an algorithm with $O(m \log n)$ work and $O(\log n)$ depth (e.g. [394, 376]) is used. This gives a (modified) algorithm with expected linear work and $O(\log n \log \log n \log^* n)$ depth w.h.p.

Let us consider a slight variation of DECOMP which breaks ties arbitrarily among frontier vertices visiting the same unvisited neighbor in a given iteration of the BFS's. This modification simplifies the implementation of the algorithm and leads to improved performance as discussed later in the chapter. This variation is equivalent to rounding down all the δ_v values to the nearest integer and again assigning each vertex v to the partition S_u that minimizes $\text{dist}_{-\delta}(u, v) = \text{dist}(u, v) - \delta_u$, but breaking ties arbitrarily. This chapter refers to this version as ***Decomp-Arb*** and shows that this modified version has the same theoretical guarantees (within a constant factor). In particular, the number of inter-component edges in the decomposition is shown to be at most $2\beta m$ in expectation (the original bound was βm).

Theorem 22. *Decomp-Arb generates a $O(2\beta, O(\log(n)/\beta))$ decomposition in $O(m)$ expected work and $O(\log^2(n)/\beta)$ depth with high probability.*

Proof. Since the algorithm still picks values from an exponential distribution, the diameter of each component is $O(\gamma \log(n)/\beta)$ with probability at least $1 - 1/n^{\gamma-1}$ as shown in [334]. Hence the depth of the algorithm is the same as the original algorithm, namely $O(\log^2(n)/\beta)$ w.h.p. The work is still $O(m)$ in expectation, since the BFS's are work-efficient. What remains is to show that the number of inter-component edges is at most $2\beta m$ in expectation.

As in [334], consider the midpoint w of an edge (u, v). Lemma 4.3 of [334] states that if u and v belong to different components, then $dist_{-\delta}(u', w)$ and $dist_{-\delta}(v', w)$ are within 1 of the minimum shifted distance to w. Decomp-Arb rounds all shifted distances down to the nearest integer. Hence when comparing two rounded shift distances, their difference is at most 1 if and only if the two original shift distances were within 2 of each other. In other words, suppose the two distances being compared are d_1 and d_2. Then $|\lfloor d_2 \rfloor - \lfloor d_1 \rfloor| \leq 1$ if and only if $|d_2 - d_1| < 2$. Hence Lemma 4.3 of [334] can be modified to state that if u and v belong to different components, then $dist_{-\delta}(u', w)$ and $dist_{-\delta}(v', w)$ (using the original shift distances) are within 2 of the minimum shifted distance to w.

Lemma 4.4 of [334] uses properties of the exponential distribution to show that the probability that the smallest and second smallest shifted distance to w (corresponding to the first two BFS's that arrive at w) has a difference of less than c is at most βc. In this case, $c = 2$, so the probability that an edge is an inter-component edge is at most 2β. By linearity of expectations, the expected total number of inter-component edges is at most $2\beta m$. $\square$

Plugging in Decomp-Arb into the proof of Theorem 21 results in a linear-work parallel connectivity algorithm for $0 < \beta < 1/2$.

9.4 Implementation Details

This section describes the algorithmic engineering efforts to obtain a fast implementation of Algorithm 11. The section describes three versions of DECOMP, referring to the original algorithm as Decomp-Min, the version which breaks ties arbitrarily as Decomp-Arb, and a variant of Decomp-Arb that will be discussed later as Decomp-Arb-Hybrid.

The implementation uses the adjacency list format for graph representation, discussed in Section 2.4, where the array V stores offsets into an array of edges E. The targets of the outgoing edges of vertex i are then stored in $E[V[i]], \ldots, E[V[i+1]] - 1$ (to deal with the edge case, $V[n]$ is set to m). The graphs are undirected so each edge is stored in both directions. The implementation also maintains an array D, where $D[i]$ stores the degree of the i'th vertex. Initially $D[i]$ is set to $V[i+1] - V[i]$, and is updated during the algorithm to avoid revisiting edges when possible.

As suggested in [334], the implementations simulate the assignment of values from the exponential distribution to vertices by generating a random permutation (in parallel), and in each round adding chunks of vertices starting from the beginning of the permutation as start centers for new BFS's, where the chunk size grows exponentially. If a vertex in a chunk has already been visited, then it is not added as a start center. Each vertex also draws a random integer from a large enough range to simulate the fractional part of its shift value (denoted by δ'_v for vertex v), used to break ties if multiple BFS's visit the same unvisited neighbor. The active frontier of the BFS's is maintained using a single array. New BFS centers are simply added to the end of this array in parallel. Note that parallel BFS can also be implemented using Cilk reducers [296] with similar performance.

Since the algorithm does not need to keep around the inter-component edges in recursive calls to CC, the inter-component edges are packed out as they are encountered. Therefore as vertices are explored, the incident edge to the explored vertex is determined on-the-fly whether it is an inter-component edge or an intra-component edge.

In contrast to the description in the proof of Theorem 21, the implementations do not store the frontiers of the BFS's and offsets of each BFS into the frontiers. Therefore the vertices of the same component will not be able to be accessed contiguously in memory. Instead, in the contraction phase an integer sort is used to collect all the vertices of the same component together. Experimentally, this was found to be more efficient than the method described in the proof of Theorem 21 because the amount of bookkeeping is reduced and the integer sort is only performed over the remaining inter-component edges, which is usually much fewer than the number of original edges. The implementations use the $O(m/\epsilon)$ work and $O((1/\epsilon)m^{\epsilon})$ depth ($0 < \epsilon < 1$) integer sorting algorithm from the Problem Based Benchmark Suite.

The first implementation, ***Decomp-Min***, is split into two phases over the frontier vertices (pseudocode shown in Algorithm 12). In the implementation, an array C is used to store both the component ID's of the vertices and to store the values that vertices write to resolve conflicts. In particular, the array C stores pairs (c_1, c_2) where for a vertex v, c_1 is used for markings from frontier vertices competing to visit v, and c_2 stores the component ID of vertex v. The pseudocode uses $C_1[v]$ and $C_2[v]$ to refer to the first and second value of the pair $C[v]$, respectively. Decomp-Min uses the ***writeMin*** operation, which is an instantiation of the priority update described in Chapter 6. The element type of writeMin is an integer pair, and the comparison function (not shown in the pseudocode) uses integer "less-than" comparison on the first value of pair. Note that instead of keeping pairs in C, the implementation could keep two arrays, one to store the component IDs and the other to resolve conflicts, but this leads to an additional cache miss per vertex visit.

The entries of C are initialized to (∞, ∞) on Line 1. The ∞ in the second value of the pair indicates that the vertex has not yet been visited, and the first value of the pair is the identity value for the writeMin function. When a vertex v is added to the BFS on Lines 5–6 (i.e., it starts a new BFS), $C[v]$ is set to $(-1, v)$—the value -1 in $C_1[v]$ indicates that v has been visited, and the value v in $C_2[v]$ indicates that the component ID of v is its own

Algorithm 12 Decomp-Min

```
1: $C = \{(\infty,\infty),\dots,(\infty,\infty)\}$
2: Frontier $= \{\}$
3: numVisited $= 0$
4: **while** (numVisited $< n$) **do**
5:     add to Frontier unvisited vertices $v$ with $\delta_v <$ round $+ 1$
6:       and set $C[v] = (-1, v)$                                    ▷ new BFS centers
7:     numVisited = numVisited + size(Frontier)
8:     NextFrontier $= \{\}$
9:     **parfor** $v \in$ Frontier **do**
10:        start $= V[v]$                                              ▷ start index of edges in $E$
11:        $k = 0$
12:        **for** $i = 0$ to $D[v] - 1$ **do**
13:            $w = E[\text{start} + i]$
14:            **if** $C_1[w] \neq -1$ **then**
15:                **if** $C_1[w] > \delta'_{C_2[v]}$ **then**
16:                    writeMin(&$C[w]$,($\delta'_{C_2[v]}, C_2[v]$))
17:                $E[\text{start} + k] = w$
18:                $k = k + 1$
19:            **else**
20:                **if** $C_2[w] \neq C_2[v]$ **then**
21:                    $E[\text{start} + k] = -C_2[w] - 1$
22:                    $k = k + 1$
23:        $D[v] = k$
24:    **parfor** $v \in$ Frontier **do**
25:        start $= V[v]$                                              ▷ start index of edges in $E$
26:        $k = 0$
27:        **for** $i = 0$ to $D[v] - 1$ **do**
28:            $w = E[\text{start} + i]$
29:            **if** $w \geq 0$ **then**
30:                **if** $C_1[w] = \delta'_{C_2[v]}$ and CAS(&$C_1[w], \delta'_{C_2[v]}, -1$) **then**
31:                    add $w$ to NextFrontier                         ▷ $v$ won on $w$
32:                **else**
33:                    **if** $C_2[w] \neq C_2[v]$ **then**
34:                        $E[\text{start} + k] = -C_2[w] - 1$
35:                        $k = k + 1$
36:            **else**
37:                $E[\text{start} + k] = w$
38:                $k = k + 1$
39:        $D[v] = k$
40:    NextFrontier = Frontier
```

vertex ID. In the implementation, inter-component edges are kept while intra-component edges are deleted on-the-fly. The implementation overwrites the edge array E as it loops over the edges (Lines 17–18 and 21–22) using a counter k indicating the current position in the array (Line 11). In the first phase, frontier vertices mark unvisited neighbors using the writeMin primitive (Lines 14–16) with the fractional part of its BFS center's shift value, $\delta'_{C_2[v]}$ (the BFS center's ID is equal to $C_2[v]$, the component ID of v). The code assumes that there are no ties as the numbers can be drawn from a large enough range to guarantee this w.h.p. Also, as long as for a neighbor w, $C_1[w] \neq -1$, this means the neighbor has not been visited in a *previous iteration*. In this case, the edge needs to be kept (Lines 17–18) as it is not currently known whether it is an intra- or inter-component edge (this can only be determined once all other frontier vertices finish doing their writeMin's). Otherwise, the neighbor w has been visited in a previous iteration and the status of the edge to w can be determined—if w has a component label different from v, then it keeps the edge as it is an inter-component edge (Lines 20–22). It labels the endpoint of the edge with its new component ID (so that it does not have to be relabeled later) but sets the sign bit of the value (negates it and subtracts 1) to indicate that this edge need not be considered again in the second phase. Otherwise, the edge is an intra-component edge and is deleted. The degree of v is set to be the number of edges kept in this phase (Line 23).

In the second phase, the remaining edges incident on v are looped over and for edges which have a non-negative value (an edge whose status has not yet been determined from the first phase), the implementation determines whether $\delta'_{C_2[v]}$ is stored on the neighbor w. If so, then v uses a compare-and-swap (CAS) to attempt to atomically set $C_1[w]$ to -1 (so that future writeMin's will not mark it again), and if successful adds w to the next frontier (Lines 30–31) and does not keep the edge (it is an intra-component edge). A CAS is required here since there could be multiple vertices from the same component exploring the same neighbor w (they all have the same $\delta'_{C_2[v]}$ value), and w should be added only once to the next frontier. If the condition on Line 30 does not hold, then the implementation checks whether the component ID of w matches that of v, and if they differ, then the edge is an inter-component edge and is kept (Lines 32–35). The sign bit of the value of its component ID is set and stored it in E (Lines 34–35). If $C_2[w] = C_2[v]$, then (v, w) is an intra-component edge and is not kept. If the edge has a negative value, then it was already processed in the first phase, and is kept (Lines 36–38). The degree of v is set to be the number of inter-component edges incident on v (Line 39). After the BFS's are finished, the sign bits of the remaining (inter-component) edges are unset, so that they can be properly processed during the relabeling phase after the call to DECOMP by the connected components algorithm.

Note that for high-degree vertices (e.g., degree greater than $k \log n$ for some constant k), the inner sequential for-loops over the neighbors of a vertex can be replaced with a parallel for-loop, marking the deleted edges with a special value and packing the edges with a parallel prefix sums after the for-loop.

Algorithm 13 Decomp-Arb

```
1: $C = \{\infty, \ldots, \infty\}$
2: Frontier = {}
3: numVisited = 0
4: while (numVisited $< n$) do
5:     add to Frontier unvisited vertices $v$ with $\delta_v <$ round + 1
6:       and set $C[v] = v$                                   ▷ new BFS centers
7:     numVisited = numVisited + size(Frontier)
8:     NextFrontier = {}
9:     parfor $v \in$ Frontier do
10:        start = $V[v]$                                    ▷ start index of edges in $E$
11:        $k = 0$
12:        for $i = 0$ to $D[v] - 1$ do
13:            $w = E[\text{start} + i]$
14:            if $C[w] = \infty$ and CAS($\&C[w], \infty, C[v]$) then
15:                add $w$ to NextFrontier
16:            else
17:                if $C[w] \neq C[v]$ then                  ▷ inter-component edge
18:                    $E[\text{start} + k] = C[w]$
19:                    $k = k + 1$
20:        $D[v] = k$
21:    NextFrontier = Frontier
```

Decomp-Min is split into two phases because it needs all the vertices to apply the writeMin on their unvisited neighbors before it can determine a winner. Hence, a synchronization point is needed between the writeMin's and the checks to see if a vertex successfully visits a neighbor. Decomp-Arb, another implementation of the decomposition algorithm that only requires one phase, is described next.

In contrast to Decomp-Min, ***Decomp-Arb*** only requires one phase over the edges of the frontier vertices and their outgoing edges (pseudocode shown in Algorithm 13). Here C stores only a single integer value, indicating the component ID's of the vertices. Each entry is initialized to ∞ (Line 1) to indicate that the vertex has not yet been visited. The code of Decomp-Arb is similar to that of Decomp-Min, except that there is only a single phase over the edges of each frontier. Instead of using a writeMin as in Decomp-Min, Decomp-Arb uses a CAS to mark an unvisited neighbor (Line 14) with the component ID of the frontier vertex. A vertex that successfully marks a neighbor can delete its edge to that neighbor since it is guaranteed to be an intra-component edge. That vertex is also responsible for adding the neighbor to the next frontier (Line 15). Otherwise, the vertex checks the component ID of its neighbor and if it differs from its own, it keeps the edge as an inter-component edge (Lines 17–19). It also marks the endpoint of the edge with its component ID so that it doesn't have to be relabeled later (Line 18). Note that although the pseudocode shown does not make use of the fact that the degree is set to the number of inter-component edges on Line 20, it is used during the relabeling phase (not shown in the pseudocode). Unlike in Decomp-Min, Decomp-Arb does not need to use the fractional part of the shift values (the δ'_v values) because an arbitrary BFS can mark an unvisited neighbor.

Decomp-Arb only requires a single phase over the edges of the frontier vertices because once a vertex w is visited by some vertex v and its component ID is set to the component ID of v, it can no longer be visited again by another vertex. At that point the implementation knows that the edge from v to w is an intra-component edge and can delete it, and any other neighbor of w with a different component ID than w that fails to mark w with the CAS has an inter-component edge to w which is kept.

During the relabeling phase, the implementations only needs to relabel the source endpoint of each remaining edge, as the target endpoint was already relabeled during DECOMP. After relabeling, the parallel hash table [421] from Chapter 5 is used to remove duplicate edges between components. On the way back up from the recursive call to CC, the implementations simply index into the labeling returned by CC with a parallel for-loop to relabel the original labels appropriately (corresponding to RELABELUP of Algorithm 11).

As shown experimentally in Section 9.5, Decomp-Arb performs better than Decomp-Min due to only requiring one pass over the edges of each frontier during the BFS's, and needing less bookkeeping overall.

We also considered the *direction-optimizing* (hybrid) BFS idea first described by Beamer et al. [32] and later implemented for general graph traversal algorithms in Ligra [420] (see Chapter 7). In BFS, the idea is that when the frontier is large, it is cheaper to have all unvisited vertices read their incoming neighbors and once a vertex finds a neighbor on the frontier, it chooses it as its parent and quits (subsequent incoming edges to this vertex do not need to be examined). If a large number of vertices' neighbors are on the frontier, then this possibly saves many edge traversals.

In contrast to a standard BFS, the connectivity algorithm presented in this chapter requires all edges to be inspected, since it must decide whether each edge is an inter-component or an intra-component edge for the recursive call. Therefore, if the direction-optimizing idea is employed, there must be a post-processing step that inspects the edges determining whether or not they should be kept, so the total number of edges inspected is not reduced. We apply this optimization to Decomp-Arb, as it allows a vertex to select an arbitrary neighbor's component ID, and thus can exit the loop over the neighbors early. One modification is that edges that are relabeled on-the-fly during the write-based computation (on Line 18 of Algorithm 13) must be marked that they have been relabeled, so that they are not processed again during the post-processing phase (the sign bit in the label is used for this purpose). The experiments in Section 9.5 show that even though no edge traversals are saved, switching to the read-based computation when the frontier is large (the fraction of vertices on the frontier is greater than 20%) helps for some graphs, as the read-based computation is more cache-friendly, and does not require using an atomic operation, in contrast to the original Decomp-Arb which uses compare-and-swaps to resolve conflicts. The direction-optimizing version of Decomp-Arb is referred to as ***Decomp-Arb-Hybrid***.

9.5 Experiments

This section compares the three implementations of the connectivity algorithm to the fastest available parallel connectivity algorithms at the time this work was initially published [425]. The section refers to the connectivity algorithm using Decomp-Min as ***decomp-min-CC***, Decomp-Arb as ***decomp-arb-CC***, and Decomp-Arb-Hybrid as ***decomp-arb-hybrid-CC***. We also tried parallelizing over the edges for the high-degree vertices in our implementations (as discussed in Section 9.4), but due to the modest core count of the machine used in the experiments, we did not find a performance improvement. Patwary et al. [372] describe two parallel spanning forest implementations—a lock-based one and a verification-based one. The experiments use only their lock-based implementation (***parallel-SF-PRM***) since the verification-based one sometimes failed to terminate. Furthermore, they found that their lock-based implementation usually outperforms their verification-based one. The experiments also compare with the parallel spanning forest implementation in the Problem Based Benchmark Suite (***parallel-SF-PBBS***), implemented using deterministic reservations as described in Section 3.4.4. Note that these existing spanning forest-based parallel implementations are not theoretically work-efficient. As for connectivity based on BFS, the experiments compare with the direction-optimizing BFS [32] available as part of Ligra (Chapter 7), performed on each component of the graph. This implementation is referred to as ***hybrid-BFS-CC***. This approach is work-efficient but the depth can be linear in the worst case. Independently of this work, Slota et al. [434] describe a connected components algorithm which combines direction-optimizing BFS with label propagation (***multistep-CC***). Label propagation is the method used by the connected components implementation in Ligra (see Section 7.4.4). In the worst case, the algorithm of Slota et al. requires quadratic work and linear depth. All of the parallel implementations are compared to a simple sequential spanning forest-based connectivity algorithm using union-find (***serial-SF***) from the PBBS. The single-thread times for hybrid-BFS-CC and multistep-CC are sometimes better than serial-SF, and can also be used as a sequential baseline. For the spanning forest-based connectivity algorithms, the reported timings include a post-processing step that finds the ID of the root of the tree for each vertex (done in parallel for the parallel implementations).

The experiments are performed on the 40-core (with two-way hyper-threading) Intel machine described in Section 2.7. The parallel codes use Cilk Plus to express parallelism, and are compiled with the `g++` compiler. The experiments use a variety of synthetic graphs, the first three of which are taken from the Problem Based Benchmark Suite, and a

Input Graph	Number of Vertices	Number of Edges
random	10^8	5×10^8
rMat	2^{27}	5×10^8
rMat2	2^{20}	4.2×10^8
3D-grid	10^8	3×10^8
line	5×10^8	5×10^8
com-Orkut	3,072,627	117,185,083

Table 9.1: Input graphs for connected components.

Implementation	random		rMat		rMat2		3D-grid		line		com-Orkut	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-SF	19.5	–	21.5	–	2.86*	–	17.5	–	68.6	–	0.82*	–
decomp-arb-CC	43.1	1.97	46.7	2.5	6.95	0.256	30.1	1.36	254	6.49	2.35	0.115
decomp-arb-hybrid-CC	38.7	1.89	39.8	2.22	4.11	0.116	30.6	1.39	247	6.5	1.22	0.058
decomp-min-CC	74.8	2.86	76.3	3.49	7.22	0.221	57.9	2.11	348	9.11	2.39	0.132
parallel-SF-PBBS	70.9	1.91	79.2	2.13	9.79	0.515	41.1	1.53	174	5.22	2.98	0.156
parallel-SF-PRM	48.8	1.64	42.2	1.3	4.51	0.1	30.3	1.33	313	4.02	1.25	0.04
hybrid-BFS-CC	28	1.3	25.9	13.3	0.111	0.009	22.1	1.51	304	304†	0.191	0.021
multistep-CC	9.74	1.29	15.9	2.06	0.23	0.05	27.0	1.22	343	343†	0.16	0.06

Table 9.2: Times (seconds) for connected components labeling. (40h) indicates 40 cores with hyper-threading. *The timing for the sequential spanning forest code from Patwary et al. [372] is used as it was faster than the PBBS implementation. †The sequential time is reported due to overheads of parallel execution.

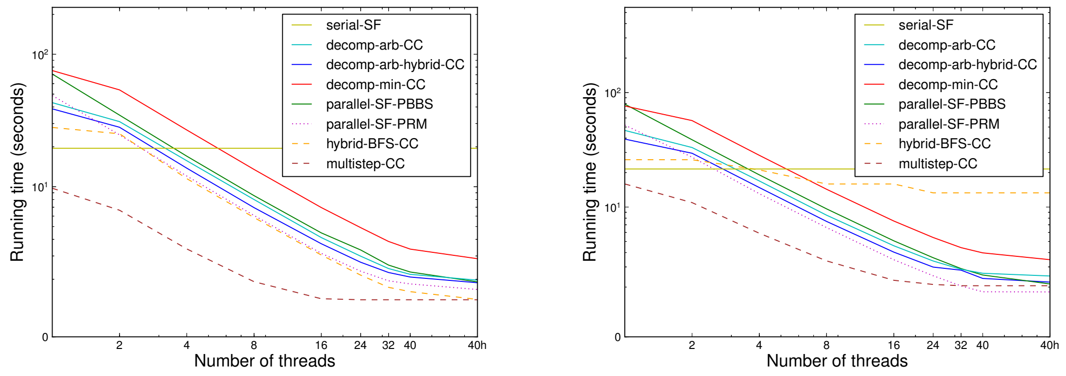
real-world graph. ***random*** is a random graph where every vertex has five edges to neighbors chosen randomly. The ***rMat*** graph [87] is a graph with a power-law degree distribution. ***rMat2*** uses the same generator as rMat, but with a higher edge-to-vertex ratio, giving a denser graph. ***3D-grid*** is a grid graph in 3-dimensional space where every vertex has six edges, each connecting it to its 2 neighbors in each dimension. ***line*** is a path of length $n-1$ (i.e., each vertex has two neighbors except for the first and the last vertex in the path). This is a degenerate graph with diameter $n-1$. ***com-Orkut*** is a social network graph downloaded from the Stanford Network Analysis Project [298]. For the synthetic graphs, the vertex labels are randomly assigned. The sizes of the graphs are shown in Table 9.1. The decomposition-based algorithms described in this chapter store an edge in each direction, so use twice the number of edges than as reported in Table 9.1, while for the spanning forest-based algorithms, edges only need to be stored in one direction.

The serial and parallel running times of the implementations on the various inputs are summarized in Table 9.2. The times reported are based on a median of three trials. Observe that decomp-arb-CC and decomp-arb-hybrid-CC usually outperform decomp-min-CC (by up to 2.3 times). This is because (1) decomp-arb-CC and decomp-arb-hybrid-CC require only one pass over the edges of the frontier instead of two passes in decomp-min-CC and (2) the vertices store less data when computing the labeling. Decomp-arb-hybrid-CC is faster than decomp-arb-CC for most of the graphs, especially for the graphs whose frontier grows very large (e.g., about 2x faster for rMat2 and com-Orkut), as these graphs benefit more from the optimization of using a read-based computation for the large frontiers. For the 3D-grid and line graphs, the times are about the same for decomp-arb-CC and decomp-arb-hybrid-CC, since in decomp-arb-hybrid-CC the frontier never grows large enough to switch to the read-based computation. Among the two spanning forest-based parallel implementations, parallel-SF-PRM is faster than parallel-SF-PBBS in parallel. Compared to parallel-SF-PRM, decomp-arb-hybrid-CC is at most 70% slower in parallel, and faster sequentially. On 40 cores with hyper-threading, the parallel implementations developed in this chapter achieve a self-relative speedup of between 18 and 39.

The experiments show that the implementations based on a single direction-optimizing BFS (hybrid-BFS-CC and multistep-CC) work well for dense graphs with low-diameter, such as random, rMat2, and com-Orkut, outperforming the other implementations both sequentially and in parallel on these graphs. For the dense rMat2 graph, which requires only 5 levels of BFS to completely traverse, even the sequential times of these implementations are competitive with the parallel times of the other implementations. This is because the read-based optimization of direction-optimizing BFS significantly reduces the number of edges traversed. For graphs with many components (e.g., rMat with over 13 million components), hybrid-BFS-CC does poorly in parallel since it visits the components one-by-one, while multistep-CC does better because it uses parallel BFS to compute only one component, and then switches to label propagation to compute the rest. For the line graph, both implementations perform poorly and get no speedup due to the large diameter of the graph. The fastest parallel implementation from this chapter (decomp-arb-hybrid-CC) is faster than hybrid-BFS-CC and multistep-CC for the line graph, competitive for the rMat and 3D-grid graphs, and slower for the random, rMat2, and com-Orkut graphs. For graphs with only one component (random, rMat2, 3D-grid, and line), multistep-CC and hybrid-BFS-CC both perform exactly one BFS, and the differences in running times are due to the choice of when to switch to the read-based computation, starting vertex of the BFS, and slight implementation differences. Note that on a single thread, multistep-CC outperforms serial-SF for four of the graphs, since the read-based optimization allows it to traverse many fewer edges for these graphs.

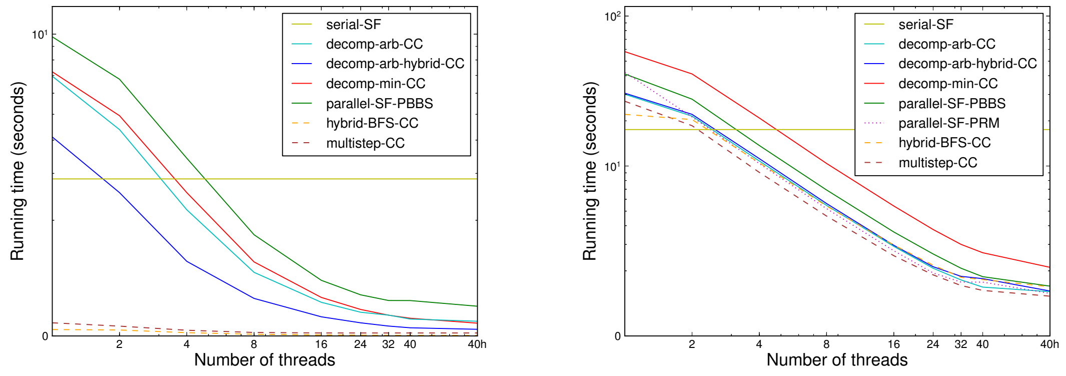
Compared to the best single-thread times among serial-SF, hybrid-BFS-CC and multistep-CC, on 40 cores the fastest implementation developed in this chapter (decomp-arb-hybrid-CC) achieves up to a 13 times speedup. For the dense rMat2 graph, on 40 cores decomp-arb-hybrid-CC is actually slower than hybrid-BFS-CC run on a single thread, but this is a special case on which the direction-optimizing BFS approach works particularly well.

Figure 9.2, 9.3, and 9.4 show the running time versus the number of threads for the different implementations on the input graphs. For the line graph, hybrid-BFS-CC, and



(a) random (b) rMat

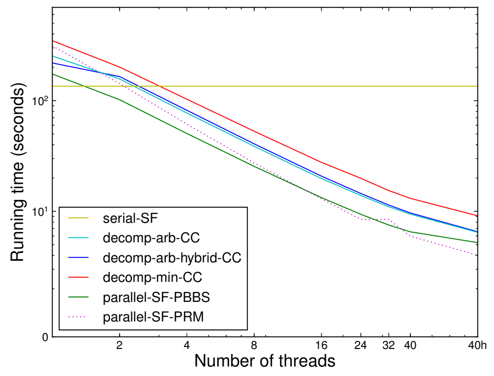
Figure 9.2: Times versus number of threads on a 40-core machine with hyper-threading of connected components implementations on random and rMat. "40h" indicates 80 hyper-threads.



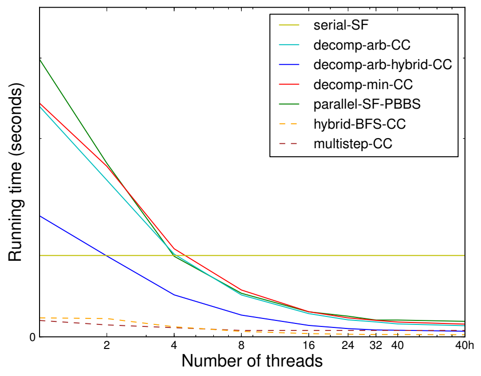
(a) rMat2 (b) 3D-grid

Figure 9.3: Times versus number of threads on a 40-core machine with hyper-threading of connected components implementations on rMat2 and 3D-grid. "40h" indicates 80 hyper-threads.

multistep-CC are not plotted as they perform very poorly and get no speedup. Observe that the decomposition-based parallel implementations get good speedup, and except for rMat2 and com-Orkut, outperform the best sequential time with a modest number of threads. The parallel implementations developed in this chapter (decomp-arb-CC, decomp-arb-hybrid-CC, and decomp-min-CC) perform reasonably well and are competitive with the other parallel implementations implementations (which are not theoretically linear-work and polylogarithmic-depth) for all graphs except rMat2 and com-Orkut, on which the direction-optimizing BFS implementations perform exceptionally well. While the decomposition-based parallel implementations do not achieve the fastest performance for



(a) line



(b) com-Orkut

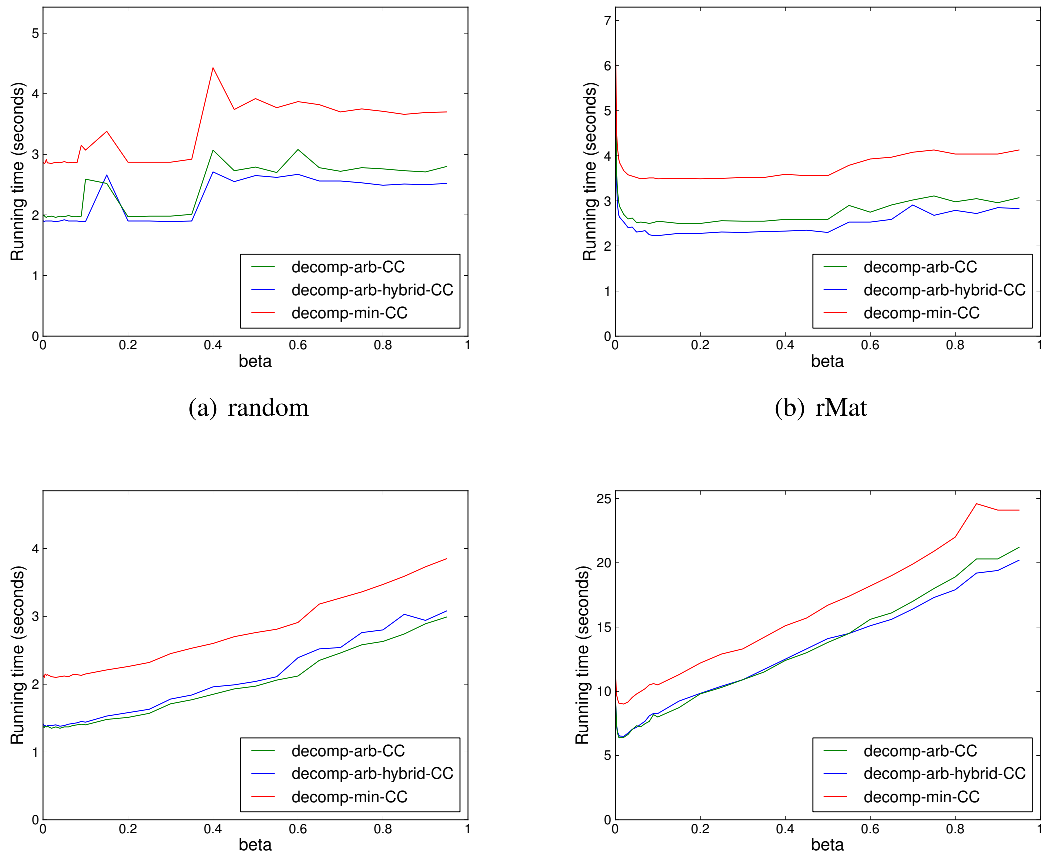
Figure 9.4: Times versus number of threads on a 40-core machine with hyper-threading of connected components implementations on the line graph and com-Orkut. “40h” indicates 80 hyper-threads.

any particular graph, due to their theoretical guarantees, they perform reasonable well across all inputs and do not suffer from poor performance on any “worst-case” inputs.

Figure 9.5 shows the 40-core running time of decomp-arb-CC, decomp-arb-hybrid-CC, and decomp-min-CC as a function of the parameter β for several graphs. The reader can observe that the trends for the implementations are similar, and the β leading to the fastest running times is between 0.05 and 0.2. Figure 9.6 shows the number of edges remaining per iteration for decomp-arb-hybrid-CC as a function of β. As expected, the number of edges drops more quickly for smaller β, leading to fewer phases before reaching the base case. Furthermore, the upper bound of a 2β-fraction of edges being removed (or β-fraction for decomp-min-CC) per iteration does not account for the removal of duplicate edges between contracted components. For all of the inputs except the line graph, there are (many) duplicate edges between components that are removed, leading to a much sharper decrease (up to an order of magnitude more than predicted by the upper bound) in the number of remaining edges per iteration.

Figure 9.7 shows the breakdown of the 40-core running time for decomp-min-CC on several graphs. In the figure, “init” refers to the time for generating random permutations and initializing arrays, “bfsPre” refers to adding new vertices to the BFS frontier and computing offsets into shared arrays for the frontier vertices, “bfsPhase1” refers to the first phase (Lines 9–23 of Algorithm 12), “bfsPhase2” refers to the second phase (Lines 24–39 of Algorithm 12), and “contractGraph” includes the time for removing duplicate edges, renumbering vertices and edges, creating the contracted graph for the recursive call, and relabeling after the recursive call. The figure shows that 80–90% of the time is spent in the two BFS phases, with the first phase being the more expensive of the two.

Figure 9.8 shows the breakdown of the running time for decomp-arb-CC on 40 cores



(a) random (b) rMat

(c) 3D-grid (d) line

Figure 9.5: Running time versus β on various input graphs on a 40-core machine using 80 hyper-threads.

on several inputs. “bfsMain” refers to the single phase of the BFS iteration (Lines 9–20 of Algorithm 13), and the other sub-timings have the same meaning as in the previous paragraph. The majority of the time (55–75%) is spent in the main BFS phase. Compared to decomp-min-CC, the savings in running time of decomp-arb-CC comes from this part of the computation due to requiring only one pass over the edges.

Figure 9.9 shows the breakdown of the 40-core running time for decomp-arb-hybrid-CC. “bfsSparse” refers to the time spent in the main phase of the BFS when performing the write-based computation for sparse frontiers, and “bfsDense” refers to the time spent in the main phase performing the read-based computation on the dense frontiers. As noted in Section 9.4, a post-processing step to filter out the intra-component edges is required, and “filterEdges” refers to this phase. For the 3D-grid and line graphs, the frontier never becomes dense enough to switch to the read-based computation, hence all of the BFS time

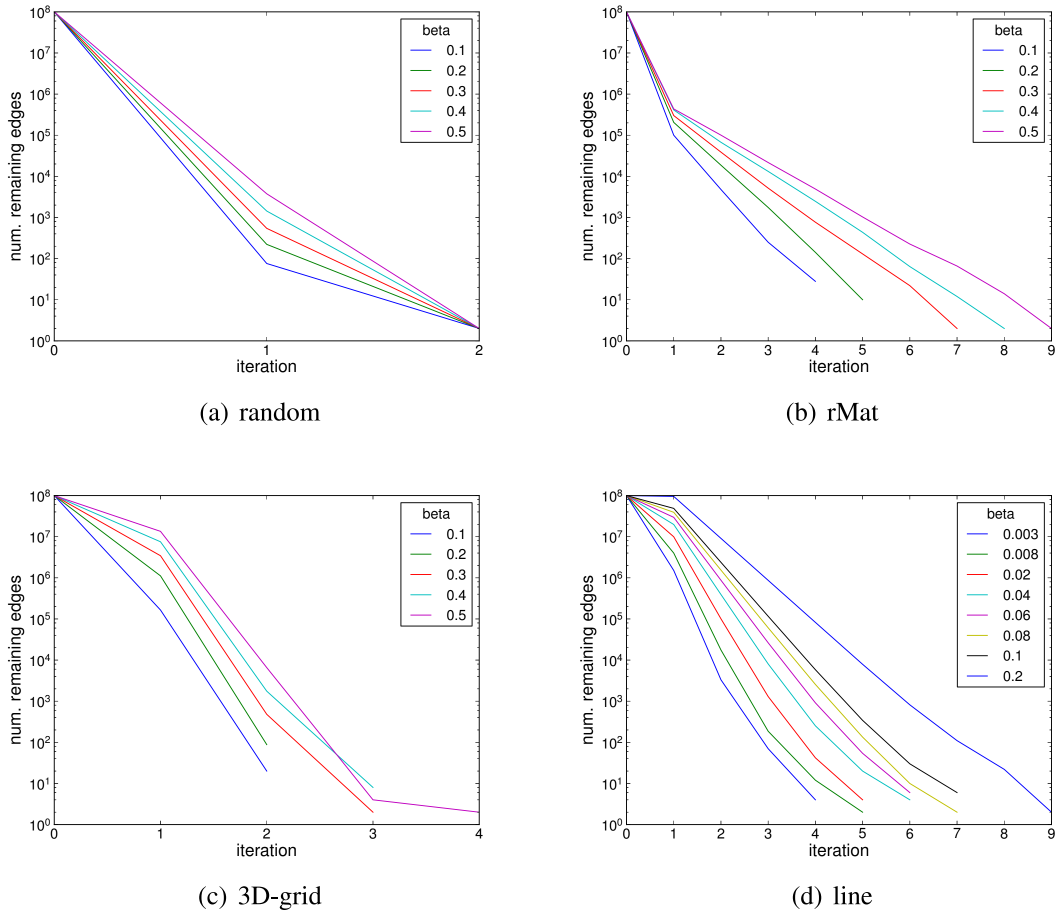


Figure 9.6: Number of remaining edges per iteration versus β of decomp-arb-hybrid-CC.

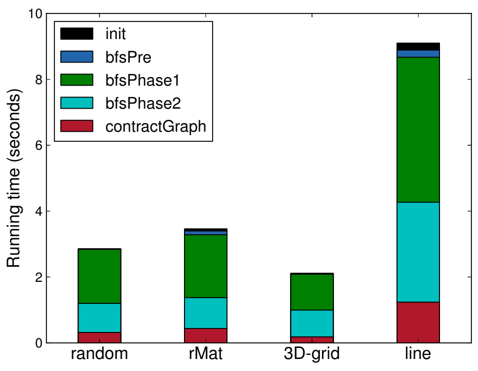


Figure 9.7: Breakdown of timings on 40 cores with hyper-threading for decomp-min-CC.

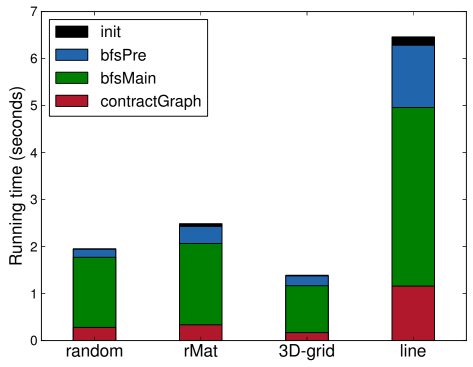


Figure 9.8: Breakdown of timings on 40 cores with hyper-threading for decomp-arb-CC.

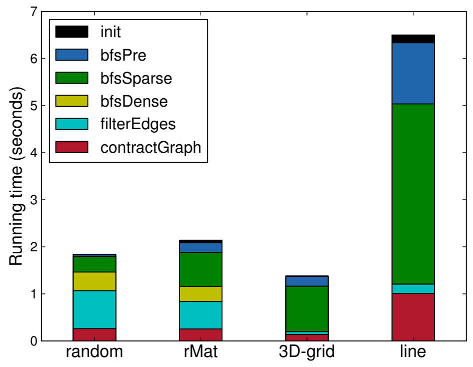


Figure 9.9: Breakdown of timings on 40 cores with hyper-threading for decomp-arb-hybrid-CC.

is captured by bfsSparse. On the other hand, random and rMat do have BFS frontiers that become dense enough where the read-based computation is invoked. Since they switch to the read-based computation, some edges do not get inspected and hence the filterEdges phase performs more work to filter out the intra-component edges. For random and rMat, about 40% of the time is spent in the main BFS phase.

Figure 9.10 shows the running time of decomp-arb-hybrid-CC on 80 hyper-threads as a function of graph size for random graphs with sizes from $m = 5 \times 10^7$ to 5×10^8, and $n = m/5$. The running time increases almost linearly as the graph size is increased.

Besides PBBS and the implementations by Patwary et al., Bader and Cong describe a parallel spanning tree implementation based on parallel depth-first search [24]. However, Patwary et al. [372] show that their implementations are faster than Bader and Cong's implementation. Galois [351] also contains implementations of connected components based on union-find, but they were slower than the implementation by Patwary et al, decomp-arb-hybrid-CC, and decomp-arb-CC for all of the input graphs used in this section. Several graph processing systems [420, 250, 289, 290] have connected components imple-

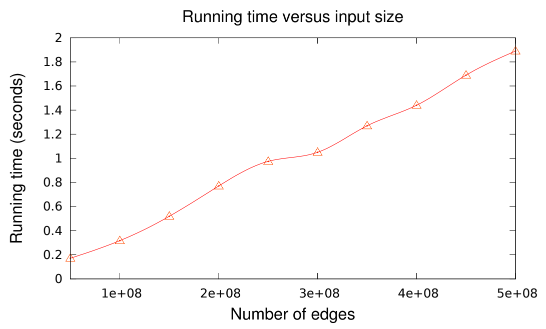


Figure 9.10: Running time of decomp-arb-hybrid-CC vs. problem size for random graphs on 40 cores with hyper-threading.

mentations based on label propagation, but the depth of the algorithm is proportional to the diameter of the graph and the algorithm is not work-efficient. As noted in Chapter 7, this algorithm usually does not perform as well as linear or near-linear work algorithms.

Chapter 10

Parallel and Cache-Oblivious Triangle Computations

10.1 Introduction

As graphs are increasingly used to model and study interactions in a variety of contexts, there is a growing need for graph analytics to process massive graphs quickly and accurately. Among various metrics of interest, the triangle count and related measures have attracted a lot of recent attention because they reveal important structural information about the network being studied. Unsurprisingly, triangle counting and enumeration has seen applications in the study of social networks [349], identifying thematic structures of networks [140], spam and fraud detection [33], link classification and recommendation [446], joining three relations in a database [350, 364], database query optimization [28]—with further examples discussed in [364, 233, 40].

Driven by such applications, several algorithms have been proposed for the distributed setting (e.g., [104, 439, 186, 368, 14, 456]) and the external-memory setting (e.g., [131, 331, 98, 289, 233, 364, 268]) as graphs of interest were deemed too big to keep in the main memory of a single computer. The distributed algorithms are not tailored for a multicore machine, and the external-memory algorithms typically do not support parallelism (with the exception of [289, 268]). However, as discussed in Chapter 1, a single multicore machine today can have tens of cores and can support several terabytes of memory—capable of storing graphs with tens or even hundreds of billions of edges. Compared to distributed-memory systems, communication costs are much cheaper in multicore systems, leading to performance benefits with a proper design. Moreover, for graph algorithms, multicores are known to be more efficient per core and per watt than an equivalent distributed system. Therefore, this chapter develops fast and simple shared-memory parallel algorithms for

Algorithm	Work	Depth	Cache Complexity
TC-Merge	$O(m^{3/2})$	$O(\log^{3/2} m)$	$O(m + m^{3/2}/B)$
TC-Hash	$O(n \log n + \alpha m)$	$O(\log^{3/2} m)$	$O(\text{sort}(n) + \alpha m)$
Parallel-PS	$O(m^{3/2})$	$O(\log^{5/2} m)$	$O(m^{3/2}/(\sqrt{M}B))$

Table 10.1: (Randomized) complexity bounds for triangle counting algorithms, where n = number of vertices, m = number of edges, α is arboricity of the graph, M = cache size, B = cache line size, and $\text{sort}(N) = O((N/B)\log_{M/B}(N/B))$.

triangle computations using Cilk Plus and analyzes them in the work-depth model.

In addition to parallelism, the cache behavior of programs has a significant impact on performance. Writing parallel programs with good cache behavior has often required expertise. Because machines differ, this often requires fine-tuning code or parameters for each individual machine. Even then, it is still difficult to achieve good cache performance because the memory system of a modern machine has become highly sophisticated, consisting of multiple levels of caches and layers of indirection.

To sidestep this complex issue, this chapter designs algorithms that make efficient use of caches without needing to know the specific memory/cache parameters (e.g., cache size, cache line size). Such parallel algorithms are known as *parallel cache-oblivious* algorithms, as they are oblivious to cache parameters [431, 54, 157]. Parallel cache-oblivious algorithms free the programmer from optimizing the cache parameters for specific machines, as they run efficiently on all shared-memory machines. These algorithms are analyzed for parallel cache complexity as a function of the problem size n, the cache size M, and the cache line size B.

Contributions. This chapter presents fast and simple shared-memory parallel algorithms for triangle counting, both exact and approximate, that are able to scale to billions of vertices and edges. The algorithms take full advantage of parallelism in a multicore system and are optimized for the memory hierarchy by being cache-oblivious. The main contributions are as follows:

1. *Parallel Algorithms.* This chapter designs parallel algorithms for triangle counting, one which uses merging for intersecting adjacency lists (TC-Merge) and one which uses hashing for intersection (TC-Hash). The algorithms are based on Latapy's sequential algorithm [292], and are shown to have good theoretical bounds in the Parallel Cache Complexity (PCC) model [54, 431]. The work, depth, and cache complexity bounds are shown in Table 10.1. In addition, the chapter describes how to extend the algorithms to approximate triangle counting, directed triangle counting, triangle enumeration, local triangle counting, and computing clustering coefficients. The algorithms are easy to implement and do not require parameter tuning. In addition, a parallelization of the recent sequential cache-oblivious triangle enumeration algorithm of Pagh and Silvestri [364]

(Parallel-PS) is presented, obtaining the complexity bounds shown in Table 10.1, which may be of independent interest.

2. *Performance Evaluation.* An extensive empirical evaluation is performed on a 40-core Intel machine with two-way hyper-threading as well as a 64-core AMD machine. The Cilk Plus implementations of the parallel exact global and local triangle counting algorithms achieve speedups of 17–50x and outperform previous algorithms for the same task. On the large-scale Yahoo! Web graph (with over 6 billion edges), the fastest algorithm from this chapter computes the triangle count in under 1.5 minutes. For approximate triangle counting, the parallel implementation from this chapter approximates the triangle count for the Yahoo! graph to within 99.6% accuracy in under 10 seconds, and is much faster than existing parallel approximate triangle counting implementations for a given accuracy.

3. *Analysis of Cache Behavior.* To further understand how these performance benefits come about, this chapter analyzes the cache performance of the implementations on several graphs, showing that cache performance is consistent with the theory and that cache efficiency is crucial for performance.

10.2 Preliminaries

For a simple, undirected graph $G = (V, E)$, a ***triangle*** is a set of three vertices $v_1, v_2, v_3 \in V$ such that the undirected edges (v_1, v_2), (v_2, v_3), and (v_1, v_3) are present in E. The ***triangle counting*** problem takes an undirected graph G and returns a count of the number of triangles in G. For ***triangle listing***, all of the triangles in the graph are output. The ***triangle enumeration*** problem takes an `emit` function that is called on each triangle discovery (hence, each triangle must appear in memory). Algorithms for ***local triangle counting/listing*** return the count/list of triangles incident on each vertex $v \in V$.

In this chapter, graphs are represented using the adjacency list format, as described in Section 2.4. This chapter assumes, without loss of generality, that the graph does not have any isolated vertices (they can be removed within the complexity bounds of the algorithms described). The ***arboricity*** α of a graph is the minimum number of forests its edges can be partitioned into (hence, $\alpha \geq 1$). This is upper bounded by $O(\sqrt{m})$ for general graphs and $O(1)$ for planar graphs [92]. Furthermore, it is known that $\sum_{(u,v)\in E} \min\{d(u), d(v)\} = O(\alpha m)$.

Cache Complexity. For cache complexity analysis, this chapter uses the parallel cache complexity (PCC) model [54, 431], a parallel variant of the cache-oblivious model [157]. A cache-oblivious algorithm has the advantage of being able to make efficient use of the memory hierarchy without knowing the specific cache parameters (e.g., cache size, cache

Algorithm 14 High-level parallel triangle counting algorithm

```
1: procedure RANK-BY-DEGREE($G = (V, E)$)
2:     Compute an array $R$ such that if $R[v] < R[w]$ then $d(v) \leq d(w)$
3:     parfor $v \in V$ do
4:         $A^+[v] = \{w \in N(v) \mid R[v] < R[w]\}$
5:     return $A^+$
6: procedure TC($A^+$)
7:     Allocate an array $C$ of size $\sum_{v \in V} |A^+[v]|$
8:     parfor $v \in V$ do
9:         parfor $w \in A^+[v]$ do
10:            $I = \texttt{intersect}(A^+[v], A^+[w])$
11:            $C[\rho(v, w)] = |I|$                    ▷ $\rho(\cdot)$ gives a unique index in $C$
12:    count = sum of values in $C$
13:    return count
```

line size). In the PCC model, the cache complexity of an algorithm is given as a function of cache size M and cache line size B, assuming the optimal offline replacement policy. This function reflects how the algorithm behaves for a particular cache/line size, although this information is unknown to the algorithm. For a parallel machine, it represents the number of cache misses across all cores for a particular level (e.g., L2, L3, etc.). An algorithm is analyzed assuming a single level of cache, but since the algorithm is oblivious to the cache parameters, the bounds simultaneously hold across all levels of the memory hierarchy, which can contain both private and shared caches.

This chapter uses scan(N) and sort(N) to denote the cache complexity of scanning (prefix sum) and sorting, respectively, on an input of size N. In the PCC model, it has been shown that scan$(N) = O(N/B)$ and sort$(N) = O((N/B) \log_{M/B}(N/B))$, under the standard assumption $M = \Omega(B^2)$, which is readily met in practice. In the PCC model, scan requires $O(N)$ work and $O(\log N)$ depth, and sort requires $O(N \log N)$ work and $O(\log^{3/2} N)$ depth with probability at least $1 - 1/N^c$ for some constant $c > 0$ and large enough N (or $O(\log^2 N)$ depth deterministically) [57, 54, 431]. Merging two sorted sequences of lengths N_1 and N_2 requires $O(N_1 + N_2)$ work, $O(\log(N_1 + N_2))$ depth and a cache complexity of scan$(N_1 + N_2)$ [57, 54, 431].

10.3 Triangle Counting

This section describes a conceptual algorithm for triangle counting that exposes substantial parallelism. Later sections describe how to derive efficient implementations for it.

The conceptual algorithm follows Latapy's sequential *compact-forward* algorithm [292] for triangle counting. This chapter extends Latapy's algorithm because it was shown to perform well sequentially, and is amenable to parallelization. To count the number of triangles in a graph, the algorithm performs two main steps, as shown in Algorithm 14.

Step 1: *Ranking*—form a directed graph where each undirected input edge gives rise to exactly one directed edge. The ranking helps to improve the asymptotic performance and ensures each triangle is counted only once.

Step 2: *Counting*—count triangles of a particular form in the directed graph formed in the previous step.

For the ranking step, the RANK-BY-DEGREE function on Lines 1–5 takes an undirected graph G, and computes a rank array R ordering the vertices by non-decreasing degree.[1] R contains unique integers, and for any two vertices v and w, if $R[v] < R[w]$ then $d(v) \leq d(w)$. On Lines 3–4, it goes over the vertices of G in parallel, storing for each vertex v, the higher-ranked neighbors of v in $A^+[v]$. Finally, it returns the *ranked adjacency list* A^+.

For the counting step, the triangle counting function TC on Lines 6–13 takes as input a ranked adjacency list A^+. An array C of size equal to the number of directed edges ($\sum_{v \in V} |A^+[v]|$) is initialized on Line 7. Each edge (v, w) is assigned a unique location in C, denoted by $\rho(v, w)$. On Lines 8–11, all vertices are processed, and for each vertex v, its neighbors w in $A^+[v]$ are inspected, and the intersection between $A^+[v]$ and $A^+[w]$ is computed. Each common out-neighbor u corresponds to a triangle (v, w, u) where $R[v] < R[w] < R[u]$. The count of triangles incident on (v, w) is thus set to the size of the intersection (Line 11). In Line 12, the individual counts are summed, and finally returned on Line 13.

Two observations are in order: First, because of the ranking step, all triangles will be counted exactly once. Second, since the intersection can be computed on all directed (v, w) pairs in parallel, this algorithm already has abundant parallelism.

The following example (Fig. 10.1) illustrates these steps. Notice the degree of parallelism the algorithm obtains (Fig. 10.2).

Example. Figure 10.1 shows an example graph and the graph after ranking by degree, which contains directed edges from lower to higher-ranked vertices. The rank of the vertices are stored in an array R:

Vertex	0	1	2	3	4
R	1	4	0	3	2

Figure 10.1 (right) shows the edges after running RANK-BY-DEGREE. Then, running TC on this graph will compute the set intersections of multiple pairs, as shown in Fig. 10.2. Notice that at this point, the algorithm indicates that these intersections are parallel tasks;

[1]Various ranking functions can be used, but ordering by degree in the original graph has it has been shown to perform the best in practice if both ranking and triangle counting times are included [361]. This ordering heuristic also leads to good theoretical guarantees for triangle counting [292].

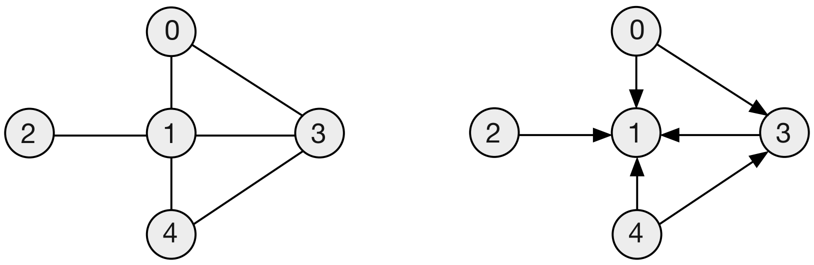


Figure 10.1: Example of a graph (**left**) and its directed edges after ranking by degree (**right**). The contents of A^+ are $A^+[0] = \{1,3\}$, $A^+[1] = \{\}$, $A^+[2] = \{1\}$, $A^+[3] = \{1\}$, and $A^+[4] = \{1,3\}$. The triangles found are $(0,3,1)$ and $(4,3,1)$, discovered by `intersect`$(A^+[0], A^+[3])$ and `intersect`$(A^+[4], A^+[3])$.

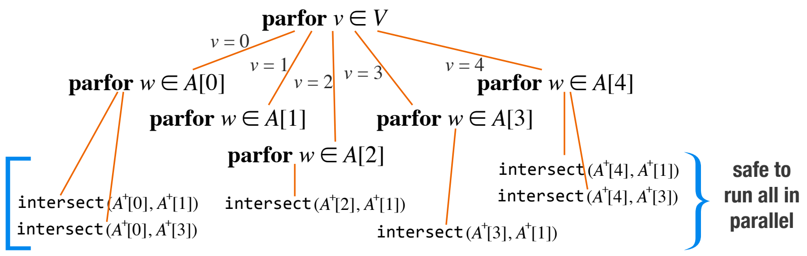


Figure 10.2: Example of how the parallel triangle counting algorithm performs in action.

however, in the context of dynamic multithreading, the exact combination of tasks that will be run simultaneously depends on the scheduler. Subsequently, for each of these pairs, the size of the intersection is recorded in C (e.g., $C[\rho(0,3)] = 1$ as $|A^+[0] \cap A^+[3]| = 1$ and $C[\rho(4,3)] = 1$ as $|A^+[4] \cap A^+[3]| = 1$).

10.4 Exact Triangle Counting

This section introduces efficient parallel algorithms for exact triangle counting based on the conceptual algorithm in the previous section. In particular, the section describes how the ranking and counting steps are implemented.

10.4.1 Ranking

To implement the ranking step, a rank array R is first constructed. Assume that the degrees of the vertices are stored in an array D of size n in order of vertex ID (i.e., $D[i]$ is the degree of the i'th vertex). By sorting the vertices by degree and breaking ties by ID, an array R can be constructed such that R contains unique integers, and if $R[u] < R[v]$ then $D[u] \le D[v]$. The sort requires $O(n \log n)$ work, $O(\log^{3/2} n)$ depth and $O(\text{sort}(n))$ cache misses w.h.p., as mentioned in Section 10.2.

Then, given the ranking array R, the algorithm looks up the rank for each endpoint of every edge and chooses which direction to retain. In particular, each vertex looks up the rank of each of its neighbors and applies a parallel filter, keeping only the higher-ranked neighbors. Each vertex will incur a cache miss to access the start of its adjacency list, for a total of $O(n)$ cache misses. The filters require $O(m)$ work, $O(\log m)$ depth and scan(m) cache misses overall. Looking up the rank of the neighbors requires $O(m)$ work, $O(1)$ depth and $O(m)$ cache misses overall (since the neighbors can appear anywhere in R). The following lemma summarizes the complexity of ranking:

Lemma 22. RANK-BY-DEGREE *can be implemented in* $O(n \log n + m)$ *work,* $O(\log^{3/2} n)$ *depth and* $O(\text{sort}(n) + m)$ *cache misses w.h.p.*

It is worth noting that a cache complexity of $O(\text{sort}(m))$ w.h.p. can be obtained for ranking (while increasing the work to $O(m \log m)$) by using sorting routines. However this approach is more expensive in practice, and furthermore it does not improve the overall complexity of triangle counting, so it is not elaborated on here.

10.4.2 Counting

This section describes the counting algorithm TC assuming that the ranked adjacency list A^+ has already been computed. The size of C and the unique locations $\rho(v, w)$ in C for each directed edge (v, w) can be computed with a parallel scan over the directed edges. In particular, each vertex v writes the length of $A^+[v]$ into a shared array at location v, and then a scan with the $+$ operator is applied to generate the starting offset o_v for each vertex. The offset for element i in $A^+[v]$ is computed as $o_v + i$. The result of the scan also gives the size of C. This requires $O(m)$ work, $O(\log m)$ depth and $O(\text{scan}(m))$ cache misses. On Line 12, the individual counts in C are added together using a prefix sum. Two implementations of Lines 10–11, differing in how the `intersect` function is implemented, are described next.

Algorithm I: Merge-based Algorithm: The first algorithm, called ***TC-Merge***, implements Line 10 by using a merge on the directed adjacency lists of v and w. It requires sorting the adjacency lists as a pre-processing step, which requires $O(m \log m)$ work, $O(\log^{3/2} m)$ depth and $O(\text{sort}(m)+n)$ cache misses w.h.p. Merging the sorted lists gives the intersection and its size, requiring work linear in the size of the two lists. Sequentially, the total amount of work done in merging has been shown to be $O(m^{3/2})$ [292], and since the merge is done in the same asymptotic work in parallel, the bound is the same (hence, it is work-efficient). The depth for merging is $O(\log(m^{3/2})) = O(\log m)$ and cache complexity is $O(\text{scan}(m^{3/2})) = O(m^{3/2}/B)$ (note that this dominates the cache complexity of sorting). Accessing the adjacency list for each edge involves a random access, adding a total of $O(m)$ cache misses. The complexity of counting dominates the complexity of ranking. This gives the following theorem:

Theorem 23. *Ranking and triangle counting using TC-Merge can be performed in* $O(m^{3/2})$ *work,* $O(\log^{3/2} m)$ *depth and* $O(m + m^{3/2}/B)$ *cache misses w.h.p.*

Note that if $B = O(\sqrt{m})$, then $m^{3/2}/B$ is the dominant term in the cache complexity. This condition is readily met in practice for in-memory algorithms since a typical cache line is 64 bytes, which holds at most 16 edges in a standard implementation,[2] and typical graphs of interest have at least tens of thousands of edges (the graphs used in this thesis have tens of millions to billions of edges). This condition will likely continue to hold in the future when analyzing large graphs (i.e., graph sizes in terms of number of edges will grow faster than $\Omega(B^2)$). The situation may be different for the external-memory setting, as was pointed out in [233].

Algorithm II: Hash-based Algorithm: The second algorithm, ***TC-Hash***, uses a hash table storing the edges of A^+ to compute the intersection on Line 10 of Algorithm 14. A hash table can be implemented in parallel to support worst-case $O(1)$ work and depth queries [324], and so Line 8 can be implemented in $O(\min\{A^+[v], A^+[w]\})$ work by looping over the smaller of $A^+[v]$ and $A^+[w]$ and querying the hash table of the other vertex. Insertion of the edges into the hash tables can be done in $O(m)$ work, $O(\log m)$ depth and $O(m)$ cache misses w.h.p. [324].

Since $|A^+[v]| \leq d(v)$ for all v, this gives an overall work bound of $O(\alpha m)$ for TC-Hash, where $\alpha \geq 1$ denotes the arboricity of the graph (recall from Section 10.2 that $\sum_{(u,v)\in E} \min\{d(u), d(v)\} = O(\alpha m)$). Note that since $\alpha = O(\sqrt{m})$, this bound is tighter than $O(m^{3/2})$ and is in fact optimal. However, each hash table look-up incurs $O(1)$ cache misses, leading to $O(\alpha m)$ total cache misses. Looking up the adjacency list of each edge involves a random access, leading to $O(m)$ cache misses. Computing the size of each intersection can be done work-efficiently with a parallel scan. Hence, this can be implemented in $O(\alpha m)$ work, $O(\log m)$ depth and a parallel cache complexity of $O(\alpha m)$.

Putting together the bounds for ranking and counting gives the following theorem:

Theorem 24. *Ranking and triangle counting using TC-Hash that performs $O(n \log n + \alpha m)$ work, $O(\log^{3/2} m)$ depth and $O(\text{sort}(n) + \alpha m)$ cache misses w.h.p.*

10.5 Approximate Triangle Counting

If some amount of error can be tolerated, the running time of triangle counting can be reduced using approximate counting algorithms. This section extends the parallel algorithms for exact triangle counting to approximate triangle counting. As will be discussed in Section 10.9, many approximate triangle counting schemes have been proposed [17, 448, 449, 446, 365, 411, 456], and the recent colorful triangle counting scheme of Pagh and Tsourakakis (PT) [365] is one of the most efficient. This section uses the PT colorful

[2]Compression techniques could be used to store edges more compactly, however $B = O(\sqrt{m})$ still holds for large graphs.

```
Algorithm 15 Pagh-Tsourakakis Sampling
    Input: a graph $G = (V, E)$ and a parameter $0 < p \leq 1$
    Output: a sampled subgraph $H = (V_H, E_H)$ of $G$.
1: Assign a random color $c(v) \in \{1, \ldots, C\}$ to every vertex $v$, where $C = \lceil 1/p \rceil$.
2: Construct $E_H = \{(u, v) \in E \mid c(u) = c(v)\}$ and $V_H \subseteq V$ if the vertex has at least one neighbor.
3: Return $H = (V_H, E_H)$.
```

triangle counting scheme to develop parallel and cache-oblivious approximate triangle counting algorithms.

The PT algorithm works by first sampling edges from the input graph using Algorithm 15. An exact triangle counting algorithm is then run on the subgraph. If the exact triangle counting algorithm reports T triangles, then the PT algorithm reports an estimate of T/p^2 triangles.

Pagh and Tsourakakis [365] show that the estimate T/p^2 is an unbiased estimate (i.e., its expectation equals the true triangle count) as each triangle is included in the subgraph with probability p^2 (if two edges in a triangle are present in the subgraph, then the third edge must also be present). They also prove that the estimate is tightly concentrated around its mean for appropriate values of p. Note that a larger p value leads to higher quality estimates and vice versa.

Using TC-Merge after sampling gives the following lemma:

Lemma 23. *For a parameter $0 < p \leq 1$, approximating the number of triangles in a graph can be done in $O(m + (pm)^{3/2})$ work, $O(\log^{3/2} m)$ depth and a parallel cache complexity of $O(scan(m) + pm + (pm)^{3/2}/B)$ in expectation.*

Proof. To form the subgraph, representation of the graph is first converted to an *edge array* representation, which is an array of length m storing pairs of vertices that have an edge between them. Since the adjacency list representation stores the neighbor arrays contiguously in memory, the conversion can be done using a scan. Then, a parallel filter is applied to the edge set keeping only edges with both endpoints having the same color. The algorithm assumes that the color of a vertex can be computed with a hash function, and so does not involve a memory access. The scan and filter can be done in $O(m)$ work, $O(\log m)$ depth and $O(\text{scan}(m))$ cache misses. Then any singleton (isolated) vertices are removed and remaining vertices and the edges are relabeled so that the vertex ID's are in a consecutive range. This packing step can be done using standard techniques involving prefix sums in $O(pm)$ work and cache misses. Afterward, the edge array is converted back to the adjacency list representation using prefix sums. Using TC-Merge on the subgraph and applying Theorem 23 on a subgraph with an expected number of edges equal to pm proves the lemma. $\square$

The following lemma can be obtained by using TC-Hash instead of TC-Merge on the subgraph, where n_H is the number of vertices in the subgraph ($n_H = O(pm)$ in expectation, since singleton vertices are removed) and $\alpha_H \geq 1$ is the arboricity of the subgraph.

Lemma 24. *For a parameter $0 < p \leq 1$, approximating the number of triangles in a graph can be done in $O(m + n_H \log n_H + p\alpha_H m)$ work, $O(\log^{3/2} m)$ depth and a parallel cache complexity of $O(\mathit{scan}(m) + \mathit{sort}(n_H) + p\alpha_H m)$ in expectation.*

10.6 Extensions

10.6.1 Triangle Enumeration

To adapt TC-Merge and TC-Hash for triangle enumeration, only the implementation of Line 10 of Algorithm 14 needs to be modified so that `emit` is called whenever a triangle is present in memory. Note that since the `emit` function may be called in parallel, one must ensure that any modifications to shared structures are atomic.

For example, to list all the triangles in the graph, an algorithm can initialize a concurrent hash table, and have the `emit` function add the triangle to the hash table when it finds one.[3] With a good hash function and large enough hash table, the probability that two triangles hash to the same location is small, and hence memory contention will be small. After all triangles are added, the algorithm can write out the contents of the hash table.

Without accounting for the cost of `emit` (consistent with the analysis in [364]), which varies with the application, the complexity is the same as that of exact triangle counting.

10.6.2 Directed Triangle Counting and Enumeration

Triangle computations on directed graphs have also attracted recent interest [186, 410]. The goal is to count triangles of different configurations of directed edges. For example, the GraphLab directed triangle counting implementation [186] counts four types of triangles: in-, out-, through, and cycle triangles. If a vertex v with two incoming edges participates in a triangle, it is said to be an *in-triangle* incident on v. If a vertex v with two outgoing edges participates in a triangle, it is said to be an *out-triangle* incident on v. Finally, if a vertex v with one incoming edge and one outgoing edge participates in a triangle, and the final triangle edge forms a cycle, then the triangle is a *cycle triangle* incident on v; otherwise it is said to participate in a *through triangle*.

Let us now look at how to modify TC-Merge and TC-Hash using Algorithm 14 to count the 4 types of directed triangles described above. When the graph is symmetrized for the ranking phase, additional information is stored indicating which direction(s) the edge appears in the original graph. The array of counts C on Line 7 is modified to store

[3]If threads are explicitly managed then the program can initialize a list for each thread, and whenever a thread finds a triangle it simply adds the triangle to its list. The lists are then be joined at the end.

4-tuples per entry, where $C[\rho(u, w)]$ stores the count of each type of triangle incident on edge (u, w). Then on Lines 10–11, the counts of each type of directed triangle is computed and stored into $C[\rho(u, w)]$. The type(s) of each triangle can be computed locally with constant work/depth and no memory accesses. Finally, to sum the counts on Line 12, element-wise sums of the 4-tuples of C are computed using a prefix sum, and a single 4-tuple is returned. If the enumeration variant is instead desired, `emit` can be modified to take additional information about the orientation of edges in the triangle. The work, depth, and cache complexity bounds of Theorems 23 and 24 are preserved for directed triangle counting.

10.6.3 Local Triangle Counting

The *local triangle counting* problem takes a graph and returns for each vertex, the number of triangles incident on it. TC-Merge and TC-Hash as described in Section 10.4 only count each triangle once, instead of 3 times, since the ranking phase keeps each edge in only one direction. So just returning the array of counts C in Algorithm 14 would not produce the correct answer. One way to perform local triangle counting is to first store all of the triangles in an array using a triangle enumeration algorithm. To obtain the local counts, the array of triangles are sorted, using the first endpoint of the triangle as the key. After the sort, the triangles sharing the first endpoint will be in consecutive order. Then standard techniques involving prefix sum operations can be used to compute the partial local counts per vertex. This procedure (sorting and computing partial local counts) is repeated on the second and third endpoints of the triangles, and the result will be the local triangle counts for each vertex. The cost of this method is dominated by sorting the triangles, and since there are $O(\alpha m)$ triangles, the work is $O(\alpha m \log m)$, depth is $O(\log^{3/2} m)$, and cache complexity is $O(\text{sort}(\alpha m))$ w.h.p. Including the cost of triangle enumeration using TC-Hash increases the cache complexity to $O(\alpha m + \text{sort}(\alpha m))$ w.h.p. If TC-Merge is used, then the work becomes $O(m^{3/2} + \alpha m \log m)$ and cache complexity becomes $O(m + m^{3/2}/B + \text{sort}(\alpha m))$ w.h.p.

If an atomic increment operation is assumed to take $O(1)$ work, then the bounds can be improved with the following scheme. In practice, this assumption can be met, for example, by using `x86`'s atomic add instructions and controlling contention at each location. An array of size n is created to store the local count of each vertex (initialized to 0). Whenever a triangle is identified in the triangle counting algorithm, an atomic increment is performed on the locations in the array corresponding to each of the three triangle endpoints. Since these locations can be anywhere, each triangle found causes $O(1)$ cache misses. The total number of triangles is bounded by $O(\alpha m)$ so if TC-Hash is used for counting, this gives an algorithm with $O(n \log n + \alpha m)$ work, $O(\log^{3/2} m)$ depth and $O(\text{sort}(n) + \alpha m)$ cache misses w.h.p. The experiments in Section 10.7 use TC-Merge for counting as it performs better in practice, although the theoretical bounds of local triangle counting become weaker—the work bound increases to $O(m^{3/2})$ and cache complexity increases to $O(m^{3/2}/B + \alpha m)$.

Local triangle counting also works for the directed setting. In the first method, a directed triangle enumeration algorithm which gives the type of each triangle can be used. After each sort, which groups the triangles by a certain endpoint, the algorithm can sort within the groups by triangle type. Then the sizes of these subgroups as well as the groups are computed using prefix sums. For the second method, the algorithm can store 4-tuples in the global array of local counts, and atomically increment the appropriate element(s) in the tuples based on the triangle type(s).

10.6.4 Clustering Coefficients and Transitivity Ratio

The *local clustering coefficient* [458] for a vertex v is defined to be the number of triangles incident on v divided by $d(v)(d(v) - 1)/2$ (the number of potential triangles incident on v). The *global clustering coefficient* is the average over all local clustering coefficients. Both quantities can be computed using the algorithms for local triangle counting.

The *transitivity ratio* of a graph is defined to be the ratio of 3 times the number of triangles to the number of length-2 paths (wedges), which can be computed as $\sum_{v\in V}(d(v)(d(v)-1)/2)$. The number of triangles is already returned by TC-Merge and TC-Hash and the number of wedges can be computed with a prefix sum. Hence, the bounds for computing the transitivity ratio are the same as in Theorems 23 and 24.

10.7 Evaluation

This section experimentally evaluates how the algorithms developed in this chapter perform in practice, specifically how well they scale with the number of threads, how fast they are compared to existing alternatives, and whether they are cache-efficient. To this end, this section reports and discusses the running times, parallel speedups, and cache misses for the exact algorithms, as well as the accuracy of the approximation algorithm versus its running time. Overall, the results indicate that *the algorithms are very fast in practice, scaling well with the number of cores.*

Input Data. The input graphs include a variety of real-world networks from the Stanford Network Analysis Project [298], and several synthetic graphs generated from the Problem Based Benchmark Suite. The experiments also use the Twitter graph [288] and the Yahoo! Web graph [466]. These graphs are drawn from many fields and have different characteristics, and many are graphs stemming from social media, where triangle computations often see applications. The graph sizes and triangle counts are shown in Table 10.2. The table reports the number of undirected edges (i.e., an edge between u and v is counted once), but the implementations store, in the intermediate representation, each edge in both directions,

Input Graph	Number of Vertices	Number of Edges*	Number of Triangles
random	100,000,000	491,001,390	24,899,692
rMat	134,217,728	498,586,618	539914
3D-grid	99,897,344	299,692,032	0
soc-LJ	4,847,571	42,851,237	285,730,264
Patents	3,774,768	16,518,947	7,515,023
com-LJ	3,997,962	34,681,189	177,820,130
Orkut	3,072,441	117,185,083	627,584,181
Twitter	41,652,231	1,202,513,046	34,824,916,864
Yahoo!	1,413,511,391	6,434,561,035	85,782,928,684

Table 10.2: Graph inputs for triangle computations. *Number of unique undirected edges.

so store twice as many edges. Therefore, the implementations effectively symmetrize all of the graphs. The graphs are also pre-processed to remove self-loops and duplicate edges.

Environment. The experiments are performed on both the 40-core (with two-way hyper-threading) Intel machine and the 64-core AMD machine described in Section 2.7. Most of the reported results are obtained from the Intel machine, but some results on the AMD machine are also reported, showing that the algorithms exhibit the same performance trends on different machines. The codes use Cilk Plus to express parallelism, and are compiled with the `g++` compiler.

10.7.1 Implementation

The implementations use the parallel primitives prefix sum, filter, and sort, from the Problem Based Benchmark Suite, which are all cache-oblivious. In the implementations of Algorithm 14, the for-loop on Line 3 and nested parallel for-loops on Lines 8 and 9 use the `cilk_for` construct. Note that already, the counting code has abundant parallelism (a lot more than the number of cores available) because all of the `intersect` calls are made in parallel (Lines 10–11 of Algorithm 14). Consequently, for TC-Merge, it suffices for each `intersect` to use a sequential merge; making the merge parallel does not improve the speedup as has been experimentally confirmed. Each merge terminates when one of the lists has been fully traversed. For TC-Hash, the concurrent hash table described in Chapter 5 is used. Before counting, each vertex creates a hash table of its neighbors in $A^+[v]$. During counting, which intersects $A^+[v]$ and $A^+[w]$ for each directed edge (v, w), the implementation loops through the smaller adjacency list and queries the table of the vertex with the larger adjacency list. Again, due to abundant parallelism in the nested parallel for-loop, the hash table look-ups for each `intersect` are done sequentially, as there was no performance improvement observed from parallelizing it.

Algorithm		random	rMat	3D-grid	soc-LJ	Patents	com-LJ	Orkut	Twitter	Yahoo!
serial-OB										
	T_1	278	298	133	24.52	6.23	18.15	95.4	–	–
Green et al.										
	T_{40h}	6.92	9.54	3.66	2.55	0.31	1.61	17.98	–	–
GraphLab										
	T_{40h}	58.0	56.1	51.3	3.45	1.7	2.33	5.7	178.7	–
TC-Merge	T_1	106	155	60.4	15.2	3.22	10.7	94.1	2680	1740
	T_{40h}	3.13	3.89	1.75	0.49	0.079	0.389	1.92	55.9	77.7
	T_1/T_{40h}	33.9	39.8	34.5	31.0	40.8	27.5	49.0	47.9	22.4
TC-Hash	T_1	193	279	107	27.5	6.92	19.5	158	4850	2960
	T_{40h}	5.33	7.21	3.25	0.931	0.198	0.723	3.3	93	104
	T_1/T_{40h}	36.2	38.7	32.9	29.5	34.9	27.0	47.9	50.2	28.5
TC-Local	T_1	119	166	64.9	17.3	3.72	12.2	101	2900	2090
	T_{40h}	3.28	3.99	1.76	0.639	0.088	0.397	2.09	163	90.7
	T_1/T_{40h}	36.3	41.6	36.9	27.1	42.3	30.7	48.3	17.8	23.0

Table 10.3: Triangle counting times (seconds) on the Intel machine: T_1 is single-thread time; T_{40h} the time on 40 cores with hyper-threading; and T_1/T_{40h} the parallel speedup.

10.7.2 Exact Triangle Counting

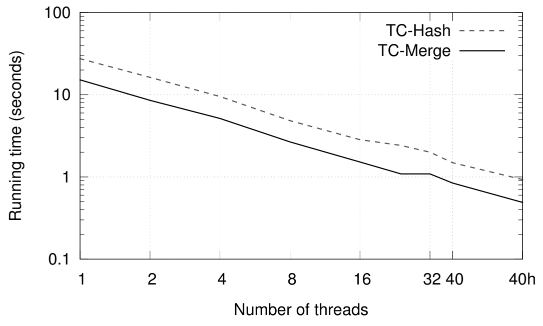
The first set of experiments is concerned with exact triangle counting. The times on the Intel machine are shown in Table 10.3, and the times on the AMD machine are shown in Table 10.4. The times include both ranking and counting, and are based on a median of three trials. The parallel speedup is also reported by dividing the time on a single thread by the parallel time (40 cores with hyper-threading for the Intel machine and 64 cores for the AMD machine). For some graphs, a speedup factor of over 40 is obtained on the Intel machine due to the effects of hyper-threading. Overall, the times on the Intel machine are faster than on the AMD machine, but the parallel speedups are comparable. This section later discusses the parallel performance of the algorithms developed in this chapter is compared with recent parallel/distributed algorithms.

Several things are worth discussing: First, *the single-threaded performance of TC-Merge is competitive with existing implementations*. To see whether the implementations incur high overhead due to parallelization, we ran the Ortmann and Brandes serial implementations (***serial-OB***) [361] on the same set of graphs and report the running time for their best implementation on each input on the Intel machine (Table 10.3). We do not have their times on the Twitter and Yahoo! graphs, as we could not run their implementations on them. When running single-threaded, the TC-Merge implementation is faster than their implementation. Their paper includes a comprehensive evaluation of other serial algorithms, which are described in Section 10.9.

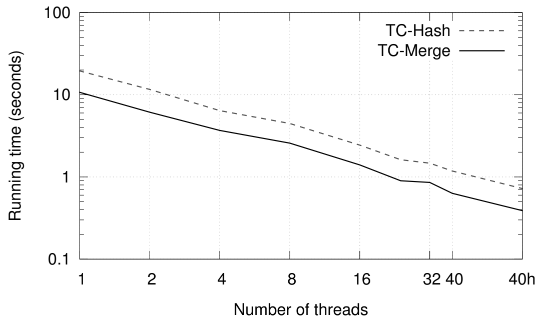
Second, *both TC-Merge and TC-Hash obtain very good speedups on all graphs, between*

Algorithm		random	rMat	3D-grid	soc-LJ	Patents	com-LJ	Orkut	Twitter	Yahoo!
TC-Merge	T_1	188	283	72.4	20.2	4.29	14.3	122	3730	2420
	T_{64}	4.93	6.18	2.68	0.81	0.155	0.623	2.67	78.9	100
	T_1/T_{64}	38.1	45.8	27.0	24.9	27.7	23.0	45.7	47.3	24.2
TC-Hash	T_1	274	416	184	33.4	11.1	23.8	173	6050	4340
	T_{64}	8.26	12.0	4.95	1.39	0.321	1.12	4.24	133	183
	T_1/T_{64}	33.2	34.7	37.2	24.0	34.6	21.3	40.8	45.5	23.7
TC-Local	T_1	168	268	79.1	24.5	5.3	17.2	134	4100	4060
	T_{64}	5.29	6.26	2.65	0.886	0.172	0.628	3.15	164	152
	T_1/T_{64}	31.8	42.8	29.8	27.7	30.8	27.4	42.5	25.0	26.7

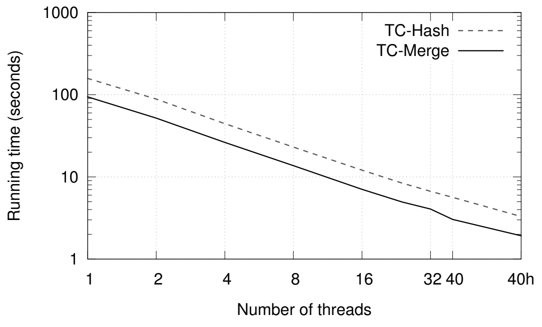
Table 10.4: Triangle counting times (seconds) on the AMD machine: T_1 is single-thread time; T_{64} the time on 64 cores; and T_1/T_{64} is the speedup.



(a) soc-LJ



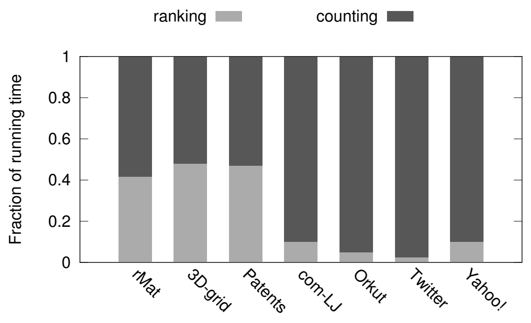
(b) com-LJ



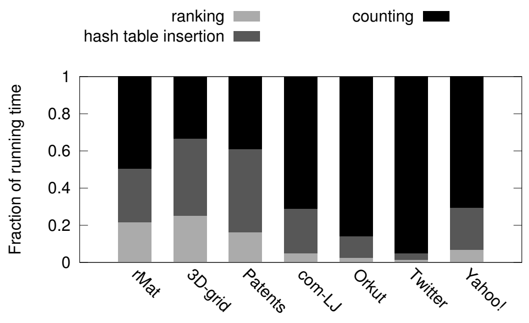
(c) Orkut

Figure 10.3: Times (seconds) for exact triangle counting (TC-Merge and TC-Hash) as the number of threads varies on a log-log scale. "40h" indicates 80 hyper-threads.

22–50x on 40 hyper-threaded cores, with TC-Merge having an edge over TC-Hash. For further detail, Figure 10.3 shows the running time versus the number of threads for several graphs on the Intel machine. Observe that both implementations scale well as the number of threads is increased, and that TC-Merge is faster than TC-Hash for all thread counts (by



(a) TC-Merge



(b) TC-Hash

Figure 10.4: Breakdown of times on 40 cores with hyper-threading on various graphs for TC-Merge and TC-Hash.

Algorithm	soc-LJ	Patents	com-LJ	Orkut
TC-Merge (L3 misses)	126M	58M	87M	762M
TC-Hash (L3 misses)	217M	90M	150M	1.2B
TC-Merge (L2 misses)	301M	134M	215M	1.4B
TC-Hash (L2 misses)	432M	182M	314M	1.8B
TC-Merge (ops)	2.54B	153M	1.7B	15.8B
TC-Hash (ops)	2.58B	164M	1.7B	18.4B

Table 10.5: L2 and L3 cache misses and work for intersection (ops) in TC-Merge and TC-Hash.

a factor of 1.3–2.5x). The trends are similar on the AMD machine, with TC-Merge again being faster than TC-Hash, although the absolute running times are slower than on the Intel machine.

Third, *for both implementations, usually the majority of the time is spent inside counting*. Figure 10.4 shows the breakdown of the parallel running times for the two implementations on the Intel machine. Observe that ranking usually takes a small fraction of the total time. For most of the real-world graphs (except Patents), the time for ranking in TC-Merge is at most 10%, although it is higher for the synthetic graphs (as high as 48% for the 3D-grid graph). This is because the number of potential triangles is much lower in the synthetic graphs, so the fraction of time spent in the counting portion of the computation is lower. For TC-Hash, at most 25% of the time is spent in ranking. The experiments also measure the time for inserting the edges into the hash tables, and the results show that for most of the real-world graphs this step takes longer than ranking, but less time than counting. For most of the real-world graphs (except Patents), this step also takes at most 25% of the total time.

Fourth, *despite the bounds, in practice, TC-Hash performs about the same amount of work as TC-Merge—but, as predicted from the theoretical bounds, TC-Hash incurs many* *more cache misses than the TC-Merge.* Table 10.5 shows the number of L2 and L3 cache misses for TC-Merge and TC-Hash on several input graphs. The numbers are collected on the 32-core Intel machine described in Section 2.7, since we did not have root access on the 40-core Intel machine. The cache misses reported are for an execution using all hyper-threads; however, the cache misses for all thread counts was similar.

Table 10.5 also reports the total number of operations inside `intersect` for each implementation. For TC-Merge, the number of operations is computed by the number of comparisons done in the merge between elements in the adjacency lists. For TC-Hash, the number of operations is computed by the number of locations inspected in the hash table, for both insertions and finds. The reader can observe that TC-Hash performs about the same amount of work as TC-Merge, but the key differentiating factor is the number of cache misses. This confirms that cache efficiency is crucial for algorithm performance.

Parallel Pagh-Silvestri Algorithm. Pagh and Silvestri (PS) [364] recently present a sequential cache-oblivious algorithm, which my co-author and I parallelize and experiment with (more details appear in Section 10.8). We found that our parallel PS implementation achieves reasonable parallel self-relative speedup; however, it is orders of magnitude slower than TC-Merge and TC-Hash. When run sequentially, we also found it to be orders of magnitude slower than other sequential triangle counting algorithms. This is because the PS algorithm makes many more passes over the edges of the graph, and does many sorts, which makes it expensive in practice. As far as we know, there is no public implementation of the PS algorithm available. Engineering the algorithm to run fast in practice, both sequentially and in parallel, would be an interesting direction for future work.

Comparison with other work. Several parallel triangle counting algorithms for distributed-memory have been proposed, and run on recent machines with comparable specifications to ours. Arifuzzaman et al. [14] propose PATRIC, which is an MPI-based parallel algorithm based on a variant of the node-iterator algorithm. Using 200 processors, they require 9.4 minutes to process the Twitter graph. Park and Chung [368] propose a MapReduce algorithm for counting triangles, which requires 213 minutes to process the Twitter graph on a cluster server with 47 nodes. They show that their algorithm outperforms the MapReduce algorithms of Cohen [104] and Suri and Vassilvitskii [439]. The MapReduce triangle enumeration algorithm of Park et al. [369] takes several hours on the Twitter graph, although they are solving the more expensive task of enumerating all triangles instead of just counting them. GraphLab implements triangle counting using MPI, and achieves better performance than the other algorithms—they process the Twitter graph in 1.5 minutes using 64 16-core machines [186]. In contrast to the distributed-memory algorithms, the TC-Merge algorithm from this chapter is able to process Twitter in under a minute on a single 40-core machine. Note that while the algorithms in this chapter are much faster than the distributed-memory algorithms, they are constrained to graphs that fit in the memory of a single machine.

The experiments in this section also compare with the implementations of Green et al. [192], the fastest in-memory implementations of triangle counting. The parallel time on the Intel machine for their fastest implementation per graph is reported in Table 10.3 for the graphs which their implementations successfully ran on. Their times do not include the time for sorting the edges per vertex (required for merging), although this would be a small fraction of the total time for most graphs. In parallel, TC-Merge is 2–9 times faster than their fastest algorithm. Their algorithms are parallel versions of the node-iterator algorithm without any ordering heuristic, and uses merging for intersection. Therefore their algorithms take $O(\sum_{v\in V}(d(v)^2+\sum_{w\in N(v)} d(w)))$ work, which in general is higher than the work of our algorithms. We believe that the difference in empirical performance between their algorithms and ours is largely due to the algorithmic difference. They also perform load balancing by estimating the work per vertex and dividing vertices and edges among threads appropriately, whereas we take the simpler approach of leaving the scheduling to the run-time system, which we found to work well in practice. In addition, the experiments compare with running GraphLab on a single machine (the 40-core Intel machine) and the times are reported in Table 10.3 for all of the input graphs except Yahoo!, which caused their program to thrash. The experiments show it to be several times slower than the implementations from this chapter as well as the implementations of Green et al. [192], as the GraphLab implementation is designed for distributed-memory using MPI, which has additional overheads when run on a single machine.

It is worth noting that there has been recent work showing that hash-based joins are usually better than sort-merge-based joins on multicores [26]. However, the setting of this work is that only two tables are joined and hence only a single join needs to be performed. Thus, the cost for sorting and hash table insertions dominate the cost. In contrast, in triangle computations each vertex participates in many intersections, but the sorting and hash table insertions for each vertex only needs to be done once, so this pre-processing cost is amortized over all of the subsequent intersections. Another difference is that for a single hash-based join, the elements of the smaller set are inserted into a hash table, with the elements from the larger set querying it, while to obtain good complexity bounds for triangle computations, the elements from the smaller adjacency list are queried in the hash table of the vertex with a larger adjacency list. Therefore, the conclusion of [26] does not directly apply to the context of this chapter.

10.7.3 Approximate Triangle Counting

The previous section showed that TC-Merge is fast and scales well with the number of threads. This section studies the parallel approximate triangle counting implementation from Section 10.5, which sparsifies the input graph using the colorful triangle counting scheme of Pagh and Tsourakakis [365], and applies TC-Merge on the sampled subgraph.

TC-Approx		random	rMat	3D-grid	soc-LJ	Patents	com-LJ	Orkut	Twitter	Yahoo!
$p = 1/25$	T_1	43.5	47.8	30.1	1.39	1.05	1.11	2.64	42.4	300
	T_{40h}	1.38	1.54	0.95	0.04	0.031	0.033	0.067	2.4	9.1
	Err.(%)	0.48	3.06	0.0	0.31	0.99	0.48	0.23	0.1	0.39
	σ^2	0.003	0.11	0.0	0.001	0.014	0.003	0.0	0.0	0.002
$p = 1/10$	T_1	56.5	62.3	40.1	1.77	1.22	1.39	4.05	79.4	350
	T_{40h}	1.6	1.77	1.11	0.05	0.036	0.042	0.1	5.88	14.5
	Err.(%)	0.19	0.8	0.0	0.34	0.38	0.4	0.17	0.12	0.18
	σ^2	0.0	0.007	0.0	0.002	0.003	0.001	0.0	0.0	0.0

Table 10.6: Times (seconds) and accuracy for approximate triangle counting on the Intel machine for $p = 1/25$ (**top**) and $p = 1/10$ (**bottom**). T_1 indicates single-thread time, and T_{40h} indicates the time on 40 cores with hyper-threading.

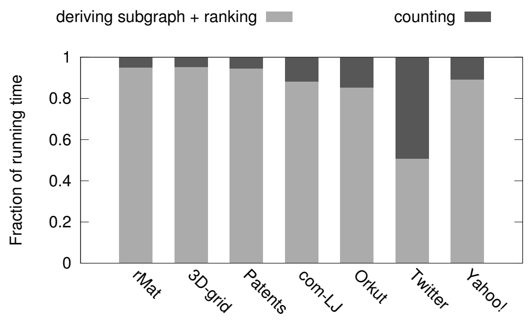


Figure 10.5: Breakdown of time for TC-approx on 40 cores with hyper-threading.

This algorithm is referred to as ***TC-Approx***. In the implementation, the ranking step is combined with the subgraph creation step to improve overall performance. In addition, the implementation operates directly on the adjacency list representation, and has each vertex separately apply a filter on its edges, instead of converting to the edge array representation and back as described in Section 10.5. While this adds an extra $O(V)$ term to the cache complexity, it performs better in practice as less work is performed.

The times on the 40-core Intel machine for $p = 1/25$ and $p = 1/10$ are shown in Table 10.6. The times include sampling edges from the original graph, and performing ranking and counting on the sampled subgraph. The reported times are based on an average of 10 trials, and the average error and variance of the estimates are also reported. The reader can observe that the times are much lower than those for exact triangle counting, and the error and variance of the estimates are very small and well-controlled. For graphs where the number of edges is much larger than the number of vertices, the speedup of TC-Approx over TC-Merge in parallel is significant (28.7x for Orkut and 23.3x for Twitter with $p = 1/25$), although for sparser graphs the savings is not as high. For the real-world graphs, the average error is less than 1% for a sampling factor of $p = 1/25$.

Figure 10.5 shows the breakdown of the parallel running time on the Intel machine of

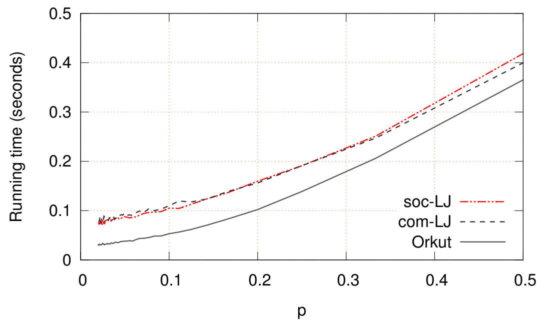


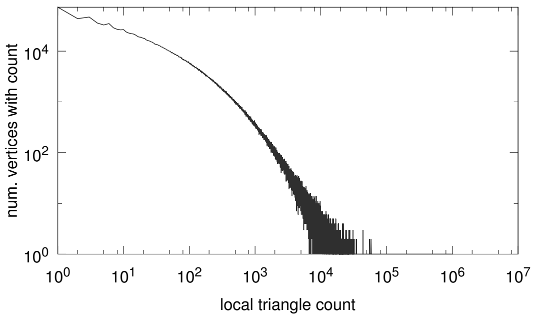
Figure 10.6: The fraction of time taken by TC-Approx relative to TC-Merge without sampling (vertical axis) as the sampling rate p (horizontal axis) varies, on the input graphs soc-LJ, com-LJ, and Orkut.

TC-Approx for $p = 1/25$. The time spent on computing the subgraph and ranking is a large fraction of the total time (at least 80% for all graphs except for the Twitter graph) because all of the edges are inspected. In contrast, the time spent on counting is a small fraction of the overall time for most graphs because there are much fewer edges in the sampled subgraph than in the original graph.

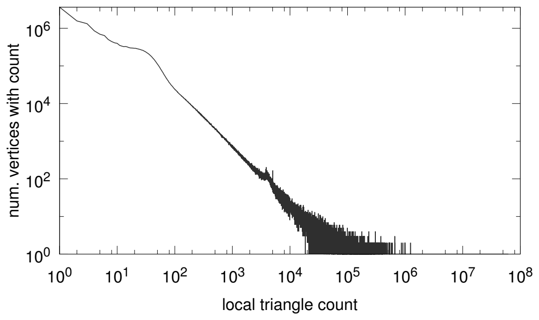
Figure 10.6 shows the parallel running time of TC-Approx relative to TC-Merge on the Intel machine as a function of the parameter p for several graphs. Overall, the time goes up as p increases, as this corresponds to a larger sample of edges.

Comparison with other work. TC-Approx is much faster than the multicore algorithm for approximate triangle counting by Rahman and Al Hasan [387]. For the Wikipedia-2007-02-06 graph[4] that they report times for (which has 3.566 million vertices and 42.375 million undirected edges), on 16 threads TC-approx obtains a 99.5% accuracy in 0.13 seconds (for $p = 1/10$), while they require 10.68 seconds to achieve 99.07% accuracy using 16 threads. The machines used in both cases are comparable, but even after adjusting for any small differences, TC-Approx would still be significantly faster. The exact algorithm TC-Merge is also faster than their algorithm on the same graph, running in 1.45 seconds on 16 threads. Recent work has extended wedge sampling to the MapReduce setting [272]. Their experiments use 32 4-core machines with hyper-threading, and they show that the overhead of MapReduce in their algorithm is already 225 seconds, and require about 10 minutes on the Twitter graph, which is slower than the parallel times for exact counting using 40 cores shown in Table 10.3. Papers for other approximate algorithms [449, 365] do not have parallel running times, and so could not be compared against.

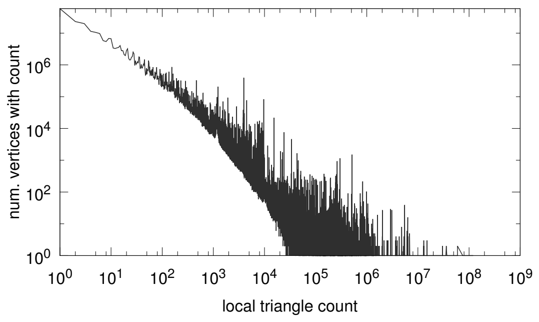
[4]http://www.cise.ufl.edu/research/sparse/matrices/



(a) Orkut



(b) Twitter



(c) Yahoo!

Figure 10.7: Distribution of local triangle counts (log-log scale), showing local triangle count (horizontal axis) vs. the number of vertices with that count (vertical axis).

10.7.4 Local Triangle Counting

We have also implemented a parallel algorithm for local triangle counting (***TC-Local***). For this algorithm, we modify TC-Merge to keep a count for every vertex in the graph. We use an atomic add (using the `x86` atomic instruction `xadd`) to a global array of local counts when a triangle is found. We use the following optimization to reduce work/contention: for a triangle discovered by looping over vertex v and vertex $w \in A^+[v]$ with a third vertex being $u \in A^+[v] \cap A^+[w]$, we atomically increment the count of u when it is discovered; for the second endpoint (w), we atomically add the count (if nonzero) after the intersection $A^+[v] \cap A^+[w]$ is finished, and for the first endpoint (v), we atomically add the count (if nonzero) after all merges with neighbors are finished.

The experiments show that *TC-Local also scales well with the number of cores*. Tables 10.3 and 10.4 show the times for local triangle counting (TC-Local) on the Intel machine and the AMD machine, respectively. As expected, it is slightly slower than global triangle counting because whenever a triangle is found, an atomic increment to a global array is performed (which likely involves a cache miss). Compared to TC-Merge, on 40 cores with hyper-threading on the Intel machine TC-Local is at most 30% slower for most graphs, but almost 3 times slower for the Twitter graph possibly due to contention with the atomic increment (Twitter has many high-degree vertices). TC-Local achieves 17–48x speedup over all of the inputs. The trends on the AMD machine are similar, although the absolute running times are slower.

As a simple application, this section extends the analysis of Tsourakakis [447] to much larger graphs. Tsourakakis observes that in real-world graphs the relationship between local triangle count and the number of vertices with such a count follows a power law [447], though the graphs used were much smaller than the inputs used in this section. Figure 10.7 plots the relationship in log-log scale for the larger real-world input graphs and confirms that this relationship does indeed quite closely resemble a power law. Due to the efficiency of the algorithm developed in this chapter, these plots for some of the largest publicly-available graphs can be generated in just a few minutes.

10.8 Parallelization of the Pagh-Silvestri Algorithm

Pagh and Silvestri [364] recently describe a sequential cache-oblivious algorithm for triangle enumeration with an expected cache complexity of $O(m^{3/2}/(\sqrt{M}B))$. This section reviews their sequential algorithm and then shows how to parallelize it. Their algorithm uses the edge array representation of the graph, which uses an array of length m storing pairs of vertices that have an edge between them.

Pagh and Silvestri first show that enumerating all triangles containing a given vertex v can be done with $O(\text{sort}(m))$ cache misses. They do this by (1) finding all of v's neighbors via a scan and sorting them lexicographically, (2) sorting the edge array by the source vertex and intersecting it with v's neighbors to get the outgoing edges of v's neighbors, and (3) sorting the result of step 2 by target vertex and intersecting it with v's neighbors to get all edges with both endpoints in v's neighbor set. The result of this is all the triangles incident on v. Since these operations are known to be implementable in a parallel and cache-oblivious manner, this gives the following lemma:

Lemma 25. *There is an algorithm for enumerating all triangles incident on a vertex v that requires $O(m \log m)$ work, $O(\log^{3/2} m)$ depth, and $O(\text{sort}(m))$ cache misses w.h.p.*

However, naively using this for each vertex is too costly, and hence their algorithm only uses this step for high-degree vertices and then uses a novel coloring scheme to recursively solve the problem on subgraphs. Using their definitions, a triangle (u, v, w) satisfies the (c_0, c_1, c_2) coloring if $c(u) = c_0$, $c(v) = c_1$ and $c(w) = c_2$ where c is the coloring function. An edge (u, v) is *compatible* with a coloring (c_0, c_1, c_2) if $(c(u), c(v)) \in$

$\{(c_0, c_1), (c_1, c_2), (c_0, c_2)\}$. The ***Pagh-Silvestri (PS) algorithm*** is a recursive algorithm with 3 steps:

Algorithm 16 Pagh-Silvestri (PS) algorithm

```
procedure PS-ENUM($G = (V, E), (c_0, c_1, c_2)$)
  (1) For each high-degree vertex (degree at least $m/8$), enumerate all triangles satisfying the
      $(c_0, c_1, c_2)$ coloring, and construct $G'$ by removing these high-degree vertices.
  (2) On $G'$, assign new colorings to the vertices by adding a random bit to its least significant
      position in its current coloring.
  (3) Recursively call PS-ENUM on $G'$ on the 8 colorings in $(c'_0, c'_1, c'_2) \in \{2c_0 - 1, 2c_0\} \times$
      $\{2c_1 - 1, 2c_1\} \times \{2c_2 - 1, 2c_2\}$, where each subproblem contains only compatible edges.
```

The algorithm is initially called on the original edge set E with a coloring $(1, 1, 1)$, and all vertices assigned a color of 1.

Step 1 applies the subroutine described above to at most 16 vertices, and so requires $O(\text{sort}(m))$ cache misses. Step 2 requires $O(\text{scan}(n))$ cache misses. Pagh and Silvestri show that each subproblem in step 3 contains at most $m/4$ edges in expectation and uses this to show an expected cache complexity of $O(m^{3/2}/(\sqrt{M}B))$. The work of their algorithm is $O(m^{3/2})$.

Each of the three steps of the PS algorithm can be parallelized, as discussed below. Step 1 requires at most 16 calls to the subroutine that finds all triangles incident on a vertex, hence can be done in the bounds stated in Lemma 25. Step 2 can be implemented with a parallel scan in $O(n)$ work, $O(\log n)$ depth, and $O(\text{scan}(n))$ cache misses. The new colors of the endpoints of the edges can be computed by sorting the edges by the first endpoint, merging with the array of colors, then sorting by the second endpoint and doing the same. For each subproblem in Step 3, generating the subset of edges belonging to the subproblem can be done with a parallel filter in $O(m)$ work, $O(\log m)$ depth, and $O(\text{scan}(m))$ cache misses. As the expected size of each subproblem is at most $m/4$, there are $O(\log m)$ levels of recursion w.h.p. This gives an overall depth of $O(\log^{5/2} m)$ w.h.p. The parallel algorithm requires $O(m^{3/2})$ work since every sequential routine that is replaced with a parallel routine has the same asymptotic work bound. The parallel cache complexity is $O(m^{3/2}/(\sqrt{M}B))$ in expectation as the cache complexity of the parallel routines match those of the sequential routines. This gives the following theorem:

Theorem 25. *A parallel version of the PS algorithm can be implemented in $O(m^{3/2})$ work, $O(\log^{5/2} m)$ depth and a parallel cache complexity of $O(m^{3/2}/(\sqrt{M}B))$ in expectation.*

While the cache complexity of the parallel PS algorithm is better than that of TC-Merge and TC-Hash, in practice we found our implementation to be much slower due to large constants in the bounds, as discussed in Section 10.7.

10.9 Prior and Related Work

Exact sequential algorithms. Sequential algorithms for exact triangle counting and enumeration have a long history (see, e.g., [240, 408, 407, 292, 361]). For sparse graphs, of particular interest is the line of work starting from Schank and Wagner [408], who describe an algorithm, called *forward*, that achieves a work bound of $O(m^{3/2})$ with a space bound of $O(n+m)$. The algorithm ranks the vertices in order of non-decreasing degree, but it populates the neighborhood A^+ sequentially while computing the intersection. Improving upon the constants in the space bounds, Latapy [292] describes an algorithm *compact-forward*, on which the algorithms in this chapter are based. Both algorithms are sequential and require $O(m^{3/2})$ work and $O(n+m)$ space. By using hash tables for intersection, the work of both algorithms can be improved to $O(\alpha m)$ [92]. Experimentally, Latapy shows that forward and compact-forward yield the best running time with compact-forward consuming less space [292], consistent with Schank's findings [407].

The *node-iterator* algorithm [407] iterates over all vertices $v \in V$, and intersects the adjacency lists of each pair of v's neighbors. This algorithm requires $O(\sum_{v\in V}(d(v)^2 + \sum_{w\in N(v)} d(w))) = O(md_{max})$ work and $O(n+m)$ space. Green and Bader describe an optimization to this algorithm using vertex covers, which improves its performance in practice [191]. The *edge-iterator* algorithm [240] iterates over the edges instead of the vertices. For each edge, it intersects the adjacency lists of the two endpoints.

Ortmann and Brandes [361] describe a framework for designing triangle listing algorithms and explore many variations of the previous algorithms.

For a graph with Δ triangles, Bjorklund et al. [44] give the best work bounds for triangle listing, requiring roughly $O(n^\omega + n^{3(\omega-1)/(5-\omega)}\Delta^{2(3-\omega)/(5-\omega)})$ work for dense graphs, and $O(m^{2\omega/(\omega+1)} + m^{3(\omega-1)/(\omega+1)}\Delta^{(3-\omega)/(\omega+1)})$ work for sparse graphs, where ω is the matrix multiplication exponent ($\omega \approx 2.3729$, using the current-best algorithm [464]).

Triangle counting, but not listing, can also be solved using matrix multiplication in $O(n^\omega)$ work [240]. For sparse graphs, this can be improved to $O(m^{2\omega/(\omega+1)})$ [9]. Other algorithms and variants can be found in [407, 292, 361] and the references therein.

Exact parallel algorithms. There has been recent work on adapting sequential triangle counting/listing/enumeration algorithms to the parallel setting. Several algorithms have been designed for distributed-memory using MapReduce [104, 439, 368, 456, 369]. Arifuzzaman et al. describe a distributed-memory algorithm using MPI [14], and GraphLab also contains an MPI implementation [186]. A multicore implementation of the node-iterator algorithm is presented by Green et al. [192]. Triangle counting has also been implemented on the GPU [465, 193].

I/O complexity of triangle computations. Various triangle counting/listing/enumeration algorithms have been designed for I/O efficiency, either in terms of disk accesses or cache misses. Triangle enumeration can be computed by using a natural join of three relations using $O(m^3/(M^2B))$ I/O's [364]. An external-memory version of compact-forward was described by Menegola [331], requiring $O(m + m^{3/2}/B)$ I/O's. An external-memory version of node-iterator was described by Dementiev [131], requiring $O((m^{3/2}/B)\log_{M/B}(m/B))$ I/O's. Chu and Cheng [98] describe an algorithm using graph partitioning with an I/O complexity $O(m^2/(MB) + \Delta/B)$, where Δ is the number of triangles in the graph. Their algorithm requires that each partitions fits in memory, that $n \leq M$, and that $M = \Omega(\sqrt{mB})$. Later, Hu et al. [233] describe an algorithm achieving the same I/O complexity of $O(m^2/(MB) + \Delta/B)$, without the restrictions of the previous algorithm. These algorithms are designed for the external-memory model, where the algorithm must be tuned for the parameters M and B of the specific machine. Recently, Pagh and Silvestri [364] describe a cache-oblivious algorithm requiring $O(m^{3/2}/(\sqrt{M}B))$ expected I/O's (cache misses), which is described in Section 10.8. They also describe a deterministic cache-aware algorithm requiring $O(m^{3/2}/(\sqrt{M}B))$ I/O's (cache misses) with the requirement $M \geq m^{\Omega(1)}$ [364]. None of the above algorithms have been parallelized. Kyrola et al. [289] and Kim et al. [268] present parallel disk-based triangle counting implementations, which require parameter tuning.

Approximate counting schemes. To speed up triangle counting, many approximation schemes have been proposed. These do not work for triangle listing/enumeration, as not all triangles are even generated. DOULION is among the first approximation schemes proposed [449]. Pagh and Tsourakakis [365] later give a more accurate scheme that improves upon DOULION, called colorful triangle counting, which is described in Section 10.5. A recent scheme based on sampling wedges was presented by Seshadri et al. [411]. Hadoop implementations have been described for some of these schemes (e.g., [365, 456, 272]). Several other approximation schemes have been proposed based on computing eigenvalues of the graph [17, 446, 448]. The performance of these methods depend on the spectrum of the graphs. Rahman and Al Hasan recently present approximate counting algorithms for multicores based on the edge-iterator algorithm [387], which is compared with in Section 10.7.

Streaming algorithms. Triangle counting has also been studied in streaming settings as an alternative means to processing massive graphs (see, e.g., [33, 28, 273, 80, 141, 245, 373, 440, 287] among many others).

Part IV

Parallel String Algorithms

Introduction

This part of the thesis develops shared-memory string algorithms that are efficient both in theory and in practice. Chapter 11 presents a simple linear-work and space, and polylogarithmic depth parallel algorithm for generating multiway Cartesian trees, and uses it in conjunction with a suffix array algorithm to generate suffix trees in linear work and polylogarithmic depth. This gives the first linear-work and polylogarithmic-depth parallel suffix tree algorithm that is also practical. Chapter 12 proposes simple parallel algorithms for computing the longest common prefix (LCP) array given the suffix array as input, and shows that they are efficient both in theory and in practice. In Chapter 13, a simple linear-work and polylogarithmic-depth parallel algorithm for Lempel-Ziv factorization based on suffix arrays is presented, resulting in the first practical parallel algorithm for this problem that is also theoretically-efficient. Finally, Chapter 14 develops the first polylogarithmic-depth parallel algorithms for constructing wavelet trees, a building block for many compressed data structures. Each chapter presents experimental results on a modern multicore machine showing that the implementations of the parallel algorithms outperform existing parallel implementations for the same problem, and achieve significant speedups over the corresponding sequential solutions.

The results in this part of the thesis have appeared in the following publications:

- Julian Shun and Guy Blelloch. A Simple Parallel Cartesian Tree Algorithm and its Application to Parallel Suffix Tree Construction, *ACM Transactions on Parallel Computing (TOPC)*, Vol. 1 Issue 1, Article No. 8, 2014.
- Julian Shun. Fast Parallel Computation of Longest Common Prefixes. *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 387–398, 2014.
- Julian Shun and Fuyao Zhao. Practical Parallel Lempel-Ziv Factorization. *Proceedings of the IEEE Data Compression Conference (DCC)*, pp. 123–132, 2013.
- Julian Shun. Parallel Wavelet Tree Construction. *Proceedings of the IEEE Data Compression Conference (DCC)*, pp. 63–72, 2015.

Chapter 11

Parallel Cartesian Tree and Suffix Tree Construction

11.1 Introduction

A Cartesian tree on a sequence of elements taken from a total order is a binary tree that satisfies two properties: (1) heap order on values, i.e. a node has an equal or lesser value than any of its descendants, and (2) an in-order traversal of the tree defines the sequence order.

Given the suffix array SA and its corresponding LCP array (refer to Section 2.6.3 for the definitions) for a string, a Cartesian tree on the string formed by interleaving SA and LCP can be used to answer queries related to the string. By adding downward pointers (e.g., using a hash table), this gives a suffix tree for binary alphabets. The approach can be generalized to arbitrary alphabets by using multiway Cartesian trees (Cartesian trees where connected components of equal value are contracted into single nodes) without much difficulty.

For a string S of length n over a character set $\Sigma \subseteq \{0, \ldots, n-1\}$[1] the suffix tree data structure stores all the suffixes of s in a patricia tree (defined in Section 2.6.3). In addition to supporting searches in S for any string $t \in \Sigma^*$ in $O(|t|)$ expected work,[2] suffix trees efficiently support many other operations on strings, such as finding the longest common substring, maximal repeats, longest repeated substrings, and the longest palindrome, among many others [201]. As such, it is one of the most important data structures for string processing. For example, it is used in several bioinformatic applications, such as REPuter [286], MUMmer [129], OASIS [327], and Trellis+ [378, 377]. Both suffix trees and a linear-work

[1]More general alphabets can be used by first sorting the characters and then labeling them from 0 to $n-1$.
[2]Worst-case work for constant-sized alphabets.

algorithm for constructing them were introduced by Weiner [460] (although he used the term position tree). Since then various similar constructions have been described [325, 450] and there have been many implementations of these algorithms. Although originally designed for fixed-sized alphabets with deterministic linear work, Weiner's algorithm can work on an alphabet $\{0, \ldots, n-1\}$, henceforth $[n]$, in linear expected work simply by using hashing to access the children of a node.

The algorithm of Weiner and its derivatives are all incremental and inherently sequential. The first parallel algorithm for suffix trees was given by Apostolico et al. [13] and was based on a quite different doubling approach. For a parameter $0 < \epsilon \leq 1$ the algorithm runs in $O((1/\epsilon)\log n)$ depth, $O((n/\epsilon)\log n)$ work and $O(n^{1+\epsilon})$ space on the CRCW PRAM for arbitrary alphabets. Although reasonably simple, this algorithm is likely not practical since it is not work-efficient and uses super-linear memory (by a polynomial factor). The parallel construction of suffix trees was later improved to linear work and polynomial space by Sahinalp [404], with an algorithm taking $O(\log^2 n)$ depth on the CRCW PRAM (they note that linear space can be obtained by using hashing and randomization) and linear work and linear space by Hariharan [215], with an algorithm taking $O(\log^4 n)$ depth on the CREW PRAM. Farach and Muthukrishnan improved the depth to $O(\log n)$ with high probability on the CRCW PRAM [148]. These later results are for constant-sized alphabets, are "considerably non-trivial", and do not seem to be amenable to efficient implementations.

As mentioned earlier, one way to construct a suffix tree is to first generate a suffix array and then convert it to a suffix tree using a Cartesian tree algorithm. Using suffix arrays is attractive since in recent years there has been considerable theoretical and practical advances in the generation of suffix arrays (see, e.g., [384]). The interest is partly due to their need in the widely used Burrows-Wheeler compression algorithm [81], and also as a more space-efficient alternative to suffix trees. As such, there have been dozens of papers on efficient implementations of suffix arrays. Among these, Kärkkäinen, Sanders, and Burkhardt have developed a quite simple and efficient parallel algorithm for suffix arrays [256, 257] that can also generate the lcp values.

The story with generating Cartesian trees in parallel is less satisfactory. Berkman et al. [38] describe a parallel algorithm for the all nearest smaller values (ANSV) problem, which can be used to generate a binary Cartesian tree. However, it cannot directly be used to generate a multiway Cartesian tree, and the algorithm is very complicated. Iliopoulos and Rytter [236] present two much simpler algorithms for generating suffix trees from suffix arrays, one based on merging and one based on a variant of the ANSV problem that allows for multiway Cartesian trees. However they both require $O(n \log n)$ work.

This chapter describes a linear-work, linear-space, and polylogarithmic-depth algorithm for generating multiway Cartesian trees. The algorithm is based on divide-and-conquer and the chapter presents two versions that differ in whether the merging step is done sequentially or in parallel. The first based on a sequential merge, is very simple, and for a tree of height d, it runs in $O(\min\{d \log n, n\})$ depth. The second version is only slightly more complicated and runs in $O(\log^2 n)$ depth. They both use linear work and space.[3]

Using the multiway Cartesian tree algorithm in conjunction with any linear-work and space algorithm for generating a suffix array and corresponding LCPs using $O(S(n))$ depth results in a linear work and space algorithm for generating suffix trees in $O(S(n) + \log^2 n)$ depth. For example, using the skew algorithm [256, 257], the algorithm has $O(\log^2 n)$ depth with high probability for constant-sized alphabets and $O((1/\epsilon)n^\epsilon)$ depth $(0 < \epsilon < 1)$ for the alphabet $[n]$. It is worth noting that a polylogarithmic-depth, linear-work, and linear-space algorithm for the alphabet $[n]$ would imply stable radix sort on $[n]$ in the same bounds, which is a long-standing open problem [388].

For comparison, this chapter also presents a technique for using the ANSV problem to generate multiway Cartesian trees on arbitrary alphabets in linear work and space. The algorithm runs in $O(I(n) + \log \log n)$ depth, where $I(n)$ is the best depth bound for a linear-work stable sorting of integers from $[n]$. Of independent interest, this chapter shows that the Cartesian tree can be used to solve the ANSV problem in linear work and $O(\log^2 n)$ depth, and the algorithm is much simpler than that of previous work [38].

This chapter gives an implementation of the first version of the parallel Cartesian tree algorithm and presents various experimental results analyzing the algorithm on a shared-memory multicore machine on a variety of inputs. First, the parallel Cartesian tree algorithm is compared to a simple stack-based sequential implementation. On a single thread, the parallel algorithm is about 3x slower, but achieves about 35x speedup (about 12x with respect to the sequential implementation) on 40 cores with two-way hyper-threading. The chapter shows three queries on strings that can be answered with the Cartesian tree on the suffix array and LCP array of the string. First, the number of leaves in the subtree at each internal node of the Cartesian tree can be computed in order to support various types of queries relating to counts of repeated substrings in the input. As an example, the experiments use this information to compute the longest substring that occurs at least k times in the input. The third query is to compute the minimum position of a suffix in the subtree of internal nodes, which is useful for computing the Lempel-Ziv decomposition of a string. These computations only require some basic parallel operations and are fast compared to the suffix array and LCP construction times.

The experiments also analyze the Cartesian tree algorithm when used as part of code to generate a suffix tree from the original string. This code is compared to the ANSV-based algorithm described in the previous paragraph and to the fastest existing sequential implementation of suffix trees. The experiments show that the Cartesian tree-based algorithm

[3]Very recently, Poon and Yuan improve the depth bound by describing a modification to the algorithm in this chapter that runs in $O(n)$ work and $O(\log n)$ depth [381].

is always faster than the ANSV-based algorithm. The algorithm is competitive with the sequential code on a single thread, and achieves good speedup on 40 cores. The chapter presents timings for searching multiple strings in the suffix trees constructed using the algorithm developed in this chapter. On one thread, the search times are always faster than searching with the sequential suffix tree and are an order of magnitude faster on 40 cores using hyper-threading.

11.2 Preliminaries

This chapter uses the patricia tree and suffix tree, which are defined in Section 2.6.3, and assumes an integer alphabet $\Sigma \subseteq [n]$ where n is the total number of characters. The patricia tree and suffix tree are assumed to support the following queries on a node in constant expected work: finding the child edge based on the first character of the edge, finding the first child, finding the next and previous sibling in the character order, and finding the parent. If the alphabet size is constant, then all of these operations can easily be implemented in constant worst-case work.

A ***Cartesian tree*** [455] on a sequence of elements taken from a total order is a binary tree that satisfies two properties: (1) heap order on values, i.e. a node has an equal or lesser value than any of its descendants, and (2) an in-order traversal of the tree defines the sequence order. If the elements in the sequence are distinct then the tree is unique, otherwise it might not be. When elements are not distinct, a connected component of equal value nodes in a Cartesian tree is referred to as a ***cluster***. A ***multiway Cartesian tree*** is derived from a Cartesian tree by contracting each cluster into a single node while maintaining the order of the children. A multiway Cartesian tree of a sequence is always unique.

Let $LCP(s_i, s_j)$ be the length of the longest common prefix of S_i and S_j. Given a sorted sequence of strings $\mathcal{S} = [\mathsf{S}_1, \ldots, \mathsf{S}_n]$, if the string lengths are interleaved with the length of their longest common prefixes (i.e., $[| \mathsf{S}_1 |, LCP(\mathsf{S}_1, \mathsf{S}_2), | \mathsf{S}_2 |, \ldots, LCP(\mathsf{S}_{n-1}, \mathsf{S}_n), | \mathsf{S}_n |]$), then the corresponding multiway Cartesian tree has the structure of the patricia tree for $\mathcal{S}$. The patricia tree can be generated by adding strings to the edges, which is easy to do—e.g., for a node with value $v = LCP(\mathsf{S}_i, \mathsf{S}_{i+1})$ and parent with value v', the edge corresponds to the substring $\mathsf{S}_i[v' + 1, \ldots, v]$. As a special case, interleaving a suffix array with its lcp values for a string S and generating the multiway Cartesian tree gives the suffix tree structure for S. Adding the nodes to a hash table to allow for efficient downward traversals completes the suffix tree construction.

```
struct node { node* parent; int value; };

void merge(node* left, node* right) {
  node* head;
  if (left->value > right->value) {
    head = left; left = left->parent;}
  else { head = right; right= right->parent; }

  while(1) {
    if (left == NULL) { head->parent = right; break; }
    if (right == NULL) { head->parent = left; break; }
    if (left->value > right->value) {
      head->parent = left; left = left->parent; }
    else { head->parent = right; right = right->parent; }
    head = head->parent; }}

void cartesianTree (node* Nodes, int n) {
  if (n < 2) return;
  cilk_spawn cartesianTree (Nodes, n/2);
  cartesianTree (Nodes+n/2, n-n/2);
  cilk_sync;
  merge(Nodes+n/2-1, Nodes+n/2);}
```

Figure 11.1: C++ code for Algorithm 1a for constructing a Cartesian tree.

11.3 Parallel Cartesian Trees

This section develops a work-efficient parallel divide-and-conquer algorithm for constructing a Cartesian tree. The algorithm works recursively by splitting the input array A into two subarrays, generating the Cartesian tree for each subarray, and then merging the results into a Cartesian tree for A. Define the ***right-spine*** (***left-spine***) of a tree to consist of all nodes on the path from the root to the rightmost (leftmost) node of the tree. The merge works by merging the right-spine of the left tree and the left-spine of the right tree based on the value stored at each node. This algorithm is similar to the $O(n \log n)$ work divide-and-conquer suffix array to suffix tree algorithm of Iliopoulos and Rytter [236], but the most important difference is that our algorithm only looks at the nodes on the spines at or deeper than the deeper of the two roots and the fully parallel version of the algorithm developed in this chapter uses trees instead of arrays to represent the spines. This leads to the $O(n)$ work bound. In addition, Iliopoulos and Rytter's algorithm works directly on the suffix array rather than solving the Cartesian tree problem so the specifics are different.

Two versions of the algorithm are described: a partially parallel version of this algorithm (Algorithm 1a) and a fully parallel version (Algorithm 1b). Algorithm 1a is very simple, and takes up to $O(\min(d \log n, n))$ depth, where d is the depth of the resulting tree, although

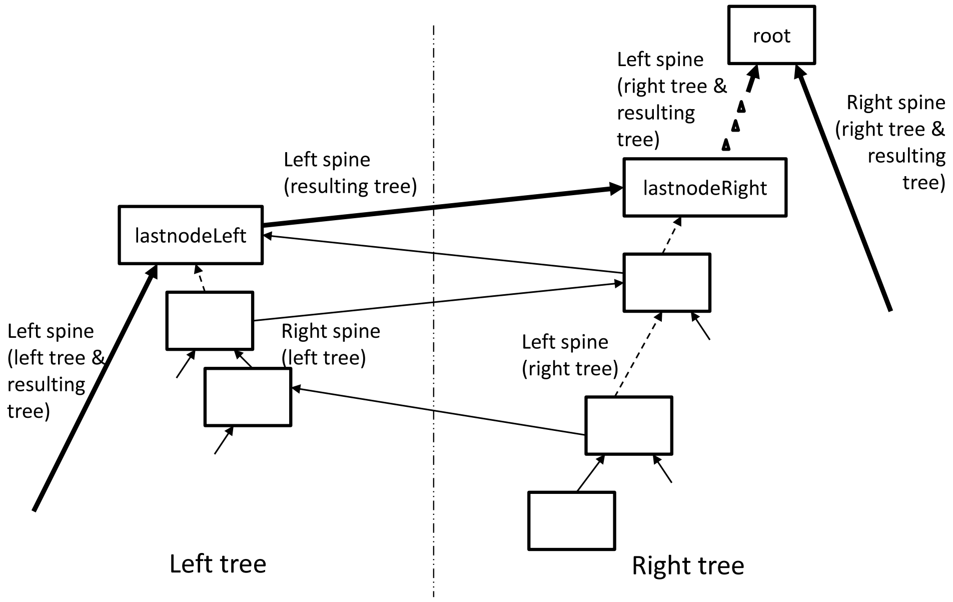


Figure 11.2: Merging two spines of Cartesian trees. Thick lines represent the spines of the resulting tree; dashed lines represent edges that existed before the merge but not after the merge; dotted edges represent an arbitrary number of nodes; all non-dashed lines represent edges in the resulting tree.

for most inputs it takes significantly less depth (e.g., for the sequence $[0, 1, \ldots, n-1]$ it takes $O(\log n)$ depth even though the resulting tree has depth n). The algorithm only needs to maintain parent pointers for the nodes in the Cartesian tree. The complete `C++` code is provided in Figure 11.1 and line numbers from it will be referenced throughout our description.

The algorithm takes as input an array of n elements (`Nodes`) and recursively splits the array into two halves (Lines 19-21), creates a Cartesian tree for each half, and then merges them into a single Cartesian tree (Line 22). For the merge (Lines 3-15), the algorithm combines the right spine of the left subtree with the left spine of the right subtree (see Figure 11.2). The right (left) spine of the left (right) subtree can be accessed by following parent pointers from the rightmost (leftmost) node of the left (right) subtree. The leftmost and rightmost nodes of a tree are simply the first and last elements in the input array of nodes. Note that once the merge reaches the deeper of the two roots, it stops and needs not process the remaining nodes on the other spine. The code in Figure 11.1 does not keep child pointers since they are not needed for the experiments, but it is easy to add a left and right child pointer and maintain them.

Theorem 26. *Algorithm 1a produces a Cartesian tree on its input array.*

Proof. The proof shows that at every step in the algorithm, both the heap and the in-order properties of the Cartesian trees are maintained. At the base case, a Cartesian tree of one node trivially satisfies the two properties. During a merge, the heap property is maintained because a node's parent pointer only changes to point to a node with equal or lesser value. Consider modifications to the left tree. Only the right children of the right spine can be changed. Any new right children of a node come from the right tree, and hence correspond to elements later in the original sequence. An in-order traversal will correctly traverse these new children of a node after the node itself. A symmetric argument holds for nodes on the left spine. Furthermore, the order within each of the two trees is maintained since any node that is a descendant on the right (left) in the trees before merging remains a descendant on the right (left) after the merge. □

Theorem 27. *Algorithm 1a for constructing a Cartesian tree requires $O(n)$ work, $O(\min(d \log n, n))$ depth and $O(n)$ space.*

Proof. The following definitions are used to help with proving the complexity bounds of the algorithm: A node in a tree is ***left-protected*** if it does not appear on the left spine of its tree, and a node is ***right-protected*** if it does not appear on the right spine of its tree. A node is ***protected*** if it is both left-protected and right-protected.

In the algorithm, once a node becomes protected, it will always be protected and will never have to be looked at again since the algorithm only ever processes the left and right spines of a tree. The proof shows that during a merge, all but two of the nodes that are looked at become protected, and the cost of processing those two nodes is charged to the merge itself. Call the last node looked at on the right spine of the left tree ***lastnodeLeft*** and the last node looked at on the left spine of the right tree ***lastnodeRight*** (see Figure 11.2).

All nodes that are looked at, except for lastnodeLeft and lastnodeRight will be left-protected by lastnodeLeft. This is because those nodes become either descendants of the right child of lastnodeLeft (when lastnodeLeft is below lastnodeRight) or descendants of lastnodeRight (when lastnodeRight is below lastnodeLeft). A symmetric argument holds for nodes being right-protected. Therefore, all nodes looked at, except for lastnodeLeft and lastnodeRight, become protected after this sequence of operations. The cost for processing lastnodeLeft and lastnodeRight is charged to the merge itself.

Other than when a node appears as lastnodeRight or lastnodeLeft, it is only looked at once and then becomes protected. Therefore, the total number of nodes looked at is $2n - 2$ for lastnodeRight or lastnodeLeft on the $n - 1$ merges, and at most n for the nodes that become protected for a total work of $O(n)$.

Although Algorithm 1a makes parallel recursive calls, it uses a sequential merge routine. In the worst case, this has depth equal to the depth of the tree per level of recursion. As there are $O(\log n)$ levels of recursion, the depth is $O(\min(d \log n, n))$.

Since each node only maintains a constant amount of data, the space required is $O(n)$. □

The algorithm can be converted to an PRAM algorithm by iterating over the levels of the recursion tree synchronously. Since each level evenly divides the problem size in half, the algorithm can easily assign cores to the sub-problems in constant work. The parallel recursive calls are on different parts of the data, so no concurrent reads/writes are needed, and hence it runs on the EREW PRAM.

A fully parallel version of the algorithm, referred to as Algorithm 1b, is described below. The algorithm maintains binary search trees for each spine, and substitutes the sequential merge with a parallel merge. The algorithm will use split and join operations on the spines. A split goes down the spine tree and cuts it at a specified value v so that all values less or equal to v are in one tree and values greater than v are in another tree. A join takes two trees such that all values in the second are greater than or equal to the largest value in the first and joins them into one. Both operations can run in depth proportional to the depth of the spine tree and the join adds at most one to the height of the larger of the two trees.

Without loss of generality, assume that the root of the right Cartesian tree has a smaller value than the root of the left Cartesian tree (as in Figure 11.2). For the left tree, the end point of the merge will be its root. To find where to stop merging on the right tree, the algorithm searches the left-spine of the right tree for the root value of the left tree and splits the spine at that point. Now it merges the whole right-spine of the left tree and the deeper portion of the left-spine of the right tree. After the merge, these two parts of the spine can be discarded since their nodes have become protected. Finally, the algorithm joins the shallower portion of the left spine of the right tree with the left spine of the left tree to form the new left spine. The right-spine of the resulting Cartesian tree is the same as that of the right Cartesian tree before the merge.

Theorem 28. *Algorithm 1b for constructing a Cartesian tree requires $O(n)$ work, $O(\log^2 n)$ depth, and $O(n)$ space.*

Proof. The trees used to represent the spines are never deeper than $O(\log n)$ since each merge does only one join, which adds only one node to the depth. All splits and joins therefore take $O(\log n)$ depth. The merge can be done using a parallel merging algorithm that runs in $O(\log n)$ depth and $O(n)$ work [207], where n is the number of elements being merged. The depth of Algorithm 1b's recursion is $O(\log n)$, which gives a $O(\log^2 n)$ depth bound. The $O(n)$ work bound follows from a similar analysis to that of Algorithm 1a, with the exception that splits and joins in the spine cost an extra $O(\log n)$ per merge, the extra cost follows a recurrence of $W(n) = 2W(n/2) + O(\log n)$, which solves to $O(n)$. The trees on the spines take linear space so the $O(n)$ space bound still holds. □

The parallel merging algorithm runs on the EREW PRAM so this algorithm can be mapped onto the EREW PRAM. Processor allocation on each level of recursion is straightforward to do within $O(\log n)$ depth.

Lemma 26. *The outputs of Algorithm 1a and Algorithm 1b can be used to construct a multiway Cartesian tree in $O(n)$ work and space. This requires $O(d)$ depth using path compression or $O(\log n)$ depth using tree contraction.*

Proof. Path compression can be used to compress all clusters of the same value into the root of the cluster, which can then be used as the "representative" of the cluster. All parent pointers to nodes in a cluster will now point to the "representative" of that cluster. This is done sequentially and requires linear work and $O(d)$ depth. Path compression can also be substituted with a parallel tree contraction algorithm, which requires $O(n)$ work and $O(\log n)$ depth [393]. $\square$

Both path compression and tree contraction can be done on the EREW PRAM.

For non-constant sized alphabets if one wants to search in the tree efficiently ($O(1)$ expected depth per edge), the edges need to be inserted into a hash table, which can be done in $O(\log n)$ depth and $O(n)$ work (both with high probability) [324], and the process can be done on a CRCW PRAM.

Corollary 4. *Given a suffix array for a string over the alphabet $[n]$ and the longest common prefixes between adjacent elements, a suffix tree can be generated in hash table format with Algorithm 1b, tree contraction and hash table insertion using $O(n)$ work and space, and $O(\log^2 n)$ depth with high probability.*

Proof. This follows directly from Theorem 28, Lemma 26 and the bounds for hash table insertion. $\square$

11.4 Cartesian Trees and the ANSV Problem

The ***all nearest smaller values (ANSV)*** problem is defined as follows: for each element in a sequence of elements from a total ordering, find the closest smaller element to the left of it and the closest smaller element to the right of it. This section augments the ANSV-based Cartesian tree algorithm of Berkman et al. [38] to support multiway Cartesian trees, and also shows how to use Cartesian trees to solve the ANSV problem.

The algorithm of Berkman et al. solves the ANSV problem in $O(n)$ work and $O(\log\log n)$ depth on the CRCW PRAM. The ANSV can then be used to generate a Cartesian tree by noting that the parent of a node has to be the nearest smaller value in one of the two directions (in particular, the larger of the two nearest smaller values is the parent). To convert their Cartesian tree to the multiway Cartesian tree, one needs to group all nodes pointing to the same parent and coming from the same direction together. If $I(n)$ is the best depth bound for stably sorting integers from $[n]$ using linear space and work, then the grouping can be done in linear work and $O(I(n) + \log\log n)$ depth by sorting on the parent ID numbers of the nodes. Stability is important since a suffix tree needs to maintain the relative order among the children of a node.

Theorem 29. *A multiway Cartesian tree on an array of elements can be generated in $O(n)$ work and space, and $O(I(n) + \log\log n)$ depth.*

Proof. This follows from the bounds of the ANSV algorithm and of integer sorting. □

It is not currently known whether $I(n)$ is polylogarithmic so at present this result seems weaker than the result from the previous section. The experimental section (Section 11.5) compares the algorithms on various inputs. In a related work, Iliopoulos and Rytter [236] present an $O(n \log n)$-work polylogarithmic-depth algorithm based on a variant of ANSV.

11.4.1 Cartesian Tree to ANSV

This section describes a method for obtaining the ANSVs from a Cartesian tree in parallel using tree contraction. Note that for any node in the Cartesian tree, both of its nearest smaller neighbors (if they exist) must be on the path from the node to the root (one neighbor is trivially the node's parent). First, a simple linear-work algorithm for the task that takes depth equal to the depth of the Cartesian tree is presented. Let d denote the depth of the tree, with the root being at depth 1. The following algorithm returns the left nearest neighbors of all nodes. A symmetric algorithm returns the right nearest neighbors.

1. For every node, maintain two variables, ***node.index*** which is set to the node's index in the sequence corresponding to the in-order traversal of the Cartesian tree and never changed, and ***node.inherited***, which is initialized to *null*.

2. For each level i of the tree from 1 to d: In parallel, for all nodes at level i: pass *node.inherited* to its left child and *node.index* to its right child. The child stores the passed value in its *inherited* variable.

3. For all nodes in parallel: if *node.inherited* $\neq null$, then *node.inherited* denotes the index of the node's left smaller neighbor. Otherwise it does not have a left smaller neighbor.

By using parallel tree contraction [393], a linear-work and polylogarithmic-depth algorithm for computing the ANSVs can be obtained, as described in the following theorem.

Theorem 30. *There is an linear-work algorithm for computing the ANSVs of a sequence using $O(\log^2 n)$ depth.*

Proof. The algorithm first computes the binary Cartesian tree of the input sequence. Then it performs tree contraction on the resulting Cartesian tree. The following describes tree contraction operations for finding the smaller left neighbors; the procedure for finding the smaller right neighbors is symmetric. To find the left neighbors, compressing and decompressing the tree for several configurations is described, and the rest of the configurations have a symmetric argument. For compression, there are the left-left and right-left configurations. The *left-left* configuration consists of three nodes A, B, and C, with B being the left child of A and C being the left child of B. For this configuration, B is the compressed node, and during decompression B takes the *inherited* value of A and passes its *inherited* value to C. The *right-left* configuration consists of three nodes A, B, and C with B being the right child of A and C being the left child of B. For this configuration, B is again the compressed node, and during decompression takes the *index* value of A and passes its *inherited* value to C. The *right-right* and *left-right* configurations are defined similarly and have symmetric properties. A raked left leaf takes the *inherited* value of its parent when it is unraked, and a raked right leaf takes the *index* value of its parent when it is unraked. Note that values are only passed during decompression and unraking, and not during compression and raking. Tree contraction requires $O(n)$ work and $O(\log n)$ depth. Combined with the complexity bounds for generating the Cartesian tree of Theorem 28, this gives us a $O(n)$ work and $O(\log^2 n)$ depth algorithm for computing the all nearest smaller values. □

As tree contraction can be done on the EREW PRAM, this algorithm can be mapped to the EREW PRAM. Although the depth complexity is higher, this algorithm is much simpler than the linear-work algorithms of Berkman et al. [38].

11.5 Experiments

The goal of the experiments is to analyze the efficiency of our parallel Cartesian tree algorithm both on its own and also as part of code to generate suffix trees. This section describes three applications of the Cartesian tree for answering queries on strings. In addition, the experiments compare the Cartesian tree-based suffix tree algorithm to the ANSV-based algorithm and to the best available sequential code for suffix trees. In the discussion, the two variants of the main algorithm (Section 11.3) are referred to as Algorithm 1a and Algorithm 1b, and the ANSV-based algorithm is referred to as Algorithm 2. For the experiments, in addition to implementing Algorithm 1a and a variant of Algorithm 2, we implemented parallel code for computing suffix arrays and their corresponding lcp values, and parallel code for inserting the tree nodes into a hash table to allow for efficient search (these codes are now part of the Problem Based Benchmark Suite). All of the experiments were performed on a 40-core parallel machine (with two-way hyper-threading) using a variety of real-world and artificial strings.

Auxiliary Code. To generate the suffix array and LCP array, we implemented a parallel version of the skew algorithm [256, 257]. The implementation uses a parallel radix sort, requiring $O(n/\epsilon)$ work and $O((1/\epsilon)n^\epsilon)$ depth for some constant $0 < \epsilon < 1$. The LCP code is based on an $O(n \log n)$ work solution for the range-minima problem instead of the optimal $O(n)$. The $O(n \log n)$ work solution creates a table with $\log n$ levels, where the i'th level of the table stores the minimum value of every interval of length 2^i in the sequence (computed in parallel from the $i-1$'st level). We did implement a parallel version of the linear-work range-minima algorithm by [154], but found that it was slower. Due to better locality in the parallel radix sort than the sequential one, our code on a single thread is actually faster than a version of [256, 257] implemented in their paper and available online, even though that version does not compute the LCP array. Our code achieves a 9 to 27 fold speedup on a 40 core machine. Compared to the parallel implementation of suffix arrays by [281], our times are faster on 40 cores than the 64-core numbers reported by them (10.8 seconds vs. 37.8 seconds on 522 million characters), although their clock speed is slower than ours and it is a different system so it is hard to compare directly. Mori provides a parallel suffix array implementation using OpenMP [338], but we found it to be slower than their corresponding sequential implementation. Our parallel implementation significantly outperforms that of Mori.

Note that recent sequential suffix array codes are faster than ours running on one thread [384, 338, 339], but most of them do not compute the LCP array (though these could be computed sequentially in a post-processing step [262, 255], or in parallel as discussed in Chapter 12). For real-world texts, those programs are faster than our code due to many optimizations that these programs make. We expect that many of these optimizations can be parallelized and could significantly improve the performance of parallel suffix array construction, but this was not the purpose of our studies. One advantage of basing suffix tree code on suffix array code, however, is that improvements made to parallel suffix arrays will improve the performance of the suffix tree code as well.

We use the parallel hash table described in Chapter 5 to allow for fast search in the suffix tree. Furthermore, we optimized the code so that most entries near leaves of the tree are not inserted into the hash table and a linear search is used instead. In particular, since the Cartesian tree code stores the tree nodes as an in-order traversal of the suffixes of the suffix tree, a child and parent near the leaf are likely to be near each other in this array. In the code, if the child is within some constant c (16 in the experiments) in the array, then we do not store it in the hash table and instead use a linear search to find it.

For Algorithm 2, we use an optimized $O(n \log n)$ work and $O(\log n)$ depth ANSV algorithm, which was part of the code of [429], instead of the much more complicated work-optimal version of [38].

Experimental Setup. The experiments were performed on the 40-core Intel machine (with two-way hyper-threading) described in Section 2.7. The parallel programs are written using Cilk Plus and compiled using Intel's `icpc` compiler. The sequential programs are compiled using `g++`.

For comparison to sequential suffix tree code, the publicly-available code of [444] and Kurtz's code from the MUMmer project [129, 285] were used. Only the results of Kurtz are reported because they are superior to those of [444] for all of the input files used. Kurtz's code is based on McCreight's suffix tree construction algorithm [325]—it is inherently sequential and completely different from the algorithms developed in this chapter. Other researchers have experimented with building suffix trees in parallel [167, 445] and the running times of the algorithm in this chapter appear significantly faster than those reported in the corresponding papers, even after accounting for differences in machine specifications. Iliopoulos and Rytter describe how to transform a suffix array into a suffix tree in parallel [236] in $O(n \log n)$ work, but they do not have an implementation available. More recent parallel disk-based suffix tree implementations [320, 108] will be compared against later in this section.

The experiments use a variety of strings available online (`http://people.unipmn.it/manzini/lightweight/corpus/`), a Microsoft Word document (thesaurus.doc), XML code from Wikipedia samples (wikisamp*.xml), the human genome (`http://webhome.cs.uvic.ca/~thomo/HG18.fasta.tar.gz`) (HG18.fasta), and artificial inputs. The artificial inputs are all of size 10^8 and include an all identical string (100Midentical), random strings with an alphabet size of 10 (100Mrandom), and a string with an alphabet size of 2 where every 10^4'th position contains one character and all other positions contain the other character (100Msqrtn). The inputs also include two files of integers, one with random integers ranging from 1 to 10^4 (100MrandomInts10K), and one with random integers ranging from 1 to 2^{31} (100MrandomIntsImax), to show that the algorithms run efficiently on arbitrary integer alphabets. See Table 11.2 for all of the input sizes.

Cartesian Trees. First, the experiments compare the Cartesian tree algorithm from Algorithm 1 to the linear-work stack-based sequential algorithm of [161]. There is also a linear-work sequential algorithm based on ANSVs, but we verified that the stack-based algorithm outperforms the ANSV one so only times for the former are reported. Figure 11.3 shows the speedup of the parallel Cartesian tree algorithm with respect to the sequential stack-based algorithm on the interleaved SA and LCP arrays of various inputs. The parallel algorithm outperforms the sequential algorithm with 4 or more cores, and achieves about 35x speedup (about 12x speedup with respect to the sequential algorithm) on 40 cores. The performance is consistent across the different inputs.

Applications of Cartesian Trees. The Cartesian tree built on the interleaved SA and LCP

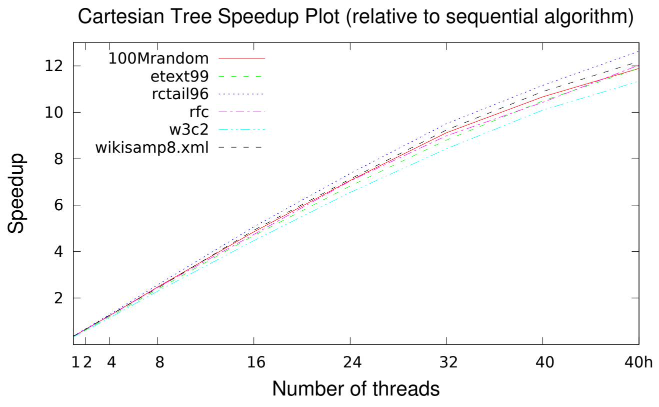


Figure 11.3: Speedup of the parallel Cartesian tree algorithm relative to the stack-based sequential algorithm on a 40 core machine. “40h” indicates 80 hyper-threads.

arrays of the string, which is essentially a suffix tree without the downward pointers, is able to answer certain string queries by performing bottom-up traversals. Abouelhoda et al. [1] show how to perform certain suffix tree queries using just SA and LCP. Their sequential stack-based method essentially computes the ANSVs on LCP to generate the tree structure, similar to the classic sequential stack-based ANSV algorithm. Berkman et al. showed how to parallelize the ANSV algorithm [38], which Section 11.4 generalizes. At the end of the day, however, the experiments in this section confirmed that building the Cartesian tree directly is more efficient than using the ANSV method, at least in parallel (see the “Cartesian tree” timings in Figure 11.5 versus the “ANSV”, “Compute parents”, plus “Create internal nodes” timings in Figure 11.6). As with the Abouelhoda et al. method, building the Cartesian tree is so fast that it can be re-computed per bottom-up computation, only requiring one to store the SA and LCP arrays between computations. For the experiments, the times both for constructing the Cartesian tree and for answering the queries (see Table 11.1) are reported. The times include parallel times on 40 cores with two-way hyper-threading (T_{40h}), times using a single thread (T_1), and the parallel speedup (SU). The code for the Abouelhoda et al. method is not available online so timings could not be obtained.

The first application is to use the Cartesian tree on the interleaved SA and LCP array of a string to compute for each internal node the number of leaf nodes in its subtree. This information can be used to answer queries related to repeated substrings, such as the number of repeated substrings of a given length that appear at least x times, or the number of repeated substrings of length at least y.

To compute the number of leaves contained in the subtree of each internal node, the Cartesian tree is processed in a bottom-up manner where initially all of the leaves are *active* and all active nodes pass the number of leaves in its subtree to its parent, which records

Text	Cartesian tree			Leaf counts			Longest substring ($k = 10$)			Leftmost suffix positions		
	T_{40h}	T_1	SU	T_{40h}	T_1	SU	T_{40h}	T_1	SU	T_{40h}	T_1	SU
100Midentical	0.23	6.61	28.7	1.7	3.27	1.92	1.69	3.61	2.14	1.73	3.37	1.95
etext99	0.26	9.2	35.4	0.12	4.77	39.8	0.14	5.19	37.1	0.12	4.87	40.6
rctail96	0.28	9.58	34.2	0.15	5.1	34	0.16	5.71	35.7	0.14	5.22	37.3
rfc	0.28	9.78	34.9	0.15	5.16	34.4	0.16	5.66	35.4	0.14	5.27	37.6
w3c2	0.26	9.14	35.2	0.13	4.37	33.6	0.14	4.83	34.4	0.13	4.47	34.4
wikisamp8.xml	0.25	8.52	34.1	0.12	4.48	37.3	0.13	4.99	38.4	0.12	4.58	38.2

Table 11.1: Times (seconds) for computing number of leaves per subtree on a 40 core machine with hyper-threading. T_{40h} is the time for our parallel algorithm on 40 cores (80 hyper-threads), T_1 is the single-thread time, and SU is the speedup computed as T_1/T_{40h}.

the sum of these values it receives. Once a node receives values from all of its children, it becomes active and passes its value to its parent. This process is work-efficient but requires depth proportional to the height of the tree. The times for this query are shown in the "Leaf counts" column in Table 11.1.

The second application is to use the Cartesian tree to compute the longest substring which appears at least k times in the text. To answer this query, the previous computation is modified to return the deepest node in the tree which has a subtree of at least size k. The times for this query for $k = 10$ are shown in the "Longest substring ($k = 10$)" column in Table 11.1.

The final application is to use the Cartesian tree on the interleaved SA and LCP array of a string to compute the leftmost starting position of any suffix in the subtree of each node. This is useful for computing the Lempel-Ziv decomposition [475] of a string (studied in Chapter 13), as described in [1]. The code for doing this is very similar to computing the number of leaves per subtree. Instead of summing the children's values, each parent takes the minimum value of its children and leaves start with a value equal to the starting position of their corresponding suffix in the original string. The times for this query are shown in the "Leftmost suffix positions" column in Table 11.1.

For most real-world strings, the height of the Cartesian tree of the interleaved SA and LCP arrays is not very large and these three applications get good speedup. As expected this process does not get much speedup for the all identical string, whose tree has linear height (the slight speedup comes from the pre-processing and post-processing steps). For the real-world strings, the cost of building the Cartesian tree is just about twice the cost of the query, which makes it reasonable to store just the SA and LCP arrays and build the Cartesian tree on-the-fly when performing a query. Other queries, such as finding maximal repeated pairs [1] and finding the longest common substring of two strings [201] can also be computed by a bottom-up traversal of the Cartesian tree.

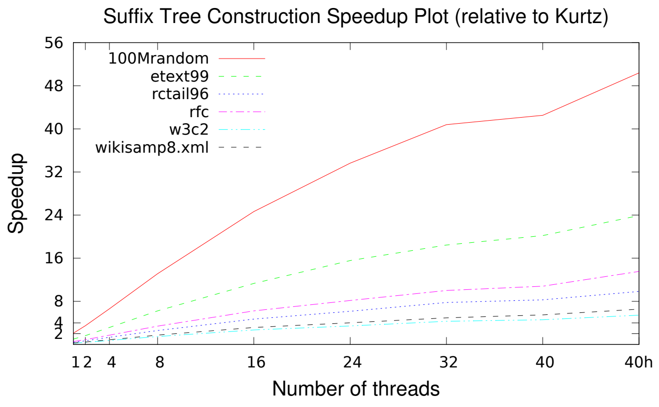


Figure 11.4: Speedup of Algorithm 1a relative to Kurtz's sequential algorithm on a 40 core machine. "40h" indicates 80 hyper-threads.

Suffix Trees. This section evaluates the performance of Algorithm 1a and 2 used with parallel suffix array and hash table code to generate suffix trees on strings. Table 11.2 presents the runtimes for generating the suffix tree based on Algorithm 1a, Algorithm 2, and Kurtz's code. For the implementations based on Algorithm 1a and Algorithm 2, both sequential (single thread) running times (T_1) and parallel running times on 40 cores with hyper-threading (T_{40h}) are reported. The parallel speedup (SU) is computed as T_1/T_{40h}. The experiments show that the speedup ranges from 14 to 24. Compared to Kurtz's code, the parallel code developed in this chapter running sequentially is between 2.1x faster and 5.3x slower. In parallel, however, the code is always faster than Kurtz's code and up to 50.4x faster. Comparatively, Kurtz's code performs best on strings with lots of regularity (e.g., the all identical string). This is because the incremental sequential algorithms based on McCreight's algorithm are particularly efficient on these strings. The runtime for the parallel code is affected much less by the type of input string. Kurtz's code only supports reading in text files with fewer than 537 million characters, so timings for wikisamp9.xml (1 billion characters) and HG18.fasta (3.08 GB) could not be obtained. Also since the code reads the input files as ASCII (alphabet size of 128), timings for integer files with larger alphabet sizes could not be obtained. The speedup of Algorithm 1a relative to Kurtz's sequential algorithm on various inputs is shown in Figure 11.4. The speedup varies widely based on the input file, with as much as 50.4x speedup for 100Mrandom and as little as 5.4x speedup for w3c2.

Figures 11.5 and 11.6 show the breakdown of the times for the implementations of Algorithm 1a and Algorithm 2 respectively when run on 40 cores with hyper-threading. In Figure 11.5, "Cartesian tree" refers to the time to construct the binary Cartesian tree and "Grouping internal nodes" refers to the time to convert to a multiway Cartesian tree. In Figure 11.6, "ANSV" is the time to compute the nearest smaller neighbors in the LCP

Text	Size (MB)	Kurtz	Alg 1a T_{40h}	Alg 1a T_1	Alg 1a SU	Alg 2 T_{40h}	Alg 2 T_1	Alg 2 SU
100Midentical	100	9.53	2.299	41.7	18.14	2.812	44.75	15.91
100Mrandom	100	168.9	3.352	80.6	24.05	3.971	84.2	21.2
100Msqrtn	100	14.52	3.518	55.2	15.69	4.023	57.97	14.41
100MrandomInts10K	100	–	5.24	81.1	15.48	5.774	84.8	14.69
100MrandomIntsImax	100	–	3.88	61.3	15.8	4.141	64.1	15.48
chr22.dna	34.6	24.5	1.469	32.62	22.21	1.728	34.45	19.94
etext99	105	119	4.977	120.3	24.17	5.75	125	21.74
howto	39.4	27.31	1.785	41.02	22.98	2.062	42.87	20.79
jdk13c	69.7	14.69	3.278	78.22	23.86	3.833	81.73	21.33
rctail96	115	55.13	5.61	133.2	23.74	6.34	138.9	21.91
rfc	116	71.77	5.619	133	23.67	6.476	139.2	21.49
sprot34.dat	110	75.11	5.299	126.2	23.82	6.048	131.6	21.76
thesaurus.doc	11.2	8.61	0.485	7.677	15.83	0.564	8.19	14.52
w3c2	104	28.44	5.24	121.2	23.13	5.913	126.1	21.33
wikisamp8.xml	100	31.48	4.808	117.2	24.37	5.612	124.8	22.24
wikisamp9.xml	1000	–	53	1280	24.15	61.88	1339	21.64
HG18.fasta	3083	–	168	3402	20.25	–†	–†	–†

Table 11.2: Comparison of running times (seconds) of Kurtz's sequential algorithm and our algorithms for suffix tree construction on different inputs on a 40 core machine with hyper-threading. T_{40h} is the time using 40 cores (80 hyper-threads) and T_1 is the time using a single thread. SU is the speedup computed as T_1/T_{40h}. †Times for Algorithm 2 on HG18.fasta are not reported since for this file, the algorithm uses more memory than the machine has available.

array, "Compute parents" is the time to select a smaller neighbor to be the parent, and "Create internal nodes" does an integer sort to create the internal nodes of the tree. In both figures, "Hash table insertion" is the time to create a hash table for downward traversal, and completes the suffix tree construction. Figure 11.7 shows the breakdown of the time for generating the suffix array and LCP array. For Algorithm 1a, more than 80% of the total time is spent in generating the suffix array, less than 10% in inserting into the hash table, and less than 5% on generating the Cartesian tree from the suffix array (i.e., the code shown in Figure 11.1). For Algorithm 2, note that the ANSV portion takes less than 2% of the total time even though it is an $O(n \log n)$ work algorithm. Improvements to the suffix array or hash table code will likely lead to an improvement in the overall code performance. Figure 11.8 shows the performance of Algorithm 1a in terms of characters per second on random character strings of varying sizes. Observe that the ratio remains nearly constant as the input size increases, indicating good scalability. While the implementation of Algorithm 1a is not truly parallel, it is incredibly straightforward and performs better than Algorithm 2.

Independent of this work, Mansour et al. have developed a disk-based parallel suffix tree

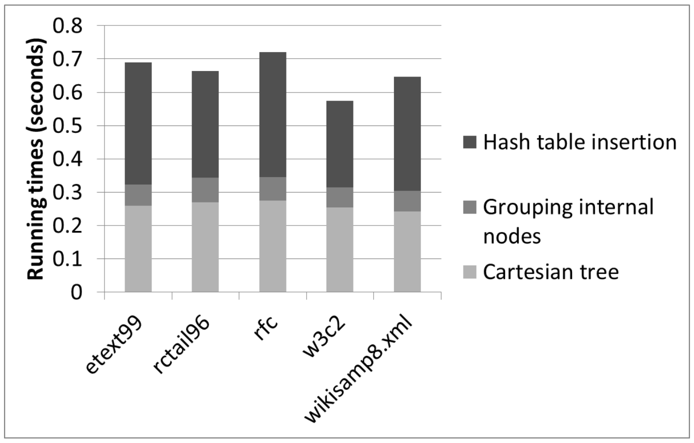


Figure 11.5: Breakdown of running times for converting a suffix array to suffix tree using Algorithm 1a on 40 cores with hyper-threading.

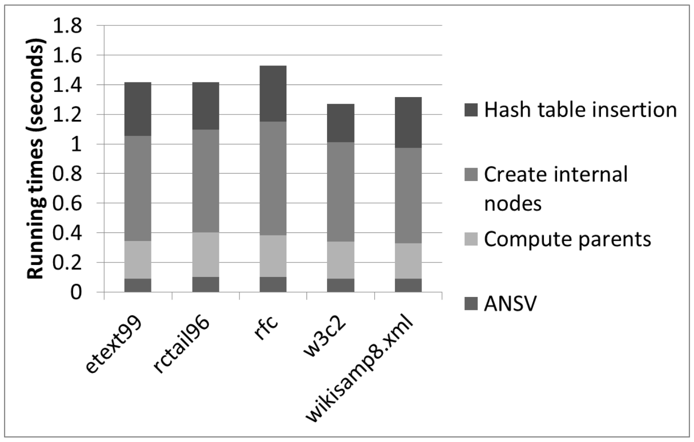


Figure 11.6: Breakdown of running times for the suffix tree portion of Algorithm 2 on 40 cores with hyper-threading.

algorithm [320] which works for input strings that do not fit in main memory. Algorithm 1a from this chapter is faster than theirs on a per-core basis—on the human genome (3.08 GB), our algorithm takes 168 seconds using 40 cores while the algorithm of Mansour et al. takes 19 minutes on 8 cores and 11.3 minutes on 16 cores. However, their algorithm is disk-based and requires less memory than ours. To account for machine differences, we ran their code on the human genome using all 80 hyper-threads on our 40-core machine, allowing each thread to use 2 GB of memory (for a total of 160 GB of memory, more memory than required for our algorithm on the human genome). The running time was approximately 400 seconds, more than a factor of 2 higher than the running time of our algorithm. A comparison of the running times of our algorithm with their algorithm on the

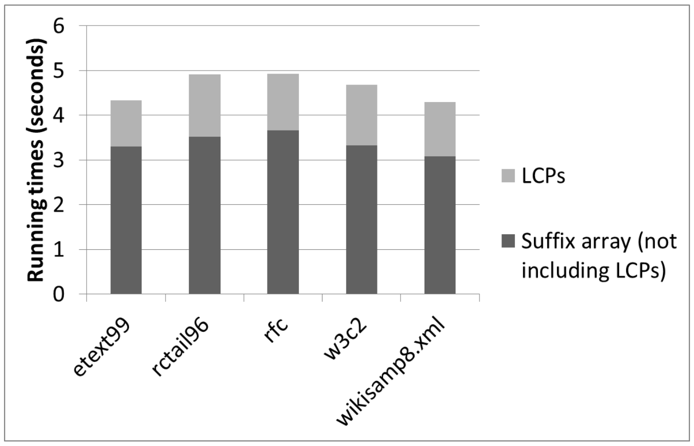


Figure 11.7: Breakdown of running times for the suffix array portion of Algorithm 1a and Algorithm 2 on 40 cores with hyper-threading.

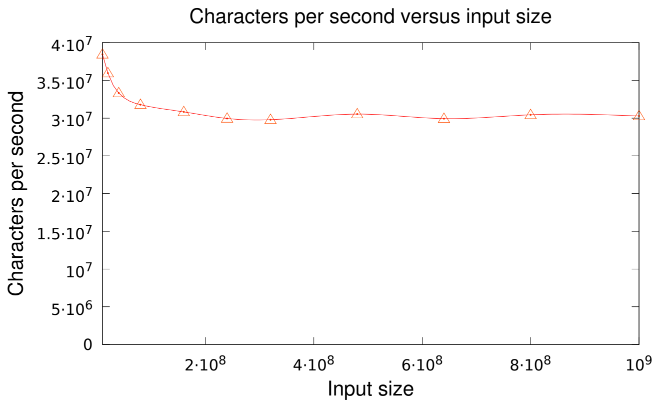


Figure 11.8: Performance (characters per second) of Algorithm 1a on random character strings of varying sizes on 40 cores with hyper-threading.

human genome is shown in Figure 11.9. It is worth noting that their algorithm requires super-linear work and depth in the worst case. Very recently, Comin and Farreras describe a parallel disk-based algorithm implemented using MPI [108]. For the human genome, they report a running time of 7 minutes using 172 processors, which is slower than our algorithm using 40 cores (see Figure 11.9). However, their algorithm again is disk-based, and their experiments were done on older hardware. Again, the algorithm takes super-linear work.

Searching the Suffix Tree. Experiments were performed to measure the time for existential queries (searching) on random strings in the suffix trees of several texts constructed using the code developed in this chapter as well as Kurtz's code. Times for searches using Manber and Myer's suffix array code [319] are also reported, as Abouelhoda et al. [1] show that this code (*mamy*) performs searches more quickly than Kurtz's code does. The suffix array

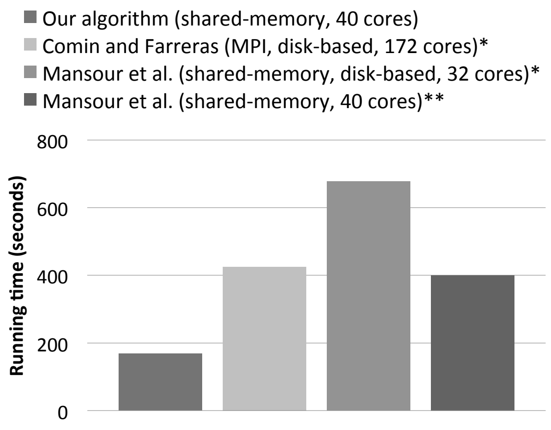


Figure 11.9: Parallel running times of suffix tree construction on the human genome. *Reported time from the literature [320, 108]. **Code from [320] run on our 40 core machine with a memory budget of 160 GB.

Text	Alg 1a T_{40h}	Alg1a T_1	Alg1a SU	Kurtz T_1	mamy T_1
100Mrandom	0.017	0.78	45.88	1.65	1.05
etext99	0.019	0.9	47.37	6.32	1.38
sprot34.dat	0.014	0.681	48.64	3.29	1.3

Table 11.3: Comparison of times (seconds) for searching (existential queries) 1,000,000 strings of lengths 1 to 50 on a 40 core machine with hyper-threading. T_{40h} is the time using 40 cores (80 hyper-threads) and T_1 is the time using a single thread. SU is the speedup computed as T_1/T_{40h}.

code uses the LCP array and answers queries in $O(m + \log n)$ time where m is the length of the pattern. For each text, the experiments search 500,000 random substrings of the text (these should all be found) and 500,000 random strings (most of these will not be found) with lengths uniformly distributed between 1 to 50. For all searches, the starting position in the text of the search string is reported if found.

Existential query times are reported in Table 11.3. Searches done in our code are on integers, while those done in Kurtz's code and Myer and Manber's code (*mamy*) are done on characters, giving a slight disadvantage to our code. Both sequential and parallel search times for our code are reported. The results show that sequentially, our code performs searches faster than Kurtz's code (2.1–7x) and *mamy* (1.3–1.9x). Abouelhoda et al. [1] report being 1.2–1.7x faster than *mamy* for searches on strings with small alphabets, but are up to 16x slower than *mamy* on larger alphabets. In contrast, the search performance of our code does not degrade with increasing alphabet size, since we use a hash table to store children of internal nodes.

The layout of our nodes in memory is in suffix array order, so listing occurrences can also be done in a cache-friendly manner by scanning the nearby nodes, similar to *mamy*.

Component	**Space (number of bytes)**
Computing SA + LCP	$32n$
SA + LCP data structure	$8n$
Node initialization	$24n$
Building Cartesian Tree	$16n$
Cartesian Tree data structure	$16n$
Finding roots	$16n$
Hash table insertion	$31n$
Suffix Tree data structure	$29n$

Table 11.4: Space requirements for the different components of Algorithm 1a for suffix tree construction.

Space Requirements. Since suffix trees are often constructed on large texts (e.g., the human genome), it is important to keep the space requirements minimal. As such, there has been related work on compactly representing suffix trees [172, 1, 402, 347, 180]. The suffix tree code developed in this chapter uses 3 integers per node (leaf and internal) and about $5n$ bytes for the hash table, which totals to about $29n$ bytes. This compares to about $12n$ bytes for Kurtz's code, which has been optimized for space [129, 285]. Table 11.4 shows the space requirements (in bytes) for the different portions and data structures of our implementation. Further optimization of the space requirements of our implementation is left to future work.

Chapter 12

Parallel Computation of Longest Common Prefixes

12.1 Introduction

Suffix arrays [319] along with the corresponding longest common prefix array (refer to Section 2.6.3 for their definitions) have applications in many fields, including bioinformatics, information retrieval, and data compression. Many applications of suffix arrays require the longest common prefix (LCP) array as well. For example, the lcp values are used for efficient pattern matching with a suffix array [319], and are used along with the suffix array to build a suffix tree [422] (as discussed in Chapter 11) or simulate suffix tree traversals [1]. The suffix array and its corresponding LCP array are often preferred over suffix trees for text indexing due to their lower space requirements [201]. With the rapid growth in data sizes, having fast parallel algorithms for suffix arrays and LCP arrays are particularly important. While there exists algorithms that compute both the suffix array and LCP array together, sometimes the suffix array is already available, and it is beneficial to have a fast algorithm for computing just the LCP array. Furthermore, separating the computation of SA and LCP allows one to use a fast SA algorithm that does not compute the lcp values, followed by a fast LCP algorithm. With such a separation, improvements in either suffix array algorithms or LCP algorithms improve the overall running time of the SA+LCP computation.

The suffix array and the first algorithm for constructing it were described by Manber and Myers [319]. Their sequential algorithm requires $O(n \log n)$ work, and also produces the LCP array. The first linear-work suffix array algorithms were described independently by Kärkkäinen and Sanders [256], Ko and Aluru [271], and Kim et al. [266]. Among these, the skew algorithm [256] (also named DC3 in [257]) of Kärkkäinen and Sanders can also compute the LCP array. Fischer [153] later describes a sequential linear-work algorithm which computes both the SA and LCP, and is based on a modification of the sequential linear-work suffix array algorithm of Nong et al. [357]. In addition, many superlinear-work suffix array algorithms exist (see, e.g., [384]), and some are faster in practice than the linear-work algorithms for certain inputs.

As for sequential standalone LCP algorithms (which compute the LCP array given the SA as input), a brute-force method is to directly compute the lcp value between every pair of adjacent suffixes in the SA, requiring quadratic work in the worst-case. The first linear-work LCP algorithm was described by Kasai et al. [263]. Kärkkäinen et al. [255] later describe a linear-work algorithm for computing the permuted longest common prefix (PLCP) array. The LCP array can easily be computed from the PLCP array, and Kärkkäinen et al. show that their approach is more efficient in practice than that of Kasai et al. They also discuss another technique in the same paper based on irreducible LCP values, which requires $O(n \log n)$ work. The details of these algorithms will be described in Section 12.2. Gog and Ohlebusch [179] present a more space-efficient sequential LCP algorithm that requires the Burrows-Wheeler Transform [81] as input and requires $O(n^2)$ work in the worst case. There have also been many papers describing how to reduce the working space requirements of LCP computation [34, 321, 385, 255, 433, 179, 30, 180] and adapting them to external memory [305, 42, 253].

As for parallel algorithms, besides simply parallelizing the brute-force method, there are two existing methods for computing the LCP array. The first method is to use the skew algorithm of Kärkkäinen and Sanders [256], which runs in linear work and $O(\log^2 n)$ depth with high probability. Note that the skew algorithm is not a standalone LCP algorithm as it computes both the SA and LCP array together. Deo and Keely [133] present a standalone parallel LCP algorithm for GPUs that is based on a parallelization of the sequential algorithm by Kasai et al. [263].

Note that by first constructing the suffix tree, the lcp values can be obtained by inspecting the depth of each internal node in the tree (refer to the construction of the suffix tree described in Chapter 11). However, this approach is less satisfactory since constructing the suffix tree is less efficient in practice than constructing the SA and LCP array together. In fact, the fastest shared-memory parallel suffix tree algorithm in practice requires first constructing the SA and LCP array, as described in Chapter 11.

With a fast parallel LCP algorithm, the performance of parallel applications that require the SA and LCP array can be improved. For example, the fastest shared-memory parallel algorithms for suffix tree construction (Chapter 11) and Lempel-Ziv factorization (Chapter 13) require computing the SA and LCP array, which is the dominant cost of the algorithms (at least 80% of the total running time).

Contributions. This chapter presents several parallel standalone LCP algorithms. The first two are based on a parallelization of the sequential algorithms of [263] and [255]

Algorithm	Work	Depth
klaap-LCP (seq.)	$O(n)$	$O(n)$
kmp-LCP (seq.)	$O(n)$	$O(n)$
naive-LCP	$O(nl_{\text{avg}})$	$O(l_{\text{max}})$
skew-SA+LCP	$O(n)$ w.h.p.	$O(\log^2 n)$ w.h.p.
skew-LCP	$O(n)$	$O(\log^2 n)$
par-iLCP	$O(n \log n)$	$O(\log n + l_{\text{max}})$
par-LCP	$O(n + Kl_{\text{max}})$	$O(n/K + l_{\text{max}})$
	$O(n + Kl_{\text{avg}})$ expected	$O(n/K + l_{\text{max}})$
par-PLCP	$O(n + Kl_{\text{max}})$	$O(n/K + l_{\text{max}})$
	$O(n + Kl_{\text{avg}})$ expected	$O(n/K + l_{\text{max}})$
dk-LCP	$O(n + Kl_{\text{max}})$	$O(n/K + \log n + l_{\text{max}})$
	$O(n + Kl_{\text{avg}})$ expected	$O(n/K + \log n + l_{\text{max}})$
dk-PLCP	$O(n + Kl_{\text{max}})$	$O(n/K + \log n + l_{\text{max}})$
	$O(n + Kl_{\text{avg}})$ expected	$O(n/K + \log n + l_{\text{max}})$

Table 12.1: Work and depth bounds for LCP algorithms. n = input size, l_{max} = maximum lcp value, l_{avg} = average lcp value, and K is an algorithm parameter, which trades off between work and depth. The new algorithms are shown in bold font.

(par-LCP and par-PLCP, respectively), and require $O(n + Kl_{\text{max}})$ work and $O(n/K + l_{\text{max}})$ depth for a parameter $K \leq n$, where l_{max} is the maximum LCP value of the suffixes of the string. The parameter K represents a trade-off between work and parallelism. This chapter discusses variants of these algorithms that improve the work to $O(n + Kl_{\text{avg}})$ in expectation. The third algorithm (skew-LCP) is a slight modification of the skew algorithm [256], and requires linear work and $O(\log^2 n)$ depth in the worst case. This chapter also applies Deo and Keely's parallelization idea (dk-LCP) to the sequential algorithm of Kärkkäinen et al. [255] (this variant is referred to as dk-PLCP). Finally, a straightforward parallelization of the irreducible LCP algorithm of Kärkkäinen et al. [255] (par-iLCP) is presented, requiring $O(n \log n)$ work and $O(\log n + l_{\text{max}})$ depth. Note that the only two parallel algorithms that require $O(n)$ work and polylogarithmic depth independent of the LCP values of the string (i.e., are work-efficient) are the original skew algorithm (skew-SA+LCP) and skew-LCP, the variant for standalone LCP computation developed in this chapter. For reference, a table of the work and depth bounds for LCP algorithms is provided in Table 12.1, with the new algorithms/variants shown in bold font.

This chapter presents the first comprehensive evaluation of shared-memory implementations of parallel LCP algorithms, comparing the new algorithms along with a CPU implementation of the parallel algorithm of Deo and Keely [133] and an implementation of the original parallel skew algorithm. The parallel implementations are also compared with the fastest sequential algorithms for computing the LCP array. Experiments on a 40-core shared-memory machine using a variety of real-world and artificial inputs show that par-PLCP usually performs the fastest among the parallel implementations, and outperforms

i	S$[i]$	SA$[i]$	LCP$[i]$	PLCP$[i]$	suf$_i$
0	b	6	0	0	$\$$
1	a	5	0	3	$a\$$
2	n	3	1	2	$ana\$$
3	a	1	3	1	$anana\$$
4	n	0	0	0	$banana\$$
5	a	4	0	0	$na\$$
6	$\$$	2	2	0	$nana\$$

Figure 12.1: Example: SA, LCP, and PLCP arrays for S $= banana\$$.

the CPU implementation of Deo and Keely's algorithm by a factor of 1.5 to 2.3 in parallel. Compared to the fastest sequential LCP algorithm, par-PLCP is 14.4–21.8 times faster on 40 cores on the real-world inputs. The experiments also show that while the linear-work and polylogarithmic depth skew-LCP algorithm is 6–11x slower than par-PLCP, it outperforms the only existing algorithm with the same theoretical guarantees (skew-SA+LCP) by 1.4–2x in parallel.

12.2 Preliminaries

The SA and LCP arrays are defined in Section 2.6.3. For a suffix array SA, the ***inverse array*** Rank stores the rank of each suffix in SA. In particular, Rank$[j] = i$ if and only if SA$[i] = j$. The ***permuted longest common prefix*** array [255] is an array PLCP of length n that stores the lcp's in the order that they appear in S instead of their order in SA. In other words, PLCP$[SA[i]]$ = LCP$[i]$. As an example, Figure 12.1 shows the SA, LCP, and PLCP arrays for the string S $= banana\$$.

A ***standalone LCP algorithm*** takes as input the string S, its suffix array SA, and its length n, and outputs the LCP array. Let us now review the existing sequential and parallel standalone LCP algorithms.

naive-LCP. The LCP array can be computed in a brute-force manner by comparing every pair of adjacent suffixes one character at a time from the beginning of the suffixes. This approach can easily be parallelized as the comparison of each suffix pair is independent of any other suffix pair. The work is proportional to the sum of all lcp values, which can be bounded by $O(nl_{\text{avg}})$, where l_{avg} is the average lcp value, but is quadratic in the worst case. The depth is proportional to the maximum lcp value, l_{max}. The pseudocode for this brute-force algorithm, which is referred to as ***naive-LCP***, is shown in Figure 12.2.

klaap-LCP. The first linear-work sequential LCP algorithm was described by Kasai et al. [263], which is referred to as ***klaap-LCP***. The pseudocode for the klaap-LCP algorithm is shown in Figure 12.3, and is adapted from [263]. The klaap-LCP algorithm uses the observation LCP[Rank$[i]$] $\geq$ LCP[Rank$[i-1]$] $- 1$ to reduce redundant computation. The algorithm first computes the Rank array (Lines 2–3). It then uses the Rank array to iterate

```
1: procedure NAIVE-LCP(S, SA, $n$)
2:     $\mathsf{LCP}[0] = 0$
3:     parfor $i = 1$ to $n - 1$ do
4:         $h = 0$
5:         $j = \mathsf{SA}[i]$
6:         $k = \mathsf{SA}[i - 1]$
7:         while $\mathsf{S}[j + h] == \mathsf{S}[k + h]$ do
8:             $h = h + 1$
9:         $\mathsf{LCP}[i] = h$
```

Figure 12.2: naive-LCP: naive parallel LCP algorithm.

```
1: procedure KLAAP-LCP(S, SA, $n$)
2:     for $i = 0$ to $n - 1$ do                              ▷ Compute Rank array
3:         $\mathsf{Rank}[\mathsf{SA}[i]] = i$
4:     $\mathsf{LCP}[0] = 0$
5:     $h = 0$
6:     for $i = 0$ to $n - 1$ do
7:         if $\mathsf{Rank}[i] \neq 0$ then
8:             $k = \mathsf{SA}[\mathsf{Rank}[i] - 1]$
9:             while $\mathsf{S}[i + h] == \mathsf{S}[k + h]$ do
10:                $h = h + 1$
11:            $\mathsf{LCP}[\mathsf{Rank}[i]] = h$
12:            if $h > 0$ then
13:                $h = h - 1$
```

Figure 12.3: klaap-LCP: sequential LCP algorithm of Kasai et al.

over the suffixes in the order that they appear in the original string, keeping a counter h of the lcp value of the current suffix. To compute the lcp value of the next suffix in original string order, character comparisons are performed between the suffix and its previous suffix in SA order, starting with the $(h - 1)$'st character of the suffixes. Kasai et al. show that this algorithm requires at most $2n$ character comparisons, giving an $O(n)$ work algorithm.

kmp-LCP. Kärkkäinen et al. [255] describe a modification of the klaap-LCP algorithm, which writes out the lcp values in a permuted order. This chapter refers to their algorithm as ***kmp-LCP***. The pseudocode for the algorithm is shown in Figure 12.4, and is adapted from [255]. In particular, it writes the lcp value of the i'th suffix in S in position i in the PLCP array (Line 15). Obtaining the LCP array is done in a post-processing phase (Lines 16–17), by applying the relation $\mathsf{LCP}[i] = \mathsf{PLCP}[\mathsf{SA}[i]]$. Another difference from klaap-LCP is that in the pre-processing phase (Lines 3–4) kmp-LCP computes the index of the preceding suffix in SA for each suffix (stored in the Φ array), whereas klaap-LCP does this in the main loop using the Rank array (Line 8 of Algorithm 12.3). This saves a random read to SA, since the read to $\mathsf{SA}[i - 1]$ on Line 4 of kmp-LCP is already in cache, whereas Line 8 of klaap-LCP involves a random read to SA.

```
procedure KMP-LCP(S, SA, $n$)
    $\Phi[\mathsf{SA}[0]] = -1$
    for $i = 1$ to $n - 1$ do                          ▷ Compute $\Phi$ array
        $\Phi[\mathsf{SA}[i]] = \mathsf{SA}[i - 1]$
    $h = 0$
    for $i = 0$ to $n - 1$ do
        if $\Phi[i] == -1$ then
            $h = 0$
        else
            $k = \Phi[i]$
            while $\mathsf{S}[i + h] == \mathsf{S}[k + h]$ do
                $h = h + 1$
            if $h > 0$ then
                $h = h - 1$
        $\mathsf{PLCP}[i] = h$
    for $i = 0$ to $n - 1$ do                          ▷ Convert PLCP to LCP
        $\mathsf{LCP}[i] = \mathsf{PLCP}[\mathsf{SA}[i]]$
```

Figure 12.4: kmp-LCP: sequential LCP algorithm of Kärkkäinen et al.

As in klaap-LCP, the number of character comparisons in kmp-LCP is at most $2n$, but kmp-LCP was shown to perform faster in practice (by about 50%) than klaap-LCP due to requiring fewer random reads and writes. The authors of [255] discuss space-saving variants which computes only n/q entries of PLCP but requires $O(q)$ work for a random access. They also discuss certain applications where the PLCP array may be used instead of the LCP array [401]. This chapter assumes that the entire LCP array must be computed.

dk-LCP. Deo and Keely describe a parallel version of klaap-LCP for GPUs [133]. The pseudocode for an implementation of their algorithm is shown in Figure 12.5, and is referred to as ***dk-LCP***. Lines 2–3 are the same as in klaap-LCP, except done in parallel. The algorithm finds all of the indices i_j such that $\mathsf{LCP}[\mathsf{Rank}[i_j]] = 0$, which can be done by comparing the first character of each suffix with the first character of its previous suffix in SA, and applying a parallel filter (Line 5).[1] In particular, all the indices i such that $\mathsf{S}[i] \neq \mathsf{S}[\mathsf{SA}[\mathsf{Rank}[i]] - 1]$ are marked and a filter is applied to keep just the marked indices. These indices form intervals $[i_j, \ldots, i_{j+1} - 1]$, and since the intervals could be large (especially for strings from a small alphabet), each interval that is larger than some threshold is split into sub-intervals, and in parallel the sequential klaap-LCP algorithm is applied to all sub-intervals (Lines 6–19). This chapter's implementation uses a threshold of $\lfloor n/K \rfloor$ for some input parameter $K \leq n$ (for simplicity, the pseudocode assumes K evenly divides n, but can be adapted for the general case). Since the first suffix of a sub-interval may not

[1]The number of these indices is at most $|\Sigma|$. Without loss of generality, the pseudocode assumes that all characters in Σ appear in the string.

```
1: procedure DK-LCP(S, SA, $n$)
2:     parfor $i = 0$ to $n-1$ do                                  ▷ Compute Rank array
3:         Rank[SA[$i$]] $= i$
4:     LCP[0] $= 0$, $i_0 = 0$
5:     Compute indices $i_1 < i_2 < \ldots < i_{|\Sigma|-1}$ such that for all
           $1 \le j < |\Sigma|$, S[$i$] $\neq$ S[SA[Rank[$i$]] $- 1$]          ▷ lcp is 0
6:     parfor $j = 0$ to $|\Sigma| - 1$ do                          ▷ Parallelize over intervals
7:         $B = \lceil \frac{(i_{j+1} - i_j)K}{n} \rceil$                   ▷ Number of sub-intervals
8:         parfor $b = 0$ to $B - 1$ do                             ▷ Parallelize over sub-intervals
9:             $h = 0$
10:            start $= i_j + \frac{bn}{K}$
11:            end $= \min\{i_j + \frac{(b+1)n}{K}, i_{j+1}\}$
12:            for $i$ = start to end $- 1$ do                      ▷ Sequential klaap-LCP
13:                if Rank[$i$] $\neq 0$ then
14:                    $k$ = SA[Rank[$i$] $- 1$]
15:                    while S[$i + h$] == S[$k + h$] do
16:                        $h = h + 1$
17:                    LCP[Rank[$i$]] $= h$
18:                    if $h > 0$ then
19:                        $h = h - 1$
```

Figure 12.5: dk-LCP: parallel LCP algorithm of Deo and Keely.

have an lcp value of 0, there is extra work done relative to klaap-LCP in computing its lcp value (unlike in klaap-LCP, it does not know the lcp value of its previous suffix). Hence the total work can no longer be bounded by $O(n)$. An analysis of the algorithm is provided in Section 12.3. Deo and Keely's original GPU algorithm also includes a load-balancing component, but this chapter uses a CPU implementation that leaves load-balancing to the run-time scheduler.

skew-SA+LCP. The ***skew algorithm*** [256] is a linear-work parallel suffix array construction algorithm, and can be used to also compute the LCP array during the suffix array construction. The skew algorithm works in 4 steps:

1. Recursively construct the suffix array SA_{12} and longest common prefix array LCP_{12} of the suffixes starting at positions i in S where $i \bmod 3 \neq 0$.

2. Use SA_{12} to construct the suffix array SA_0 of the positions i in S where $i \bmod 3 = 0$.

3. Merge SA_{12} and SA_0 together to form SA.

4. Use SA and LCP_{12} to compute the full LCP array.

To perform step (1) it assigns lexicographic integer labels $s'_i \in [1, \ldots, 2n/3]$ to the triples S$[i, i+1, i+2]$ for $i \bmod 3 \neq 0$ using a stable integer sort followed by a prefix sum. If the names are all unique then the array of labels is the suffix array SA_{12}, and LCP_{12} contains all 0's; otherwise it recurses on the string $\mathsf{S}' = s_1 s_2$ where s_1 is formed by concatenating all of the labels s'_i for $i \bmod 3 = 1$ in order of i and s_2 is formed by concatenating all of the labels s'_j for $j \bmod 3 = 2$ in order of j. The authors of [256] show that the stable integer sorting here can be done in linear work and $O(\log n)$ depth w.h.p. for an initial alphabet of constant size by combining techniques from [388, 206].

To perform step (2), the suffixes at positions i where $i \bmod 3 = 0$ can be sorted by sorting the pairs $(\mathsf{S}[i], \mathsf{suf}_{i+1})$ using an integer sort, as the suffixes suf_{i+1} are at mod 1 positions and hence already in sorted order in SA_{12} from step (1). The integer sort requires $O(n)$ work and $O(\log n)$ depth w.h.p.

The merge in step (3) can be performed by using pairs $(\mathsf{S}[i], \mathsf{suf}_{i+1})$ if comparing a mod 0 suffix with a mod 1 suffix, and triples $(\mathsf{S}[i], \mathsf{S}[i+1], \mathsf{suf}_{i+2})$ if comparing a mod 0 suffix with a mod 2 suffix. This ensures that the suffixes appearing in the pairs or triples already appear in sorted order in SA_{12}. Computing the relative order of two suffixes in SA_{12} can be done in constant work by pre-computing an inverse array mapping each suffix to its position in SA_{12}. The inverse array can be computed in linear work and $O(1)$ depth. The merge can be done using a parallel merging algorithm in $O(n)$ work and $O(\log n)$ depth [243].

Finally, to perform step (4) the algorithm uses the fact that an lcp value in LCP corresponds to 3 times the corresponding value in LCP_{12}, and the fact that the lcp value between the two suffixes at positions i and j of the LCP_{12} array is equal to $\min_{i \leq k < j} \mathsf{LCP}_{12}[k]$. For two suffixes $\mathsf{suf}_{\mathsf{SA}[i-1]}$ and $\mathsf{suf}_{\mathsf{SA}[i]}$, the algorithm first compares c characters $(0 \leq c \leq 2)$ from the beginning of the suffixes until both $(\mathsf{SA}[i-1]+c) \bmod 3 \neq 0$ and $(\mathsf{SA}[i]+c) \bmod 3 \neq 0$. If fewer than c characters match, then $\mathsf{LCP}[i] = c'$, where c' is the length of the prefix that matches. Otherwise, let l be equal to the lcp between $\mathsf{suf}_{\mathsf{SA}[i-1]+c}$ and $\mathsf{suf}_{\mathsf{SA}[i]+c}$. These suffixes are represented in LCP_{12} because they are at mod 1 and/or mod 2 positions, and the positions in LCP_{12} can be looked up using the inverse array from step (3). However, the suffixes may not be adjacent in LCP_{12}, and so a range minima query between the two positions in LCP_{12} is done if necessary to give the lcp value between the suffixes. Then $\mathsf{LCP}[i]$ is equal to $c+3l+l'$, where l' is the lcp value between $\mathsf{suf}_{\mathsf{SA}[i-1]+c+3l}$ and $\mathsf{suf}_{\mathsf{SA}[i]+c+3l}$. l' is at most 2 and is computed by comparing the characters of the suffixes one-by-one. To answer range minima queries in $O(1)$ work/depth, the algorithm builds a range minima query table over LCP_{12}, which requires $O(n)$ work and $O(\log n)$ depth [243].

The overall work of the algorithm is $O(n)$ since each level of recursion requires linear work and reduces the problem size to $2n/3$. The depth is $O(\log^2 n)$ w.h.p. as there are $O(\log n)$ levels of recursion, each requiring $O(\log n)$ depth w.h.p. This chapter later shows how to modify the skew algorithm to compute the LCP array given the suffix array as input.

irreducible-LCP. Kärkkäinen et al. [255] describe a technique for computing the PLCP array based on irreducible lcp values, which this chapter refers to as ***irreducible-LCP***.

```
1: procedure PAR-LCP(S, SA, $n$)
2:     parfor $i = 0$ to $n - 1$ do                          ▷ Compute Rank array
3:         Rank[SA[$i$]] $= i$
4:     LCP[0] $= 0$
5:     parfor $j = 0$ to $K - 1$ do                          ▷ Parallelize over intervals
6:         $h = 0$
7:         for $i = \frac{jn}{K}$ to $\frac{(j+1)n}{K} - 1$ do          ▷ Sequential klaap-LCP
8:             if Rank[$i$] $\neq 0$ then
9:                 $k =$ SA[Rank[$i$] $- 1$]
10:                while S[$i + h$] $==$ S[$k + h$] do
11:                    $h = h + 1$
12:                LCP[Rank[$i$]] $= h$
13:                if $h > 0$ then
14:                    $h = h - 1$
```

Figure 12.6: par-LCP: parallelization of klaap-LCP.

PLCP[i] is *reducible* if S[$i-1$] $=$ S[$\Phi[i]-1$] and *irreducible* otherwise, where Φ is computed as in kmp-LCP, i.e. Φ[SA[i]] $=$ SA[$i-1$]. For reducible values, it can be shown [321, 255] that PLCP[i] $=$ PLCP[$i-1$] -1. The algorithm works by computing the PLCP values corresponding to the irreducible lcp's using the brute-force method of comparing the suffixes from the beginning, and using the results to compute each remaining PLCP value in constant work. The authors of [255] show that the sum of all irreducible lcp values is at most $2n \log n$. Hence, the overall work is $O(n \log n)$ (note that this is not work-efficient). The authors also show that in practice the algorithm is slower than kmp-LCP. This chapter later presents a straightforward parallelization of this algorithm.

12.3 Algorithms and Analysis

This section presents several parallel algorithms for computing the longest common prefix array given a string and its corresponding suffix array. The work and depth bounds of the algorithms are also analyzed.

par-LCP and par-PLCP. This first approach developed in this chapter is similar to that of Deo and Keely [133], but instead of requiring a pre-processing step to find the intervals that are processed in parallel, the input is split into equal-sized intervals. This approach can be used to parallelize both klaap-LCP and kmp-LCP. The algorithms use a parameter $K \leq n$, which trades off between parallelism and work, and split the input into intervals of size at most $\lfloor n/K \rfloor$ (there are either K or $K+1$ intervals). This chapter refers to the parallelization of klaap-LCP using this approach as ***par-LCP*** (pseudocode shown in Figure 12.6) and the parallelization of kmp-LCP as ***par-PLCP*** (pseudocode shown in Figure 12.7). For simplicity, the pseudocode assumes K evenly divides n, but can be adapted for the general case. The intervals are processed in parallel, where each interval

```
1: procedure PAR-PLCP(S, SA, $n$)
2:     $\Phi$[SA[0]] $= -1$
3:     parfor $i = 1$ to $n-1$ do                                   ▷ Compute $\Phi$ array
4:         $\Phi$[SA[$i$]] $=$ SA[$i-1$]
5:     parfor $j = 0$ to $K-1$ do                                   ▷ Parallelize over intervals
6:         $h = 0$
7:         for $i = \frac{jn}{K}$ to $\frac{(j+1)n}{K} - 1$ do              ▷ Sequential kmp-LCP
8:             if $\Phi[i] == -1$ then
9:                 $h = 0$
10:            else
11:                $k = \Phi[i]$
12:                while S[$i+h$] $==$ S[$k+h$] do
13:                    $h = h+1$
14:                if $h > 0$ then
15:                    $h = h-1$
16:            PLCP[$i$] $= h$
17:    parfor $i = 0$ to $n-1$ do                                   ▷ Convert PLCP to LCP
18:        LCP[$i$] $=$ PLCP[SA[$i$]]
```

Figure 12.7: par-PLCP: parallelization of kmp-LCP.

runs klaap-LCP or kmp-LCP sequentially, with a counter h starting at 0. The parameter K could, for example, be set to $O(P)$ where P is the number of cores available to the computation, and this is what is used in the experiments in Section 12.4.

In par-LCP (Figure 12.6), the Rank array is computed in parallel on Lines 2–3. Then Line 5 is a parallel for-loop splitting the indices into equal-sized chunks, where each chunk is processed sequentially in Lines 6–14 using klaap-LCP. For par-PLCP (Figure 12.7), the loops computing Φ (Lines 3–4) and computing LCP (Lines 17–18) can be trivially parallelized. Again, on Line 5, the indices are split in a parallel for-loop, and each chunk is processed sequentially in Lines 6–16 using kmp-LCP.

In contrast to dk-LCP (and dk-PLCP, which is described next), par-LCP and par-PLCP do not have a pre-processing phase to find all the indices for which the lcp value is 0, therefore leading to splits that perform more extra work on average for the first element of each chunk. However, the experiments later show that the extra work is insignificant compared to the work required for pre-processing.

dk-PLCP. This chapter observes that the approach of Deo and Keely can also be used to parallelize kmp-LCP. This variant is referred to as ***dk-PLCP***. The code for this algorithm is very similar to that of dk-LCP and is not shown here.

Analysis. Let us now analyze the theoretical performance of the four parallel algorithms (par-LCP, par-PLCP, dk-LCP, and dk-PLCP) based on splitting the computation into intervals (and sub-intervals). In the analysis, K is assumed to evenly divide n, but the bounds still hold in the general case. The performance is based on the maximum or average lcp value of the suffixes of the string, which are denoted as l_{max} and l_{avg}, respectively.

Theorem 31. *For a parameter $K \leq n$, par-LCP and par-PLCP require $O(n + Kl_{max})$ work and $O(n/K + l_{max})$ depth.*

Proof. For each interval, the maximum value of the counter h is l_{max} and there are n/K decrements, so the number of character comparisons (equal to the number of times h is incremented) is at most $n/K + l_{\text{max}}$. This analysis is similar to that of [263]. Over all K intervals, the number of character comparisons is at most $n + Kl_{\text{max}}$. The work of the main loop (Lines 5–14 of par-LCP and Lines 5–16 of par-PLCP) is thus $O(n + Kl_{\text{max}})$.

An alternative argument for the work bound of the main loop is that except for the first element of each interval, the work for the rest of the elements is exactly the same as in the sequential algorithm and hence bounded by $O(n)$. The first element of an interval can do at most l_{max} comparisons, and over all K intervals, this contributes $O(Kl_{\text{max}})$ to the work. Hence the total work is bounded by $O(n + Kl_{\text{max}})$.

The intervals can be processed in parallel, but each interval is done sequentially doing at most $n/K + l_{\text{max}}$ comparisons, so the depth of the main loop is $O(n/K + l_{\text{max}})$. The parallel loops on Lines 2–3 of par-LCP, and Lines 3–4 and 17–18 of par-PLCP require $O(n)$ work and $O(1)$ depth. Therefore, the work of the algorithms is $O(n + Kl_{\text{max}})$ and depth is $O(n/K + l_{\text{max}})$. $\square$

Note that if $K = \omega(n/l_{\text{max}})$, then par-LCP and par-PLCP do more than $O(n)$ work in the worst case. However, in the experiments K is set to be the number of threads, which is less than n/l_{max} for most inputs. Also, for real-world strings the $O(Kl_{\text{max}})$ term is usually very loose as it is unlikely that the first elements of many intervals have an lcp value close to l_{max}.

By using randomization, the work bound can be improved to $O(n+Kl_{\text{avg}})$ in expectation, as discussed in Lemma 27 below. This improvement is significant when $l_{\text{avg}} \ll l_{\text{max}}$.

Lemma 27. *Modified versions of par-LCP and par-PLCP require $O(n + Kl_{avg})$ expected work and $O(n/K + l_{max})$ depth.*

Proof. Instead of fixing the interval start indices at jn/K for $0 \leq j < K$, the algorithm picks an integer uniformly at random between 0 and $n/K - 1$ and shift all start indices to the right by this amount. A start index at $i = 0$ is added back (if it was shifted) to guarantee that all elements are processed.

Consider the extra work performed for the first elements of the intervals, except for at $i = 0$. Summing over all possible random shifts, each first element where $i > 0$ will be a first element of an interval exactly once, and the total extra work for these elements can be upper bounded by nl_{avg} (the sum over all lcp values). Each random shift is picked with $1/(n/K) = K/n$ probability, so the expected work for these elements for a single execution is at most $(K/n)nl_{\text{avg}} = Kl_{\text{avg}}$. The extra work for the first element at $i = 0$ can be bounded by l_{max}. The remainder of the work done in the main loop is the same as in the sequential algorithm, and so contributes $O(n)$ to the total work. Therefore, the total expected work is $O(n + Kl_{\text{avg}})$.

Again, the depth is bounded by the maximum size of an interval plus l_{max}, giving a bound of $O(n/K + l_{\text{max}})$. □

An analysis of dk-LCP and dk-PLCP, which is similar to that of par-LCP and par-PLCP, is provided in the following lemma.

Lemma 28. *dk-LCP and dk-PLCP require $O(n + Kl_{max})$ work and $O(n/K + \log n + l_{max})$ depth.*

Proof. For dk-LCP, computing the indices where the lcp value is 0 (Line 5) is done with a parallel filter, which requires $O(n)$ work and $O(\log n)$ depth. Lines 2–3 can be done in $O(n)$ work and $O(1)$ depth. Each interval larger than size n/K is divided into sub-intervals of size n/K (except for the last sub-interval which may contain fewer than n/K elements). Similar to the analysis of par-LCP and par-PLCP, the number of character comparisons for each sub-interval is $O(n/K + l_{\text{max}})$. The intervals that were not sub-divided do no more work than the sequential algorithm as the first lcp value is 0, and hence contribute $O(n)$ work. The maximum number of sub-intervals is $O(K)$ so this gives an overall work of $O(n + Kl_{\text{max}})$. The overall depth including the filter is $O(n/K + \log n + l_{\text{max}})$ as the maximum interval and sub-interval size is n/K. The analysis for dk-PLCP is similar. □

Analogous to Lemma 27, for dk-LCP and dk-PLCP, the sub-intervals in each interval can be shifted by a random amount to obtain the following lemma. The proof is omitted as it is similar to the proof of Lemma 27.

Lemma 29. *Modified versions of dk-LCP and dk-PLCP require $O(n + Kl_{avg})$ expected work and $O(n/K + \log n + l_{max})$ depth.*

Random Strings. Let us now analyze the behavior of the algorithms on random strings. For random strings from the alphabet Σ, where each character of the string is chosen uniformly at random from Σ, and $|\Sigma| \geq 2$, the expected length of the longest repeated substring of a random string has been shown to be $O(\log_{|\Sigma|} n)$ [258, 319]. This is also an upper bound (in expectation) of the maximum lcp value, since the longest common prefix of any two suffixes is a repeated substring in the string.

Lemma 30. *For a random string from an alphabet of size $|\Sigma| \geq 2$, par-LCP, par-PLCP, dk-LCP, and dk-PLCP require $O(n)$ work and $O(\log n)$ depth in expectation for $K = O(n/\log n)$.*

Proof. The expected maximum lcp value of a suffix is $O(\log_{|\Sigma|} n)$ which is $O(\log n)$ for $|\Sigma| \geq 2$. Apply Theorem 31 and Lemma 28 with $l_{\text{max}} = O(\log n)$ and $K = O(n/\log n)$. □

If $|\Sigma|$ is known beforehand then K can be set to $K = O(n/\log_{|\Sigma|} n)$, giving $O(n)$ work and $O(\log_{|\Sigma|} n)$ depth in expectation for par-LCP and par-PLCP.

skew-LCP—Standalone LCP Computation with the Skew Algorithm. A slight modification of the skew algorithm [256] that can be used as a standalone LCP algorithm (referred to as ***skew-LCP***) given the suffix array SA as input is described below. Refer to the steps of the skew algorithm as described in Section 12.2.

For step (1), construct SA_{12} by marking the indices i such that $\mathsf{SA}[i] \bmod 3 \neq 0$, and apply a parallel filter keeping just the elements at these indices. Computing the new lexicographic names is still done by comparing triples and using a parallel prefix sum to compute the new name of each triple. However, since the suffixes in SA_{12} are already sorted (SA is sorted), the algorithm assigns new lexicographic names in the range $[1, \ldots, 2n/3]$ based on the suffix's index in SA_{12}, instead of using an integer sort. Creating the string S' to recurse on is done as before—by moving all of the mod 1 suffixes to the beginning and mod 2 suffixes to the end of the string using a parallel for-loop. Steps (2) and (3) are no longer required since the algorithm does not need to generate SA. Step (4) to generate the LCP array from LCP_{12} remains the same as before.

Theorem 32. *skew-LCP requires $O(n)$ work and $O(\log^2 n)$ depth.*

Proof. For each level of recursion, the prefix sum and filter take linear work and $O(\log n)$ depth, and to answer range minima queries in $O(1)$ work and depth in step (4), a range minima query look-up table can be built in linear work and $O(\log n)$ depth [243]. As each recursive call reduces the problem to two-thirds of the original size, the work recurrence is $W(n) = W(2n/3) + O(n)$ and depth recurrence is $D(n) = D(2n/3) + O(\log n)$. Solving the recurrences gives the theorem. □

Note that the bounds of the original skew algorithm [256] are $O(n)$ work and $O(\log^2 n)$ depth w.h.p. for a constant alphabet. The bounds required use of integer sorting algorithms [388, 206] which limited the alphabet size. Since skew-LCP does not involve integer sorting, the bounds hold for general alphabets.

Just like the original skew algorithm, skew-LCP can be adapted to other models of computation using the ideas in [256]. In the Bulk Synchronous Parallel (BSP) model [451],

```
1: procedure PAR-ILCP(S, SA, $n$)
2:     $\Phi[\mathsf{SA}[0]] = -1$
3:     parfor $i = 1$ to $n - 1$ do                              ▷ Compute $\Phi$ array
4:         $\Phi[\mathsf{SA}[i]] = \mathsf{SA}[i - 1]$
5:     Compute all indices $i_1 < i_2 < \ldots < i_{m-1}$, such that
           $\mathsf{S}[i_j - 1] \neq \mathsf{S}[\Phi[i_j] - 1]$
6:     $i_0 = 0,\ i_m = n$
7:     parfor $j = 0$ to $m - 1$ do
8:         $h = 0$
9:         if $\Phi[i_j] \neq -1$ then
10:            $k = \Phi[i_j]$
11:            while $\mathsf{S}[i_j + h] == \mathsf{S}[k + h]$ do
12:                $h = h + 1$
13:        $\mathsf{PLCP}[i_j] = h$                                ▷ Irreducible lcp value
14:        parfor $l = i_j + 1$ to $i_{j+1} - 1$ do
15:            $\mathsf{PLCP}[l] = h - (l - i_j)$                   ▷ Reducible lcp values
16:    parfor $i = 0$ to $n - 1$ do                              ▷ Convert PLCP to LCP
17:        $\mathsf{LCP}[i] = \mathsf{PLCP}[\mathsf{SA}[i]]$
```

Figure 12.8: parallel-iLCP: parallel irreducible LCP algorithm.

skew-LCP requires $O(n/P + L\log^2 P + gn/P)$ time for a communication parameter g, synchronization cost L and number of cores P. This bound was true only for $P = O(n^{1-\epsilon})$ in the original skew algorithm due to the need for integer sorting. The bounds for skew-LCP in the external-memory and cache-oblivious models are the same as for the original skew algorithm—that is $O((n/B)\log_{M/B}(n/B))$ I/O's (external-memory) or cache misses (cache-oblivious) for a block size of B and a fast memory size of M.

par-iLCP—A Parallel Irreducible LCP algorithm. A straightforward parallelization of the irreducible-LCP algorithm described in Section 12.2 is discussed below, and is referred to as ***par-iLCP***. The pseudocode is shown in Figure 12.8. The parallel for-loops on Lines 3–4 and 16–17 are the same as in par-PLCP, since the algorithm first computes the PLCP array before converting it to the LCP array. On Line 5, all of the indices i_j, where $\mathsf{PLCP}[i_j]$ corresponds to an irreducible lcp value (an ***irreducible index***), are computed. This is done with a parallel filter with the predicate $\mathsf{S}[i_j - 1] \neq \mathsf{S}[\Phi[i_j] - 1]$, and requires $O(n)$ work and $O(\log n)$ depth.

Then for each irreducible index in parallel (Line 7), the algorithm first computes its PLCP value by comparing characters one-by-one (Lines 8–13). All of the indices after the irreducible index i_j and before the next irreducible index i_{j+1} correspond to reducible lcp values, so the algorithm then applies the formula $\mathsf{PLCP}[l] = \mathsf{PLCP}[i_j] - (l - i_j)$ from [321, 255] for all $i_j < l < i_{j+1}$ in parallel (Lines 14–15). The work of the main loop (Lines 7–15) is the same as in the sequential irreducible-LCP algorithm, namely

$O(n \log n)$. The work for the rest of the algorithm is $O(n)$. The depth is $O(l_{\text{max}} + \log n)$ as computing the lcp values for the irreducible indices requires $O(l_{\text{max}})$ depth and the parallel filter requires $O(\log n)$ depth. This gives the following theorem:

Theorem 33. *par-iLCP requires $O(n \log n)$ work and $O(\log n + l_{max})$ depth.*

12.4 Experiments

This section presents a detailed experimental evaluation of LCP algorithms in a shared-memory setting. The implementations and experimental setup are first discussed. Then, the performance of the standalone LCP implementations is evaluated. Finally, this section studies the performance of the implementations when used in conjunction with suffix array code. Additional experiments can be found in [416].

This chapter implements all of the algorithms listed in Table 12.1, and as a reminder, among the parallel LCP algorithms compared, par-LCP, par-PLCP, dk-PLCP, skew-LCP, and par-iLCP are new, and naive-LCP, dk-LCP, and skew-SA+LCP are existing algorithms. The main findings of the experimental study are summarized below:

1. On a 40-core machine with two-way hyper-threading, par-PLCP achieves the best parallel running times for most real-world inputs. It is 1.5–2.3x faster than this chapter's CPU implementation of the existing parallel LCP algorithm of Deo and Keely [133].

2. While skew-LCP has better worst-case theoretical guarantees than par-PLCP, it is 6–11x slower in parallel.

3. For real-world inputs, the performance of par-LCP, par-PLCP, dk-LCP and dk-PLCP is quite robust to the choice of the parameter K as long as K is not too extreme.

4. par-PLCP achieves good parallel speedup relative to kmp-LCP (up to 21.8x on 40 cores), the fastest sequential LCP algorithm.

5. All of the parallel algorithms achieve good self-relative speedup on most inputs.

6. In parallel, computing the SA and LCP arrays separately is 1.2–2.1x faster than computing them together with the skew algorithm.

7. Comparing the two parallel LCP algorithms which require $O(n)$ work and polylogarithmic depth, in parallel skew-LCP is 1.4–2x times faster than the original skew algorithm.

Implementations. We implement the parallel algorithms using Cilk Plus [294]. In the implementations of par-LCP and par-PLCP, K is set to be the number of available threads P (except for the experiment in Figure 12.10). Therefore the interval size is at most $\lfloor n/P \rfloor$ and number of intervals is either P or $P+1$. In practice, this gave the best balance between the extra work spent in computing the lcp values for the first element of each chunk and the amount of parallelism. The modified versions of par-LCP and par-PLCP using random shifting as discussed in Lemma 27 were also implemented, but there was no improvement over the original versions. This is because in the original versions, the work for computing the first element of each interval is usually much lower than l_{max} in practice.

The dk-LCP and dk-PLCP algorithms are implemented using the parallel filter code (which uses prefix sum) from the Problem Based Benchmark Suite. K is set to $2P$ (except for the experiment in Figure 12.10) and each interval with size greater than $\lfloor n/K \rfloor$ is split into sub-intervals of size $\lfloor n/K \rfloor$, except for the last sub-interval, which may be smaller. For single-threaded execution K is set to 1. This setting gave the best performance across all inputs. Note that the value of K here is higher than in par-LCP and par-PLCP. This is because the sizes of the intervals and sub-intervals in dk-LCP and dk-PLCP vary more, and creating more parallel tasks gives more flexibility to the run-time scheduler to achieve better load-balancing.

The implementation of par-iLCP uses the parallel filter code from the Problem Based Benchmark Suite, and the for-loop over the indices between two irreducible values is only parallelized when the size is greater than 1000 (to avoid the overhead of a parallel for-loop for smaller sizes). We also implement the naive parallel LCP algorithm (naive-LCP) from Figure 12.2.

We implement skew-LCP, the standalone LCP algorithm described in Section 12.3, by making the necessary modifications to the parallel implementation of the skew algorithm from the Problem Based Benchmark Suite. The implementations of the sequential klaap-LCP and kmp-LCP algorithms follow the pseudocode shown in Figure 12.3 and 12.4, respectively.

Gog and Ohlebusch [179] describe a sequential LCP algorithm that requires the Burrows-Wheeler transform array as input. Its implementation [178] uses compressed integers and are semi-external, leading to lower space usage but higher running time, and hence it is difficult to perform a direct comparison with the internal memory implementations in this chapter that do not use compressed integers.

Experimental Setup. The experiments are performed on the 40-core (with two-way hyper-threading) Intel machine described in Section 2.7. The implementations are compiled with the `g++` compiler. The times reported are based on a median of three trials.

The experiments use a variety of strings available online (`http://people.unipmn.it/manzini/lightweight/corpus/`), XML code from Wikipedia samples (***wik-***

isamp8 and ***wikisamp9***), human genomic data (http://webhome.cs.uvic.ca/~thomo/HG18.fasta.tar.gz) (***HG18.fasta***), protein data (http://pizzachili.dcc.uchile.cl/texts/protein/) (***proteins***), short reads of a DNA sequence (ftp://ftp.ncbi.nih.gov/pub/TraceDB/Personal_Genomics/Venter/) (***Venter0***), and artificial inputs. The artificial inputs are all of size 10^8 and include a random string with an alphabet size of 10 (***random***), an all identical string (***identical***), and a binary string where every 10^4'th position contains one character and all other positions contain the other character (***sqrtn***). One byte is used to represent each character for all inputs. Table 12.2 shows the file size, alphabet size ($|\Sigma|$), maximum lcp value (l_{max}), and average lcp value (l_{avg}) for each input.

Comparison of LCP algorithms. Table 12.2 shows the single-threaded times (T_1), 40-core with hyper-threading times (T_{40h}), and parallel speedups (T_1/T_{40h}) for all of the standalone LCP implementations. The fastest parallel time per input in Table 12.2 is shown in bold.

First, let us look at the performance of naive-LCP, the brute-force parallel algorithm. As expected, naive-LCP performs relatively well for inputs with small average lcp values, but significantly worse for inputs with large lcp values. For Venter0 and the random string, naive-LCP performs the best among all implementations due to the small lcp values. For several inputs, naive-LCP did not finish in a reasonable amount of time due to large lcp values, and hence the running time is not reported.

Figure 12.9 shows a bar chart comparing the running times for the parallel implementations using 80 hyper-threads on several inputs (for clarity of presentation, naive-LCP is not included as it is an order of magnitude slower on some inputs). From Table 12.2 and Figure 12.9, we see that *par-PLCP performs the fastest on most of the inputs*. We do see some exceptions, however. For the identical and sqrtn strings, par-LCP performs the best. This is because most contiguous suffixes in the suffix array also appear contiguously in the original string, and thus most memory accesses are cache-friendly. par-PLCP is designed to reduce random accesses at the cost of an extra phase to convert the PLCP array into the LCP array so this makes it slower than par-LCP for these two strings. For Venter0, which has small lcp values, par-PLCP performs almost as fast as naive-LCP. For the random string, which has even smaller lcp values, par-PLCP is about two times slower than naive-LCP. However, even though par-PLCP is not the fastest on these inputs, it still performs reasonably well. For all of the other inputs, par-PLCP is the fastest in parallel, so without prior knowledge about an input, par-PLCP will likely give the best performance.

We see that *skew-LCP is 6–11 times slower than par-PLCP in parallel*, even though it has a better worst-case complexity than par-PLCP. This is because the constants in its work bound are higher than for par-PLCP, and the extra work in computing the first element of each interval in par-PLCP (the $O(Kl_{\text{max}})$ term) is not high in practice. For par-iLCP, although it is not the fastest on any input, it is at most 3 times slower than the fastest

	chr22	etext99	HG18.fasta	howto	jdk13c	proteins	rctail96	rfc	sprot34	Venter0	w3c2	wikisamp8	wikisamp9	random	identical	sqrtn
size (MB)	34.6	105	3083	39.4	69.7	1184	115	116	110	427	104	100	1000	100	100	100
$\|\Sigma\|$	5	146	27	197	113	27	93	120	66	5	256	204	207	10	1	2
$l_{\max}$	$2 \cdot 10^5$	$3 \cdot 10^5$	$2 \cdot 10^7$	70720	37334	$6 \cdot 10^5$	26597	3445	7373	1139	10^6	1265	2032	15	10^8	10^8
l_{avg}	1979	1109	$4 \cdot 10^5$	268	679	1422	282	93	89.1	44	42300	53.2	68	7.31	$5 \cdot 10^7$	$5 \cdot 10^7$
klaap-LCP (seq.)	2.34	6.67	315	2.16	2.94	76.7	5.69	6.09	5.71	31.8	4.27	4.8	56.1	7.39	0.522	1.86
kmp-LCP (seq.)	1.67	5.53	233	1.67	2.56	58.9	4.78	5.14	4.83	26.3	3.74	4.08	43.8	6.07	0.726	1.57
naive-LCP (T_1)	51.9	93.5	–	10	43.6	1420	37.2	17.6	19.2	61.3	3250	12.9	191	5.37	–	–
naive-LCP (T_{40h})	2.11	2.82	–	0.326	1.32	45.5	0.965	0.403	0.373	**1.41**	119	0.256	4.01	**0.169**	–	–
naive-LCP (T_1/T_{40h})	24.6	33.2	–	30.7	33	31.2	38.5	43.7	51.5	43.5	27.3	50.4	47.6	31.8	–	–
skew-LCP (T_1)	15.2	58.4	2610	18.5	45.6	887	91.7	82.2	67.3	257	69	63.4	784	34.1	18.9	34.9
skew-LCP (T_{40h})	0.584	1.99	64	0.705	1.48	26.6	2.45	2.28	2.21	8.34	2.31	2.06	21.9	1.26	0.814	2.07
skew-LCP (T_1/T_{40h})	26	29.3	40.8	26.2	30.8	33.3	37.4	36.1	30.4	30.8	29.9	30.8	35.8	27.1	23.2	16.9
par-iLCP (T_1)	2.97	9.27	407	2.5	3.11	87.2	6.68	7.8	6.79	49.3	4.64	5.57	62.7	11.3	0.976	1.81
par-iLCP (T_{40h})	0.115	0.41	15.8	0.12	0.196	4.85	0.354	0.384	0.355	2.03	0.3	0.31	3.31	0.51	0.243	0.261
par-iLCP (T_1/T_{40h})	25.8	22.6	25.8	20.8	15.9	18	18.9	20.3	19.1	24.3	15.5	18	18.9	22.2	4	6.9
par-LCP (T_1)	2.29	6.5	311	2.12	2.93	76.2	5.61	5.95	5.63	30.2	4.22	4.76	55.9	7.31	0.568	1.91
par-LCP (T_{40h})	0.144	0.44	14.2	0.138	0.215	4.88	0.388	0.389	0.359	1.93	0.31	0.312	3.32	0.481	**0.119**	**0.179**
par-LCP (T_1/T_{40h})	15.9	14.8	21.9	15.4	13.6	15.6	14.5	15.3	15.7	15.6	13.6	15.3	16.8	15.2	4.8	10.7
par-PLCP (T_1)	1.68	5.51	233	1.66	2.56	58.8	4.78	5.16	4.84	25.1	3.85	4.07	44.1	6.98	0.767	1.58
par-PLCP (T_{40h})	**0.083**	**0.31**	**10.7**	**0.095**	**0.173**	**3.89**	**0.293**	**0.31**	**0.287**	1.42	**0.268**	**0.251**	**2.73**	0.343	0.143	0.186
par-PLCP (T_1/T_{40h})	20.2	17.8	21.8	17.5	14.8	15.1	16.3	16.6	16.9	17.7	14.4	16.2	16.2	20.3	5.4	8.5
dk-LCP (T_1)	3.25	9.32	384	2.98	4.01	106	7.76	8.37	7.83	55.1	5.83	6.63	76.4	10.5	1.06	3.1
dk-LCP (T_{40h})	0.195	0.606	20.1	0.185	0.265	6.51	0.523	0.535	0.495	2.7	0.389	0.406	4.56	0.663	0.212	0.301
dk-LCP (T_1/T_{40h})	16.7	15.4	19.1	16.1	15.1	16.3	14.8	15.6	15.8	20.4	15	16.3	16.8	15.8	5	10.3
dk-PLCP (T_1)	2.06	6.78	328	1.99	2.99	71.7	5.68	6.23	5.81	31	4.35	4.79	52.1	7.55	1.14	1.98
dk-PLCP (T_{40h})	0.107	0.386	13.3	0.117	0.196	4.47	0.34	0.358	0.335	1.77	0.302	0.306	3.16	0.446	0.227	0.236
dk-PLCP (T_1/T_{40h})	19.3	17.6	24.7	17	15.3	16	16.7	17.4	17.3	17.5	14.4	15.7	16.5	16.9	5	8.4

Table 12.2: Running times (seconds) of the LCP algorithms on different inputs on a 40-core machine with hyper-threading. The new algorithms are shown in bold font. T_1 is the time using a single thread, T_{40h} is the time using 40 cores (80 hyper-threads), and T_1/T_{40h} is the parallel speedup . The numbers in bold indicate the fastest parallel LCP running time for an input among all implementations. The entries labeled "–" indicate that the experiment did not finish running in a reasonable amount of time.

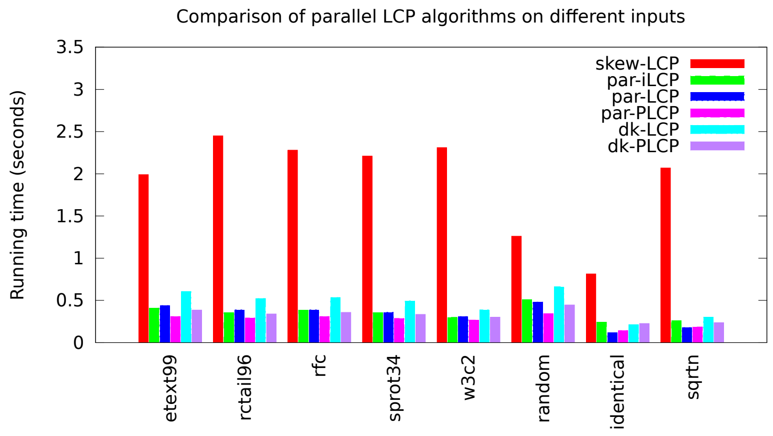


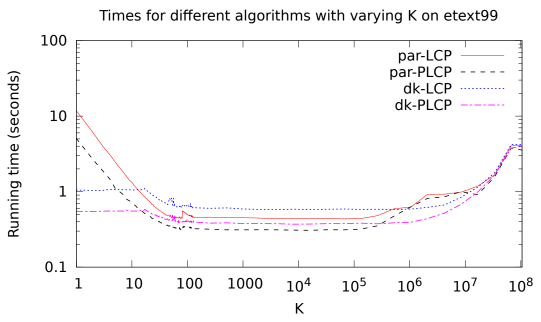
Figure 12.9: Comparison of running times of parallel LCP algorithms using 40 cores (80 hyper-threads).

implementation in parallel. Furthermore, it always outperforms skew-LCP. This is likely because for most inputs, the amount of work performed is less than its worst-case bound of $O(n \log n)$.

Note that par-PLCP is overall faster than par-LCP, and dk-PLCP is overall faster than dk-LCP. This is consistent with the study of sequential LCP implementations by Kärkkäinen et al. [255], showing that kmp-LCP is faster than klaap-LCP.

Observe that in parallel par-LCP outperforms dk-LCP by 23–78%, and par-PLCP outperforms dk-PLCP by 13–59%. dk-LCP and dk-PLCP guarantee that the elements with an lcp value of 0 are at the beginning of intervals with the goal of performing less wasted work compared to the corresponding sequential algorithm. However, it requires a pre-processing phase to identify the indices of elements for which the lcp value is 0 using a parallel filter. Therefore the overall time becomes slower than that of par-LCP and par-PLCP, which simply work on equal-sized chunks. *Compared to dk-LCP, the only existing parallel standalone LCP algorithm, the fastest LCP algorithm developed in this chapter, par-PLCP, is 1.5–2.3x faster on 40 cores with hyper-threading*.

Varying K. In the complexity bounds of par-LCP, par-PLCP, dk-LCP, and dk-PLCP, the parameter K represents a trade-off between work and parallelism. To see how it affects performance in practice, this section measures the parallel running times as K is varied. Figure 12.10 shows the running time of the four implementations using 40 cores (80 hyper-threads) as a function of K for etext99 and wikisamp8. For par-LCP and par-PLCP, the interval size is $\lfloor n/K \rfloor$, except for possibly the last interval. For dk-LCP and dk-PLCP, the number of intervals beginning with an lcp value of 0 is fixed (at most $|\Sigma|$), but the algorithms divide each interval larger than size $\lfloor n/K \rfloor$ into sub-intervals of size $\lfloor n/K \rfloor$, except for possibly the last sub-interval.



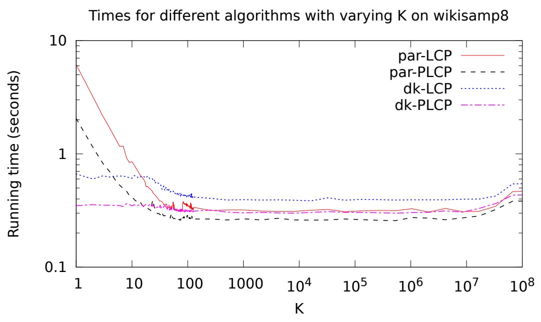


Figure 12.10: Parallel running times versus K for different algorithms on etext99 (**left**) and wikisamp8 (**right**). The y-axis is in log-scale.

For small values of K, dk-LCP and dk-PLCP are faster than par-LCP and par-PLCP, respectively, as they exhibit more parallelism due to having separate intervals starting at all indices corresponding to an lcp value of 0. For larger values of K there is enough parallelism and par-LCP and par-PLCP are faster due to not requiring a parallel filter. Figure 12.10 shows that *the performance of the algorithms is quite robust across different values of K as long as it is not too small or too large.* Similar behavior was observed for the other real-world inputs.

Comparing to sequential. As shown in Table 12.2, on a single thread, par-LCP and par-PLCP do just as well as klaap-LCP and kmp-LCP, respectively. This is because in the implementations, when there is only a single thread, only one interval is used ($K = 1$) and the parallel implementations do the same amount of work as their sequential counterparts. The speedup curves of par-PLCP with respect to kmp-LCP for several inputs are plotted in Figure 12.11. Compared to the sequential kmp-LCP code, par-PLCP achieves a speedup of 14.4–20.3x for the inputs in Figure 12.11 (and 21.8x for HG18.fasta). For the identical and sqrtn strings, the speedups are only 5.4x and 8.5x, respectively, since the parallel version does much more work than the sequential version due to the large lcp values; the speedup comes from the parallelism in generating the Φ array and converting the PLCP array to the LCP array.

Note that since K is varied based on the number of threads available, the amount of work done at each data point is not the same. In particular, with more threads there are more intervals, leading to more work compared to a single-threaded execution. Adjusting K is done to minimize the work, while taking advantage of all of the available parallelism. For inputs with high lcp values (e.g., HG18.fasta, identical, and sqrtn), this leads to lower speedup than if K had been fixed for different thread counts. For the other inputs, this effect was minimal for the modest values of K used in the experiments (between 1 and 80), as the extra work done (the $O(Kl_{max})$ term) is small.

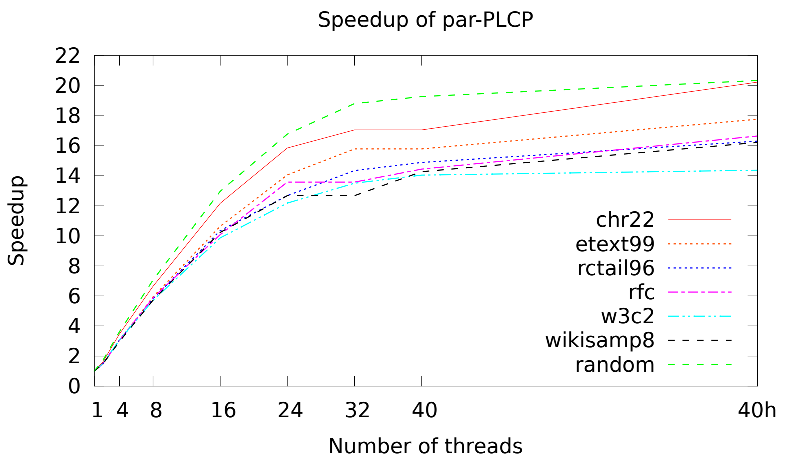
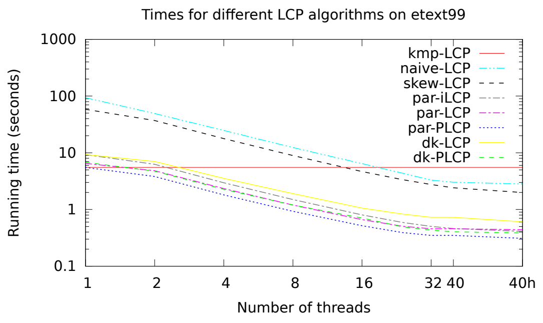


Figure 12.11: Speedup of par-PLCP with respect to kmp-LCP. "40h" indicates 80 hyper-threads.



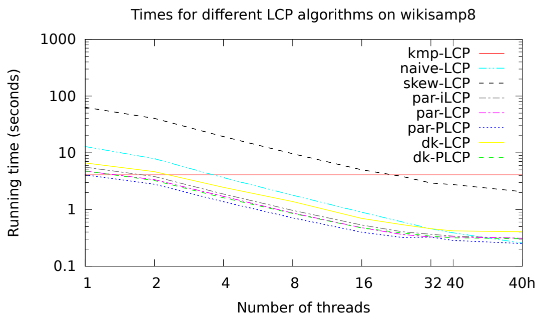


Figure 12.12: Running times versus number of threads of LCP algorithms on etext99 (**left**) and wikisamp8 (**right**) in log-log scale. "40h" indicates 80 hyper-threads.

Self-relative speedup. *All of the parallel implementations achieve good self-relative speedup on the real-world inputs*. For the implementations whose work is independent of the number of threads, on 80 hyper-threads, naive-LCP, skew-LCP, and par-iLCP achieve speedups of up to 51.5x, 40.8x, and 25.8x respectively (see Table 12.2). par-iLCP does not achieve good speedups on the identical and sqrtn strings as the available parallelism is low due to the large lcp values. As for the implementations whose work varies with thread count (par-LCP, par-PLCP, dk-LCP, and dk-PLCP), the self-relative speedups are lower, ranging from 13.6x to 24.7x on the real-world inputs. Again, these implementations do not get good speedup on the identical and sqrtn strings due to the large lcp values. Since the implementations perform many random memory accesses, the speedups are also likely limited by the memory bandwidth of the machine and the latency associated with memory contention.

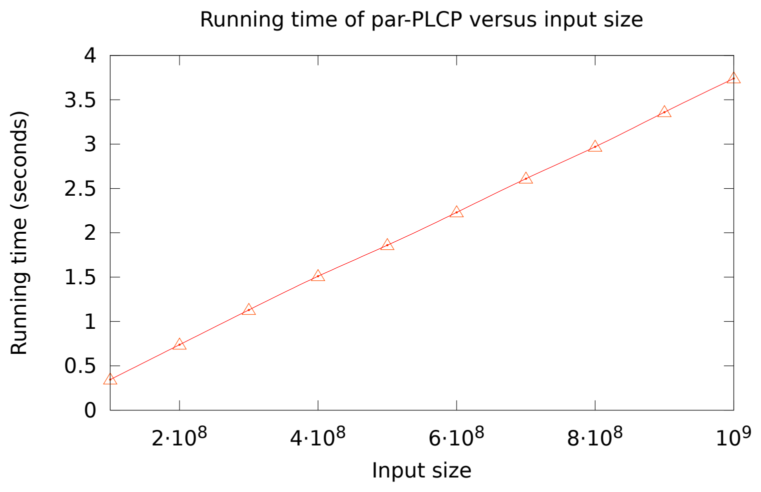


Figure 12.13: Running time versus input size of random text for par-PLCP using 40 cores (80 hyper-threads).

Varying thread count. Figure 12.12 shows the running time as a function of thread count for the different LCP implementations on etext99 and wikisamp8. Except for naive-LCP and skew-LCP, all of the parallel implementations outperform the best sequential implementation (kmp-LCP) with 4 or more threads.

Varying input size. To show scalability with increasing input size, par-PLCP was run on random strings of varying sizes ($|\Sigma| = 10$). Figure 12.13 shows the 40-core running time of par-PLCP as a function of input size. We observe that the running time scales linearly with the input size.

12.4.1 Performance of suffix array and LCP construction

In addition to studying the performance of the LCP algorithms on their own, this section also studies the overall performance of suffix array and LCP construction. The experiments show that in the parallel setting, separating suffix array and LCP construction leads to performance improvements in practice over constructing both arrays together. In this sub-section, the suffix array algorithms used are first discussed, and then their performance when combined with LCP algorithms is presented.

Performance of suffix array algorithms. Table 12.3 reports the times for suffix array computation using the fastest available parallel algorithms, skew-SA and range-SA, which are part of the Problem Based Benchmark Suite. ***skew-SA*** is the parallel implementation of the skew algorithm that does not compute the LCP array. ***range-SA*** is a parallel algorithm based on the prefix-doubling idea of sorting prefixes of suffixes with the prefix sizes increasing in powers of two. This idea has been used in several sequential suffix array algorithms [384] and also in parallel suffix tree algorithms [243]. range-SA requires $O(n \log n)$ work in the worst-case and does not generate the LCP array. The times for

	chr22	etext99	HG18.fasta	howto	jdk13c	proteins	rctail96	rfc	sprot34	Venter0	w3c2	wikisamp8	wikisamp9	random	identical	sqrtn
divsufsort-SA (seq.)	4.21	17.3	†	4.65	8.48	268.5	16.6	15	15.7	83.7	13.2	14.6	190.9	20.7	0.62	1.69
range-SA (T_1)	6.82	38.9	1130	12.1	35.3	548	53.2	43.4	40.1	99.8	75.7	37.3	421	16.2	135	119
range-SA (T_{40h})	**0.609**	**2.52**	**79.9**	**0.88**	2.15	**29.3**	**3.31**	**2.76**	**2.59**	**5.53**	4.58	**2.78**	**27.9**	**0.84**	8.58	6.39
skew-SA (T_1)	15.2	57.8	2020	19.5	34.6	736	59.4	60.9	57.7	214	55.1	50.4	555	34	14.6	19.8
skew-SA (T_{40h})	0.931	3.26	97.4	1.16	**1.98**	39.3	3.41	3.48	3.31	12.7	**3.22**	2.99	32.8	1.99	**1.07**	**1.63**
divsufsort-SA + kmp-LCP (seq.)	5.88	22.8	†	6.32	11	327.4	21.4	20.1	20.5	110	16.9	18.7	234.7	26.7	1.35	3.26
skew-SA+LCP (T_{40h})	1.15	4.01	122	1.44	2.71	50.8	4.56	4.48	4.28	16	4.34	3.98	43.1	2.48	1.45	2.84
parallel-SA + par-PLCP (T_{40h})	**0.692**	**2.83**	**90.6**	**0.975**	**2.153**	**33.19**	**3.6**	**3.07**	**2.877**	**6.95**	**3.49**	**3.03**	**30.63**	**1.18**	**1.21**	**1.82**

Table 12.3: **Top:** Running times (seconds) of SA algorithms on a single thread (T_1) and on 40 cores with hyper-threading (T_{40h}). The numbers in bold indicate the fastest parallel SA running time for an input. **Bottom:** Running times (seconds) of the various SA+LCP combinations. The numbers in bold indicate the fastest parallel SA+LCP running time for an input. **Note:** The entries labeled † indicate that the implementation failed to run. (Refer to Table 12.2 for input statistics.)

standalone suffix array construction in using the fastest available sequential algorithm (***divsufsort-SA***) implemented by Mori [338] are also reported in Table 12.3. Mori also provides a parallel implementation of divsufsort-SA using OpenMP [338], however we were unable to obtain any speedup compared to the corresponding sequential implementation.

The fastest parallel suffix array time per input is shown in bold in Table 12.3, and observe that in parallel there is no clear winner between range-SA and skew-SA. Compared to the sequential divsufsort-SA, the faster parallel implementation achieves a speedup of 4.1–15.1x on the real-world inputs. On the random string, the faster parallel implementation achieves a 24.6 fold speedup over divsufsort-SA, while for the identical and sqrtn strings, it performs about the same or worse, as the two parallel implementations are not well-suited for inputs with a lot of repeated structure. divsufsort-SA is faster than both range-SA and skew-SA on a single thread for all inputs except HG18.fasta, on which it failed to run, and the random string, on which it loses to range-SA.

Generating both the suffix array and LCP array. Table 12.3 reports the times for computing both the suffix array and the LCP array. For sequential times, the table reports the time for divsufsort-SA followed by kmp-LCP (***divsufsort-SA + kmp-LCP***). We also tried the implementation of Fischer's sequential algorithm [153, 339], which generates both the suffix array and LCP array, but found it to be slower than divsufsort-SA followed by kmp-LCP for all of the inputs. For parallel times, Table 12.3 reports the time for the parallel skew algorithm from the PBBS that generates both the suffix array and LCP array (***skew-SA+LCP***) and also the time for running the fastest parallel suffix array algorithm for the input followed by par-PLCP (***parallel-SA + par-PLCP***).

In parallel, the faster parallel SA algorithm followed by par-PLCP always outperforms skew-SA+LCP, with a speedup factor ranging from 1.2 to 2.1, confirming that *separating LCP construction from the suffix array construction leads to improved performance in the parallel setting*. Separating the construction of the two arrays allows one to use a faster parallel SA algorithm that does not compute the lcp values followed by a fast LCP algorithm. Furthermore, improvements in either parallel SA algorithms or parallel LCP algorithms lead to an overall performance improvement in the construction process.

The improvement in the parallel running time of SA and LCP array construction improves the overall running time of parallel applications that require SA and LCP, such as suffix tree construction (Chapter 11) [422] and Lempel-Ziv factorization (Chapter 13) [429]. The improvements are significant as the $\mathsf{SA} + \mathsf{LCP}$ computation is the dominant part of the computation in these applications (at least 80% of the total running time).

Compared to the sequential method of applying divsufsort-SA followed by kmp-LCP, applying the faster parallel suffix array algorithm followed by par-PLCP achieves a speedup of 4.8–15.8x on 40 cores for the real-world inputs. For the random string, the speedup is 22.6x, which is higher than for the real-world inputs, due to the good performance of the parallel suffix array algorithm. For the identical and sqrt strings, the speedup is less than 2x mostly due to the poor performance of the parallel suffix array algorithm relative to divsufsort-SA.

Linear work and polylogarithmic depth algorithms. skew-SA+LCP and skew-LCP are the two LCP algorithms with linear work and polylogarithmic depth without dependence on the lcp values of the suffixes of the input. From Tables 12.2 and 12.3, we observe that *in parallel, skew-LCP outperforms skew-SA+LCP by 1.8–2x for the real-world inputs and 1.4–2x for the artificial inputs.*

Chapter 13

Parallel Lempel-Ziv Factorization

13.1 Introduction

Compression techniques are widely studied as a means of reducing the space of storing data. The techniques studied fall into two categories—lossless and lossy. For *lossless* methods (e.g., Lempel-Ziv compression [475, 476], arithmetic coding [397], Huffman coding [234] and Burrows-Wheeler [81]), no information is lost when the data is compressed, while compression with *lossy* methods (e.g., JPEG and MPEG) can result in some information loss.

Lempel-Ziv-77 (LZ77) [475] and Lempel-Ziv-78 (LZ78) [476] form the basis for the family of Lempel-Ziv methods. They are dictionary coders, meaning that the encoder searches a dictionary for matches of substrings of the text, and returns a pointer to the substring's location in the dictionary. In LZ77, the encoder uses a sliding window (implicit dictionary) over the text to search for previous occurrences of substrings. Lempel-Ziv-Storer-Szymanski (LZSS) [438] is a variant of LZ77 that returns a pointer to the dictionary only if the matched substring is "long enough". LZ78 stores an explicit dictionary containing substrings previously seen, and in each iteration searches this dictionary to find the longest substring that exists in the dictionary, and then inserts a new entry into the dictionary. Lempel-Ziv-Welch [461] is a variant of LZ78 that uses a pre-initialized dictionary.

This chapter studies LZ77 rather than LZ78, since LZ77 admits efficient parallel solutions, whereas LZ78 has been shown to be P-complete (unlikely to have an efficient parallel solution) [125, 126].

LZ77 is a lossless dynamic compression method that has been popular due to its simplicity and computational efficiency. It is a component of the DEFLATE algorithm, which is used in software packages such as gzip and PKZIP. It has also been used in algorithms for detecting maximal repetitions in strings [274, 202]. The LZ77 algorithm consists of a compression stage, which computes the Lempel-Ziv factorization (henceforth ***LZ-factorization***) of the input string, and a decompression stage, which recovers the original string from the compressed string. The LZ-factorization can be computed sequentially [398] in linear work with a suffix tree [325], and decompression can be done sequentially in linear work with a scan. The first parallel algorithms for LZ-factorization were described independently by Noar [343] and Crochemore and Rytter [117]. For a string of length n, their algorithms require $O(\log n)$ depth and $O(n \log n)$ work, making them not work-efficient. Farach and Muthukrishnan [147] give the first linear-work algorithms for both LZ-factorization and decompression, each requiring $O(\log n)$ depth. These parallel algorithms all make use of parallel suffix trees. LZ77 decompression is much simpler and faster than LZ-factorization, so this chapter focuses on the latter.

There has been much research done in designing practical sequential algorithms for computing the LZ-factorization. Recently, researchers have proposed the use of suffix arrays instead of suffix trees to obtain faster and more space-efficient algorithms for LZ-factorization [116, 114, 89, 115, 358]. Since suffix arrays can be computed in linear work [256], these LZ-factorization algorithms are also able to run in linear work. The aforementioned sequential algorithms have been shown to perform well in practice.

The only parallel implementations of LZ-factorization described in the literature prior to the publication of this work [429] are those of Klein and Wiseman (using CPUs) [269] and Ozsoy and Swany (using GPUs) [362]. Both implementations involve splitting the input string among cores and having each core independently compute the factorization of its substring. Because in these implementations the cores do not necessarily have access to the entire input string, they do not always compute the same LZ-factorization as would be computed sequentially, and thus can produce larger compressed files. Furthermore, the corresponding papers [269, 362] do not provide any complexity bounds on work and depth. Subsequent to the publication of the results in this chapter, other GPU implementations of LZ-factorization have been presented in [95, 477], although again the algorithms do not return the same factorization as the sequential algorithm. Previous work on parallel algorithms for computing the same LZ-factorization as would be computed sequentially do not include any implementations or experiments [147, 343, 117]. The linear-work algorithm of Farach and Muthukrishnan [147] does not lead to a practical implementation, as it involves complicated parallel methods for tree contraction, least common ancestors, and Euler tours.

This chapter presents a simple linear-work parallel algorithm for LZ-factorization and practical shared-memory implementations of the algorithm. The algorithm computes the same factorization as would be computed sequentially. The algorithm is based on parallel suffix arrays [256], finding all nearest smaller values [38], and uses simple parallel routines such as prefix sums and leaffix computation [243]. Theoretically, the algorithm requires

i	$\mathsf{S}[i]$	$\mathsf{SA}[i]$	$\mathsf{LCP}[i]$	suf_i	$\mathsf{LPF}[\mathsf{SA}[i]]$	$\mathsf{prevOcc}[\mathsf{SA}[i]]$	$\mathsf{LZ}[i]$
0	a	14	0	\$	0	-1	0
1	b	8	0	$aaabab\$$	2	3	1
2	b	9	2	$aabab\$$	3	3	2
3	a	3	3	$aabbbaaabab\$$	1	0	3
4	a	12	1	$ab\$$	2	10	4
5	b	10	2	$abab\$$	2	0	7
6	b	0	2	$abbaabbbaaabab\$$	0	-1	10
7	b	4	3	$abbbaaabab\$$	3	0	12
8	a	13	0	$b\$$	1	7	14
9	a	7	1	$baaabab\$$	3	2	–
10	a	2	3	$baabbbaaabab\$$	1	1	–
11	b	11	2	$bab\$$	2	2	–
12	a	6	1	$bbaaabab\$$	4	1	–
13	b	1	4	$bbaabbbaaabab\$$	0	-1	–
14	\$	5	2	$bbbaaabab\$$	2	1	–

Figure 13.1: Example: SA, LCP, LPF, prevOcc and LZ for $\mathsf{S} = abbaabbbaaabab\$$.

$O(n)$ work and $O(\log^2 n)$ depth w.h.p. due to the use of suffix arrays [256], so does not achieve the $O(\log n)$ depth bound of Farach and Muthukrishnan [147]. However, it lends itself to a practical implementation. This chapter shows experimentally that on 40 cores with hyper-threading the parallel LZ-factorization algorithm achieve speedups between 11.1 and 23.1 compared to running the algorithm on a single thread. A sequential algorithm for LZ-factorization that is faster than previous algorithms[1] is also presented, and the parallel algorithm achieves a 7.9–16.6 fold speedup on 40 cores over this sequential algorithm.

13.2 Preliminaries

The LZ-factorization of a string $\mathsf{S}[0,\ldots,n-1]$ is $\mathsf{S} = \omega_0\omega_1\ldots\omega_{m-1}$, where $m \le n$ and for each $0 \le i < m$, ω_i (called the i'th ***factor*** of the string) is either a single character which does not appear in $\omega_0\ldots\omega_{i-1}$ or is the longest prefix of $\omega_i\ldots\omega_{m-1}$ that also appears starting at a position to the left of ω_i in S. For example, the string $abbaabbbaaabab\$$ has the factorization $\mathsf{S} = \omega_0\ldots\omega_8$ where $\omega_0 = a$, $\omega_1 = b$, $\omega_2 = b$, $\omega_3 = a$, $\omega_4 = abb$, $\omega_5 = baa$, $\omega_6 = ab$, $\omega_7 = ab$, and $\omega_8 = \$$ (example borrowed from [116]). To achieve compression, the ω values are not explicitly returned. Instead, LZ77 returns a sequence of pairs $(\mathsf{start}_i, \mathsf{prev}_i)$ where start_i indicates the starting position of ω_i in S and prev_i indicates the position of ω_i's left match in S if it exists (coded), and otherwise stores the character at position start_i (uncoded). For decompression, each pair i can reconstruct its factor by looking at prev_i and either directly copying prev_i if it is a character or copying $(\mathsf{start}_{i+1} - \mathsf{start}_i)$ characters starting at the position stored in prev_i (the value of start_0 is

[1]Faster sequential algorithms have been independently described in [264, 187, 254].

```
1: procedure LPFtoLZ(LPF,$n$)
2:     $\mathsf{LZ}[0] = 0$
3:     $i = 0$
4:     while $\mathsf{LZ}[i] < n$ do
5:         $\mathsf{LZ}[i+1] = \mathsf{LZ}[i] + \max(1,\ \mathsf{LPF}[\mathsf{LZ}[i]])$
6:         $i = i + 1$
7:     return LZ
```

Figure 13.2: LPFtoLZ: Algorithm for generating the Lempel-Ziv factorization from the longest previous factors.

defined to be 0 and start_m is defined to be n). The sequence of pairs returned for the string $abbaabbbaaabab\$$ is $[(0,a),(1,b),(2,1),(3,0),(4,0),(7,2),(10,0),(12,10),(14,\$)]$.

Throughout this chapter, the LZ-factorization of a string is denoted by an array LZ of size m where $\mathsf{LZ}[i]$ stores only the start_i value of the pair. To obtain the LZ77 representation, the prev_i value of the pair can easily be computed given the previous occurrence array, defined later in this section. This can easily be modified to return the LZSS representation [438].

This chapter will use the suffix array SA and longest common prefix array LCP, as defined in Section 2.6.3. The ***longest previous factor*** of an index i in S is equal to the maximum value of $\mathrm{lcp}(\mathsf{suf}_i, \mathsf{suf}_j)$, for all $j < i$. LPF is defined to be the longest previous factor array, where $\mathsf{LPF}[i]$ stores the longest previous factor of index i (0 if none). prevOcc is defined to be the ***previous occurrence array***, where $\mathsf{prevOcc}[i]$ stores the starting location of the longest previous factor of suf_i in S (-1 if none). Figure 13.1 shows the SA, LCP, LPF, prevOcc, and LZ arrays for the string $abbaabbbaaabab\$$. The chapter will also use algorithms for solving the all nearest smaller values (ANSV) problem as defined in Section 11.4. An algorithm for ANSV returns two arrays LN and RN where $\mathsf{LN}[i]$ ($\mathsf{RN}[i]$) contains the index of the nearest smaller element to the left (right) of element i (-1 if none).

13.3 Parallel Lempel-Ziv Factorization Algorithm

The parallel algorithm developed in this chapter is based on the sequential algorithm described by Crochemore, Ilie and Smyth (henceforth ***CIS***) [116], which first computes the LPF array. Computing the LZ-factorization can then be computed with a single pass over the LPF array [114]. The psuedocode for computing LZ from LPF is shown in Figure 13.2.

We now review Farach and Muthukrishnan's method of parallelizing **LPFtoLZ** given the LPF array as an input [147]. Their method creates a size $n+1$ array of pointers, next, where $\mathsf{next}[i] = \min(i + \max(\mathsf{LPF}[i], 1), n)$ for $i < n$ and $\mathsf{next}[n] = -1$. Following the indices (pointers) starting at $\mathsf{next}[0]$ until reaching a value of -1 is sufficient to determine the indices in LZ. Using a parallel leaffix algorithm [243] with the value at index 0 set to 1 and the remaining values set to 0, the result is an array of flags indicating which indices are in the LZ-factorization. This can be done in $O(n)$ work and $O(\log n)$ depth. A prefix sums [243] is then done on the array of flags to get the start values for the elements in the LZ-factorization, which can also be done in $O(n)$ work and $O(\log n)$ depth.

Now what remains is to show how to compute the LPF array. As done in CIS, the suffix array SA is first computed. While CIS computes the LCP array after computing SA, the algorithm in this chapter computes LCP while computing SA. Using the skew algorithm of Karkkainen and Sanders [256], both SA and LCP can be computed in parallel using $O(n)$ work and $O(\log^2 n)$ depth w.h.p. for constant-sized alphabets and $O(n/\epsilon)$ work and $O((1/\epsilon)n^\epsilon)$ depth for $0 < \epsilon < 1$ on integer alphabets.

After computing SA and LCP, the algorithm uses the following lemma due to Crochemore et al. [115], which states that any $\mathsf{LPF}[i]$ can be computed using an ANSV computation and range minima queries on SA and LCP. To deal with boundary cases, let us assume that $\mathsf{suf}_{\mathsf{SA}[-1]}$ evaluates to the empty string (and therefore has an lcp of 0 with any other string).

Lemma 31. *Let* $\mathsf{LN}[i]$ *and* $\mathsf{RN}[i]$ *be the left and right nearest smaller neighbors of element* i *in* SA. *Then* $\mathsf{LPF}[i] = \max(\mathrm{lcp}(\mathsf{suf}_{\mathsf{SA}[i]}, \mathsf{suf}_{\mathsf{SA}[\mathsf{LN}[i]]}), \mathrm{lcp}(\mathsf{suf}_{\mathsf{SA}[i]}, \mathsf{suf}_{\mathsf{SA}[\mathsf{RN}[i]]}))$.

Berkman, Schieber and Vishkin [38] show that ANSVs can be computed in $O(n)$ work and $O(\log \log n)$ depth. It can be shown that for any $0 \leq i < j < n$, $\mathrm{lcp}(\mathsf{suf}_{\mathsf{SA}[i]}, \mathsf{suf}_{\mathsf{SA}[j]}) = \min_{i<k\leq j} \mathsf{LCP}[k]$, so using range minima queries, one can compute the lcp values and hence the LPF values [256]. $\mathsf{prevOcc}[i]$ is set to $\mathsf{LN}[i]$ if $\mathsf{suf}_{\mathsf{SA}[\mathsf{LN}[i]]}$ has a longer lcp with $\mathsf{suf}_{\mathsf{SA}[i]}$, and $\mathsf{RN}[i]$ otherwise. Range minima queries can be performed in $O(1)$ work and depth, and require $O(n)$ work and $O(\log n)$ depth for pre-processing [243].

In the example shown in Figure 13.1, to determine $\mathsf{LPF}[7]$ and $\mathsf{prevOcc}[7]$ (corresponding to suffix $baaabab\$$), look at its left nearest smaller value in SA, which is 4, and its right nearest smaller value, which is 2, and then select the one corresponding to the suffix with a larger lcp with $baaabab\$$, which is of length 3. Therefore, $\mathsf{LPF}[7] = 3$ and $\mathsf{prevOcc}[7] = 2$.

Let us now look at two variants of the parallel LZ-factorization algorithm, differing only in how LPF is computed. The first variant (***PLZ1***) uses Lemma 31 directly. It builds a range minima query table on the LCP array for constant-time queries and then in parallel does range minima queries to compute each $\mathsf{LPF}[i]$. The n queries require a total of $O(n)$ work and $O(1)$ depth.

The second variant (***PLZ2***) uses as a component the sequential algorithm of Crochemore et al. [114], which takes the ANSVs as input and does a single pass over the string to compute the LPF array. Their crucial observation is that $\mathsf{LPF}[i] \geq \mathsf{LPF}[i-1] - 1$, and using this dependence they derive a linear-work algorithm for computing LPF.

Unlike PLZ1, PLZ2 does not build a range minima query table for constant-time queries but instead builds a segment tree [127] on the LCP array, an idea which was also investigated by Canovas and Navarro [86]. The segment tree is a binary tree whose leaves store the elements of LCP and internal nodes store the minimum value of its children. It requires $O(n)$ work and $O(\log n)$ depth to construct. Range minima queries can be answered by traversing the $O(\log n)$ levels of the tree, hence requiring $O(\log n)$ work and depth. PLZ2 then divides the input into $n/\log n$ blocks and computes the LPF values of each block. The longest previous factor of the first element is computed using a range minima query on the segment tree described above, and since $\mathsf{LPF}[i]$ only depends on $\mathsf{LPF}[i-1]$, the sequential algorithm of Crochemore et al. [114] can be used to compute the remaining longest previous factors of each block. Since one query is performed for each of the $n/\log n$ blocks in parallel, this leads to a cost of $O(n)$ work and $O(\log n)$ depth. Running the linear-work sequential algorithm per block in parallel takes a total of $O(n)$ work and $O(\log n)$ depth, since the size of each block is $O(\log n)$. The motivation for designing PLZ2 was that constructing the segment tree is simpler than constructing the table for constant-time queries, and since queries are only performed on a subset of the elements, experimentally the decreased construction time more than makes up for the increased query times.

The steps for LZ-factorization are summarized below. PLZ1 and PLZ2 differ only in the computation of step 3.

1. Compute the suffix array, SA, and longest common prefix array, LCP, for S.
2. Compute the left and right smaller neighbor arrays, LN and RN, on SA using an ANSV algorithm.
3. Compute the LPF and prevOcc arrays.
4. Return **LPFtoLZ(**LPF**,** n**)**

From the above discussion, it can be seen that all the steps require $O(n)$ work, and the depth is dominated by suffix array construction. This gives the following lemma:

Lemma 32. *Our parallel algorithm for computing the Lempel-Ziv factorization requires $O(n)$ work and $O(\log^2 n)$ depth with high probability for constant-sized alphabets and $O(n/\epsilon)$ work and $O((1/\epsilon)n^\epsilon)$ depth ($0 < \epsilon < 1$) for integer alphabets.*

The algorithm can be mapped onto the CRCW PRAM, as concurrent writes are required by the suffix array algorithm.

Excluding the suffix array and lcp computation, the algorithm takes only $O(\log n)$ depth for arbitrary alphabets, so improvements to the bounds for suffix array and lcp computation can improve the overall bounds as well. This algorithm is amenable to implementation, as described in the next section.

13.4 Implementations

Parallel LZ-factorization. This section describes the implementations of PLZ1 and PLZ2, as well as a simple variant of PLZ2 that avoids computing the LCP array. For suffix arrays, the linear-work and $O((1/\epsilon)n^\epsilon)$ depth (for some constant $0 < \epsilon < 1$) implementation from the Problem Based Benchmark Suite, which is an implementation of the skew algorithm of Karkkainen and Sanders [256], was used. My co-author and I implemented an optimized version of the $O(n \log n)$ work and $O(\log n)$ depth ANSV algorithm of Berkman et al. [38] instead of their much more complicated linear-work version. For **LPFtoLZ**, we implemented a random sampling-based leaffix algorithm [243] and used the parallel sequence routines from the Problem Based Benchmark Suite. For the range minima query table used for computing the LCP array inside the suffix array algorithm, we used an $O(n \log n)$ work and $O(\log n)$ depth construction algorithm for constant-time range minima queries. For PLZ1, we built a range minima table on the resulting LCP array using the same construction.

In PLZ2, the number of blocks was set to $n/8196$, as this gave the best results experimentally. We implemented the sequential algorithm of Crochemore et al. [114], which is used in each block. The variant of PLZ2, referred to as ***PLZ3***, does not compute the LCP array, but instead computes the lcp values of the first element of each block with its nearest smaller neighbors using naive string comparison, and uses this to compute its LPF value. The rest of each block is computed in the same way as in PLZ2.

Sequential LZ-factorization. This section describes a simple sequential algorithm for LZ-factorization that is more efficient in practice than existing sequential algorithms at the time this work was initially published [429]. This algorithm is used in the experiments in Section 13.5 as a sequential baseline. This sequential algorithm (***LZ-ANSV***) first computes the suffix array (without lcp values), and then computes the ANSVs on the suffix array sequentially using the stack-based algorithm of Gabow et al. [161]. It then loops through the suffixes in their original order, and for the positions appearing in the LZ-factorization, it computes the longest previous factor with the suffixes corresponding to the positions of their left and right smaller neighbors in SA using naive string comparison. By incrementing the index of the loop by the length of the longest previous factor after computing it for an element, it bypasses the LPF computation for the elements not appearing in the LZ-factorization. LZ-ANSV requires $O(n)$ work.

13.5 Experiments

This section experimentally compares the performance of the different implementations of the parallel LZ-factorization algorithm as well as sequential algorithms. We are not aware of any existing parallel implementations for computing the same LZ-factorization as would be computed sequentially. Previous parallel algorithms for doing so [147, 343, 117] use parallel suffix trees and are relatively complicated (no implementations are available). The experiments will show that the entire LZ-factorization algorithm developed in this chapter is faster than the parallel suffix tree algorithm from Chapter 11 (the fastest shared-memory parallel suffix tree algorithm) on most strings; hence it is unlikely that a parallel implementation of LZ-factorization that uses suffix trees will outperform the implementation developed in this chapter.

The parallel implementations are compared with our sequential LZ-ANSV code and the sequential algorithm of Ohlebusch and Gog [358], the fastest sequential algorithm at the time the results of this chapter were published [429]. The code was obtained from Ohlebusch and Gog, and is referred to as ***LZ-OG***. All of the implementations in the experiments compute pairs containing the starting position and previous occurrence for each factor in the LZ-factorization. For fair comparison, all of the implementations use the same suffix array code from the Problem Based Benchmark Suite.

Experimental Setup. The experiments were performed on the 40-core Intel machine (with two-way hyper-threading) described in Section 2.7. The parallel programs were written using Cilk Plus and compiled with Intel's `icpc` compiler. The sequential programs were compiled using `g++`.

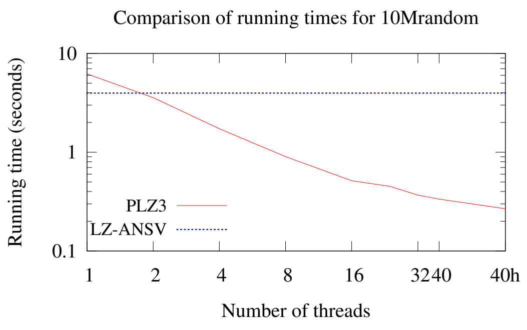
The experiments use a variety of real-world strings available online (`http://people.unipmn.it/manzini/lightweight/corpus/` and `http://pizzachili.dcc.uchile.cl/texts.html`), XML code from Wikipedia samples (wikisamp*.xml), and artificial inputs. The artificial inputs are of size 10^7 and include an all identical string (10Midentical), a random string with an alphabet size of 10 (10Mrandom), and a string with an alphabet size of 2 where every $\sqrt{10^7}$'th position contains the first character and all other positions contain the second character (10Msqrtn).

Experimental Results. The experiments first compare the three variants of the parallel LZ-factorization algorithm. The parallel running times on 40 cores with two-way hyper-threading (T_{40h}) for all three variants are shown in Table 13.1. Among the three variants, PLZ3 gives the best absolute performance across the board, both in parallel and sequentially. This is due to the fact that PLZ3 does not need to compute the LCP array (which takes about one-third of the time of the suffix array code), and this more than makes up for the extra time spent in performing naive string comparisons for the first element of each block. On average over all inputs, PLZ3 is 1.33x faster than PLZ1 in parallel and 1.36x faster than PLZ2 in parallel.

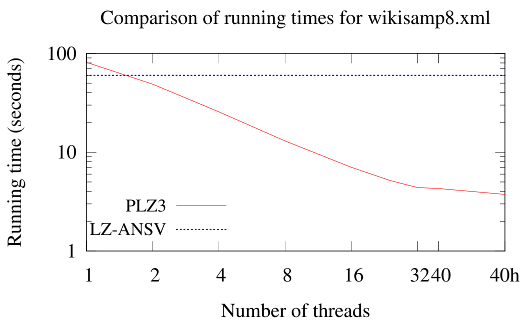
The experiments also compare PLZ3 to the two sequential algorithms LZ-ANSV and LZ-OG (see Table 13.1). T_1 is the time (in seconds) for running PLZ3 on a single thread and the speedup is computed as T_1/T_{40h}. The results show that the sequential algorithm described in Section 13.4 (LZ-ANSV) outperforms LZ-OG on all of the input strings. Note

Text	Size (MB)	LZ-ANSV	LZ-OG	PLZ3 T_1	PLZ3 T_{40h}	PLZ3 Speedup	PLZ1 T_{40h}	PLZ2 T_{40h}
10Midentical	10	1.68	1.74	2.35	0.212	11.08	0.313	0.318
10Mrandom	10	3.97	4.67	6.2	0.268	23.13	0.312	0.331
10Msqrtn	10	2.14	2.44	3.36	0.279	12.04	0.418	0.379
chr22.dna	34.6	19.4	22.0	28.9	1.3	22.23	1.71	1.75
etext99	105	69.9	75.2	99.0	4.47	22.15	5.23	5.71
howto.txt	39.4	24.0	25.5	33.4	1.53	21.83	2.02	2.1
jdk13c	69.7	40.4	41.4	54.1	2.5	21.64	3.67	3.67
pitches	55.8	31.8	34.3	43	1.92	22.4	2.61	2.61
proteins	210	147	172	203	9.25	21.95	11.1	11.9
rctail96	115	70.0	72.9	96.5	4.42	21.83	5.94	6.16
rfc	116	72.8	76.6	100	4.46	22.42	5.87	6.12
sources	211	140	163	186	8.74	21.28	11.1	11.5
sprot34.dat	110	69.0	72.2	93.7	4.23	22.15	5.64	5.79
w3c2	104	63.1	64.7	84.1	4.03	20.87	5.64	5.63
wikisamp8.xml	100	59.9	61.4	81.2	3.74	21.71	5.13	5.17
wikisamp9.xml	1000	653	670	894	40.8	21.92	50.9	53.6

Table 13.1: Comparison of running times (seconds) of parallel and sequential LZ-factorization algorithms on different inputs on a 40-core machine with two-way hyper-threading.



(a) 10Mrandom

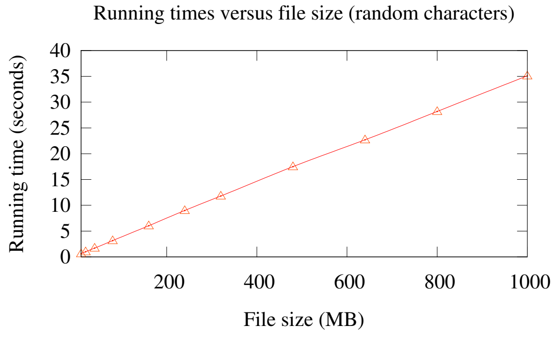


(b) wikisamp8.xml

Figure 13.3: Log-log plots of running times on a 40-core machine (with two-way hyper-threading). “40h” corresponds to 80 hyper-threads.

that, however, LZ-ANSV does not compute the entire LPF array whereas LZ-OG does, so for applications where the entire LPF array is required, LZ-ANSV will not suffice. On a single thread, PLZ3 is 1.3–1.6 times slower than LZ-ANSV. On 40 cores with hyper-threading, PLZ3 achieves 11.1–23.1 times speedup with respect to its single-thread running time, and achieves a 7.9–16.6 times speedup with respect to LZ-ANSV.

The running times of PLZ3 and LZ-ANSV as a function of the number of threads



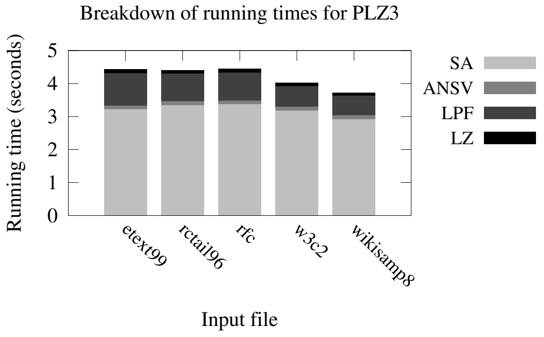


Figure 13.4: Left: Running time versus input size of PLZ3 on 40 cores. **Right:** Breakdown of running time of PLZ3 on 40 cores.

for the 10Mrandom and wikisamp8.xml inputs are shown in Figures 13.3(a) and 13.3(b), respectively. PLZ3 achieves good speedup and outperforms LZ-ANSV with just 2 or more threads. Figure 13.4 (left) shows the running time of PLZ3 on 40 cores as a function of the input size on random characters with an alphabet size of 10. We see that the performance of PLZ3 scales gracefully with input size. Figure 13.4 (right) shows the breakdown of the running time of PLZ3 on several input strings. For PLZ3, the suffix array takes 70–80% of the time. If the lcp values are also computed (as in PLZ1 and PLZ2), then the suffix array time becomes about 1.5 times slower,[2] which explains why PLZ3 improves over PLZ1 and PLZ2 by not computing the LCP array. The LPF computation takes about 15–20% of the overall time, and the ANSV computation and conversion from LPF to LZ take very little time. The suffix array portion of the code achieves the lowest speedup, so improvements in parallel suffix array code will likely improve the LZ-factorization code as well.

Comparing the 40-core times for LZ-factorization (Table 13.1) with the times for suffix tree construction using the code from Chapter 11 (Table 11.2), it can be seen that the suffix tree algorithm takes more time than the entire PLZ3 algorithm for the inputs appearing in both tables.[3] Since a suffix tree-based parallel LZ-factorization algorithm involves many other procedures (e.g., tree contraction, least common ancestors, and Euler tours), it is unlikely that such an algorithm will have a better overall performance.

[2]This could be improved by using the algorithms in Chapter 12, which were developed subsequent to the work in this chapter.

[3]The hash table portion of the suffix tree code is not needed for LZ-factorization, but this portion takes less than 10% of the overall time, as shown in Chapter 11. Even after adjusting for this, PLZ3 is still as fast as or faster than the suffix tree code.

Chapter 14

Parallel Wavelet Tree Construction

14.1 Introduction

The *wavelet tree* was first described by Grossi et al. [199], where it was used in compressed suffix arrays. It is a space-efficient data structure that supports access, rank, and select queries on a sequence in $O(\log \sigma)$ work, where σ is the alphabet size of the sequence. Since its initial use, wavelet trees have found many other applications, for example in compressed representations of sequences, permutations, grids, graphs, self-indexes based on the Burrows-Wheeler transform [81], images, two-dimensional range queries [316], among many others (see [346, 317] for surveys of applications). While applications of wavelet trees have attracted significant attention, wavelet tree construction has not been widely studied. This is not surprising, as the standard sequential algorithm for wavelet tree construction is very straightforward. The algorithm requires $O(n \log \sigma)$ work for a sequence of length n. However, constructing the wavelet tree of large sequences (with large alphabets) can be time-consuming, and hence parallelizing the construction is important. A step in this direction was taken recently by Fuentes-Sepulveda et al. [160], who describe parallel algorithms for constructing wavelet trees that require $O(n)$ depth.

This chapter presents parallel algorithms for wavelet tree construction that exhibit much more parallelism (in particular, polylogarithmic depth) [418]. First, an algorithm that constructs the tree level-by-level is introduced, and shown to require $O(n \log \sigma)$ work and $O(\log n \log \sigma)$ depth. Then, a second algorithm that requires $O(W_{sort}(n) \log \sigma)$ work and $O(D_{sort}(n) + \log n)$ depth is presented, where $W_{sort}(n)$ and $D_{sort}(n)$ are the work and depth, respectively, of the parallel stable integer sorting routine used in the algorithm. Using a linear-work integer sort [388], this gives a work bound of $O((n/\epsilon) \log \sigma)$ and depth bound of $O((1/\epsilon)(\sigma^{\epsilon} + \log n))$ for some constant $0 < \epsilon < 1$, which is sub-linear. For alphabets of polylogarithmic size, this gives an algorithm with $O(\log n)$ depth. Using a super-linear work integer sort [389, 41], a work bound of $O(n \log \log n \log \sigma)$ and depth bound of $O(\log n)$ for all alphabets can be obtained. In addition to having good theoretical bounds, the algorithms developed in this chapter are also efficient in practice. We implement the algorithms using Cilk Plus and show experiments on a 40-core shared-memory machine (with two-way hyper-threading) indicating that they outperform the existing parallel algorithms for wavelet tree construction by 1.3–5.6x and achieve up to 27x speedup over the sequential algorithm. The experiments also show that the implementations scale well with increasing thread count, input size, and alphabet size. The parallel construction of rank/select structures on binary sequences, which are an essential component to wavelet trees, is then described. Finally, the chapter describes how to adapt the algorithms to variants of wavelet trees—Huffman-shaped wavelet trees [155], multiary wavelet trees [151], and wavelet matrices [101].

14.2 Preliminaries

For a sequence S, $\mathsf{access}(\mathsf{S}, i)$ returns the symbol at position i of S, $\mathsf{rank}_c(\mathsf{S}, i)$ returns the number of times c appears in S from positions 0 to i, and $\mathsf{select}_c(\mathsf{S}, i)$ returns the position of the i'th occurrence of c in S.

A ***wavelet tree*** is a data structure that supports access, rank and select operations on a sequence in $O(\log \sigma)$ work. The standard wavelet tree is a binary tree where each node represents a range of the symbols in Σ using a bitmap (binary sequence). This chapter assumes $\sigma \leq n$ as the symbols can be mapped to a contiguous range otherwise. The structure of the wavelet tree is defined recursively as follows: The root represents the symbols $[0, \dots, 2^{\lceil \log \sigma \rceil} - 1]$. A node v which represents the symbols $[a, \dots, b]$ stores a bitmap which has a 0 in position i if the i'th symbol in the range $[a, \dots, b]$ is in the range $[a, \dots, ((a + b + 1)/2) - 1]$, and 1 otherwise. It will have a left child that represents the symbols $[a, \dots, ((a + b + 1)/2) - 1]$ and a right child that represents the symbols $[(a + b + 1)/2, \dots, b]$. The recursion stops when the size of the range is 2 or less or if a node has no symbols to represent. Note that the original wavelet tree description in [199] uses a root whose range is not necessarily a power of 2. However, the definition used here gives the same query complexities and leads to a simpler description of the algorithms.

Along with the bitmaps, each node stores a *succinct* rank/select structure (whose size is sub-linear in the bitmap length) to allow for constant work rank and select queries. The structure of a wavelet tree requires $n\lceil \log \sigma \rceil + o(n \log \sigma)$ bits (the lower-order term is for the rank/select structures). The tree topology (parent and child pointers) requires $O(\sigma \log n)$ bits, though this can be reduced or removed by modifying the queries accordingly [316, 100]. The standard sequential algorithm for wavelet tree construction takes $O(n \log \sigma)$ work.

14.3 Related Work

Fuentes-Sepulveda et al. [160] describe a parallel algorithm for constructing a wavelet tree. They observe that for an alphabet where the symbols are contiguous in $[0, \sigma - 1]$, the node at which a symbol s is represented at level i of the wavelet tree can be computed as $s \gg \lceil \log \sigma \rceil - i$, requiring constant work. With this observation they can compute the bitmaps of each level independently. Each level is computed sequentially, requiring $O(n)$ work and depth. Thus, their algorithm requires an overall work of $O(n \log \sigma)$ and $O(n)$ depth. They describe a second algorithm which splits the input sequence into P sub-sequences, where P is the number of cores available. In the first step, the wavelet tree for each sub-sequence is computed sequentially and independently. Then in the second step, the partial wavelet trees are merged. The merging step requires $O(n)$ depth. Thus the algorithm again requires $O(n \log \sigma)$ work and $O(n)$ depth. This algorithm was shown to perform better than the first algorithm due to the high parallelism in the first step.

Multiple queries on the wavelet tree can be answered in parallel since they do not modify the tree. Furthermore, they can be batched to take advantage of cache locality [160].

Arroyuelo et al. [16] explore the use of wavelet trees in distributed search engines. They do not construct the wavelet tree for the entire text in parallel, but instead sequentially construct the wavelet tree for parts of the text on each machine.

Tischler [442] and Claude et al. [102] discuss how to reduce the space usage of sequential wavelet tree construction. Foschini et al. [155] describe an improved algorithm for sequentially constructing the wavelet tree in compressed format, requiring $O(n + \min(n, nH_h) \log \sigma)$ work, where H_h is the h'th order entropy of the input. The approach only works if the object produced is the wavelet tree compressed using run-length encoding. Parallelizing these techniques is a direction for future work. Very recently, Babenko et al. [19] and Munro et al. [342] describe sequential wavelet tree construction algorithms that require $O(n \log \sigma / \sqrt{\log n})$ work. The algorithms pack small integers into words, and require extensive bit manipulation. As far as we know, there are no implementations of the algorithms available. Designing practical (parallel) implementations of these algorithms is left for future work.

14.4 Parallel Wavelet Tree Construction

This section describes new parallel algorithms for wavelet tree construction. The construction requires a rank/select data structure for binary sequences. For now, assume that such structures can be created in linear work and logarithmic depth, and the description of how to do so is deferred to Section 14.6.

```
1:  procedure LEVELWT(S, $n$, $\sigma$)
2:      Nodes = {},   $L = \lceil \log \sigma \rceil$,   $\mathsf{S}' = \mathsf{S}$,   $A = \{(0, n, 0, 2^L)\}$,   $A' = \{\}$
3:      for $l = 0$ to $L - 1$ do                                                    ▷ Process level-by-level
4:          mask = $2^{L-(l+1)}$,   $B$ = bitmap of length $n$                       ▷ Mask and bitmap for this level
5:          parfor $j = 0$ to $|A| - 1$ do
6:              start = $A[j]$.start,   len = $A[j]$.len,   id = $A[j]$.id,   $r = A[j]$.range
7:              Nodes[id].bitmap = $B$ + start,   Nodes[id].len = len
8:              if $r \leq 2$ then                                                   ▷ Node has no children
9:                  parfor $i = 0$ to len $- 1$ do
10:                     if (S[start + $i$] & mask $\neq 0$) then $B$[start + $i$] = 1 else $B$[start + $i$] = 0
11:             else
12:                 $X = \{\}$                                       ▷ Array used to store target positions into S′
13:                 parfor $i = 0$ to len $- 1$ do { if (S[start + $i$] & mask = 0) $X[i] = 1$ else $X[i] = 0$ }
14:                 Perform prefix sum on $X$ to get offsets of "left" characters ($X_s$ is the total sum, i.e. number of "left" characters)
15:                 parfor $i = 0$ to len $- 1$ do
16:                     if (S[start + $i$] & mask $\neq 0$) then $B$[start + $i$] = 1,   S′[start + $X_s$ + $i$ − $X[i]$] = S[start + $i$]
17:                     else $B$[start + $i$] = 0,   S′[start + $X[i]$] = S[start + $i$]
18:                 if $X_s > 0$ then $A'[2 * j]$ = (start, $X_s$, $2 *$ id + 1, $r/2$)                  ▷ Left child
19:                 if (len − $X_s$) > 0 then $A'[2 * j + 1]$ = (start + $X_s$, len − $X_s$, $2 *$ id + 2, $r/2$)   ▷ Right child
20:         Filter out empty $A'$ entries and store into $A$
21:         swap(S, S′)
22:     return Nodes
```

Figure 14.1: levelWT: Level-by-level parallel algorithm for wavelet tree construction.

14.4.1 LevelWT Algorithm

The first algorithm, ***levelWT***, constructs the wavelet tree level-by-level. On each level, the nodes and their bitmaps are constructed in parallel in $O(n)$ work and $O(\log n)$ depth, which gives an overall complexity of $O(n \log \sigma)$ work and $O(\log n \log \sigma)$ depth since there are $O(\log \sigma)$ levels in the tree. The pseudocode for levelWT is shown in Figure 14.1. The algorithm maintains a bitmap B of length n shared by all nodes on each level (Line 4). Each node will simply store its starting point in this array along with its bitmap length. To keep track of which nodes need to be constructed on each level, the algorithm maintains an array A of information for nodes to be added at the next level. Each entry in A stores the starting point (*start*) in the level bitmap (also the starting point in the sequence for the level) for the node, bitmap length (*len*), node identifier (*id*), and length of the alphabet range that it represents (*range*). An entry of A is represented as a 4-tuple (start, len, id, range).

Initially, the array A contains just the root node, with a starting index of 0, length of n (it represents all elements), node ID of 0, and a range of $2^{\lceil \log \sigma \rceil}$ (Line 2). The array A' is used as the output array for the level. The algorithm proceeds one level at a time for $\lceil \log \sigma \rceil$ levels (Line 3). The bitmaps on each level are determined by the $l + 1$'st highest bit in the symbols, so the algorithm uses a mask to determine the sign of this bit in the symbols (Line 4). The algorithm then loops through all the nodes on the current level in parallel (Lines 5–21). For each node, it sets its bitmap pointer and length in the Nodes array (Line 7). If the alphabet range of the node is 2 or less, then it has no children (Line 8). The algorithm then just loops over the node's symbols in S in parallel, and sets each bit in the bitmap according to the sign of the symbol's $l+1$'st highest bit (Lines 9–10).[1] Otherwise, the symbols in S are rearranged (and stored into S') so that they are in the correct order on the next level. To do this in parallel, the algorithm first counts the number of symbols that go to the left child ($l+1$'st highest bit is 0), and the offset of each such symbol using a prefix sum (Lines 12–14). With the prefix sum array X and result X_s, the symbols that go to the right child can be computed as well using the formula $X_s + i - X[i]$ (the number of symbols with an $l+1$'st highest bit of 0 is X_s, so this is the number of symbols to offset, and then the number of symbols with an $l+1$'st highest bit of 1 up to index i is $i - X[i]$). In Lines 15–17, the bits in the bitmap and positions in S' are set in parallel. Children nodes are placed into A' on Lines 18–19 if the number of symbols represented for the child is greater than 0. The children's node IDs are computed as twice the current node ID plus one for the left child and twice the current node ID plus two for the right child in order to give all nodes unique IDs. The starting point in the bitmap of the next level is the same as the current starting point for the left child, and is the current starting point plus the number of elements on the left (X_s) for the right child. The length is stored from the computation before. The range of each child is half of the current range. After each level, the non-empty entries of A' are filtered out and stored into A (Line 20). The roles of S and S' are swapped for the next level (Line 21).

Note that setting the bits in B (Lines 10, 16, and 17) must be done atomically since multiple cores may write to the same word concurrently. This can be implemented using a loop with a compare-and-swap until successful. An optimization is to only perform atomic writes if a word is shared between two nodes (there can be at most two shared words per node, as the bits for each node are contiguous in B), and for the remaining words we parallelize at the granularity of a word. Inside each word, the updates are done sequentially. This allows the algorithm to use regular writes for all but at most two words per node.

Note that the algorithm is able to stop early when the range of a node is 2 or less since the construction of the previous level provides this information. The algorithm of [160] (and also the next algorithm described in this section) processes all levels independently, so it is not easy to stop early.

Let us now analyze the complexity of levelWT. For each level, there is a total of $O(n)$ work performed, since each symbol is processed a constant number of times. The prefix sum on Lines 14 and the filter on Line 20 require linear work and $O(\log n)$ depth per level. The parallel for-loops on Lines 9–10, 13, and 15–17 require linear work and $O(1)$ depth per level. There are $O(\log \sigma)$ levels so the total work is $O(n \log \sigma)$ and depth is $O(\log n \log \sigma)$. Since $\sigma \leq n$, the depth is polylogarithmic in n. This gives the following theorem:

Theorem 34. *levelWT requires $O(n \log \sigma)$ work and $O(\log n \log \sigma)$ depth.*

[1]The actual code requires bit arithmetic to access the appropriate bit in the word, which is omitted for simplicity.

```
procedure SORTWT(S, $n$, $\sigma$)
    Nodes = {},    $L = \lceil \log \sigma \rceil$
    parfor $l = 0$ to $L - 1$ do
        mask = $2^{L-(l+1)}$,    $B$ = bitmap of length $n$          ▷ Mask and bitmap for this level
        if $l = 0$ then                                               ▷ No sorting required for first level
            parfor $i = 0$ to $n - 1$ do { if (S[$i$] & mask $\neq 0$) $B[i] = 1$ else $B[i] = 0$ }
            Nodes[0].bitmap = 0,    Nodes[0].len = $n$
        else
            S$'$ = S stably sorted by top $l$ bits
            parfor $i = 0$ to $n - 1$ do { if (S$'[i]$ & mask $\neq 0$) $B[i] = 1$ else $B[i] = 0$ }
            $O$ = indices $i$ such that (S$'[i] \gg L - l$) $\neq$ (S$'[i-1] \gg L - l$)      ▷ Using a filter
            parfor $i = 0$ to $|O| - 1$ do
                id = $2^l - 1 +$ (S$'[O[i]] \gg L - l$),    Nodes[id].bitmap = $O[i]$,    Nodes[id].len = $O[i+1] - O[i]$
    return Nodes
```

Figure 14.2: sortWT: Sorting-based parallel algorithm for wavelet tree construction.

14.4.2 SortWT Algorithm

This section describes the second wavelet tree construction algorithm, which constructs all levels of the wavelet tree in parallel, and is referred to as ***sortWT***. Since preceding levels cannot provide information to later levels, independent computation must be performed per level to obtain the correct ordering of the sequence for the level. The algorithm makes use of the observation of Fuentes-Sepulveda et al. [160] that the node at which a symbol s is represented at level l of the wavelet tree ($l = 0$ for the root) is encoded in the top l bits. For level l, the algorithm sorts S using the top l bits as the key, which gives the correct ordering of the sequence for the level. Note that the sort must be stable since the relative ordering of nodes with the same top l bits must be preserved in the wavelet tree.

The pseudocode for sortWT is shown in Figure 14.2. As in levelWT, a mask and a bitmap B is used for each level (Line 4). For the first level ($l = 0$), no sorting of S is required, so the algorithm simply fills the bitmap according to the highest bit of each symbol (Lines 5–6). For each subsequent level, the algorithm first stably sorts S by the top l bits to obtain the symbols in the correct order, and stores it in S$'$ (Line 9). The bitmap is filled according to the $l + 1$'st highest bit of each symbol (Line 10). To compute the length of each node's bitmap, a filter is used to find all the indices where the symbol's top l bits differ from the previous symbol's top l bits in S$'$ (Line 11). These mark the bitmaps of each node since S$'$ is sorted by the top l bits. The length of each bitmap can be computed by the difference in indices. On Lines 12–13, the algorithm sets the bitmap pointers and lengths for the nodes on the current level. The IDs of the nodes start at $2^l - 1$, since there are up to $2^l - 1$ nodes in previous levels, and each node ID is offset by the top l bits of the symbols that it represents, as this determines the node's position in the level. As in levelWT, updates to the bitmaps are done in parallel at word granularity.

Let us now analyze the algorithm's complexity. Let $W_{sort}(n)$ and $D_{sort}(n)$ be the work and depth, respectively, of the stable sort on Line 9. The filter on Line 11 requires $O(n)$

work and $O(\log n)$ depth. The parallel for-loops on Lines 6, 10 and 12–13 require $O(n)$ work and $O(1)$ depth. The overall work is $O(W_{sort}(n) \log \sigma)$ and since all levels can be computed in parallel, the overall depth is $O(D_{sort}(n) + \log n)$. This gives the following theorem:

Theorem 35. *sortWT requires $O(W_{sort}(n) \log \sigma)$ work and $O(D_{sort}(n) + \log n)$ depth.*

Using linear-work parallel stable integer sorting [388], where $W_{sort}(n) = O(n/\epsilon)$ and $D_{sort}(n) = O((1/\epsilon)(\sigma^{\epsilon} + \log n))$ for some constant $0 < \epsilon < 1$, gives a work bound of $O((n/\epsilon) \log \sigma)$ and depth bound of $O((1/\epsilon)(\sigma^{\epsilon} + \log n))$, which is sub-linear. For $\sigma = O(\log^c n)$ for any constant c, this gives $O(\log n)$ depth (by setting ϵ appropriately). Alternatively, a stable integer sorting algorithm with $W_{sort}(n) = O(n \log \log n)$ and $D_{sort} = O(\log n / \log \log n)$ (either using randomization [389] or using super-linear space [41]) can be used to obtain an overall work of $O(n \log \log n \log \sigma)$ and depth of $O(\log n)$ for any alphabet size.

14.4.3 Space usage

The input and output of the algorithms is $O(n \log \sigma)$ bits. levelWT requires two auxiliary arrays for the prefix sum on each level (that can be reused per level), which takes $O(n \log n)$ bits. For sortWT, since all levels are processed in parallel, and each level requires $O(n \log n)$ bits for the integer sort, the total space usage is $O(n \log n \log \sigma)$ bits.

Due to the high space usage and hence memory footprint of sortWT, processing the levels one-by-one gives better performance in practice, as will be discussed in Section 14.5 (although this increases the depth by a factor of $O(\log \sigma)$). This modified version is referred to as ***msortWT***. On each level, msortWT sorts the sequence from the previous level. Since the levels are processed one-by-one, msortWT has a space usage of $O(n \log n)$ bits.

14.5 Experiments

Implementations. This section compares implementations of levelWT, sortWT, and msortWT with existing parallel implementations as well as a sequential implementation. The implementations all use the levelwise representation of the bitmaps, where one bitmap of length n is stored per level, and nodes have pointers into the bitmaps. sortWT and msortWT use linear-work parallel stable integer sorting. The implementations use the parallel prefix sum, filter, and integer sorting routines from the Problem Based Benchmark Suite [424]. The experiments compare the parallel implementations developed in this chapter with the implementations of Fuentes-Sepulveda et al. [160], one which computes each level of the wavelet tree independently in parallel (***FEFS***), and the other which computes a partial wavelet tree for each thread, and then merges them together (***FEFS2***). Both implementations require the alphabet size to be a power of 2, so times are reported only for the inputs with such an alphabet size. We implemented a sequential version of wavelet tree construction (***serialWT***), and found its performance to be competitive with the times of the sequential algorithm reported in [200]. We also tried the serial implementation in SDSL [178] for constructing a balanced wavelet tree, but found it to be slower than serialWT on the inputs used in this section. However, the SDSL implementation is more space-efficient, and sometimes faster on the Burrows-Wheeler transformed inputs. A comparison with SDSL is presented in the full version of the paper that this chapter is based on [419].

Experimental Setup. The experiments are performed on the 40-core (with two-way hyper-threading) machine described in Section 2.7. The parallel codes use Cilk Plus, and are compiled with `g++`. The times reported are based on a median of 3 trials.

The experiments use a variety of real-world and artificial sequences. The real-world sequences include strings from `http://people.unipmn.it/manzini/lightweight/corpus/`, XML code from Wikipedia (*wikisamp*), protein data from `http://pizzachili.dcc.uchile.cl/texts/protein/` (*proteins*), the human genome from `http://webhome.cs.uvic.ca/~thomo/HG18.fasta.tar.gz` (*HG18*), and a document array of text collections (*trec8*). The artificial inputs (*rand*), parameterized by σ, are generated by drawing each symbol uniformly at random from the range $[0, \ldots, \sigma - 1]$. The lengths and alphabet sizes of the inputs are listed in Table 14.1.

Due to the various choices for rank/select structures, each of which has different space/time trade-offs, their construction times are not included in the wavelet tree construction time. The FEFS and FEFS2 codes were modified accordingly. The times for the implementations developed in this chapter include generating the parent/child pointers for the nodes, although these could be removed using techniques from [316, 100]. FEFS and FEFS2 do not generate these pointers.

Results. Table 14.1 shows the single-thread (T_1) and 40-core with two-way hyper-threading (T_{40h}) running times on the inputs for the various implementations. The experimental results show that levelWT is faster than sortWT and msortWT both sequentially and in parallel. This is because sortWT and msortWT use sorting, which has a larger overhead. msortWT is slightly faster than sortWT due to its smaller memory footprint. Compared to serialWT, levelWT is 1.2–1.8x slower on a single thread, and 13–27x faster on 40 cores with hyper-threading. The self-relative speedup of levelWT ranges from 23 to 35. On 40 cores, sortWT and msortWT are 6–19x and 7–22x faster than serialWT, respectively. sortWT and msortWT achieve self-relative speedups of 17–31 and 22–34, respectively.

FEFS2 always outperforms FEFS on 40 cores with two-way hyper-threading because FEFS splits the work among only $\log \sigma$ threads, which is less than 80 on all of the inputs ($\sigma < 2^{80}$), whereas FEFS2 splits the work among all available threads in its first step. The second (merging) step of FEFS2, however, only makes use of $\log \sigma$ threads, but this is a

Text	n	σ	serialWT (T_1)	levelWT (T_1)	levelWT (T_{40h})	sortWT (T_1)	sortWT (T_{40h})	msortWT (T_1)	msortWT (T_{40h})	FEFS (T_1)	FEFS (T_{40h})	FEFS2 (T_1)	FEFS2 (T_{40h})
chr22	$3.35 \cdot 10^7$	4	0.486	0.611	0.018	0.786	0.046	0.768	0.029	1.03	0.53	0.98	0.1
etext99	$1.05 \cdot 10^8$	146	4.12	6.99	0.28	10.5	0.393	9.79	0.364	–	–	–	–
HG18	$2.83 \cdot 10^9$	4	35.5	45.5	1.32	57.7	1.88	56.9	1.67	76.6	39.1	72.2	2.55
howto	$3.94 \cdot 10^7$	197	1.65	2.69	0.105	4.07	0.161	3.86	0.144	–	–	–	–
jdk13c	$6.97 \cdot 10^7$	113	2.51	3.9	0.159	6.13	0.234	5.63	0.22	–	–	–	–
proteins	$1.18 \cdot 10^9$	27	31.3	52.2	1.82	75.3	2.59	71.3	2.28	–	–	–	–
rctail96	$1.15 \cdot 10^8$	93	3.54	5.88	0.231	10.2	0.373	9.39	0.34	–	–	–	–
rfc	$1.16 \cdot 10^8$	120	3.8	6.51	0.261	10.1	0.37	9.28	0.348	–	–	–	–
sprot34	$1.1 \cdot 10^8$	66	3.69	6.26	0.248	9.48	0.36	8.8	0.328	–	–	–	–
trec8	$2.43 \cdot 10^8$	528155	33.3	50.4	2.08	138	5.5	104	4.55	–	–	–	–
w3c2	$1.04 \cdot 10^8$	256	3.82	6.66	0.275	10.6	0.388	9.78	0.357	11.1	2.0	10.6	0.51
wikisamp	10^8	204	3.52	6.16	0.264	9.78	0.374	9.08	0.349	–	–	–	–
rand-2^8	10^8	2^8	5.76	8.58	0.36	14.3	0.652	12.1	0.533	12.4	1.71	12.3	0.5
rand-2^{10}	10^8	2^{10}	6.88	11	0.456	19	0.857	15.9	0.708	15.0	1.71	15.3	0.58
rand-2^{12}	10^8	2^{12}	8.32	12.4	0.525	24.5	1.11	20.4	0.922	18.7	1.78	17.4	0.67
rand-2^{16}	10^8	2^{16}	11.2	16.4	0.655	36	1.59	29.5	1.34	34.4	3.26	32.4	1.28
rand-2^{20}	10^8	2^{20}	14	20.4	0.772	49.5	2.19	40.6	1.85	65.7	7.14	64.8	3.94

Table 14.1: Comparison of running times (seconds) of wavelet tree construction algorithms on a 40-core machine with hyper-threading. T_{40h} is the time using 40 cores (80 hyper-threads) and T_1 is the time using a single thread.

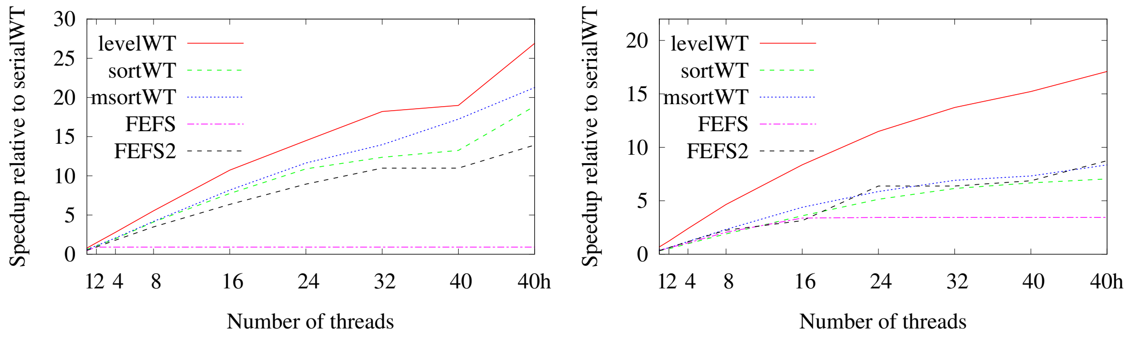


Figure 14.3: Speedup of implementations relative to serialWT for HG18 (**left**) and rand-2^{16} (**right**). "40h" corresponds to 80 hyper-threads.

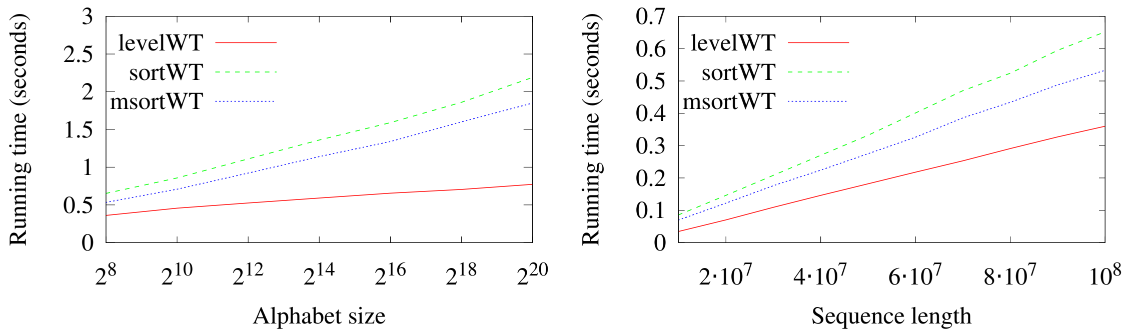


Figure 14.4: 40-core (with hyper-threading) running times vs. σ (**left**, x-axis in log-scale) and vs. n (**right**) on random sequences ($\sigma = 2^8$).

smaller fraction of the total time. On 40 cores with hyper-threading, the best implementation from this chapter (levelWT) outperforms FEFS2 by a factor of 1.3–5.6x, and FEFS by much more. Compared to msortWT, FEFS2 is faster in some cases and slower in others.

Figure 14.3 shows the speedup of the parallel implementations relative to serialWT as a function of the number of threads for HG18 and rand-2^{16}. For HG18, the implementations developed in this chapter and FEFS2 scale well up to 80 hyper-threads. FEFS only scales up to 2 threads due to the small alphabet size. For rand-2^{16}, the implementations developed in this chapter again exhibit good scalability. FEFS scales up to 16 threads as there are 16 levels in the tree, while FEFS2 scales to more threads. FEFS2 is competitive with msortWT, but slower than levelWT.

Figure 14.4 (left) shows the 40-core parallel running time of the three implementations from this chapter as a function of the alphabet size for random sequences of length 10^8. We see that for fixed n, the running times increase nearly linearly with $O(\log \sigma)$, which is expected since the total work is $O(n \log \sigma)$. Figure 14.4 (right) shows the running time of the implementations as a function of n for $\sigma = 2^8$ on random sequences, and we see that the times increase linearly with n as expected. The algorithms introduced in this chapter exhibit similar parallel speedups on 40 cores as σ or n is varied, since the core count is much lower than the available parallelism of the algorithms.

In summary, the experiments show that the parallel algorithms for wavelet tree construction developed in this chapter scale well with the number of threads, input length, and alphabet size. levelWT outperforms sortWT and msortWT as it does not have the overheads of sorting, and achieves good speedup over serialWT. Overall, levelWT outperforms FEFS and FEFS2.

14.6 Parallel Construction of Rank/Select Structures

Wavelet trees make use of succinct rank/select structures which support constant-work rank and select queries on binary sequences. This section describes how to construct these structures in parallel using $O(n)$ work and $O(\log n)$ depth for a binary sequence of length n. *Sequential* construction of rank/select structures in $o(n)$ work have been described in [19, 342], however parallel construction in linear work suffices for the purposes of using them in the wavelet tree construction algorithms in this chapter.

The rank structure of Jacobson [242] stores the rank of every $\log^2 n$'th bit in a first-level directory, and the rank of every $\log n$'th bit in each of the ranges in a second-level directory. Rank queries in each range of size $\log n$ can be answered by at most two table look-ups, where the table stores the rank of all bit-strings of length up to $\log(n)/2$. The first- and second-level directories of can be constructed by converting the bit-string to a length n array of 0's and 1's and computing a prefix sum on the array in $O(n)$ work and $O(\log n)$ depth. Entries in the second-level directory require $\log \log n$ bits each, and for space efficiency, they need to be packed into words. This can be done by processing groups of $O(\log n/\log\log n)$ entries (the number that fits in a word) in parallel, and packing each group into a word sequentially. There are $O(n/\log n)$ entries, and word operations take $O(1)$ work and depth per entry, so this process takes $O(n)$ work and $O(\log n)$ depth. The look-up table can be constructed in $o(n)$ work and $O(\log n)$ depth, as the number of 1's in bit-strings of size $O(\log n)$ can be computed in $O(\log n)$ work and depth, and there are $O(2^{\log(n)/2}\log n) = O(\sqrt{n}\log n)$ such bit-strings.

Clark's select structure [99] stores the position of every $\log n \log\log n$'th 1 bit in a first-level directory. Then for each range r between the positions, if $r \geq \log^2 n(\log\log n)^2$, then the $\log n\log\log n$ answers in the range are stored directly. Otherwise, the position of every $\log r \log\log n$'th 1 bit is stored in a second-level directory. The sub-ranges r' in the second-level directory are again considered, and if $r' \geq \log r' \log r(\log\log n)^2$, then all answers in the range are stored directly. Otherwise, a look-up table is constructed for all bit-strings of length less than r'. To parallelize the construction, the bit-string is first converted to an array of 0's and 1's, and then the positions of all the 1 bits are computed using a prefix sum and filter in $O(n)$ work and $O(\log n)$ depth. This allows all of the ranges to be processed in parallel. Constructing each second-level directory again uses prefix sum and filter. Over all directories, this sums to $O(n)$ work and $O(\log n)$ depth. The look-up table can be constructed in $o(n)$ work and $O(1)$ depth, similar to the rank structure. Packing entries into words can also be done within the complexity bounds.

It is worth noting that there have been more practical variants of rank/select structures (see, e.g., [473, 181] and references therein) that have a similar high-level structure.

14.7 Extensions

The ***Huffman-shaped wavelet tree***, where each node is placed at a level proportional to the length of its Huffman code, was introduced to improve compression and average query performance [155]. The Huffman-shaped wavelet tree can be constructed in parallel by first computing the Huffman tree and prefix codes in $O(\sigma + n)$ work and $O(\sigma + \log n)$ depth using the algorithm of [142]. The construction then follows the strategy of levelWT. To decide how to set the bitmaps in the internal nodes and rearrange S, the algorithm must know the side of the tree that each symbol is located on. The algorithm maps each symbol to an integer corresponding to the location of its leaf in an in-order traversal of the Huffman tree, which can be done in parallel using an Euler tour algorithm in $O(\sigma)$ work and $O(\log \sigma)$ depth [243]. At each internal node, the symbols to the left and to the right are in consecutive ranges, and the highest mapped integer of nodes in its left sub-tree is stored during the Euler tour computation. Then the decision of how to set the bitmap and where to place the symbol in S' can be made with a single comparison with the mapped integers. The rest of the computation follows the logic of levelWT. For a tree of height h, the overall work (including Huffman encoding) is $O(nh)$ as linear work is done per level. The overall depth is $O(\sigma + h \log n)$, as each level of the tree takes $O(\log n)$ depth.

Ferragina et al. [151] describe the ***multiary wavelet tree*** where each node has up to d children for some value d, and stores sequences of symbols in the range $[0, \ldots, d-1]$. The height of the tree is $O(\log_d \sigma)$. This section describes the parallel construction for the case where $d = O(\log^\epsilon n)$ for $\epsilon < 1/3$, and d is a power of two.[2] To do this, sortWT is modified to process the levels one-by-one and save the sorted sequence S' for the next level. On level l, S' is already sorted by the top $(l-1)\log d$ bits, so the algorithm only needs to sort the next $\log d$ highest bits within each of the sub-sequences sharing the same top $(l-1)\log d$ bits. The sub-sequence boundaries can be identified with a filter, and for each sub-sequence the algorithm applies a stable integer sort using the $\log d$ appropriate bits for the level as the key. The bitmap B is substituted with a sequence of $\log d$-bit entries, and in parallel each

[2]The requirement $\epsilon < 1/3$ is necessary for the analysis of the rank/select structure [19] that is parallelized.

entry is set according to the value of the appropriate $\log d$ bits of each symbol. There will be up to d^l nodes on level l, and the offset to the Nodes array is $d^l - 1$. Since $d = O(\log^\epsilon n)$, the integer sort on each level requires $O(n)$ work and $O(\log n)$ depth, giving an overall work of $O(n \log_d \sigma)$ and depth of $O(\log n \log_d \sigma)$. The parallel construction of rank/select structures on sequences with larger alphabets, which are used in the multiary wavelet tree, is described in [419].

The ***wavelet matrix*** [101] is a variant of the wavelet tree where on level l, all symbols with a 0 as their l'th highest bit are represented on the left side of the level's bitmap and all symbols with a 1 as their l'th highest bit are represented on the right. Each level stores the number of 0's on the level. The bitmap is filled based on the $l + 1$'st highest bit of the symbols. To construct the wavelet matrix, the algorithm proceeds level-by-level and stably reorders S based on the l'th highest bit of the symbols using standard operations involving prefix sum (similar to levelWT) in $O(n)$ work and $O(\log n)$ depth, which also gives the number of 0's on the level. The bitmap for the level is then filled in parallel in $O(n)$ work and $O(1)$ depth. This gives an algorithm with $O(n \log \sigma)$ work and $O(\log n \log \sigma)$ depth. Alternatively, a strategy similar to sortWT can be employed, but using the reverse of the top l bits as the key when sorting.

Chapter 15

Conclusion and Future Work

15.1 Summary

The emergence of commodity shared-memory multicore machines in the past decade has allowed a wide class of problems to be solved very efficiently without resorting to expensive supercomputers. This thesis demonstrated the power of multicore machines for solving various important irregular problems in computing, providing efficient solutions that scale well with the number of cores and that are capable of (and sufficient for) processing the largest real-world data sets studied in the literature. Prior to this work, many of these problems were typically solved using distributed-memory solutions and were much less efficient on a per-core, per-dollar, and per-joule basis. To make large-scale computing using multicores more accessible to the masses, this thesis adopted a three-pronged approach of addressing challenges in the programming, algorithm design, and performance of shared-memory solutions.

Part I of the thesis introduced techniques for writing efficient internally deterministic parallel programs, guaranteeing that the final result as well as certain intermediate states are deterministic, independent of the number of threads, how they are scheduled, and any nondeterminism in the underlying hardware and software. The thesis developed a new approach for writing efficient internally deterministic code using commutative building blocks (e.g., priority updates, phase-concurrent hash tables, and disjoint sets), and introduced the deterministic reservations framework for parallelizing sequential loops with dependencies among iterations. These tools were employed to design internally deterministic parallel solutions that are simple, efficient (competitive with nondeterministic solutions for the same problem), and scalable. Furthermore, this thesis gave the first theoretical proofs that several natural sequential iterative algorithms are in fact highly parallel (polylogarithmic depth), leading to efficient and deterministic parallel implementations of the algorithms.

Part II of the thesis introduced Ligra, a lightweight graph processing system for shared-memory. The thesis showed that the two simple functions provided by the framework are sufficient for expressing a wide class of graph algorithms. The Ligra system includes optimizations that make graph traversal algorithms particularly efficient, and the thesis demonstrated that the code written in Ligra is much faster than existing graph processing systems, and competitive with highly-optimized code while being much simpler. Experiments showed that Ligra can process the largest publicly-available real-world graphs on the order of seconds to minutes on just a single shared-memory server. To enable shared-memory graph processing to become even more efficient and scalable, Ligra+, an extension of Ligra with graph compression techniques, was developed to reduce space usage while improving parallel performance at the same time. Ligra+ is the first graph processing system to use in-memory compression to reduce space usage and improve parallel performance.

Parts III and IV of this thesis bridged the gap between theory and practice in parallel algorithms by designing simple and practical shared-memory algorithms with strong theoretical guarantees for important problems on graphs and strings. Part III gave the first practical linear-work and polylogarithmic-depth algorithm for graph connectivity as well as the first work-efficient, polylogarithmic-depth, and cache-efficient algorithms for triangle computations. Part IV introduced the first practical linear-work and polylogarithmic-depth algorithm for suffix tree construction, several new theoretically-efficient parallel algorithms for computing longest common prefixes, the first practical linear-work and polylogarithmic-depth Lempel-Ziv factorization algorithm, and the first polylogarithmic-depth algorithms for wavelet tree construction. The thesis experimentally evaluated shared-memory implementations of all of the algorithms on multicore machines, and showed that they achieve good speedup relative to the best sequential solutions, outperform existing parallel implementations, and scale to the largest real-world data sets used in the literature.

By addressing challenges in programming, algorithm design, and performance of large-scale shared-memory solutions, this thesis provides evidence showing that *with appropriate programming techniques, frameworks, and algorithms, shared-memory programs can be simple, efficient, and scalable, both in theory and in practice*. The tools developed in this thesis make the use of multicores for large-scale computations more accessible to the community.

15.2 Future Work

This section describes directions for future work that build on this thesis.

Commutativity Violation Detection. The approach of this thesis to developing internally deterministic parallel programs in Chapter 3 uses nested parallelism with commuting and linearizable parallel operations. This thesis has not addressed the issue of how to verify that operations commute or are linearizable, but the techniques used are simple enough that it is quite easy to reason about the correctness. For example, in deterministic reservations a programmer only needs to verify that the operations within the reserve component and separately within the commit component commute. However, to lessen the burden on the programmer, we would like to investigate using runtime techniques for automatically detecting commutativity violations [462, 138] in conjunction with our techniques.

Proving Bounds for Sequential Iterative Algorithms. Chapter 4 of this thesis showed that the sequential maximal independent set and maximal matching algorithms have an iteration depth of $O(\log^2 n)$ with high probability. An interesting direction for future work is to investigate whether this bound is tight. In addition to the problems that studied in Chapter 4, we are also interested in proving bounds for other sequential iterative algorithms that can be parallelized, such as connected components, spanning forest, minimum spanning forest, and Delaunay triangulation/refinement.

Resizing and Automatic Phase-Concurrency. Although the experiments on the phase-concurrent hash table in Chapter 5 did not require resizing, the resizing solution described in the chapter uses locks. My co-authors and I are interested in investigating whether there is an efficient lock-free solution to resizing the hash table. We are also interested in exploring ways to automatically separate operations into phases efficiently, e.g., by using room synchronizations [51]. Finally, we are eager to study other data structures that can be simplified and made more efficient by restricting its use to the phase-concurrent setting.

Uniformity of memory access. While all cores on a multicore machine have access to the same shared memory, the memory access time is not necessarily uniform among the cores. Most multicore machines with multiple sockets (e.g., the ones used for experiments in this thesis) have non-uniform memory access (NUMA) times. In NUMA machines, each socket contains part of the global shared memory, and the latency to access a particular object in memory depends on the distance from the accessing core to where the object is located. Designing NUMA-aware programs for these machines may improve performance, although often requires explicitly controlling the threads which complicates programming. In the future, we are interested in developing programming abstractions that simplify writing NUMA-aware shared-memory code. We note that a NUMA-aware version of our Ligra graph processing framework from Part II has recently been developed [471].

Real-time Graph Algorithms. Currently, Ligra does not support algorithms based on modifying the input graph. An interesting direction for future work is to extend the system to support efficient graph modifications. This feature would also be useful for streaming algorithms in which graph updates arrive in real-time and need to be efficiently processed. There has been significant work on *graph databases*, which support efficient updates to the graph. However, the systems are mostly optimized for local queries on the graph. We would like to explore whether some of these techniques work well for traversal algorithms that explore most of the graph. Instead of restarting algorithms from scratch when updates occur, the interface could be extended with a user-defined function indicating when and where recomputation is necessary. In addition, the graph compression techniques of Ligra+ assume a static graph, and we are interested in extending the graph compression techniques to more easily cope with dynamic graph updates.

External Memory. Recent work has shown that large-scale graph computations that do not fit in memory can be performed efficiently relative to distributed memory by optimizing access to disk [289, 212, 399, 290, 470, 313, 91]. However, if the data fits in memory, then these solutions are slower than shared-memory solutions, such as those presented in this thesis. For data sets that exceed the size of memory, we are interested in designing solutions that take advantage of shared-memory processing for the data that fits in memory, while also having optimized access to data that needs to be stored on disk. In addition to having practical algorithms, we would also like to prove theoretical guarantees about the algorithms, so that they perform well under all possible settings.

Emerging Memory Technologies. In contrast to dynamic random-access memory (DRAM), which is the type of RAM on current multicore machines, emerging non-volatile memory technologies will exhibit a significant gap between writing to memory and reading from memory. These technologies include phase-change memory, Spin-Torque Transfer Magnetic RAM and Memristor-based Resistive RAM. My co-authors and I are interested in exploring how to adapt parallel algorithms for these technologies by minimizing the number of memory writes (possibly at the expense of performing more memory reads). We have done preliminary theoretical work [52] proposing a model where reads and writes have different costs (reads have unit cost while writes have a cost of k), and describing a parallel sorting algorithm that performs $O(n \log n)$ reads, $O(n)$ writes, and has $O(k \log n)$ depth with high probability. We also describe parallel external-memory and cache-oblivious algorithms for sorting, Fast Fourier Transform, and matrix multiplication which trade additional reads for fewer writes. We are interested in studying other fundamental problems in this model, where reads and writes have asymmetric costs. When such technologies become readily available on multicore machines, we are also interested in testing the actual performance of our algorithms, and using the results to guide us in finding the best cost model.

Bibliography

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, Mar. 2004. 11.5, 11.5, 11.5, 11.5, 12.1

[2] U. Acar, G. E. Blelloch, and R. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002. 2.1

[3] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *IEEE Data Compression Conference (DCC)*, pages 203–212, 2001. 8.2

[4] S. V. Adve and M. D. Hill. Weak ordering–a new definition. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 2–14, 1990. 3.1

[5] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2010. 7.5

[6] A. Agrawal, L. Nekludova, and W. Lim. A parallel $O(\log N)$ algorithm for finding connected components in planar images. In *International Conference on Parallel Processing (ICPP)*, pages 783–786, 1987. 9.1

[7] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the GPU. *ACM Trans. Graph.*, 28(5):154:1–154:9, Dec. 2009. 5.2

[8] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583, December 1986. 4.1

[9] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997. 10.9

[10] L. Alonso and R. Schott. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science*, 159(1):15–28, 1996. 4.1

[11] R. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 95–102, 1990. 4.1

[12] A. Apostolico and G. Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009. 8.4.1

[13] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3(1-4):347–365, 1988. 11.1

[14] S. Arifuzzaman, M. Khan, and M. Marathe. PATRIC: A parallel algorithm for counting triangles in massive networks. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 529–538, 2013. 10.1, 10.7.2, 10.9

[15] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. 6

[16] D. Arroyuelo, V. Gil-Costa, S. Gonzalez, M. Marin, and M. Oyarzun. Distributed search based on self-indexed compressed text. *Information Processing & Management*, 48(5):819–827, 2012. 14.3

[17] H. Avron. Counting triangles in large graphs using randomized matrix trace estimation. In *Workshop on Large-scale Data Mining: Theory and Applications*, 2010. 10.5, 10.9

[18] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. In *International Conference on Parallel Processing (ICPP)*, pages 177–187, 1983. 9.1

[19] M. A. Babenko, P. Gawrychowski, T. Kociumaka, and T. A. Starikovskaya. Wavelet trees meet suffix trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 572–591, 2015. 14.3, 14.6, 2

[20] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *International Conference on Parallel Processing (ICPP)*, pages 547–556, 2005. 4.10.2, 9.1

[21] D. A. Bader and J. JaJa. Parallel algorithms for image histogramming and connected components with an experimental study. *J. Parallel Distrib. Comput.*, 35(2):173–190, 1996. 9.1

[22] D. A. Bader, V. Kanade, and K. Madduri. SWARM: A parallel programming framework for multi-core processors. In *Workshop on Multithreaded Architectures and Applications (MTAAP)*, pages 1–8, 2007. 1.1

[23] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Workshop on Algorithms and Models for the Web-Graph (WAW)*, pages 124–137, 2007. 7.4.2

[24] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *International Conference on High Performance Computing (HiPC)*, pages 465–476, 2005. 9.1, 9.5

[25] D. A. Bader, S. Sreshta, and N. R. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In *International Conference on High Performance Computing (HiPC)*, pages 63–75. 2002. 4.10.2

[26] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *International Conference on Very Large Data Bases (VLDB)*, 7(1):85–96, 2013. 10.7.2

[27] D. S. Banerjee and K. Kothapalli. Hybrid algorithms for list ranking and graph connected components. In *International Conference on High Performance Computing (HiPC)*, pages 1–10, 2011. 9.1

[28] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 623–632, 2002. 10.1, 10.9

[29] Y. Bartal. Graph decomposition lemmas and their role in metric embedding methods. In *European Symposium on Algorithms (ESA)*, pages 89–97. 2004. 9.1

[30] M. J. Bauer, A. J. Cox, G. Rosone, and M. Sciortino. Lightweight LCP construction for next-generation sequencing datasets. In *Workshop on Algorithms in Bioinformatics (WABI)*, pages 326–337. 2012. 12.1

[31] S. Beamer, K. Asanović, and D. Patterson. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for Graph500. *Technical Report UCB/EECS-2011-117, EECS Department, University of California, Berkeley*, 2011. 7.1, 7.2.1, 7.5

[32] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 12:1–12:10, 2012. 1.6, 3.5, 7.1, 7.2.1, 7.5, 9.4, 9.5

[33] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 16–24, 2008. 10.1, 10.9

[34] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. *Journal of Discrete Algorithms*, 18:22–31, 2013. 12.1

[35] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 53–64, 2010. 3.1

[36] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010. 3.1

[37] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *ACM Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 81–96, 2009. 3.1

[38] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993. 11.1, 11.4, 11.4.1, 11.5, 11.5, 13.1, 13.3, 13.4

[39] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–14, 2007. 7.2.2

[40] J. W. Berry, L. K. Fostvedt, D. J. Nordman, C. A. Phillips, C. Seshadhri, and A. G. Wilson. Why do simple algorithms for triangle enumeration work in the real world? In *Innovations in Theoretical Computer Science (ITCS)*, pages 225–234, 2014. 10.1

[41] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. *Information and Computation*, 94(1):29–47, 1991. 14.1, 14.4.2

[42] T. Bingmann, J. Fischer, and V. Osipov. Inducing suffix and LCP arrays in external memory. In *Algorithm Engineering and Experiments (ALENEX)*, pages 88–102, 2013. 12.1

[43] M. Birn, V. Osipov, P. Sanders, C. Schulz, and N. Sitchinava. Efficient parallel and external matching. In *Euro-Par*, pages 659–670. 2013. 4.1

[44] A. Bjorklund, R. Pagh, V. V. Williams, and U. Zwick. Listing triangles. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 223–234, 2014. 10.9

[45] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 679–688, 2003. 8.2, 8.4, 8.4.1

[46] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis of a compact graph representation. In *Algorithms Engineering and Experiments (ALENEX)*, pages 49–61, 2004. 8.2, 8.3.1, 8.3.2

[47] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Computers*, 38(11):1526–1538, 1989. 2.2

[48] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990. 1.2

[49] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, 1992. 1.1, 1.2, 2.1, 9.1

[50] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996. 2.1, 3.1

[51] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Scalable room synchronizations. *Theory Comput. Syst.*, 36(5):397–430, 2003. 5.2, 15.2

[52] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015. 15.2

[53] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic algorithms can be fast. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 181–192, 2012. 1.1, 3.5, 3.5

[54] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–366, 2011. 10.1, 1, 10.2

[55] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 308–317, 2012. 1.1, 4.3

[56] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Combinable memory-block transactions. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 23–34, 2008. 6.1

[57] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low-depth cache oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 189–199, 2010. 3.4.4, 10.2

[58] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 272–282, 2007. 3.3, 5.1, 5.2, 5.3, 5.4

[59] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–225, 1996. 2.1

[60] G. E. Blelloch, A. Gupta, I. Koutis, G. L. Miller, R. Peng, and K. Tangwongsan. Near linear-work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 13–22, 2011. 9.1

[61] G. E. Blelloch, I. Koutis, G. L. Miller, and K. Tangwongsan. Hierarchical diagonal blocking and precision reduction applied to combinatorial multigrid. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2010. 8.2, 8.4.1

[62] G. E. Blelloch and B. M. Maggs. Parallel algorithms. In *The Computer Science and Engineering Handbook*, pages 277–315. 1997. 2.3

[63] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and I/O efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 82–90, 2012. 6.4.6

[64] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel and Distributed Computing*, 37(1):55–69, 1996. Elsevier. 2.1

[65] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999. 1.1, 1.2, 2.2, 5.4

[66] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *USENIX Conference on Hot Topics in Parallelism (HotPar)*, 2009. 3.1

[67] R. L. Bocchino, S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 535–548, 2011. 3.1

[68] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *International World Wide Web Conference (WWW)*, pages 587–596, 2011. 8.4.1

[69] P. Boldi, M. Santini, and S. Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, Nov. 2008. 8.4

[70] P. Boldi, M. Santini, and S. Vigna. Permuting web and social graphs. *Internet Mathematics*, 6(3):257–283, 2009. 8.4.1

[71] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *International World Wide Web Conference (WWW)*, pages 595–602, 2004. 8.2

[72] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001. 7.4.2

[73] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, Apr. 1974. 1.2, 2.2

[74] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Computer Networks and ISDN Systems*, pages 107–117, 1998. 7.4.5

[75] N. Brunelle, G. Robins, and A. Shelat. Algorithms for compressed inputs. In *IEEE Data Compression Conference (DCC)*, page 478, 2013. 8.2

[76] Z. Budimlic, V. Cave, R. Raman, J. Shirako, S. Tasirlar, J. Zhao, and V. Sarkar. The design and implementation of the habanero-java parallel programming language. In *ACM International Conference Companion on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 185–186, 2011. 1.1, 2.1

[77] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *ACM Conference on Web Search and Data Mining (WSDM)*, pages 95–106, 2008. 8.2

[78] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, Nov. 2011. 7.1, 7.2.2

[79] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 721–733, 2011. 8.2

[80] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 253–262, 2006. 10.9

[81] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, HP Labs, 1994. 11.1, 12.1, 13.1, 14.1

[82] L. Bus and P. Tvrdik. A parallel algorithm for connected components on distributed memory machines. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 280–287. 2001. 9.1

[83] E. Caceres, H. Mongelli, C. Nishibe, and S. W. Song. Experimental results of a coarse-grained parallel algorithm for spanning tree and connected components. In *High Performance Computing & Simulation*, pages 631–637, 2010. 9.1

[84] N. J. Calkin and A. M. Frieze. Probabilistic analysis of a parallel algorithm for finding maximal independent sets. *Random Struct. Algorithms*, 1(1):39–50, 1990. 4.1, 4.4, 4.4

[85] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1):67–90, 1995. 3.4.4

[86] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *Symposium on Experimental Algorithms (SEA)*, pages 94–105, 2010. 13.3

[87] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining (SDM)*, pages 442–446, 2004. 3.5, 4.10.1, 5.6, 6.5, 7.5, 9.5

[88] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005. 1.1, 2.1

[89] G. Chen, S. Puglisi, and W. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, 2008. 13.1

[90] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 298–309, 1998. 3.2.3, 3.2.3

[91] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. Lui, and C. He. VENUS: Vertex-centric streamlined graph computation on a single PC. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1131–1142, 2015. 15.2

[92] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, Feb. 1985. 10.2, 10.9

[93] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 219–228, 2009. 8.2, 8.4.1

[94] F. Y. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 25(9):659–665, Sept. 1982. 9.1

[95] B. Ching. Optimizing lempel-ziv factorization for the GPU architecture. *Master's Thesis, California Polytechnic State University–San Luis Obispo*, 2014. 13.1

[96] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel SAH k-D tree construction. In *ACM Conference on High Performance Graphics (HPG)*, pages 77–86, 2010. 3.4.4

[97] K. Chong and T. Lam. Finding connected components in $O(\log n \log \log n)$ time on the EREW PRAM. *Journal of Algorithms*, 18(3):378–402, 1995. 9.1

[98] S. Chu and J. Cheng. Triangle listing in massive networks. *Trans. Knowl. Discov. Data*, 6(4):17:1–17:32, Dec. 2012. 10.1, 10.9

[99] D. R. Clark. *Compact Pat Trees*. PhD thesis, 1996. 14.6

[100] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *String Processing and Information Retrieval (SPIRE)*, pages 176–187, 2008. 14.2, 14.5

[101] F. Claude and G. Navarro. The wavelet matrix. In *String Processing and Information Retrieval (SPIRE)*, pages 167–179. 2012. 14.1, 14.7

[102] F. Claude, P. K. Nicholson, and D. Seco. Space efficient wavelet tree construction. In *String Processing and Information Retrieval (SPIRE)*, pages 185–196, 2011. 14.3

[103] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, December 1997. 7.4.3

[104] J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Eng.*, 11(4):29–41, July 2009. 10.1, 10.7.2, 10.9

[105] R. Cole, P. N. Klein, and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 243–250, 1996. 9.1

[106] R. Cole and U. Vishkin. Approximate parallel scheduling. II. applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, 1991. 9.1

[107] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 169–178, 1989. 1.2

[108] M. Comin and M. Farreras. Efficient parallel construction of suffix trees for genomes larger than main memory. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 211–216, 2013. (document), 1.3(b), 11.5, 11.5, 11.9

[109] G. Cong and D. A. Bader. An empirical analysis of parallel random permutation algorithms on SMPs. In *International Conference on Parallel and Distributed Computing and Systems*, pages 27–34, 2005. 4.1, 4.10.2

[110] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Inf. Control*, 64(1–3):2–22, March 1985. 4.1, 1

[111] D. Coppersmith, P. Raghavan, and M. Tompa. Parallel graph algorithms that are efficient on average. *Inf. Comput.*, 81(3):318–333, June 1989. 4.1, 4.4, 4.4

[112] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. 3.3, 3.4.4, 4.5, 4.5, 6.4.4, 7.4.3, 7.4.6, 7.5

[113] D. G. Corneil, F. F. Dragan, M. Habib, and C. Paul. Diameter determination on restricted graph families. *Discrete Applied Mathematics*, 113(2–3):143–166, 2001. 7.4.3

[114] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, pages 75–80, 2008. 13.1, 13.3, 13.3, 13.4

[115] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń. LPF computation revisited. In *Combinatorial Algorithms*, pages 158–169. 2009. 13.1, 13.3

[116] M. Crochemore, L. Ilie, and W. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In *IEEE Data Compression Conference (DCC)*, pages 482–488, 2008. 13.1, 13.2, 13.3

[117] M. Crochemore and W. Rytter. Efficient parallel algorithms to test square-freeness and factorize strings. *Inf. Process. Lett.*, pages 57–60, 1991. 13.1, 13.5

[118] H. Cui, J. Simsa, Y. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 388–405, 2013. 3.1

[119] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 337–351, 2011. 3.1

[120] H. Cui, J. Wu, C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 207–221, 2010. 3.1

[121] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998. 1.3

[122] A. Czumaj, P. Kanarek, M. Kutylowski, and K. Lorys. Fast generation of random permutations via networks simulation. *Algorithmica*, pages 2–20, 1998. 4.1

[123] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48, 2013. 1.3

[124] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, Nov. 2011. 8.4

[125] S. De Agostino. P-complete problems in data compression. *Theor. Comp. Sci.*, pages 181–186, 1994. 13.1

[126] S. De Agostino. Lempel-Ziv data compression on parallel and distributed systems. *Algorithms*, 4(3):183–199, 2011. 13.1

[127] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008. 2.6.4, 3.4.4, 13.3

[128] F. Dehne and S. W. Song. Randomized parallel list ranking for distributed memory multiprocessors. *International Journal of Parallel Programming*, 25(1):1–16, 1997. 4.10.2

[129] A. Delcher, A. Phillippy, J. Carlton, and S. Salzberg. Fast algorithms for large-scale genome alignment and comparision. *Nucleic Acids Research*, 30(11):2478–2483, 2002. 11.1, 11.5, 11.5

[130] G. Della-Libera and N. Shavit. Reactive diffracting trees. *J. Parallel Distrib. Comput.*, pages 853–890, 2000. 6.1

[131] R. Dementiev. Algorithm engineering for large data sets. *PhD Thesis, Saarland University*, 2006. 10.1, 10.9

[132] R. H. Dennard, F. Gaensslen, H.-N. Yu, L. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid State Circuits*, 9(5):256–268, 1974. 1

[133] M. Deo and S. Keely. Parallel suffix array and least common prefix for the GPU. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 197–206, 2013. 12.1, 12.1, 12.2, 12.3, 1

[134] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, 2009. 3.1

[135] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A relaxed consistency deterministic computer. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 67–78, 2011. 3.1

[136] L. Devroye. A note on the height of binary search trees. *J. ACM*, 33(3):489–498, 1986. 4.6.1

[137] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD 123, Dept. of Mathematics, Technological U., Eindhoven, 1965. 3.2.1

[138] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 305–315, 2014. 15.2

[139] R. Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, 1964. 4.1, 4.6

[140] J.-P. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *PNAS*, 99(9):5825–5829, 2002. 10.1

[141] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader. Massive streaming data analytics: A case study with clustering coefficients. In *Workshop on Multithreaded Architectures and Applications (MTAAP)*, pages 1–8, 2010. 10.9

[142] J. A. Edwards and U. Vishkin. Parallel algorithms for Burrows-Wheeler compression and decompression. *Theor. Comput. Sci.*, 525:10–22, Mar. 2014. 14.7

[143] C. S. Ellis. Concurrency in linear hashing. *ACM Trans. Database Syst.*, 12(2):195–217, 1987. 5.2

[144] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, 1988. 3.2.2

[145] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: compact and concurrent MemCache with dumber caching and smarter hashing. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 371–384, 2013. 5.2

[146] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker. Active memory operations. In *ACM International Conference on Supercomputing (ICS)*, pages 232–241, 2007. 6.1

[147] M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression (extended abstract). In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 244–253, 1995. 13.1, 13.3, 13.5

[148] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 550–561, 1996. 11.1

[149] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–266, 2012. 6.1

[150] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):49–57, Feb. 2010. 6.1

[151] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), May 2007. 14.1, 14.7

[152] J.-A. Ferrez, K. Fukuda, and T. Liebling. Parallel computation of the diameter of a graph. In *High Performance Computing Systems and Applications*, pages 283–296, 1998. 7.4.3

[153] J. Fischer. Inducing the LCP-array. In *International Conference on Algorithms and Data Structures (WADS)*, pages 374–385. 2011. 12.1, 12.4.1

[154] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Combinatorial Pattern Matching (CPM)*, pages 36–48, 2006. 11.5

[155] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Trans. Algorithms*, 2(4):611–639, Oct. 2006. 14.1, 14.3, 14.7

[156] L. Freeman. A set of measures of centrality based upon betweenness. *Sociometry*, 40(1):35–41, 1977. 7.4.2

[157] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 285–298, 1999. 1.3, 10.1, 10.2

[158] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998. 1.1, 1.2, 2.1

[159] Z. Fu, B. B. Thompson, and M. Personick. Mapgraph: A high level API for fast development of high performance graph analytics on GPUs. In *Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2014. 7.2.2

[160] J. Fuentes-Sepulveda, E. Elejalde, L. Ferres, and D. Seco. Efficient wavelet tree construction and querying for multicore architectures. In *Symposium on Experimental Algorithms (SEA)*, pages 150–161, 2014. 14.1, 14.3, 14.4.1, 14.4.2, 14.5

[161] H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984. 11.5, 13.4

[162] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, 2005. 5.2, 5.6

[163] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, Dec. 1991. 9.1

[164] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *Algorithms Engineering and Experiments (ALENEX)*, pages 90–100, 2008. 7.4.2

[165] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 15–26, 1990. 3.1

[166] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 345–354, 2012. 7.2.2

[167] A. Ghoting and K. Makarychev. Indexing genomic sequences on the IBM Blue Gene. In *ACM/IEEE International Conference for High Performance Computing Networking, Storage and Analysis (SC)*, pages 1–11, 2009. 11.5

[168] P. B. Gibbons. A more practical PRAM model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 158–168, 1989. 1.2, 3.1

[169] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. *Journal of Computer and System Sciences*, 53(3):417–442, 1996. 4.1, 4.6.2, 4.10.2

[170] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. *Theoretical Computer Science*, 196(1-2):3–29, 1998. 1.2

[171] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. Comput.*, 28(2):3–29, 1999. 1.2, 4.6.2

[172] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software: Practice and Experience*, 33(11):1035–1049, 2003. 11.5

[173] J. Gil. Fast load balancing on a PRAM. In *Symposium on Parallel and Distributed Processing*, pages 10–17, 1991. 4.1, 4.10.2

[174] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991. (document), 1.1, 4.1, 4.3, 9.3

[175] J. R. Gilbert, S. Reinhardt, and V. B. Shah. A unified framework for numerical and combinatorial computing. *Computing in Sciences and Engineering*, 10(2):20–25, Mar/Apr 2008. 7.2.2

[176] Giraph. `http://giraph.apache.org`, 2012. 7.1, 7.2.2

[177] S. Goddard, S. Kumar, and J. F. Prins. Connected components algorithms for mesh-connected parallel computers. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge*, pages 43–58, 1995. 9.1

[178] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014. 12.4, 14.5

[179] S. Gog and E. Ohlebusch. Fast and lightweight LCP-array construction algorithms. In *Algorithm Engineering and Experiments (ALENEX)*, pages 25–34, 2011. 12.1, 12.4

[180] S. Gog and E. Ohlebusch. Compressed suffix trees: Efficient computation and storage of LCP-values. *J. Exp. Algorithmics*, 18(2.1):2.1:2.1–2.1:2.31, May 2013. 11.5, 12.1

[181] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2013. 14.6

[182] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. In *ACM Symposium on Theory of Computing (STOC)*, pages 315–324, 1987. 4.1

[183] M. Goldberg and T. Spencer. Constructing a maximal independent set in parallel. *SIAM Journal on Discrete Mathematics*, 2(3):322–328, August 1989. 4.1

[184] M. Goldberg and T. Spencer. A new parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 18(2):419–427, April 1989. 4.1

[185] M. K. Goldberg. Parallel algorithms for three graph problems. *Congressus Numerantium*, 54:111–121, 1986. 4.1

[186] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 17–30, 2012. 1.1, 1.3(a), 7.1, 7.1, 7.2.2, 7.5, 10.1, 10.6.2, 10.7.2, 10.9

[187] K. Goto and H. Bannai. Simpler and faster Lempel Ziv factorization. In *IEEE Data Compression Conference (DCC)*, pages 133–142, 2013. 1

[188] A. Gottlieb, R. Grishman, C. P. Kruskal, C. P. Mcauliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer—designing an MIMD parallel computer. *IEEE Trans. Comput.*, Feb. 1983. 6.1

[189] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Language and Systems*, 5(2):164–189, Apr. 1983. 6.1

[190] Graph500. `http://www.graph500.org`, 2012. 7.5

[191] O. Green and D. A. Bader. Faster clustering coefficient using vertex covers. In *ASE International Conference on Social Computing (SocialCom)*, pages 321–330, 2013. 10.9

[192] O. Green, L. M. Munguia, and D. A. Bader. Load balanced clustering coefficients. In *Workshop on Parallel Programming for Analytics Applications*, pages 3–10, 2014. 1.3(a), 10.7.2, 10.9

[193] O. Green, P. Yalamanchili, and L. M. Munguia. Fast triangle counting on the GPU. In *Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2015. 10.9

[194] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987. 3.4.4

[195] R. Greenlaw, J. H. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, USA, Apr. 1995. 4.1

[196] M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 260–269, 2002. 5.2

[197] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Workshop on Parallel Object-Oriented Scientific Computing*, 2005. 1.1, 7.1, 7.2.2

[198] J. Greiner. A comparison of parallel algorithms for connected components. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 16–25, 1994. 9.1

[199] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003. 14.1, 14.2

[200] R. Grossi, J. S. Vitter, and B. Xu. Wavelet trees: From theory to practice. In *International Conference on Data Compression, Communications and Processing (CCP)*, pages 210–221, 2011. 14.5

[201] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997. 11.1, 11.5, 12.1

[202] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, Dec. 2004. 13.1

[203] J. Gustedt. Randomized permutations in a coarse grained parallel environment. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 248–249, 2003. 4.1

[204] J. Gustedt. Engineering parallel in-place random generation of integer permutations. In *International Workshop on Experimental Algorithmics (WEA)*, pages 129–141, 2008. 4.1

[205] T. Hagerup. Fast parallel generation of random permutations. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 405–416. Springer, 1991. 4.1

[206] T. Hagerup and R. Raman. Waste makes haste: tight bounds for loose parallel sorting. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 628–637, 1992. 12.2, 12.3

[207] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Inf. Process. Lett.*, 33(4):181–185, Dec. 1989. 11.3

[208] S. Halperin and U. Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. Syst. Sci.*, 53(3):395–416, 1996. 9.1

[209] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests. In *J. Algorithms*, pages 1740–1759, 2000. 9.1

[210] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985. 3.1

[211] S. Hambrusch and L. TeWinkel. A study of connected component labeling algorithms on the MPP. In *International Conference on Supercomputing (ICS)*, pages 477–483, 1988. 9.1

[212] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 77–85, 2013. 15.2

[213] Y. Han and R. A. Wagner. An efficient and fast parallel-connected component algorithm. *J. ACM*, 37(3):626–642, July 1990. 9.1

[214] D. Hannah, C. Macdonald, and I. Ounis. Analysis of link graph compression techniques. In *European Conference on Advances in Information Retrieval*, pages 596–601, 2008. 8.2

[215] R. Hariharan. Optimal parallel suffix tree construction. In *ACM Symposium on Theory of Computing (STOC)*, pages 290–299, 1994. 11.1

[216] T. Harris, J. Larus, and R. Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010. 1.1

[217] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. Rocke. Characterizing history independent data structures. *Algorithmica*, pages 57–74, 2005. 5.2

[218] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 166–177, 2014. 4.3

[219] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 3–12, 2011. 3.1, 3.4.3, 3.5

[220] K. A. Hawick, A. Leist, and D. P. Playne. Parallel graph component labelling with GPUs and CUDA. *Parallel Comput.*, 36(12):655–678, Dec. 2010. 9.1

[221] D. Helman and J. JaJa. Designing practical efficient algorithms for symmetric multiprocessors. *Algorithm Engineering and Experimentation*, pages 37–56, 1999. 4.10.2

[222] D. Helman and J. JaJa. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265–278, 2001. 4.10.2

[223] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, 2010. 6.1

[224] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 2008. 3.1, 3.2.2, 3.2.3

[225] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012. 1.1, 5.2, 5.4

[226] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing (DISC)*, pages 350–364, 2008. 5.1, 5.2, 5.6

[227] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990. 3.1, 3.2.3

[228] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, Aug. 1979. 9.1

[229] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 349–362, 2012. 1.1, 7.2.2

[230] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? Free will to choose. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 333–334, 2011. 3.1

[231] M. Hsu and W.-P. Yang. Concurrent operations in extendible hashing. In *International Conference on Very Large Data Bases (VLDB)*, pages 241–247, 1986. 5.2

[232] T.-S. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components in graphs. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge*, pages 23–41, 1997. 9.1

[233] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. In *ACM SIGMOD Conference on Management of Data*, pages 325–336, 2013. 10.1, 10.4.2, 10.9

[234] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, pages 1098–1101, September 1952. 13.1

[235] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble. DDOS: taming nondeterminism in distributed systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 499–508, 2013. 3.1

[236] C. Iliopoulos and W. Rytter. On parallel transformations of suffix arrays into suffix trees. In *Australasian Workshop on Combinatorial Algorithms (AWOCA)*, 2004. 11.1, 11.3, 11.4, 11.5

[237] Intel Threading Building Blocks. `https://www.threadingbuildingblocks.org`. 1.1, 2.1

[238] A. Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):77–80, February 1986. 4.1

[239] A. Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):57–60, February 1986. 4.1

[240] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. In *ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 1977. 10.9

[241] K. Iwama and Y. Kambayashi. A simpler parallel algorithm for graph connectivity. *J. Algorithms*, 16(2):190–217, Mar. 1994. 9.1

[242] G. J. Jacobson. *Succinct Static Data Structures*. PhD thesis, 1988. 14.6

[243] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992. 1.2, 2.2, 4.1, 4.6.2, 4.6.2, 4.7, 4.8, 4.10.2, 6.2, 12.2, 12.3, 12.4.1, 13.1, 13.3, 13.3, 13.4, 14.7

[244] Java Fork-Join. `http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html`. 2.1

[245] M. Jha, C. Seshadhri, and A. Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 589–597, 2013. 10.9

[246] D. B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for the CREW PRAM. *Journal of Computer and System Sciences*, 54(2):227–242, 1997. 9.1

[247] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 154–165, 2014. 1.1, 1.6, 3.1, 7.2.2

[248] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. GBASE: an efficient analysis platform for large graphs. *International Conference on Very Large Data Bases (VLDB)*, 21(5):637–650, 2012. 8.2

[249] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. HADI: Mining radii of large graphs. *ACM Trans. Knowl. Discov. Data*, 5(2):8:1–8:24, Feb. 2011. 7.4.3, 7.5

[250] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011. 7.1, 7.2.2, 7.5, 9.1, 9.5

[251] C. Karande, K. Chellapilla, and R. Andersen. Speeding up algorithms on compressed web graphs. In *ACM Conference on Web Search and Data Mining (WSDM)*, pages 272–281, 2009. 8.2

[252] D. R. Karger, N. Nisan, and M. Parnas. Fast connected components algorithms for the EREW PRAM. *SIAM J. Comput.*, 28(3):1021–1034, Feb. 1999. 9.1

[253] J. Kärkkäinen and D. Kempa. LCP array construction in external memory. In *Symposium on Experimental Algorithms (SEA)*, pages 412–423, 2014. 12.1

[254] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching (CPM)*, pages 189–200, 2013. 1

[255] J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Combinatorial Pattern Matching (CPM)*, pages 181–192. 2009. 11.5, 12.1, 12.2, 12.2, 12.2, 12.3, 12.4

[256] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 943–955, 2003. 3.4.4, 11.1, 11.5, 12.1, 12.2, 12.2, 12.3, 12.3, 13.1, 13.3, 13.3, 13.4

[257] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, Nov. 2006. 11.1, 11.5, 12.1

[258] S. Karlin, G. Ghandour, F. Ost, S. Tavare, and L. J. Korn. New approaches for computer analysis of nucleic acid sequences. *Natl. Acad. Sci. USA*, 80:5660–5664, 1993. 12.3

[259] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. MIT Press, 1990. 4.1, 4.7

[260] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *J. ACM*, 32(4):762–773, Oct. 1985. 4.1

[261] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998. 5.5.4, 8.4, 8.4.1

[262] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching (CPM)*, pages 181–192, 2001. 11.5

[263] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching (CPM)*, pages 181–192. 2001. 12.1, 12.2, 12.3

[264] D. Kempa and S. J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Algorithms Engineering and Experiments (ALENEX)*, pages 103–112, 2013. 1

[265] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: Vertex-centric graph processing on GPUs. In *International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, pages 239–252, 2014. 7.2.2

[266] D. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching (CPM)*, pages 186–199. 2003. 12.1

[267] E. Kim and M.-S. Kim. Performance analysis of cache-conscious hashing techniques for multi-core CPUs. *International Journal of Control and Automation*, 6(2):121–134, Apr. 2013. 5.2

[268] J. Kim, W.-S. Han, S. Lee, K. Park, and H. Yu. OPT: A new framework for overlapped and parallel triangulation in large-scale graphs. In *ACM SIGMOD Conference on Management of Data*, pages 637–648, 2014. 10.1, 10.9

[269] S. T. Klein and Y. Wiseman. Parallel Lempel Ziv coding. *Discrete Appl. Math.*, 146(2):180–191, 2005. 13.1

[270] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969. 4.1, 4.6, 4.10.2

[271] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005. 12.1

[272] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task. Counting triangles in massive graphs with MapReduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014. 10.7.3, 10.9

[273] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics*, 8(1-2):161–185, 2012. 10.9

[274] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 596–604, 1999. 13.1

[275] V. Koubek and J. Krsnakova. Parallel algorithms for connected components in a graph. In *Fundamentals of Computation Theory*, pages 208–217. 1985. 9.1

[276] K. Kourtis, G. I. Goumas, and N. Koziris. Exploiting compression opportunities to improve SpMxV performance on shared memory systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(3):16:1–16:31, Dec. 2010. 8.2, 8.3.2

[277] K. Kourtis, V. Karakasis, G. I. Goumas, and N. Koziris. CSX: an extended compression format for spmv on shared memory systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 247–256, 2011. 8.2

[278] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge*, pages 1–21, 1994. 9.1

[279] C. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5(1-4):43–64, 1990. 9.1

[280] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 542–555, 2011. 3.1, 3.3

[281] F. Kulla and P. Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007. 11.5

[282] V. Kumar. Concurrent operations on extendible hashing and its performance. *Commun. ACM*, 33(6):681–694, 1990. 5.2

[283] L. Kuper, A. Todd, S. Tobin-Hochstadt, and R. R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with LVish. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 2–14, 2014. 3.1

[284] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 257–270, 2014. 3.1

[285] S. Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999. 1.2, 11.5, 11.5

[286] S. Kurtz and C. Schleiermacher. Reputer: Fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, 1999. 11.1

[287] K. Kutzkov and R. Pagh. Triangle counting in dynamic graph streams. In *Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 306–318, 2014. 10.9

[288] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *International World Wide Web Conference (WWW)*, pages 591–600, 2010. 1.3(a), 7.5, 7.5, 8.3.4, 10.7

[289] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012. 7.2.2, 9.1, 9.5, 10.1, 10.9, 15.2

[290] A. Kyrola, J. Shun, and G. E. Blelloch. Beyond synchronous computation: New techniques for external memory graph algorithms. In *Symposium on Experimental Algorithms (SEA)*, pages 123–137, 2014. 9.1, 9.5, 15.2

[291] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, pages 690–691, Sept. 1979. 1.1

[292] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, Nov. 2008. 1, 10.3, 1, 10.4.2, 10.9

[293] D. Lea. Hash table `util.concurrent.concurrenthashmap` in `java.util.concurrent` the Java Concurrency Package. 5.1, 5.2, 5.6

[294] C. E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3):244–257, 2010. 1.1, 7.1, 7.1, 7.5, 12.4

[295] C. E. Leiserson and I. B. Mirman. How to survive the multicore software revolution (or at least survive the hype). Cilk Arts, 2008. 1

[296] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 303–314, 2010. 3.4.4, 3.5, 9.4

[297] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 193–204, 2012. 3.2.2

[298] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014. 1.3(a), 8.4, 9.5, 10.7

[299] X. Li, D. G. Anderson, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *European Conference on Computer Systems (EuroSys)*, pages 27:1–27:14, 2014. 5.2, 5.6

[300] W. Lim, A. Agrawal, and L. Nekludova. A fast parallel algorithm for labeling connected components in image arrays. In *Tech. Report NA86-2, Thinking Machines Corporation*, 1986. 9.1

[301] Y. Lim, U. Kang, and C. Faloutsos. SlashBurn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(12):3077–3089, 2014. 8.2

[302] N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993. 9.1

[303] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 327–336, 2011. 3.1

[304] Y. Liu, K. Zhang, and M. Spear. Dynamic-sized nonblocking hash tables. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 242–251, 2014. 5.2

[305] F. A. Louza, G. P. Telles, and C. D. D. A. Ciferri. External memory generalized suffix and LCP arrays construction. In *Combinatorial Pattern Matching (CPM)*, pages 201–210. 2013. 12.1

[306] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010. 1.1, 7.1, 7.2.2

[307] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *International Conference on Very Large Data Bases (VLDB)*, 5(8):716–727, Apr. 2012. 7.1, 7.2.2, 7.4.5

[308] K. Lu, X. Zhou, X. Wang, T. Bergan, and C. Chen. An efficient and flexible deterministic framework for multithreaded programs. *J. Comput. Sci. Technol.*, 30(1):42–56, 2015. 3.1

[309] L. Lu and M. L. Scott. Toward a formal semantic framework for deterministic parallel programming. In *International Symposium on Distributed Computing (DISC)*, pages 460–474, 2011. 3.2.2

[310] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1055, November 1986. 4.1, 4.4

[311] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *SIAM International Conference on Data Mining (SDM)*, pages 930–941, 2012. 7.1, 7.2.2, 7.4.2, 7.5

[312] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–165, 1990. Springer. 3.4.4

[313] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *IEEE International Conference on Data Engineering (ICDE)*, pages 363–374, 2015. 15.2

[314] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Algorithms Engineering and Experiments (ALENEX)*, pages 23–35, 2007. 7.4.6

[315] C. Magnien, M. Latapy, and M. Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *J. Exp. Algorithmics*, 13:10:1.10–10:1.9, February 2009. 7.4.3, 7.4.3

[316] V. Makinen and G. Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007. 14.1, 14.2, 14.5

[317] C. Makris. Wavelet trees: A survey. *Comput. Sci. Inf. Syst.*, 9(2):585–625, 2012. 14.1

[318] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD Conference on Management of Data*, pages 135–146, 2010. 7.1, 1, 7.2.2, 7.5

[319] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. 2.6.3, 11.5, 12.1, 12.3

[320] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. ERA: Efficient serial and parallel suffix tree construction for very long strings. *International Conference on Very Large Data Bases (VLDB)*, 5(1):49–60, Sept. 2011. (document), 1.3(b), 11.5, 11.5, 11.9

[321] G. Manzini. Two space saving tricks for linear time LCP array computation. In *Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 372–383. 2004. 12.1, 12.2, 12.3

[322] S. Marlow, R. Newton, and S. L. P. Jones. A monad for deterministic parallelism. In *ACM SIGPLAN Symposium on Haskell*, pages 71–82, 2011. 3.1

[323] D. R. Martin and R. C. Davis. A scalable non-blocking concurrent hash table implementation with incremental rehashing. Unpublished manuscript, 1997. 5.2

[324] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, 1991. 9.3, 10.4.2, 11.3

[325] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. 11.1, 11.5, 13.1

[326] F. McSherry. A uniform approach to accelerated pagerank computation. In *International Conference on World Wide Web (WWW)*, pages 575–582, 2005. 7.4.5

[327] C. Meek, J. M. Patel, and S. Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *International Conference on Very Large Data Bases (VLDB)*, pages 910–921, 2003. 11.1

[328] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, pages 21–65, Feb. 1991. 6.1

[329] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 106–113, 1991. 6.1

[330] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–278, 1991. 6.1

[331] B. Menegola. An external memory algorithm for listing triangles. *Tech. report, Universidade Federal do Rio Grande do Sul*, 2010. 10.1, 10.9

[332] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003. 7.4.6

[333] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 73–82, 2002. 5.2

[334] G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decomposition using random shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 196–203, 2013. 9.1, 9.2, 9.3, 9.3, 9.4

[335] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 478–489, 1985. 4.1, 4.8, 4.10.2

[336] G. L. Miller and J. H. Reif. Parallel tree contraction part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991. 4.1

[337] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, pages 82–85, 1965. 1

[338] Y. Mori. libdivsufsort: A lightweight suffix-sorting library. `http://code.google.com/p/libdivsufsort`, 2010. 11.5, 12.4.1

[339] Y. Mori. sais: An implementation of the induced sorting algorithm. `http://sites.google.com/site/yuta256/sais`, 2010. 11.5, 12.4.1

[340] D. R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968. 2.6.3

[341] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. 4.6.1, 4.6.2, 6.3.2

[342] J. I. Munro, Y. Nekrich, and J. S. Vitter. Fast construction of wavelet trees. In *String Processing and Information Retrieval (SPIRE)*, pages 101–110, 2014. 14.3, 14.6

[343] M. Naor. String matching with preprocessing of text and pattern. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 739–750, 1991. 13.1, 13.5

[344] M. Naor and V. Teague. Anti-persistence: history independent data structures. In *ACM Symposium on Theory of Computing*, pages 492–501, 2001. 5.2

[345] D. Nath and S. N. Maheshwari. Parallel algorithms for the connected components and minimal spanning tree problems. *Inf. Process. Lett.*, 14(1):7–11, 1982. 9.1

[346] G. Navarro. Wavelet trees for all. In *Combinatorial Pattern Matching (CPM)*, pages 2–26. 2012. 14.1

[347] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), Apr. 2007. 11.5

[348] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992. 3.2.2

[349] M. E. J. Newman. The structure and function of complex networks. *SIAM REVIEW*, 45:167–256, 2003. 10.1

[350] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, Feb. 2014. 10.1

[351] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013. 1.1, 1.6, 7.2.2, 9.1, 9.5

[352] D. Nguyen, A. Lenharth, and K. Pingali. Deterministic galois: On-demand, portable and parameterless. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 499–512, 2014. 3.1

[353] N. Nguyen and P. Tsigas. Lock-free cuckoo hashing. In *IEEE International Conference on Distributed Computing Systems*, pages 627–636, 2014. 5.2, 5.6

[354] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992. 1.6, 4.1, 4.9, 4.9

[355] N. Nisan, E. Szemeredi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 24–29, 1992. 9.1

[356] N. Nishimura. Asynchronous shared memory parallel computation. In *ACM Symposium on Parallelism Algorithms and Architectures (SPAA)*, pages 76–84, 1990. 1.2

[357] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *IEEE Data Compression Conference (DCC)*, pages 193–202, 2009. 12.1

[358] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *Combinatorial Pattern Matching (CPM)*, pages 15–26, 2011. 13.1, 13.5

[359] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, 2009. 3.1

[360] Openmp. `http://www.openmp.org`. 1.1, 2.1

[361] M. Ortmann and U. Brandes. Triangle listing algorithms: Back from the diversion. In *Algorithms Engineering and Experiments (ALENEX)*, pages 1–8, 2014. 1, 10.7.2, 10.9

[362] A. Ozsoy and M. Swany. CULZSS: LZSS lossless data compression on CUDA. In *IEEE International Conference on Cluster Computing*, pages 403–411, 2011. 13.1

[363] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. 5.2

[364] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 224–233, 2014. 10.1, 1, 10.6.1, 10.7.2, 10.8, 10.9

[365] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a MapReduce implementation. *Inf. Process. Lett.*, 112(7):277–281, Mar. 2012. 10.5, 10.5, 10.7.3, 10.7.3, 10.9

[366] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: a fast and scalable tool for data mining in massive graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 81–90, 2002. 7.4.3

[367] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *International Symposium on Computer Architecture (ISCA)*, pages 348–354, 1984. 6.3

[368] H.-M. Park and C.-W. Chung. An efficient MapReduce algorithm for counting triangles in a very large graph. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 539–548, 2013. 10.1, 10.7.2, 10.9

[369] H.-M. Park, F. Silvestri, U. Kang, and R. Pagh. MapReduce triangle enumeration with guarantees. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 1739–1748, 2014. 10.7.2, 10.9

[370] J. Patel, A. Khokhar, and L. Jamieson. Scalable parallel implementations of list ranking on fine-grained machines. *IEEE Transactions on Parallel and Distributed Systems*, pages 1006–1018, 1997. 4.10.2

[371] S. S. Patil. Closure properties of interconnections of determinate systems. In *Record of the Project MAC conference on concurrent systems and parallel computation*, pages 107–116. 1970. 3.1

[372] M. Patwary, P. Refsnes, and F. Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 827–835, 2012. (document), 9.1, 9.1, 9.5, 9.2, 9.5

[373] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Counting and sampling triangles from a graph stream. *International Conference on Very Large Data Bases (VLDB)*, 6(14):1870–1881, 2013. 10.9

[374] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. 1996. 8.4.1

[375] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6):1879–1895, 2002. 9.1

[376] C. A. Phillips. Parallel graph contraction. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 148–157, 1989. 9.1, 9.3

[377] B. Phoophakdee and M. Zaki. Genome-scale disk-based suffix tree indexing. In *ACM SIGMOD International Conference on Management of Data*, pages 833–844, 2007. 11.1

[378] B. Phoophakdee and M. Zaki. Trellis+: An effective approach for indexing genome-scale sequences using suffix trees. In *Pacific Symposium on Biocomputing (PSB)*, volume 13, pages 90–101, 2008. 11.1

[379] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 12–25, 2011. 1.1, 3.5, 7.2.2

[380] C. K. Poon and V. Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 212–222, 1997. 9.1

[381] C. K. Poon and H. Yuan. A faster CREW PRAM algorithm for computing cartesian trees. In *International Conference on Algorithms and Complexity*, pages 336–344, 2013. 3

[382] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX Annual Technical Conference (ATC)*, pages 41–52, 2012. 7.1, 7.2.2

[383] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, 2011. 3.1, 3.4.3, 3.4.4

[384] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), July 2007. 11.1, 11.5, 12.1, 12.4.1

[385] S. J. Puglisi and A. Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 124–135. 2008. 12.1

[386] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. In *International Symposium on Distributed Computing (DISC)*, pages 108–121, 2005. 5.2

[387] M. Rahman and M. Al Hasan. Approximate triangle counting algorithms on multi-cores. In *IEEE International Conference on Big Data*, pages 127–133, 2013. 10.7.3, 10.9

[388] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989. 4.1, 4.6.2, 4.10.2, 11.1, 12.2, 12.3, 14.1, 14.4.2

[389] R. Raman. The power of collision: Randomized parallel algorithms for chaining and integer sorting. In *Foundations of Software Technology and Theoretical Computer Science*, pages 161–175, 1990. 14.1, 14.4.2

[390] K. H. Randall, R. Stata, J. L. Wiener, and R. G. Wickremesinghe. The link database: Fast access to graphs of the web. In *IEEE Data Compression Conference (DCC)*, pages 122–131, 2002. 8.2, 8.4.1

[391] M. S. Rehman, K. Kothapalli, and P. J. Narayanan. Fast and scalable list ranking on the GPU. In *ACM International Conference on Supercomputing (ICS)*, pages 235–243, 2009. 4.10.2

[392] M. Reid-Miller. List ranking and list scan on the CRAY C90. *J. Comput. Syst. Sci.*, 53(3):344–356, 1996. 4.10.2

[393] M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. 1993. 11.3, 11.4.1

[394] J. H. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. *TR-08-85, Harvard University*, 1985. 9.1, 9.3

[395] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993. 4.1, 4.7, 4.8

[396] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–991, 1997. 3.1, 3.2.3, 3.4.3

[397] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM J. Res. Dev.*, pages 149–162, 1979. 13.1

[398] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. ACM*, 28(1):16–24, Jan. 1981. 13.1

[399] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: edge-centric graph processing using streaming partitions. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488, 2013. 1.1, 1.6, 7.2.2, 15.2

[400] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *International Symposium on Computer Architecture (ISCA)*, pages 340–347, 1984. 6.1

[401] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002. 12.2

[402] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, Dec. 2007. 11.5

[403] I. Safro and B. Temkin. Multiscale approach for the network compression-friendly ordering. *Journal of Discrete Algorithms*, 9(2):190 – 202, 2011. 8.4.1

[404] S. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *ACM Symposium on Theory of Computing (STOC)*, pages 300–309, 1994. 11.1

[405] S. Salihoglu and J. Widom. GPS: A graph processing system. Technical Report InfoLab 1039, Stanford University, 2012. 7.1, 7.2.2, 7.5

[406] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Inf. Process. Lett.*, 67(6):305–309, 1998. 4.10.2

[407] T. Schank. Algorithmic aspects of triangle-based network analysis. *PhD Thesis, Universitat Karlsruhe*, 2007. 10.9

[408] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International Workshop on Experimental Algorithmics (WEA)*, pages 606–609, 2005. 10.9

[409] H. Seo, J. Kim, and M.-S. Kim. Gstream: A graph streaming processing method for large-scale graphs on gpus. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 253–254, 2015. 7.2.2

[410] C. Seshadhri, A. Pinar, N. Durak, and T. G. Kolda. The importance of directed triangles with reciprocity: patterns and algorithms. *CoRR*, abs/1302.6220, 2013. 10.6.2

[411] C. Seshadri, A. Pinar, and T. G. Kolda. Triadic measures on graphs: The power of wedge sampling. In *SIAM International Conference on Data Mining (SDM)*, pages 10–18, 2013. 10.5, 10.9

[412] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006. 5.2

[413] N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, Nov. 1996. 6.1

[414] N. Shavit and A. Zemach. Combining funnels: a dynamic approach to software combining. *J. Parallel Distrib. Comput.*, pages 1355–1387, Nov. 2000. 6.1

[415] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982. 9.1

[416] J. Shun. Fast parallel computation of longest common prefixes. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 387–398, 2014. 12.4

[417] J. Shun. An evaluation of parallel eccentricity estimation algorithms on real-world graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2015. 7.4.3

[418] J. Shun. Parallel wavelet tree construction. In *IEEE Data Compression Conference (DCC)*, pages 63–72, 2015. 14.1

[419] J. Shun. Parallel wavelet tree construction. *CoRR*, abs/1407.8142, 2015. 14.5, 14.7

[420] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 135–146, 2013. 3.5, 6.4.6, 7.2.1, 7.2.2, 7.5, 9.1, 9.1, 9.4, 9.5

[421] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014. 1.1, 5.2, 5.6, 9.4

[422] J. Shun and G. E. Blelloch. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Transactions on Parallel Computing*, 1(1):8:1–8:20, Oct. 2014. 1.2, 12.1, 12.4.1

[423] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 152–163, 2013. 1.1, 5.6, 6.3.1

[424] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012. 1.3(a), 1.4, 7, 7.1, 7.5, 9.1, 9.1, 14.5

[425] J. Shun, L. Dhulipala, and G. E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 143–153, 2014. 9.5

[426] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *IEEE Data Compression Conference (DCC)*, pages 403–412, 2015. 1.3

[427] J. Shun, Y. Gu, G. Blelloch, J. Fineman, and P. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 431–448, 2015. 1.1, 4.3, 4.10.2

[428] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *IEEE International Conference on Data Engineering (ICDE)*, pages 149–160, 2015. 1.2

[429] J. Shun and F. Zhao. Practical parallel Lempel-Ziv factorization. In *IEEE Data Compression Conference (DCC)*, pages 123–132, 2013. 11.5, 12.4.1, 13.1, 13.4, 13.5

[430] J. F. Sibeyn. Better trade-offs for parallel list ranking. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 221–230, 1997. 4.10.2

[431] H. V. Simhadri. Program-centric cost models for locality and parallelism. *PhD Thesis, Carnegie Mellon University*, 2013. 1.3, 10.1, 1, 10.2

[432] J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *Euro-Par*, pages 682–694, 2007. 1.1, 3.5

[433] J. Sirén. Sampled longest common prefix array. In *Combinatorial Pattern Matching (CPM)*, pages 227–237. 2010. 12.1

[434] G. M. Slota, S. Rajamanickam, and K. Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 550–559, 2014. 9.1, 9.1, 9.5

[435] J. Soman, K. Kishore, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2010. 9.1

[436] G. L. Steele Jr. Making asynchronous parallelism safe for the world. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 218–231, 1990. 3.1, 3.2.3, 3.2.3, 5.1, 6.1

[437] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2000. 3.1, 3.4.3

[438] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982. 13.1, 13.2

[439] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *International World Wide Web Conference (WWW)*, pages 607–614, 2011. 10.1, 10.7.2, 10.9

[440] K. Tangwongsan, A. Pavan, and S. Tirthapura. Parallel triangle counting in massive streaming graphs. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 781–786, 2013. 10.9

[441] Task Parallel Library (TPL). `https://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx`. 2.1

[442] G. Tischler. On wavelet tree construction. In *Combinatorial Pattern Matching (CPM)*, pages 208–218, 2011. 14.3

[443] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX Annual Technical Conference (ATC)*, pages 1–11, 2011. 5.2

[444] D. Tsadok and S. Yona. ANSI C implementation of a suffix tree. `http://mila.cs.technion.ac.il/~yona/suffix_tree/`, 2003. 11.5

[445] D. Tsirogiannis and N. Koudas. Suffix tree construction algorithms on modern hardware. In *International Conference on Extending Database Technology*, pages 263–274, 2010. 11.5

[446] C. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 1(2):75–81, 2011. 10.1, 10.5, 10.9

[447] C. E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *International Conference on Data Mining (ICDM)*, pages 608–617, 2008. 10.7.4

[448] C. E. Tsourakakis. Counting triangles in real-world networks using projections. *Knowl. Inf. Syst.*, 26(3):501–520, 2011. 10.5, 10.9

[449] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. DOULION: Counting triangles in massive graphs with a coin. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 837–846, 2009. 10.5, 10.7.3, 10.9

[450] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. 11.1

[451] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990. 12.3

[452] S. van der Vegt. A concurrent bidirectional linear probing algorithm. In *15th Twente Student Conference on Information Technology*, 2011. 5.2

[453] S. van der Vegt and A. Laarman. A parallel compact hash table. In *International Conference on Mathematical and Engineering Methods in Computer Science*, pages 191–204, 2011. 5.2

[454] U. Vishkin. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics*, 9(2):197–207, 1984. 9.1

[455] J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980. 11.2

[456] W. Wang, Y. Gu, Z. Wang, and G. Yu. Parallel triangle counting over large graphs. In *Database Systems for Advanced Applications*, pages 301–308. 2013. 10.1, 10.5, 10.9

[457] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 265–266, 2015. 1.1, 1.6, 7.2.2

[458] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):409–10, 1998. 10.6.4

[459] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Computers*, 37(12):1488–1505, 1988. 3.1, 3.2.3, 5.1, 6.1

[460] P. Weiner. Linear pattern matching algorithm. In *IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973. 2.6.3, 11.1

[461] T. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984. 13.1

[462] E. Westbrook, R. Raman, J. Zhao, Z. Budlilic, and V. Sarkar. Dynamic determinism checking for structured parallelism. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2014. 15.2

[463] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *ACM International Conference on Supercomputing (ICS)*, pages 307–316, 2006. 8.2

[464] V. V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In *ACM Symposium on Theory of Computing (STOC)*, pages 887–898, 2012. 10.9

[465] H. Wu, D. Zinn, M. Aref, and S. Yalamanchili. Multipredicate join algorithms for accelerating relational graph processing on GPUs. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2014. 10.9

[466] Altavista web page hyperlink connectivity graph. `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`, 2012. 1, 7.5, 10.7

[467] Y. Yasui, K. Fujisawa, and K. Goto. NUMA-optimized parallel breadth-first search on multicore single-node system. In *IEEE International Conference on Big Data*, pages 394–402, 2013. 3.5, 7.2.1

[468] Y. You, D. Bader, and M. M. Dehnavi. Designing a heuristic cross-architecture combination for breadth-first search. In *International Conference on Parallel Processing (ICPP)*, pages 70–79, 2014. 3.5, 7.2.1

[469] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 325–336, 2009. 3.1

[470] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee. Fast iterative graph computation: A path centric approach. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 401–412, 2014. 15.2

[471] K. Zhang, R. Chen, and H. Chen. NUMA-aware graph-structured analytics. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 183–193, 2015. 1.1, 1.6, 7.2.2, 15.2

[472] J. Zhong and B. He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, June 2014. 7.2.2

[473] D. Zhou, D. G. Andersen, and M. Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Symposium on Experimental Algorithms (SEA)*, pages 151–163, 2013. 14.6

[474] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 129–142, 2010. 6.1

[475] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. 11.5, 13.1

[476] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. 13.1

[477] Y. Zu and B. Hua. GLZSS: LZSS lossless data compression can be faster. In *Workshop on General Purpose Processing Using GPUs*, pages 46:46–46:53, 2014. 13.1