Program-Centric Cost Models for Locality and Parallelism

Harsha Vardhan Simhadri

CMU-CS-13-124
September 2013

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch, Chair
Phillip B. Gibbons,
Gary L. Miller,
Jeremy T. Fineman (Georgetown University),
Charles E. Leiserson (MIT)

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

This work was supported by the National Science Foundation under grants CCF-1018188 and CCR-0122581, by Intel via the Intel Labs Pittsburgh, the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program and the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by Microsoft as part of the Carnegie Mellon Center for Computaional Thinking, and by IBM.

Keywords: Parallelism, Locality, Cost Models, Metrics, Schedulers, Parallel Algorithms, Shared Memory, Cache Hierarchies, Performance of Parallel Programs, Programming Interfaces, Run-time Systems, Parallel Cache Complexity, Effective Cache Complexity, PCC Framework, Parallelizability

Abstract

Good locality is critical for the scalability of parallel computations. Many cost models that quantify locality and parallelism of a computation with respect to specific machine models have been proposed. A significant drawback of these machine-centric cost models is their lack of portability. Since the design and analysis of good algorithms in most machine-centric cost models is a non-trivial task, lack of portability can lead to a significant wastage of design effort. Therefore, a machine-independent portable cost model for locality and parallelism that is relevant to a broad class of machines can be a valuable guide for the design of portable and scalable algorithms as well as for understanding the complexity of problems.

This thesis addresses the problem of portable analysis by presenting program-centric metrics for measuring the locality and parallelism of nested-parallel programs written for shared memory machines – metrics based solely on the program structure without reference to machine parameters such as processors, caches and connections. The metrics we present for this purpose are the parallel cache complexity (Q^*) for quantifying locality, and the effective cache complexity ($\widehat{Q}_\alpha$) for both locality and the cost of load-balancing for a parameter $\alpha \geq 0$. These two program-centric metrics constitute the parallel cache complexity (PCC) framework.

We show that the PCC framework is relevant by constructing a class of provably-good schedulers that map programs to the parallel memory hierarchy (PMH) model, represented by a tree of caches. We prove optimal performance bounds on the communication costs of our schedulers in terms of Q^* and on running times in terms of $\widehat{Q}_\alpha$. For many algorithmic problems, it is possible to design algorithms for which $\widehat{Q}_\alpha = O(Q^*)$ for a range of values of $\alpha > 0$. The least upper bound on the values of α for which this asymptotic relation holds results in a new notion of parallelizability of algorithms that subsumes previous notions such as the ratio of work to depth. For such algorithms, our schedulers are asymptotically optimal in terms of load balancing cache misses across a hierarchy with *parallelism* $\beta < \alpha$, which results in asymptotically optimal running times. We also prove bounds on the space requirements of dynamically allocated computations in terms of $\widehat{Q}_\alpha$. Since the PMH machine model is a good approximation for a broad class of machines, our experimental results demonstrate that program-centric metrics can capture the cost of parallel computations on shared memory machines.

To show that the PCC framework is useful, we present optimal parallel algorithms for common building blocks such as Prefix Sums, Sorting and List Ranking. Our designs use the observation that algorithms that have low depth (preferably polylogarithmic depth as in NC algorithms), reasonable "regularity" and optimal sequential cache complexity on the ideal cache model are also optimal according to the effective cache complexity metric. We present results indicating that these algorithms also scale well in practice.

Emphasizing the separation between algorithms and schedulers, we built a framework for an empirical study of schedulers and engineered a practical variant of our schedulers for the PMH model. A comparison of our scheduler against the widely used work-stealing scheduler on a 32 core machine with one level of shared cache reveals 25–50% improvement in cache misses on shared caches for most benchmarks. We present measurements to show the extent to which this translates into improvements in running times for memory-intensive benchmarks for different bandwidths.

Acknowledgments

I am very grateful to Guy Blelloch and Phillip Gibbons for introducing me to this field and many of the problems addressed in this thesis. In addition to being an inspiration as researchers, they have been great mentors and extremely patient teachers. It has been exciting to work with them, and many of the results in this thesis have come out of discussions with them. I could not have asked for wiser advisors.

I would like to thank Jeremy Fineman for many useful discussions, both technical and otherwise. He has been a great sounding board for many of my half-baked ideas. I would like to thank Charles Leiserson for his useful comments and pointers through out the course of my thesis work. I would like to thank both Jeremy and Charles for their feedback which has greatly helped me improve the presentation of this thesis. I would like to thank Gary Miller for asking many pertinent and incisive questions regarding the thesis.

I would like to thank my co-advisees Aapo Kyrola, Julian Shun and Kanat Tangwongsan for being great collaborators. I would like to thank Cyrus Omar, Umut Acar and Kayvon Fatahalian for many useful discussions and comments about my work. I would like to thank Pranjal Awasthi, Manuel Blum, Ashwathi Krishnan, Richard Peng, Vivek Seshadri and Aravindan Vijayaraghavan for many general discussions related to the thesis and their constructive feedback on my writing and talks.

Finally, I would like to thank my parents, my grandparents and my family for encouraging me to study science, for being my first teachers, and their unwavering support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Background

1.1.1 Hardware Constraints of Parallel Machines

The simplest cost models for programs such as the Random-Access Memory (RAM) and the Parallel Random-Access Memory (PRAM) models assume one unified memory with uniform cost — every memory access takes unit time. While hardware and software interfaces may or may not present unified addressable memory to programs, the real cost of accessing memory is always far from uniform. Beneath the unified interface, memory is supported by an assortment of cache and memory banks of varying speeds and sizes knit together around processors with interconnects of limited bandwidths.

Although non-uniform organization makes reasoning about program performance harder, it is a necessary choice for improving instruction throughput. Designing a large uniform access cost memory would create a memory access latency bottleneck. Instructions would execute at the rate at which wires can transfer data across the physical spread of the memory to serve memory references. Since the time required to transfer a signal across a wire increases with the length (and size of memory), data transfer times across large memory banks would be much slower than processor clock rates. Although increasing the operating voltage of the wire can speed up memory accesses, energy considerations constrain the voltage to be set to a reasonably low level resulting in latencies of hundreds of processor cycles for memories sizes common today. Refer Figure 1.1, which depicts stagnant DRAM latency trends.

The memory access latency problem can be overcome to some extent with a faster "cache" closer to processors, separate from the slower memory banks, that contains copies of contents of slower memory that are relevant to task assigned to a processor. Including a cache that is physically smaller and closer to the processor and caching— creating copies in cache—memory locations corresponding to frequently accessed program variables can improve the latency problem for programs capable of exploiting the setup. In practice, it is also effective to include several levels of increasingly larger, and correspondingly slower, caches near the processor. For instance, Xeon X5570, a high-end Intel processor from 2009, includes three levels of caches on the processor die with increasing access latencies — $4, 10$, and 38 processor cycles respectively while accessing the slower DRAM banks takes few hundred cycles [133]. Programs designed

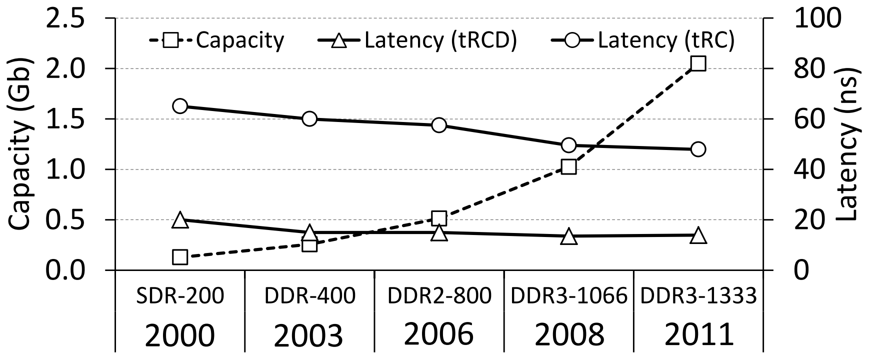


Figure 1.1: DRAM Capacity & Latency trends based on the dominant DRAM chips during the period of time [52, 104, 136, 144]. The overall DRAM access latency can be decomposed into individual timing constraints. Two of the most important timing constraints in DRAM are t_{RCD} (row-to-column delay) and t_{RC} ("row-conflict" latency). When DRAM is lightly loaded, t_{RCD} is often the bottleneck, whereas when DRAM is heavily loaded, t_{RC} is often the bottleneck. Note the relatively stagnant access latency. Figure adapted from [114].

with better variable (data) reuse and an ability to execute a greater number of instructions using a smaller working set that fits in cache would show significant benefits. In fact, many common algorithmic problems admit algorithms that have significant data reuse [72, 166].

A second approach to solving the memory access latency problem is prefetching — anticipating future memory accesses of a processor based on current access patterns and caching the location before an instruction explicitly requests it. This approach has its limitations: it is not always possible to predict future accesses with precision, and aggressive prefetching may saturate the bandwidth of the connections between cache and memory banks. Therefore, even when a processor is equipped with a prefetcher, it is important to design programs that use caches well. Moreover, prefetchers bloat the processor increasing its die size and power consumption. Several hardware designs such as GPUs choose to include only a small inaccurate prefetcher or leave it out altogether.

A third approach to the problems is to hide latency with multi-threading, also referred to as hyper-threading on some architectures. Each core is built to handle several instruction pipelines corresponding to different threads. When one pipeline halts due to slow memory access, other pipelines are brought in and processed. Although multi-threading can hide latency to some extent, it results in greater processor complexity. Programming them requires great care to ensure that the pipelines are not competing for the limited cache available to the core.

None of these hardware fixes are definitive solutions to the bandwidth problem — the gap between the rate at which processors can process data and the rate at which the connections between processors, caches and memory can transfer it. This gap has been widening every year with faster processor technologies [123]. Bandwidth, rather than the number of cores, is the bottleneck for many communication intensive programs. Working with limited bandwidth requires fundamental rethinking of the algorithm design to generate fewer data transfers rather than hardware fixes.

The performance landscape is further complicated by the arrangement of processors with respect to the cache and memory banks. Hardware designs vary widely in how they budget silicon area and power to processors and caches, the number of processors they include, and the connections between modules on and across chips. Designs also vary on assignment of caches to processors — some caches are exclusive to individual processors while others are shared by subsets of processors. Further variations include the arrangement of memory banks and the degree of coherence between them, i.e. whether the memory banks present an unified address space with certain semantic consistency or if the banks are presented as fragmented pieces to be dealt with directly by the programmer.

Irrespective of these design choices, programmers are left to deal with two common and vital issues when attempting to write fast and scalable parallel programs for any parallel machine:

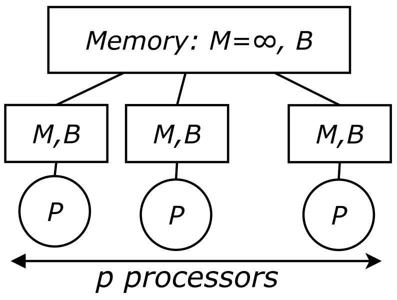
1. **Maximizing locality**: Reducing memory access latency by effective management of caches and economical usage of bandwidth between processing and memory components.
2. **Load Balance**: Maximizing utilization of processors available on the machine.

Managing the two issues simultaneously is a difficult problem that requires a good understanding of the machine that is not always solvable in an ad hoc manner. A systematic approach to address this problem is to present a programmer with a good cost model for the machine that estimates the effectiveness of their program with respect to managing these two issues. Simple cost models like PRAM make the programmer take account of load balance costs while being agnostic to locality. However, performance of data-intensive applications is constrained more by communication capabilities of the machine rather than computation capabilities. Contemporary chips such as 8-core Nehalem can easily generate and process data requests that are about a factor of two to three times greater than what the interconnect can support [133]. Placing multiple such chips on the interconnect exacerbates the problem further, and the gap between computing power and transfer capability is likely to widen in future generations of machines. Therefore, it is necessary to construct a cost model that makes the programmer take account of locality issues in addition to load balance.

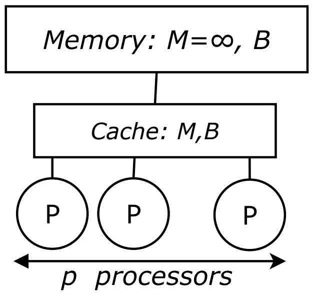
1.1.2 Modeling Machines for Locality and Parallelism

A common approach to developing cost models for parallelism and locality is to start with an abstract machine model for a class of parallel machines. The model would capture the most important aspects of the arrangements of processors with respect to the cache and memory banks. Individual machines within the class would be specified by supplying a set of parameters such as number of processors, cache levels, etc. For example, Figure 1.2(a) depicts the private cache machine model in which a certain number of processors have access to their own private cache which connect to a large shared memory. The parameters that identify a specific machine within this model are the number of processors p, the size of each cache M and the size of cache lines B. Figure 1.2(a) depicts the shared cache machine model with p processors sharing a cache of size M organized into cache lines of size B.

A machine-centric cost model would be developed on top of the machine model to reflect the cost of a program in the model. The input to the cost model is an execution: the parallel program as well as a valid execution order of instructions on individual processors (schedule) and a scheme for placing and moving data elements (routing) when the machine model requires



(a) Private cache model



(b) Shared cache model

Figure 1.2: Simple Memory Machine Models

explicit routing specification. Programs are allowed to be, and often are, dynamic. That is, parts of the program are revealed only after prerequisite portions of the program are executed. This necessitates an online scheduling policy that assigns instructions to processors on the fly.

The output of the cost model for a given execution typically includes costs such as the amount of data transferred for capturing locality, execution completion time to reflect load balance, the maximum space footprint of the program, etc. For example, machine-centric cost models developed for machine models in Figure 1.2 measure:

- Locality: total number of caches misses under the optimal cache replacement policy [150] and full associativity.
- Time: number of cycles to complete execution assuming that cache misses stall an instruction for C time units.
- Space: maximum number of distinct memory locations used by the program at any point during its execution.

The output of the cost model matches up with actual performance on a machine to the extent that the model is a realistic representation of the machine. Therefore, various machine and cost models have been proposed for different categories of machines. The two key aspects in which the underlying machine models differ are:

1. Whether the memory presents an unified address space (shared memory) or if the program is required to handle accessing distinct memory banks (distributed memory).
2. Another key aspect in which they differ include the number of levels of caches or program synchronization points. While some models allow account for locality one level, others allow hierarchies, i.e. , multiple levels of caches and synchronization.

Table 1.3 classifies previous work along these two axes. The presentation of this thesis builds on shared memory models of which a brief overview follows. Chapter 2 surveys these various machine models in greater detail.

Shared Memory. Shared memory parallel machines present a single continuously addressable memory address space. Contemporary examples include Intel Nehalem/SandyBridge/IvyBridge with QPI, IBM Z system, IBM POWER blades, SGI Altix series and Intel MIC. The simpler

	Distributed Memory	**Shared Memory**
One Level of Storage/ Synchronization	Fixed-Connection Networks [116] BSP [163] LogP[70]	Parallel External Memory [20] (Private Caches [2, 48]) Shared Cache Model [37, 134]
Hierarchy of Storage/ Synchronization	Network-Oblivious Model [32]	Multi-BSP [164] Parallel Memory Hierarchies [13]

Figure 1.3: Machine-centric cost models for parallel machines that address locality. Models are classified based on whether their memory is shared or distributed, and the number of levels of locality and synchronization costs that are accounted for in the model.

interface makes them a popular choice for most programmers as they are easier to program than distributed systems and graphics processing units. They are often modelled as PRAM machines when locality is not a consideration.

Previously studied models for shared memory machines include the Private Cache model [2, 48], the closely related parallel External Memory model (PEM) [20] and the Shared Cache model [37]. Both these models include a large unified shared memory and one level of cache, but differ in the ownership of caches. While the earlier model assigns each processor its own cache, the latter forces all processors to share one cache.

Locality on modern parallel machines can not be accurately captured with a model with only one cache level. The disparity in access latency varies over several orders of magnitude. To reflect this more accurately, models with several cache levels are required. Further, the model should include a description of the ownership or nearness relation between caches and processors. Such models include the Multi-BSP [164] and the Parallel Memory Hierarchy model [13].

The Parallel Memory Hierarchy model (PMH model, Figure 1.2(c)) describes the machine as a symmetric tree of caches with the memory at the root and the processors at the leaves. Each internal node is a cache; the closer the cache is to the processor in the tree, the faster and smaller it is. Locality is measured in terms of the number of cache misses at each level in the hierarchy, and parallelism in terms of program completion time.

The PMH model will be referred to multiple times through this thesis. It is one of the most relevant and general machine models for various reasons:

- It models many modern shared memory parallel machines reasonably well. For example, Figure 1.5 shows the memory hierarchies for the current generation desktop/servers from Intel, AMD, and IBM all of which directly correspond to the PMH model.
- It is also a good approximation for other cache organizations. Consider a 2-D grid with each node consisting of cache and processor for example. It is possible to approximate it with a PMH by recursively partitioning the grid into finer sub-grids. Figure 1.6 depicts such grid of dimension 4×4 and how such a partition might be modeled with a three level

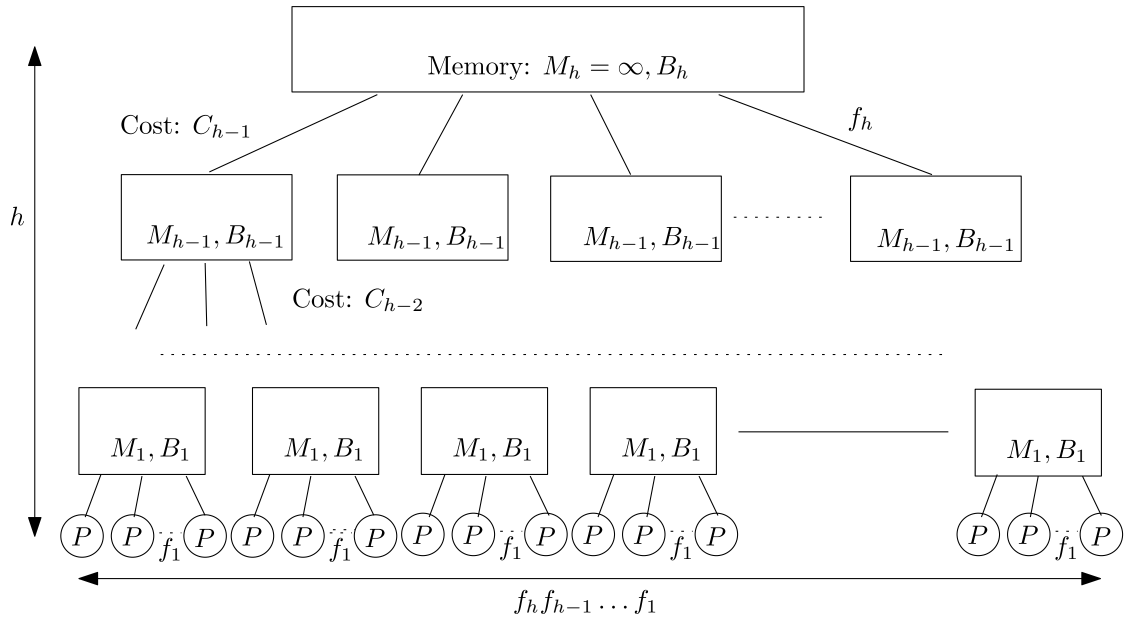


Figure 1.4: A Parallel Memory Hierarchy, modeled by an h-level tree of caches, with $\prod_{i=1}^{h} f_i$ processors.

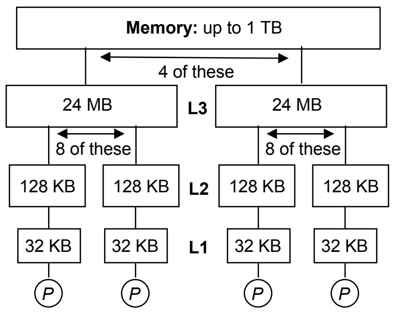
cache hierarchy.

- Since it is possible to emulate any network of a fixed volume embedded in 2 or 3 dimensions with a fat-tree (a tree with greater bandwidth towards the root) network with only poly-logarithmic factor slowdown in routing delay [121], fat-trees networks can support robust performance across programs. PMH models the fat-tree network with storage elements at each node [112, 169].

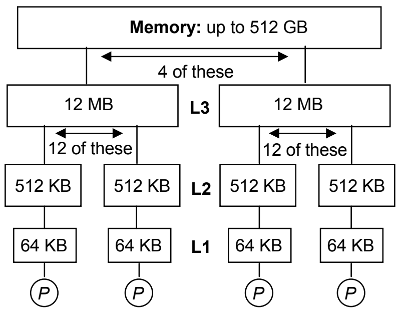
Drawbacks of Machine-centric Models. Constructing good parallel algorithms for many of these machine-centric cost models requires significant effort. Unfortunately, these algorithms are tailored to specific machine-centric cost models and may not be suitable or portable for a different machine model. This results in a duplication and waste of design effort. Even in those cases where the algorithm is adaptable with minor modifications across machine models, the analysis certifying the performance of the algorithms may not be portable across cost models.

Within a machine-centric cost model, programs are often designed specifically for a certain choice of machine parameters (number of processors and available cache size). Therefore, when the resources available on the machine fluctuate, which is often the case in practice, program's performance may degrade due to lack of robustness in program design.

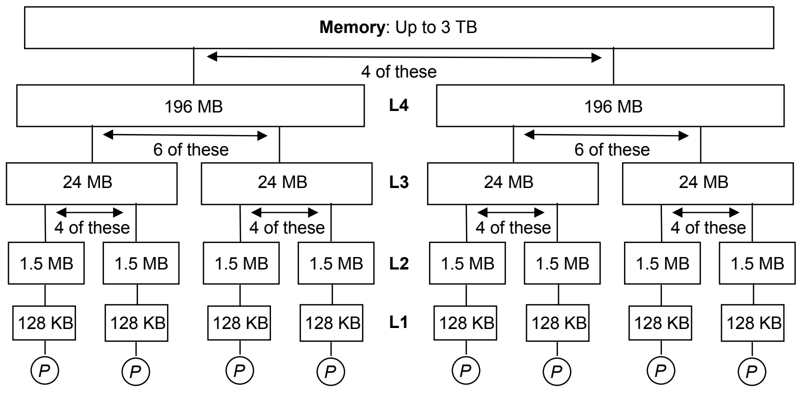
Another major drawback with this paradigm is that the cost model requires an execution rather than a program. It is cumbersome for a program designer to provide an algorithm and scheduler for every machine. More often than not, it encourages the program designer to bundle the two together and hand-tune programs for specific machines. Program portability suffers greatly as a consequence.



(a) 32-processor Intel Xeon 7500



(b) 48-processor AMD Opteron 6100



(c) 96-processor IBM z196

Figure 1.5: Memory hierarchies of current generation architectures from Intel, AMD, and IBM. Each cache (rectangle) is shared by all processors (circles) in its subtree.

1.1.3 Portability of Programs and Analysis

A more elegant, portable and productive approach is to separate program and scheduler design. The programmer would design portable programs that have robust performance across machines. The scheduler designer would handle the task of understanding machines and mapping programs on to machine. Each class of machine model would have its own family of schedulers. This paradigm is much closer to how users actually code on contemporary parallel machines using languages such as OpenMP [51], Cilk++ [96], Intel TBB [97], and the Microsoft Task Parallel Library [131].

This paradigm is also more robust to variations in computing resources. Since the programmer specifies the full parallelism of the computation, typically much more than is available on a real machine, the number of processors during execution can be dynamically varied. If the program's locality can be made "robust", it can also better cope with variations in the amount of cache and memory available at execution.

To make this approach systematic, it is critical to develop a model that quantifies locality and parallelism of the program itself rather than with respect to an execution on a particular machine

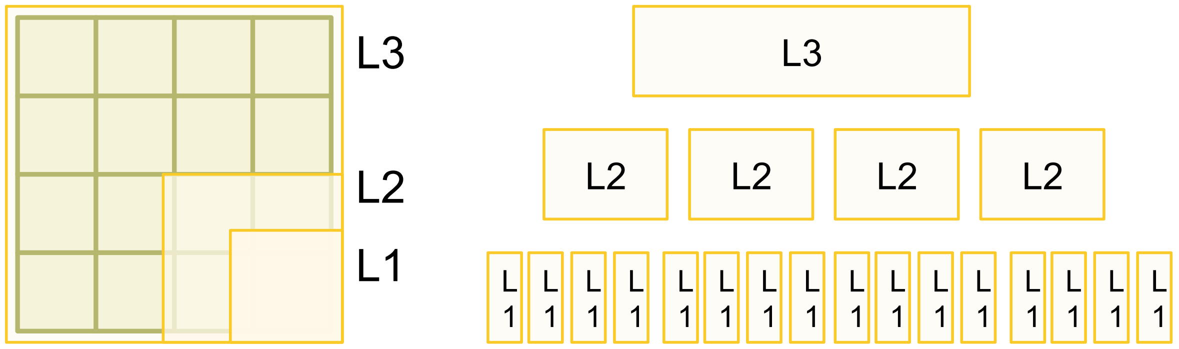


Figure 1.6: Approximating a grid with a Parallel Memory Hierarchy.

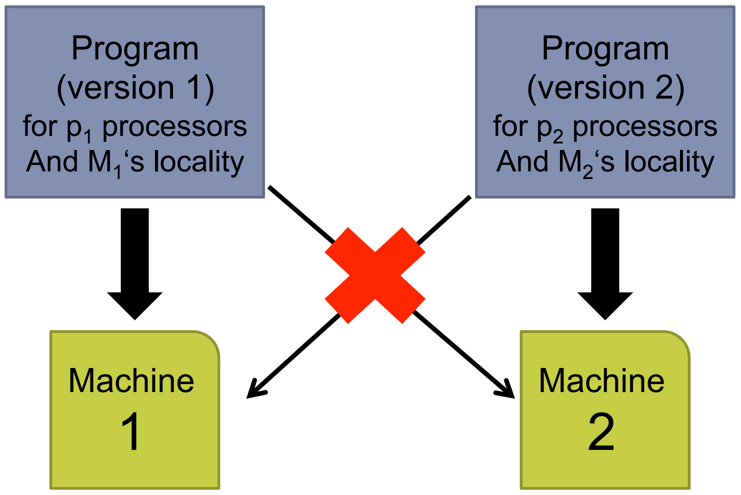

Program (version 1) for p_1 processors And M_1's locality
Program (version 2) for p_2 processors And M_2's locality
Machine 1
Machine 2

Figure 1.7: Programs designed in machine-centric cost models may not be portable either in programs or analysis.

model. Such models have little utility if they do not represent costs on real machines. Therefore, in addition to the model itself, it is critical to show general bounds on performance when mapping the program onto particular machine models. Ideally, one high-level program-centric model for locality can replace many machine-centric models, allowing portability of programs and the analysis of their locality across machines via machine-specific mappings. Such mappings make use of a scheduler designed for a particular machine model. Schedulers should come with program performance guarantees that are based solely on program-centric metrics for the program and parameters of the machine.

There are certain natural starting points for program-centric metrics. The ratio of the number of instructions (work) to the length of the critical path (depth) is a good metric for parallelism in the PRAM model. However, non-uniform communication cost creates variations in execution time of individual instructions and this ratio may not reflect the true extent of parallelism on real machines. Independently, the Cache-Oblivious framework [86] is a popular framework for measuring locality of sequential programs. In this framework, programs are measured against an abstract machine with two levels of memory: a slow unlimited RAM and a fast ideal cache with

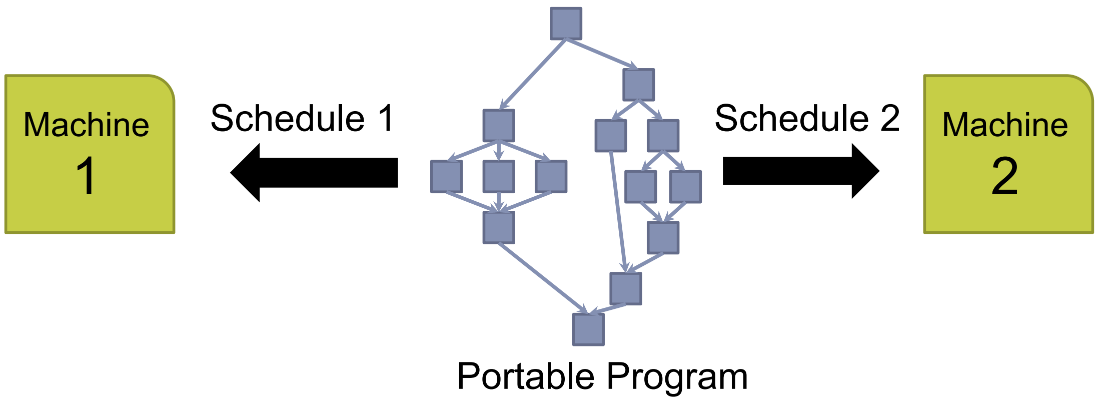


Figure 1.8: Separating program from scheduler design: programs should be portable and schedulers should be used to map program to target machines

limited size. Locality for a sequential program is expressed in terms of the cache complexity function $Q(M, B)$, defined to be the number of cache misses on an ideal cache [86] of size M and cache line size B. As long as a program does not use the parameters M and B, the metric is program-centric and the bounds for the single level are valid simultaneously across all levels of a multi-level cache hierarchy.

There has been some success in putting together a program-centric cost model for parallel programs that has relevance for simple machine models such as the one-level cache models in Figure 1.2(a),(b) [2, 35, 40, 84]. The cache-oblivious framework (a.k.a., the ideal cache model) can be used for analyzing the misses in the sequential ordering. For example, we could use the depth-first order to define a sequential schedule (Figure 1.9) to obtain a cache complexity Q_1 on the ideal cache model, say for fixed parameters M, B. One can show, for example, that any nested-parallel computation with sequential cache complexity Q_1 and depth D will cause at most $Q_1 + O(pDM/B)$ cache misses in total when run with an appropriate scheduler on p processors, each with a private cache of size M and block size B [2, 50].

Unfortunately, previous program-centric metrics have important limitations in terms of generality: they either apply to hierarchies of only private or only shared caches [2, 35, 40, 66, 84, 134], require some strict balance criteria [38, 67], or require a joint algorithm/scheduler analysis [38, 63, 64, 66, 67]. Constructing a program-centric metric relevant to realistic machine models such as PMHs seems non-trivial.

1.2 Problem Statement

This thesis addresses the following problems:

1. Constructing a program-centric cost model for parallel programs that is relevant to a broad class of realistic shared memory machine models. The cost model should be defined entirely based on the program without referring to machine attributes.
2. Constructing a locality-preserving and load-balancing scheduler to map programs to realistic machine models. While it is relatively easy to build a scheduler that addresses one of

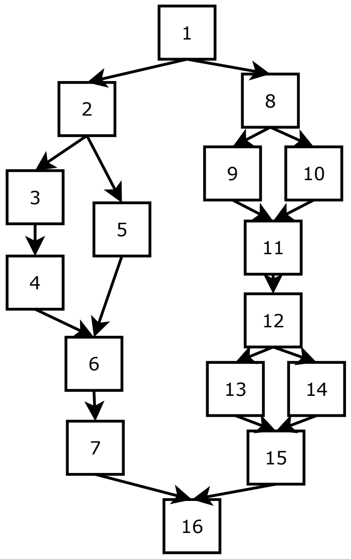


Figure 1.9: A Depth-first schedule

the two constraints, managing both constraints simultaneously is harder.

3. Proving good performance bounds in terms of program metrics and machine parameters. A scheduler with optimal performance bounds for Parallel Memory Hierarchy would serve as a test of generality for the cost models and the feasibility of the program-centric approach to locality.
4. Designing parallel algorithms for common algorithmic problems that are optimal with respect to the program-centric metrics and practically demonstrating of the performance and scalability of these algorithms.
5. Constructing an experimental framework for validation of the locality-preserving scheduler, and comparison with current schedulers over a variety of micro-benchmarks.

1.3 Solution Approach and Contributions

An execution of a parallel program can be viewed as a Directed Acyclic Graphs (DAG) with nodes representing instructions and edges representing precedence constraints between nodes. As previously mentioned, the ratio of the number of nodes (work) to the length of the critical path (depth) gives some indication of the parallelism of the program [54, 79, 88]. A more accurate estimate would involve an appropriate weighting of nodes or sets of nodes to account for the delay in executing instructions because of non-uniform memory access costs.

If the schedule were provided in addition to the program, the precise execution order of the instructions would be known and each node in the DAG could be assigned a weight depending on the time taken for the memory request initiated by the instruction to complete. This would give a fairly accurate prediction of the parallelism of the program. However, this approach violates the program-centric requirement on the cost model.

To retain the program-centric characteristic, we might place weights on nodes and sets of nodes—supernodes—corresponding to the delay that might be realizable in practice in a schedule that is asymptotically optimal. The delay on a node might correspond to access latency of the instruction, and the delay on a supernode might correspond to the cost of warming up a cache for the group of instruction that a schedule has decided to colocate on a cluster of processors sharing a cache. However, what constitutes a good schedule for an arbitrary DAG might differ based on the target machine. Even for a target machine, finding the optimal schedule may be computationally hard. Further, it is intuitively difficult to partition arbitrary DAGs into supernodes to assign weights.

This dissertation focuses only on nested-parallel DAGs—programs with dynamic nesting of fork-join constructs but no other synchronizations—as they are more tractable. Although it might seem restrictive to limit our focus to nested-parallel DAGS, this class is sufficiently expressive for many algorithmic problems[149]. It includes a very broad set of algorithms, including most divide-and-conquer, data-parallel, and CREW PRAM-like algorithms.

The solution approach starts with a cost model to analyze the "cache complexity" of nested-parallel DAGs. The cache complexity is a function that estimates the number of cache misses a nested-parallel DAG is likely to incur on a variety of reasonable schedules, as a function of the available cache size. Cache misses can be used as a proxy for objectives an execution needs to optimize such as minimizing latency and bandwidth usage (see section 2.7.1 for more details).

1.3.1 Program-centric Cost Model for Locality and Parallelism

The first contribution is a program-centric cost model for locality — the parallel cache complexity cost model first described in a joint work with Blelloch, Fineman and Gibbons [41, 46] under the name PCO cost model (Section 4.3). We also extended this cost model to account for parallelism and the cost of load balance, and defined another cost model called the effective cache complexity. We refer to the set of these two cost models — the parallel cache complexity Q^* and effective cache complexity $\widehat{Q}_\alpha$ — as the PCC (Parallel Cache Complexity) framework or the PCC model from now on. As in the Cache-Oblivious design framework the cache cost of a family of DAGs parameterized by input size n is derived in terms of a single cache size M and a single block size B giving a cache complexity $Q^*(n; M, B)$ independent of the number of processors.

Although Q_1, D, and W are good program-centric metrics for machines where one level of cache (either private or shared) dominates the programs running time, it is hard to generalize these results to multiple levels (Section 4.3). The problem arises because the depth-first order based cache analysis accounts for cache line reuse between instructions unordered in the DAG. The parallel cache complexity cost model overcomes this problem and is suitable for analyzing performance on cache hierarchies. In the PCC model, locality, denoted by $Q^*(M, B)$, is defined based solely on the composition rules used to construct nested-parallel programs. The model and analysis have no notions of processors, schedulers, or the specifics of the cache hierarchy.

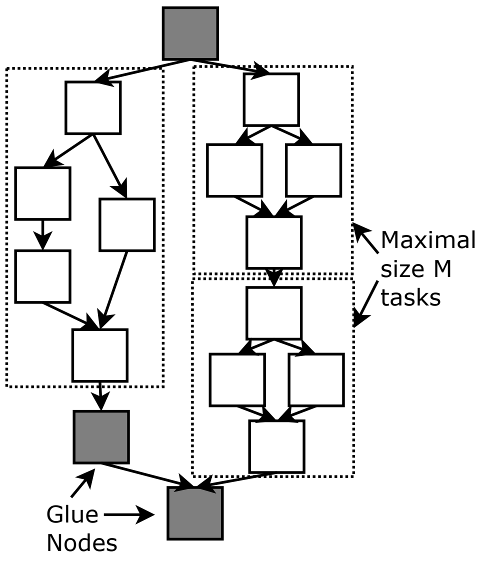


Figure 1.10: Parallel cache complexity analysis

The PCC model takes advantage of the fact that nested-parallel DAGs can be partitioned easily and naturally at various scales. Any corresponding pair of fork and join points separates the supernode nested within from the rest of the DAG. We call such supernodes including the corresponding fork and join points a *task*.

We refer to the memory footprint of a task (total number of distinct cache lines it touches) as its size. We say a task is *size M maximal* if its size is less than M but the size of its parent task is more than M. Roughly speaking, the PCC analysis decomposes the program into a collection of maximal subtasks that fit in M space, and "glue nodes" — instructions outside these subtasks (fig. 4.3). This decomposition of a DAG is unique for a given M because of the tree-like and monotonicity properties of nested-parallel DAGs described earlier. For a maximal size M task t, the parallel cache complexity $Q^*(\mathsf{t}; M, B)$ is defined to be the number of distinct cache lines it accesses, counting accesses to a cache line from unordered instructions multiple times. The model then pessimistically counts all memory instructions that fall outside of a maximal subtask (i.e., glue nodes) as cache misses. The total cache complexity of a program is the sum of the complexities of the maximal tasks, and the memory accesses outside of maximal tasks.

Although this may seem overly pessimistic, it turns out that for many well-designed parallel algorithms, including quicksort, sample sort, matrix multiplication, matrix inversion, sparse-matrix multiplication, and convex hulls, the asymptotic bounds are not affected, i.e. , $Q^* = O(Q)$ (see Section 4.3.1). The higher baseline enables a provably efficient mapping to parallel hierarchies for arbitrary nested-parallel computations.

The Q^* metric captures only the locality of a nested-parallel DAG, but not its parallelism. We need to extend the metric to capture the difficulty of load balancing the program on a machine model, say on a parallel hierarchy. Intuitively, programs in which the ratio of space to *parallelism* is uniform and balanced across subtasks are easy to load balance. Intuitively this is because on any given parallel memory hierarchy, the cache resources are linked to the processing resources: each cache is shared by a fixed number of processors. Therefore any large imbalance between space and processor requirements will require either processors to be under-utilized or caches to be over-subscribed. Theorem 3 presents a lower bound that indicates that some form of parallelism-space imbalance penalty is required.

The $Q^*(n; M, B)$ metric was extended in our work [41] to a new cost metric for a families of DAGs parameterized by the input size n—the effective cache complexity $\widehat{Q}_\alpha(n; M, B)$—that captures the extent of the imbalance (Section 4.3.2). The metric aims to estimate the degree of parallelism that can be utilized by a symmetric hierarchy as a function of the size of the computation. Intuitively, a computation of size S with "parallelism" $\alpha \geq 0$ should be able to use $p = O(S^\alpha)$ processors effectively. This intuition works well for algorithms where parallelism is polynomial in the size of the problem. Most notably, for regular recursive algorithms, the α here roughly corresponds to $(\log_b a) - c$, where a is the number of parallel recursive subproblems, n/b is the size of subproblems for a size-n problem, and $O(n^c \cdot polylog\ n)$ is the depth (span) of an algorithm with input size n.

As in the case of parallel cache complexity, for many algorithmic problems, it is possible to construct algorithms that are asymptotically optimal, i.e. well balanced according to this metric. The cost $\widehat{Q}_\alpha(n; M, B)$ for inputs of size n is asymptotically equal to that of the standard sequential cache cost $Q^*(n; M, B)$. When the imbalance is higher, the cost metric would reflect this in asymptotics—$\widehat{Q}_\alpha(n; M, B) = \Omega(Q^*(n, M, B))$—and prompt the algorithm designer to reconsider the algorithm.

1.3.2 Scheduler for Parallel Memory Hierarchies

To support the position that scheduler design should be taken up separately from algorithm design, this dissertation demonstrates a good class of schedules suitable for Parallel Memory Hierarchies. The scheduler for PMH presented here is a new space-bounded scheduler from the joint work with Blelloch, Fineman and Gibbons [41] that extends recent work of Chowdhury et al. [64].

A space-bounded scheduler accepts dynamically parallel programs that have been annotated with space requirements for each recursive subcomputation called a task. These schedulers run every task in a cache that just fits it (i.e. , no lower cache will fit it), and once assigned, tasks are not migrated across caches. Once a task is *anchored* to a cache, the instructions in the task can only be executed by the processors beneath the cache in the hierarchy. As processors traverse the DAG of a task anchored at cache X, they reveal subtasks that can be potentially anchored to smaller caches beneath X in the hierarchy. The scheduler has to decide where to anchor such subtasks and the number of processors to assign to the subtask with two objectives — keeping all processors busy, and making fast progress along the critical path in the DAG. These decisions affect the precise communication costs and running time of the algorithm.

While all space-bounded schedulers are good at preserving locality because of their anchoring property, not all are good at load-balancing. Identifying those specific schedulers which can cope with a reasonable degree of imbalance in programs is more difficult. We construct a class of such schedulers (Section 5.3). We further subselect from this class of schedulers and adapt them to attain good space bounds under dynamic space allocation. This class of schedulers can also be applied to other machine organizations by approximating them with PMH as discussed in section 1.1.2.

1.3.3 Performance Bounds for the Scheduler

We prove that the class of schedulers described above performs well on Parallel Memory Hierarchies in terms of communication costs, space and time. The bounds are in terms of the parallel and effective cache complexity analysis of the PCC model, and the parameters that characterize a PMH. Part of the analysis presented in this thesis is adapted from our earlier work [41] while some of it, specifically the bounds on space, is new.

- **Communication**. (Section 5.3.1) The scheduler guarantees that the number of misses across all caches at each level i of the machines hierarchy is at most $O(Q^*(n; M_i, B_i))$. This shows that the scheduler is capable of preserving locality of the program at every level in the hierarchy.
- **Time.** (Section 5.3.2) The scheduler allows parallel subtasks to be scheduled on different levels in the memory hierarchy, allowing significant imbalance in the sizes of tasks. Furthermore, the scheduler achieves efficient total running time, as long as the parallelism of the algorithm is sufficient greater than the *parallelism of the machine*. Specifically, we show that our scheduler executes a computation with cost $\widehat{Q}_\alpha(n; M_i, B)$ on a homogeneous h-level parallel memory hierarchy having p processors in time:

$$O\left(\frac{\sum_{i=0}^{h} \widehat{Q}_\alpha(n; M_i, B)}{p} \cdot v_h\right),$$

  where M_i is the size of each level-i cache, B is the uniform cache-line size, C_i is the cost of a level-i cache miss, and v_h is an overhead defined in Theorem 10. For any algorithms where $\widehat{Q}_\alpha(n; M_i, B)$ is asymptotically equal to the optimal sequential cache-oblivious cost $Q^*(n; M, B)$ for the problem, and under conditions where v_h is constant, this is optimal across all levels of the cache. For example, a parallel sample sort (that uses imbalanced subtasks) has $\widehat{Q}_\alpha(n; M_i, B) = O((n/B)log_{M+2}(n/B))$ (Theorem 25), which matches the optimal sequential cache complexity for sorting, implying optimality on parallel cache hierarchies using our scheduler.
- **Space.** (Section 5.3.3) In a dynamically allocated program, the space requirement depends on the schedule. A bad parallel scheduler can require many times more space than the space requirement of a sequential schedule. A natural baseline is space of the sequential depth-first schedule S_1. We demonstrate a schedule that has good provable bounds on space. In a certain range of cost-metrics and machine parameters, the extra space that the schedule needs over the sequential space requirement S_1 is asymptotically smaller than S_1.

1.3.4 Algorithms: Theory and Practice

Since this thesis advocates the separation of algorithms and schedulers, existence of good portable algorithms is essential to support this position. Specifically, it is necessary to demonstrate that algorithms that are optimal in the program-centric cost models described earlier exist.

Chapter 6 describes our contributions in terms of new algorithms. All the algorithms presented in the chapter are oblivious to machine parameters such as cache size, number of processors etc. The chapter presents various building block algorithms such as Scans (Prefix Sums), Pack, Merge, and Sort from our work [40]. The algorithms are optimal in the effective cache complexity ($\widehat{Q}_\alpha$) model and also have poly-logarithmic depth and are optimal in the sequential cache complexity model (Q_1). Many of these algorithms are new. These optimal building blocks are plugged into adaptations of existing algorithms to construct good graph algorithms (Section 6.5). We also designed new I/O-efficient combinatorial algorithms for the set-cover problem using these primitives [45] (Section 6.6). An algorithm for sparse matrix-vector multiplication is also presented (Section 6.4). We used the algorithms presented in this thesis as sub-routines for designing many baseline algorithms [149] for the problem-based benchmark suite (PBBS) [34].

These algorithms also perform extremely well in practice, in addition to being simple to program (< 1000 lines in most cases). Experiments on a 40-core shared memory machine with limited bandwidth are reported in Section 6.7. The algorithms demonstrate good scalability because they have been designed to minimize data transfers and thus use bandwidth sparingly. In many cases, the algorithms run at the maximum speed that the L3 – DRAM bandwidth permits.

1.3.5 Experimental Analysis of Schedulers

In addition to theoretical guarantees, an experimental framework would be useful to help test the efficacy of different schedulers on target machines. Experiments would address two concerns with theoretical bounds: (i) the machine model may not be a perfect representation of the machine, and (ii) the cost of scheduling overhead is not counted in our performance guarantees. Although the second problem could be addressed in theory, we leave it for future work.

To facilitate a fair comparison of schedulers on various benchmarks, a light-weight, flexible framework was designed to provides separate modular interfaces for: (i) writing portable nested parallel programs, (ii) specifying schedulers, and (iii) describing the target machine as a tree of caches in terms of parameters such as number of cache levels, size of caches etc (Chapter 7). The framework provides fine grained timers and allows access to various hardware counters to measure cache misses, data movements etc (Section 7.1).

The framework was deployed on a 32-core shared memory machine with 3 levels of cache, one of which is shared. We compare four schedulers — work-stealing, priority work-stealing (PWS) [140], and two variants of space-bounded schedulers engineered for low overheads (Section 7.2) — on both divide-and-conquer micro-benchmarks (scan-based and gather-based) and popular algorithmic kernels such as quicksort, samplesort, matrix multiplication, and quad trees (Section 7.3.1).

Our results in Section 7.3.3 indicate that space-bounded schedulers reduce the number of L3 cache misses compared to work-stealing schedulers by 25–50% for most of the benchmarks, while incurring up to 6% additional overhead. For memory-intensive benchmarks, the reduction in cache misses overcomes the added overhead, resulting in up to a 25% improvement in running time for synthetic benchmarks and about 5% improvement for algorithmic kernels. To better understand the impact of memory bandwidth on scheduler performance, we quantify runtime improvements over a 4-fold range in the available bandwidth per core and show further improvements in the running times of kernels (up to 25%) when the bandwidth is reduced.

1.4 Thesis statement

A program-centric approach to quantifying locality and parallelism in shared memory parallel programs is feasible and useful.

Chapter 2

Related Work – Cost Models for Locality

This chapter surveys models of computations — models based on circuits, machines, and programs — that quantify locality and/or parallelism. Approaches to developing complexity results (lower bounds) and constructive examples (algorithms) will be highlighted.

Chronologically, the first machine-centric model to quantify locality in parallel machines is the VLSI computation model pioneered by Thompson [160] which attracted a great deal of attention from theorists [143] (Section 2.1). Although this line of work could arguably be linked to earlier studies such as the comparison of the relative speeds of one-tape vs multi-tape Turing machines [92, 93], VLSI complexity studies were the first to deal with practical parallel computation models – logic gates and wires. In the spirit of the Turing hypothesis, Hong [94] establishes polynomial relations between sequential time (work), parallel time (depth) and space of several variants of Multi-Tape Turing Machines, Vector Models and VLSI computation models.

In this model, computations are represented as circuits embedded in a few planar layers. Greater locality in the computation allows for shorter wire lengths and a more compact circuit layout. Greater parallelism leads to fewer time steps. The complexity of algorithmic problems is explored in terms of trade-off between the area and time for VLSI solutions to the problem. Such results are usually expressed in the form $AT^\alpha = \Omega(f(n))$ where n is the problem size and α is typically set to 2 for 2-D VLSI following Thompson's initial work [160]. A scalable solution to the problem would present a circuit with the small circuit area (greater locality) and a small number of time steps (greater parallelism) making the product AT^2 as close to the lower bound $f(n)$ as possible.

Although the model provides valuable insights into the complexity of the problem and relates graph partitioning to locality, it can be used only at a very low level of abstraction – fixed logic gates and wires. Since programs today are written at a much higher level than circuits, a more program-friendly cost model such as the PCC model is required. Quantitatively mapping VLSI complexity bounds to PCC bounds and vice-versa remains an interesting problem for future.

The next step in the evolution of parallel machines were the fixed-connection networks (Section 2.2). These machines are modeled as a collection of nodes connected by a network with a specific topology. Each node includes a general purpose processor, a memory bank and a router. Programming in this model would involve specifying a schedule and routing policy in addition to the instructions. Performance of an algorithms is typically measured in terms of the number of time steps required to solve a problem of a specific size. An algorithm with a structure better suited for the machine would take lesser time to complete.

A fundamental problem with fixed-connection networks is the difficulty of porting algorithms and their analyses between models. In other words, locality in these machines is topology-specific. The relative strengths of networks, in terms of how well one network can emulate another network and algorithms written for it, have been studied in detail. These embeddings allow for portability of some analyses between topologies to within polylog-factor approximations. However, programming these machines is still inconvenient and porting the programs is not easy. Lower bounds from the VLSI complexity model can be directly mapped to finite-dimensional embeddings of fixed-connection networks.

To simplify programming and analysis on fixed-connection networks, cost models that abstract away the details of the network topology in favor of costs for bandwidth, message latency, synchronization costs, etc. have been proposed (Section 2.3). Among the early models are the Bulk-Synchronous Parallel and the LogP model. Subsequently, models for distributed memory machines with multi-level hierarchies such as the Network-Oblivous Model [32] have been proposed. These machine-centric models have the advantage of being more general in their applicability. A simpler but slightly more program-centric model inspired by the models above – the Communication Avoiding model is widely used for developing scalable algorithms for distributed machines.

Pebbling games (Section 2.4) were suggested by Hong and Kung [102] as a tool to study I/O requirements in algorithms expressed as dependence graphs, and are very useful for establishing I/O lower bounds for algorithms. Pebbling games highlight the direct relation between partitions of a DAG and I/O complexity, a theme that underlies most of the cost models for locality. A major drawback of this model is that it does not quantify parallelism or the cost of load balancing a computation.

Nevertheless, pebbling games are very versatile in their applications. In addition to modeling I/O complexity, it has been used for studying register allocation problems and space-time tradeoffs in computations. Pebbling games have been extended for multi-level hierarchies. Savage's textbook [147] surveys pebbling games in great detail in addition to many of the models described till this point.

Pebbling games are trivial for sequential programs whose DAGs are just chains. This simplified game directly resembles a one-processor machine with a slow memory (RAM) and a fast memory (cache). The first simplified model to reflect this was the External Memory Model which was made more program-centric by the Cache-Oblivious model. Both models have attracted a great deal of algorithm's research because of their simplicity. Section 2.5 highlights the similarities and differences between the two. Another model for measuring locality in sequential programs is the Hierarchical Memory Model which models memory as one continuously indexable array but with a monotonically increasing cost for accessing higher indices.

Parallel extensions to these sequential models that are well suited for shared memory machines have been studied. Section 2.6 presents a overview of some of these including one-level and multi-level cache models. Among the one-level cache models are the Parallel External-Memory Model and the Shared Cache Model. The Multi-level models include the Multi-BSP model and the Parallel Memory Hierarchy model. The PMH model is referred to througout the dissertaion.

The details of an adaptation of Parallel Memory Hierarchy model of Alpern, Carter and Ferrante [13] are described in Section 2.7. The PMH model is a realistic model that captures locality and parallelism in modern shared memory machines. While many architectures directly resemble the PMH model (Figure 1.5), others can be approximated well (Figure 1.6) with PMH. The PCC framework presented in this dissertation, while being entirely program-centric, is inspired by the PMH machine model. Section 5.3 demonstrates a scheduler that maps costs in the PCC model to a PMH with good provable guarantees.

Finally, in Section 2.8, we survey reuse distance analysis which is extremely useful for profiling the locality of sequential programs.

2.1 VLSI Complexity

VLSI complexity models represents computations as circuits embedded on few planar layers. A computation is specified by a collection of fixed logic gates connected with wires embedded on a planar VLSI substrate. All wires have the same width and can only be embedded along the rectangular axes with 90° bends. Any node with k wires connecting to it must at least k times as wide as the wires along both the axes. The embedding can not map more than a constant number of components to the same location as VLSI technologies can handle only a small number of layers. In addition, the computation also specifies the order and the placement of the input and output data on the VLSI chip.

Each VLSI technology has its own limitations on the minimum width of the wires and the signal propagation time. This places a limit on the minimum width of the wires, as well the maximum length of the wire than can successfully carry transfer a signal in a specified amount of time. In general, making the wire narrower, while reducing the area of the chip, also increases signal propagation rate and reduces the clock speed. Wires can not be made infinitely narrow and contribute to the cost of the design. Therefore, an embedding with fewer and shorter wires, i.e. smaller footprint, is better for performance.

Several variants of the model with differences in the location and timing of input and output have been proposed [124, 129, 130, 160]. In general, the objective is to design a circuit for a problem and an embedding of the circuit on a plane that minimizes the total area A of the design and the number of clock cycles to complete the computation T, assuming that a wire can carry one bit of information per clock cycle. If we assume that gates have constant fan-in and fan-out, the area of the chip strongly correlates to the total length of the wires. Models in which signal propagation time is proportional to the length of the wire have also been studied [59].

Lower Bounds. The central question in VLSI complexity is the tradeoff between A and T. Most results assume that the design is when and where oblivious, i.e. the location and the timing of the inputs and outputs are oblivious to the values of the data. The complexity of a problem is presented as a lower bound on the product AT^α for some α in terms of the size of the input instance N. Results for two dimensional circuits are often presented with α set to 2. For example, for comparison-based sorting, the first lower bound presented was $AT^2 = \Omega(N^2)$ [167], improved to $\Omega(N^2 \log N)$ by Thompson and Angluin and finally to $\Omega(N^2 \log^2 N)$ by Leighton [119]. The complexity of multiplying two binary number of length N was demonstrated to be $AT^2 = \Omega(N^2)$ by Abelson and Andreae [1] which was generalized by Brent and Kung

[53] to $AT^{2\alpha} = \Omega(N^{1+\alpha})$. Thompson [157] established the complexity of Discrete Fourier Transforms to be $AT^{\alpha} = \Omega(N^{1+\frac{\alpha}{2}})$. Savage [145] established the bounds for matrix multiplication: $AT^2 = \Omega((mp)^2)$ for multiplying matrices of dimensions $m \times n$ and $n \times p$ when $(a-n)(b-n) < n^2/2$ where $a = \max\{n,p\}, b = \max\{n,m\}$. Savage's results can be easily extended to matrix inversion and transitive closure.

Lower Bound Techniques. Most of the proofs for lower bounds follow a two-step template.

- Establishing an information transfer lower bound for the problem.
- Demonstrate that any embedding circuit transferring the established amount of information requires a certain area and time.

The first step involves finding a physical partition of the input and output wires on the VLSI and establishing that any circuit for the problem needs to transfer a minimum amount of "information" across the partition. For instance, Leighton establishes that for a comparison-based sorting, any partition of the chip that separates the output wires in half requires $\Omega(N \log N)$ bits of information across it (Theorem 4 of [119]).

The next step establishes the relation between the bisection bandwidth of the circuit graph and the area required to embed it. Theorem 1 of Thompson [157] demonstrates that any circuit which requires the removal of at least ω edges to bisect a subset of vertices (say the output nodes) requires $\Omega(\omega^2)$ area. Since it is not possible to transfer more than ω bits per cycle of information across the minimum cut, at least I/ω steps are needed to transfer I amount of information. Put together, these two imply $AT^2 = \Omega(I^2)$. Plugging in bounds for I in this inequality results in lower bounds for the complexity of the problem.

Approaches to establishing lower bounds on "information transfer" in the first step vary. Most approaches are encapsulated by the crossing sequence argument of Lipton and Sedgewick [124] and the information-theoretic approach of Yao [172]. In the crossing sequence argument of [124], the circuit is bisected (in terms of area, input, output or any convenient measure) and the number of different sequences that cross the wires and gates cut by the bisection during the computation is determined. Lower bounds are established by arguing that a minimum number of wires and time is required to support the required number of distinct crossing sequences for the input space. In [172], the left and right halves of the chip are treated as two players with inputs l and r respectively playing a game (as in [173]) to compute a function $f(l,r)$ by exchanging the minimum amount of information, which is used to establish the bounds. These techniques are compared in Aho, Ullman and Yanakakis [9] with suggestions for techniques to obtain stronger bounds.

A peculiar property of VLSI complexity is its non-composability [172]. While area and time-efficient circuits may be constructed to evaluate two composable functions f and g individually, lining up the inputs of one to the output of the other may not be easy. In fact, functions F and G exist such that the the AT^2 complexity of the composition $F \circ G$ is asymptotically greater than that of F and G.

Upper Bounds Good circuit designs for most problems mentioned above exist. Thompson's thesis presents circuits for Fourier Transforms and Sorting. Several designs with various tradeoffs are presented for FFT in [159], and sorting in [158]. Leighton presents sorting circuits that match lower bounds in [119]. Brent and Kung [53] present binary multiplication circuits with complexity $AT^{2\alpha} = O(n^{1+\alpha} \log^{l+2\alpha} n)$. Kung and Leiserson present an optimal design [111] for matrix multiplication. [110] surveys and categorizes many of these algorithms.

To place some of these results in perspective, let us consider the lower bounds for matrix multiplication and contrast two circuit designs for the problem with different degrees of locality. For multiplying two $n \times n$ matrices A and B using the straight-forward $O(n^3)$ matrix multiplication, a circuit must have complexity $AT^2 = \Omega(n^4)$. Among the two circuit designs we consider, one exposes the full parallelism of matrix multiplication ($T = O(\log n)$) for poor locality while the other trades off some parallelism ($T = O(n)$) for greater locality.

To compute the product in $T = O(\log n)$ steps would require a circuit that is a three-dimensional mesh of trees, each node of the mesh performing a multiplication operation and each node in the tree performing an addition operation. The input matrices would be plugged in at two non-parallel faces, say A at the $x = 0$ face and B at the $y = 0$ face on a mesh aligned along the 3D-axis at origin. In the first phase, the trees along the mesh lines perpendicular to $x = 0$ face copy the element in A input on one of their leaf nodes to all of their leaf nodes. Therefore, A_{jk} initially placed at $(0, j, k)$ would be copied to each of the grid nodes $\{i, j, k\}$ for $i \in \{0, 1, \ldots, n-1\}$ in the first phase in $O(logn)$ steps. B is similarly copied along the y-axis. After this phase, the elements at each grid element are point-wise multiplied and reduced using the trees along the z-axis in another $O(\log n)$ steps. This circuit design has a bisection bandwidth of $\Omega(n^2)$ and thus needs n^4 wire length when stretched out on to a few planar layers (following Thompson's theorem that relates the bisection bandwidth of a graph and the area of its planar embeddings [157]). The AT^2 cost of this design is $O(n^4 \log^2 n)$ which is close to the lower bound. This design realizes the full extent of the parallelism while paying a heavy price in terms of locality – wire lengths are $O(n)$ on average.

Another circuit design for the problem uses a two-dimensional hexagonal array design with n nodes along each edge that solves the problem in $O(n)$ time steps [111] (see Figures 2 and 3 of [110] for an illustration). It may be possible to roughly view this design as a two-dimensional projection of the 3D-mesh on to a plane perpendicular to its long diagonal and each time step of the hexagonal array design as a slice of the 3D-mesh parallel to the projected plane). The hexagonal array design is already two-dimensional and is thus embeddable with only constant length wires and thus has $A = O(n^2)$. By trading off some parallelism, this design achieves a significant improvement in locality for multiplying matrices. This solution also achieves the optimal value for AT^2.

Finally, consider the comparison sorting problem which requires $AT^2 = \Omega(n^2)$. Any poly-logarithmic time solution to the problem needs a circuit with total wire length $O(n^2/polylog(n))$ for $O(n/polylog(n))$ gates which corresponds to average length wire $O(n/polylog(n))$. This implies that solutions to this problem need relatively large bisection bandwidth. Circuits for this problem like the $O(\log^2 n)$ time shuffle-exchange networks [25] or the $O(\log n)$ time AKS sorting network [10] do indeed have good expansion properties and large bisection bandwidths.

2.2 Fixed-Connection Networks

Parallel machines built for high-performance computing span several VLSI chips. They are composed of multiple nodes, each node containing a general purpose processor, memory, and a packet router. Once fabricated, the topology of the network between the nodes is not easily changed. The cost of an algorithm on these machines is measured by the number of steps and the number of processing elements required. The maximum number of processors that can be used for problem of size N while being work-efficient can be used as an indicator of the parallelizability of the algorithm on the machine. The average number of hops that messages of the algorithm travel over the network can be used as an indicator of the locality of the algorithm with respect to the machine.

A good network should be capable of supporting efficient and fast algorithms for a variety of problems. Some criteria for a good network are large bisection bandwidth, low diameter, and good expansion. Alternatively, the number of steps required for a permutation operation – sending one packet from every processor to another processor chosen from a random permutation – is a good measure of the quality of the network.

Commonly used network topologies include meshes and the closely related tori (e.g. Cray XC, IBM Bluegene), mesh of trees, hypercubes (e.g. CM-1,CM-2) and the closely related hypercubic networks – Butterfly and Shuffle-Exchange networks, and fat-trees (e.g. CM-5). Leighton [116] presents a survey of these topologies and algorithms for them. The relative merits of the networks have been extensively studied. For example, hypercube and hypercubic networks are capable emulating many topologies such as meshes or a mesh of trees very effectively. But, on the downside, hypercubes have a larger degree than other nodes and are difficult to embed in two or three dimensions. Among networks that can be embedded in a fixed amount of *volume* in constant number of dimensions (eg. 2,3), it has been demonstrated that fat-trees are capable of emulating any other network with only a polylogarithmic slowdown [121]. Therefore, optimal algorithms for fat-trees achieve the best performance to within a polylogarithmic factor for circuits of a given *volume*.

2.2.1 Routing on Networks

While it is easy to design algorithms with one-hop data transfers for special-purpose networks — a 3D-mesh for Matrix Multiplication for example — algorithms for general-purpose networks need multi-hop packet routing and are harder to program. Ideally, the programmer of these distributed memory machines is provided a shared memory interface. Any piece of data is routed on-demand to the node requesting it.

Another common scenario that requires multi-hop routing is the emulation of a guest network on the host network by embedding the guest network on the host. In addition to a good embedding, supporting the guest network requires a good routing policy.

Consequently, a central question in the design of fixed-connection networks is the design of an online distributed routing scheme that can support data transfer between instructions across nodes with small buffers. The quality of the scheme is usually quantified in terms of steps required to route a h-relation – a packet routing request that involves at most h packets originating or terminating at any node. A permutation operation is a 1-relation.

Valiant and Brebner [165] were the first to formalize this question. They presented a randomized $O(h + \log N)$-step routing scheme for an h-relation on N-node hypercubes with $O(\log N)$-size buffers at each node. Building on this, routing schemes for shuffle-exchange and butterfly networks were constructed [11, 162]. Pippenger [138] improved the result for Butterfly networks by using constant size buffers.

These schemes enable efficient emulation of shared memories on top of distributed memory machines by presenting a uniform address space with low asymptotic access costs for every location. Karlin and Upfal [105] presented the first $O(\log N)$-step shared memory emulation on butterfly networks with $O(\log N)$ size priority queue buffers that supports the retrieval and storage of N data items. Further improvements culminated in Ranade's $O(\log N)$-step emulation with $O(1)$ size FIFO queue emulations [142].

All of the results above deal with routing on specific networks. Leighton, Maggs and Rao [118] addressed the problem of routing on *any* network. Central to the solution of this problem is a routing schedule that schedules transfer of packets between source-destination pairs along specified paths using small buffer at each node. The general routing problem can be reduced to finding a routing schedule along specific paths using ideas of random intermediate nodes from Valiant [165]. For an instance of the routing problem where any pair of source-destination nodes are separated by d hops (dilation) and any edge is used by at most c paths (congestion), Leighton, Maggs and Rao [118] proved the existence of an optimal online routing strategy that takes at most $O(c + d)$ steps with high probability of success. This result was used to design a efficient routing schemes for many fixed-connection networks in [117].

2.3 Abstract Models for Distributed Memory Machines

Fixed-connection networks require very low-level specifications such as routing, and programs written for one machine are not very portable. The Bulk-Synchronous Parallel (BSP) [163] and the LogP [70] models aim to provide a higher level cost model for programs by abstracting away the network and routing details.

2.3.1 BSP and LogP Models

In the BSP model, programs are written for a certain number of processing nodes without the necessity to deal with network details. Programs proceed in phases. In each phase, each processor executes its assigned set of instructions and can access local memory with out paying any cost. The phase ends when all processors are done with their instructions. At this point, nodes exchange one-sided messages in preparation for the next round. Network and synchronization costs are modeled as numerical parameters. The cost of exchanging an h-relation is gh, where g is network dependent. The cost of synchronization is l. The duration of a phase is

$$\max_{p \in P} \{w_p + gh_p + l\},$$

where w_p and h_p are the number of instructions executed and the number of messages sent or received at processor p respectively.

The aims of the LogP model are similar, although the model is asynchronous. Instead of waiting for phase boundaries for communication as in the BSP model, processors in the LogP model can request remote data at any time and use them as soon as they are made available. The cost parameters in the LogP model are (i) an upper bound on the latency of a message: L, (ii) overhead of sending or receiving a message: o (during which processor is inactive), (iii) the gap between two consecutive message transmissions: g (inverse of the bandwidth) and (iv) the number of processors: P. The cost of a program is the maximum of the sum of costs of all instructions and messages at any node. LogGP model [12] is a refinement of the LogP model that distiniguishes between the effective bandwidth available to small (parameter g) and large messages (parameter G).

Distributed-RAM Algorithms Another way to characterize the complexity of algorithms on fixed-connection networks is by measuring the cost of their communication across cuts in the network [121]. An algorithm that minimizes congestion — ratio of communication to cut size — across every cut in the network would make efficient use of the network. Leiserson and Maggs [122] consider the design of such data-structures and *conservative* parallel algorithms on them. Conservative algorithms are those algorithms that generate no more congestion across any cut that the congestion of the pointers in the input data-structures acrosss the same cut. Specficially they consider the design and embedding of a data-structure that supports conservative algorithms for basic list operations – random-mate and pointer jumping. They use this to design conservative tree contraction and other graph algorithms.

2.3.2 Communication Avoiding Algorithms

Inspired by the LogP model, a number of algorithms haven been adapted into or designed as *communication-avoiding* algorithms [74]. Communication-avoiding algorithms target distributed memory machines and are designed to minimize the maximum over all processors a weighted sum of (i) number of instructions multiplied time per processor clock-cycle (ii) number of messages multiplied by the network latency, and (iii) the sum of sizes of the messages multiplied by the inverse bandwidth available to each node. These measures directly quantify locality. Parallelism of a class of algorithms can be characterized by the maximum number of processors that can be supported with out asymptotic increase in communication costs. A few samples of optimal parallel algorithms analyzed in this cost model are QR and LU factorizations in [75] and the 2.5D matrix multiplication in [151].

An useful technique used to save bandwidth by the algorithms in this model is data replication. When multiple nodes simultaneously need the same variable, multiple copies are kept locally at each node at the cost of using extra memory. Solomonik and Demmel [151] consider this tradeoff in the context of matrix multiplication and explore the continuum between "2D" multiplication which uses exactly one copy of the matrices and "3D" multiplication where processors are logically arranged into a 3D-array with redundant copies at each layer. They demonstrate a continuum of *fractional-dimension* algorithms that take advantage of extra local memory to decrease number of words transmitted between processors while paying extra in terms of the number of messages.

Proofs of lower bounds on the communication costs in this model relate closely to correspondence between graph partitioning problem and communication complexity explored in Pebbling Games [102]. For example, Ballard et al. [24] demonstrate a lower bound on the number of edges required to cut any Strassen-style computation DAG into chunks that use at most M words (M being the size of local memory on each node), and relate the cut size to minimum communication cost. Irony et al. [99] use similar techniques to establish lower bounds for the standard matrix multiplication with various degrees of replication. Ballard's thesis [23] presents more comprehensive survey of lower bounds for communication-avoiding algorithms.

2.3.3 Network-Oblivious Model

Both BSP and LogP model locality by differentiating between local and remote data. While this provides a good first-order approximation, this distinction is not sufficiently refined for large machines where there are several levels of caches and memory. The network-oblivious model [32] fills this gap by modeling locality for distributed memory machines with multi-level hierarchies. An adaptation of the BSP model called the multi-BSP for shared memory machines with multi-level hierarchies is described in Section 2.6.

In the network-oblivious model, the algorithm is designed for a certain number of processing elements depending on its input size. The number of processing elements chosen (n) reflects the parallelism of the algorithm. To quantify the communication cost of the algorithm, the algorithm is then mapped onto a simple abstract machine model to measure its communication cost. The abstract machine model consists of p processors with unbounded local memory on a complete graph which communicate with blocks of size B. The processing elements that the algorithm is designed for are now mapped onto p processors. The algorithm is allowed to use $\log n$ supersteps. In each superstep, each of the p processors execute the local computation corresponding to the processing elements assigned to them and communicate with other processors at the end of the superstep. The sum of the maximum communication used by any of the p processors across all supersteps is the communication cost of the program. An optimal algorithm would optimize the communication for all values of p and B. The machine is then modeled as a hierarchy of BSP models referred to as the D-BSP model [161]. The algorithm is mapped to the machine model by using the best recursive blocking strategy for the processing elements at every level in the hierarchy. An algorithm that can be efficiently mapped to any value of p and B in the abstract machine model does well on the D-BSP model.

2.3.4 Cost Models for MapReduce

The MapReduce is a slightly restrictive programming model suited for indexing and related operations [71] on simple distributed memory machines . It takes as input a list of key-value pairs and supports two types of operations — Map, which maps each key-value pair to a set of pairs and Reduce, which collects all pairs with the same key and applies *reduces* this set to generate a new set of key-value pairs. A computation involves multiple rounds each consisting of exactly one Map and one Reduce operation. The output of a round is fed as input to the next round. Implementations that realize the MapReduce framework such as Hadoop [168] handle the details of mapping the program to the machine such as shuffling keys to get them to the right place (routing), assigning Map and Reduce operations to processors (scheduling), load balancing and fault tolerance.

The number of rounds of computation and the cost of each round determine the complexity of a MapReduce program just as in the BSP model. Given the relatively low bisection bandwidths of Ethernet-based clusters that commonly run the MapReduce framework, emphasis is often placed on minimizing the number of computation rounds at the expense of greater data replication and total number of instructions. Karloff, Suri and Vassilvitskii [106] present a simple program-centric cost model to guide algorithm design with these considerations. In their model, each Map operation on a key-value pair is allowed to take sublinear ($O(n^{1-\varepsilon})$) and space, where n is the length of the input to the round. Similarly, each Reduce operation on a collection of key-value pairs with the same key is allowed similar time and space budget. The output of each round must fit within $O(n^{2-2\varepsilon})$ space. The MapReduce class consists of those problems that admit an algorithm with a polylogarithmic number of rounds (as a function of the input size). Algorithms made of Map and Reduce operations with sublinear time and space can be mapped to clusters made of machines with small memories.

This particular model for MapReduce ignores the variation in the cost of shuffling data across the interconnect in each round and differentiates between algorithms only in terms of their round complexity. More quantitative studies relate communication costs to the replication rate – the number of key-value pairs that an input pair produces, and parallelism to the reducer size – the number of inputs that map to a reducer. Since communication costs increase with replication rates, and parallelism decreases with increase in reducer size, a good MapReduce algorithm would minimize both replication rates and reducer sizes. Trade-offs between the two for several problems are presented by Afrati et al. [5]. As in the case of other cost models, a notion of *minimal* algorithms that achieve optimal total communication, space, instructions, and parallel time (rounds of load-balanced computation) is formalized by Tao et al. [155]. They present algorithms for sorting and sliding aggregates. The sorting algorithm closely resembles sample sort which has been shown to be optimal in many other models of locality [6, 40].

2.4 Pebbling Games

Pebbling games were introduced in [95] to study the relation between the time and space complexity of Turing machines. Hong and Kung [102] modified the game to model I/O complexity – the number of transfers between a processor with a limited amount of fast memory and a larger but slower memory. The pebbling game of Hong and Kung involves two kind of pebbles – blue and red pebbles – to represent data in slow and fast memory. The red pebbles are limited in number while there is ann unlimited supply of the blue nodes. The red-blue pebbling game is played on the computation DAG. The game starts with a blue pebble placed on the starting nodes of the DAG – nodes with no incoming edges, and should end with blue pebbles on all the output nodes – nodes with no outgoing edges.

Load: A blue pebble on any node can be replaced with a red pebble. This correponds to moving the data from slow memory to fast memory.

Compute: If all the immediate precedessors of a node x are pebbled red, then a red pebble may be placed on x. In other words, if all the inputs required for x are in fast memory, node x can be computed.

Store: A red pebble on a node can be replaced with a blue pebble. This corresponds to evicting the data at the node from the fast memory and storing it in the slow memory.

A valid pebbling games corresponds to a valid execution of the DAG. The number of transitions between red and blue pebbles correponds to the number of transfers between fast and slow memories. The pebbling game that makes the least number of red-blue transitions with M red pebbles gives the I/O complexity of the DAG with a fast memory of size M.

Lower Bounds. Lower bounds on the I/O complexity are established using the following tight correspondence between red-blue pebbling games and M-paritions of the DAG. Define a M-partition of the DAG to be a partition of the nodes in to disjoint partitions such that (i) the *in*-flow into each partition is $\leq M$, (ii) the *out*-flow out of each partition is $\leq M$, and (iii) the contracted graph obtained by replacing the partitions with a supernode is acyclic. Then any pebbling game with M red pebbles and Q transfers can be related to a $2M$-partition of the DAG into h partitions with $Mh \geq Q \geq M(h-1)$. Using this technique, [102] establishes the I/O complexity of the FFT graph on n elements to be $\Omega(n \log_M n)$. [102] also establishes other techniques to prove lower bounds based on number of vertex-disjoint paths between pairs of nodes. Note that lower bounds established in this model correspond to the algorithms and not the problem.

Parallelism. A major drawback of pebbling games is that they do not model the cost of load balance or parallelism. The description does not specify whether the game is played sequentially or in parallel. Although the usual description of the game corresponds to a sequential execution, natural extensions of the games can model parallel executions. Sequential pebbling games correspond to I/O complexity in the sequential models in Section 2.5. Parallel executions of the DAG on a shared cache of size M (Section 2.6.1) correspond to simultaneous pebbling across multiple nodes in the DAG with a common pool of M red pebbles. Parallel executions of the DAG on the PEM model (Section 2.6.1) with p processors each with a M-size private cache correspond to a pebbling games with p different subtypes of red pebbles numbering M each. The compute rule of the pebbling game would be modified to allow a node to pebbled red with subtype j only if all its immediate predecessors have been pebbled red with the same subtype j.

Multi-level Pebbling Games An an extension of pebbling game to hierarchies called the Memory Hiererchy Game (MHG) has been suggested [146]. In a MHG, there are L different pebble types corresponding to each level in a L-level memory hierarchy. The number of pebbles of type $j < L$ are limited and increase with j, their posistion in hierarchy. Computation can only be performed on a node with predecessors pebbled with type-1 pebble corresponding to the fastest level in the hierarchy. Transitions are allowed between adjacent types, and correspond to transfers between adjacent levels in the hierarchy. I/O complexity at level i relates to the number of transitions between level-$(i-1)$ and level-i pebbles. Complexity in this pebbling game relates quite closely to cache complexity in the Cache-Oblivious framework applied to multi-level hierarchy (Section 2.5.2).

2.5 Models for Sequential Algorithms

This section surveys cost models for locality of sequential algorithms. We start with the one-level External Memory Model of Aggarwal and Vitter [6], and move to the program-centric Cache-Oblivious framework and the ideal cache model [86] that can be used for capturing costs on multi-level hierarchies. We finally describe explicitly multi-level cost models such as the Hierarchical Memory Model [7].

2.5.1 External-Memory Model

The external-memory model consists of a processor, a fast cache of size M organized into contiguous blocks of size B, and a slower external memory of unbounded size also organized into blocks of size B. Transfers between cache and the memory are always performed in blocks. The original model also allowed for multiple external memory units and simultaneous transfers between these units and the cache. Simplifying the model by using only one external memory does not change the model or the complexities of the problems too much. Algorithms explicitly manage transfers between cache and the memory. The locality of an algorithm is measured by its cache complexity – the number of transfers between cache and memory. This model captures both the spatial and temporal locality of the algorithm in terms of parameters B and M respectively.

Lower bounds in this model include $\Omega\left(\frac{n}{B}\right)$ for scanning n numbers, and $\Omega\left(\left\lceil \frac{n}{B}\right\rceil \log_{2+M/B}\left\lceil \frac{n}{B}\right\rceil\right)$ for sorting or permuting n numbers. Examples of external-memory algorithms and data-structures have been designed for sorting, FFT [6], buffer trees [16], geometric data structures [17] and graph algorithms [60, 174]. A good survey of External-Memory algorithms is presented in the textbooks of Vitter [166] and Arge and Zeh [18].

2.5.2 Ideal Cache Model and Cache-Oblivious Algorithms

The External-Memory Model is too low-level and machine-centric, requiring the algorithm to explicitly handle cache transfers and allowing the algorithm to be designed with knowledge of the cache sizes. An algorithm designed for one cache size may not be portable to a different cache size, and may have significantly deteriorate in performance even if ported. For greater robustness across caches, an algorithm should be designed to be optimal for any cache size without the knowledge of the cache size. Ideally, I/O complexity should degrade smoothly with decreasing cache size. Further, transfers should between caches and transfers should be automatically handled by the machine and not the algorithm.

The cache-oblivious framework presents this level of abstraction to algorithm designers. The machine is modeled as a processor connected to an *ideal* cache and an infinite memory. The cache is ideal in the sense that it is fully associative and uses the optimal offline farthest-in-the-future replacement policy (or the more practical yet competitive online least-recently-used policy [150]). This allows for algorithms to be specified at a high level – data elements can be accessed by their address without referring to caches. Algorithms are designed with out knowledge of cache size. Cache complexity in this model $Q(M, B)$ is the number of transfers generated by the automatic replacement policy between the cache with parameters M, B and the memory.

The ideal cache assumptions make for an easy and portable algorithm analysis. Many of the External-Memory algorithms can be adapted into this model, while some problems need newer design. Demaine [72] presents a survey of some cache-oblivious algorithms and data-structures designed for the ideal cache model and their analyses.

Multi-level Hierarchies. Since algorithms are oblivious to cache sizes, the analysis for the algorithms is valid for multi-level cache hierarchies as well. On a h-level hierarchy with caches of size $M_i, i = 1, 2, \ldots, h$, the algorithm incurs $Q(M_i, B)$ cache misses at level i.

Limitations. Some problems seem to admit better External-Memory algorithms than Oblivious algorithms by virtue of the cache size awareness. For example, optimality of cache complexity for many algorithms in the CO model depends on the *tall-cache assumption* – $M \leq B^{\varepsilon}$ for some $\varepsilon > 1$, usually 2. Brodal and Fagerberg [58] show that this assumption is in fact required for optimal cache-oblivious algorithms to some problems such as sorting. Further, they prove that even with the tall-cache assumption, permutations can not be generated optimally in the Cache-Oblivious model.

Use for Parallel Algorithms Although the ideal cache model (and cache-oblivious framework) has been described for sequential algorithms, it can be used to study cache complexities of parallel algorithms by imposing a sequential order on parallel programs. In particular, sequential cache complexity in the ideal cache model under the depth-first order Q_1 is a useful quantifier of locality in nested-parallel programs. See section 5.2 for more details.

2.5.3 Hierarchical Memory Models

The Hierarchical Machine Model (HMM) [7] is a machine-centric cost model in which there is only one memory, but of non-uniform costs. The cost of accessing index x is monotonically increasing function of x, say $\log x$ or x^{α} for example. Algorithms must move frequently used data to locations with lower indices. Cache-oblivious algorithms can be adapted for this model by partitioning the memory into segments with geometrically increasing accessing costs (say factor of 2) and treating the progression of segments as a multi-level cache hierarchy. An extension of the HMM which allows for blocked transfers, along with several optimal algorithms for the $\log x$ and x^{α} access cost profiles, is presented by Agarwal et al. [8]. Another model for memory hierarchies is the Uniform Memory Hierarchy model which models the machine as a sequence of discrete caches [14] with uniform access cost across each cache.

2.6 Models for Shared Memory

Shared memory parallel machines present a single continuously addressable memory address space on top of many memory units. Contemporary examples include the Intel Nehalem and Sandy/Ivy-Bridge architectures with QPI, Intel Xeon Phi, IBM Z system, IBM POWER blades, SGI Altix series and Intel MIC. The simpler interface makes them a popular choice for most programmers as they are easier to program than distributed systems and GPUs. The semantics of concurrent accesses to the shared address space [3] are specified by memory consistency policy.

2.6.1 Parallel External-Memory Model and Shared Cache Model

Models for shared memory machines with one cache level include the Parallel External Memory model [20] and the Shared Cache model [37]. Both these models include a large unified shared memory and multiple processors, but differ in the arrangement of caches (Figure 1.2). While the earlier model assigns each processor its own cache, the latter forces all processors to share one cache.

The cache complexity, i.e. number of transfers between cache and memory, of parallel programs in this model depends on the order of execution of the instructions. A scheduler capable of preserving locality in the program while load balancing the processors is necessary for mapping a portable program on to these machine models. The PEM and the share cache models require different scheduler designs. The special case of scheduling nested parallel programs on these models is well understood. Briefly put, a program-centric analysis of the parallel program for work (number of instructions), depth (length of the critical path, also known as span) and the depth-first sequential cache-complexity in the ideal cache model is sufficient to predict program performance in these two models [37, 49] (Section 4.2). Schedulers with provable bounds on performance in terms of these parameters are surveyed in Section 5.2.

2.6.2 Multi-BSP and HBP Models

Models with multiple levels of caches include the Multi-BSP [164], the Hierarchical Balanced Parallel model [68] and the Parallel Memory Hierarchy model [13]. All of these models are built for a symmetric tree of caches but differ in their approaches.

The Multi-BSP is an extension of the BSP model for a symmetric hierarchy of caches. It is a synchronous model like the BSP model. A h-level hierarchy is modeled by $4h$ parameters $\{p_i, m_i, g_i, l_i\}_{i=1}^{h}$ representing the fan-out, cache size, data rate (bandwidth) and the synchronization cost at each level in the hierarchy. Perfectly balanced divide and conquer algorithms are mapped onto the hierarchy by recursively blocking them at sizes corresponding to the cache sizes. A block of size m_i is mapped onto a level-i cache and forms one level-i super step. The block is subdivided into parts that fit in level-$(i-1)$ caches. A level-i super-step consists of multiple synchronizations of level-$(i-1)$ super-steps. Mapping algorithms with even a small degree of imbalance in their divide and conquer recursion is not considered in this model.

Cole and Ramachandran [68] take a more program-centric approach to designing algorithms for multi-level hierarchies. A notion of perfect balance in computations is formalized as Hierarchical Balanced Parallel (HBP) computations as follows. A *Balanced Parallel* computation is a divide and conquer algorithm in which the computation must fork into two exactly equal parts (in terms of space and number of instructions) after accessing $O(1)$ locations or using $O(1)$ instructions. HBP computations are defined inductively. A sequential computation is a type-0 HBP. A BP computation is a type-1 HBP. A type-i HBP is a recursive divide and conquer computation in which each divide operation is perfectly balanced and invoked only HBP computation of type-$< i$. They present a scheduler to map these perfectly balanced computations onto a hierarchy with good bounds. Scheduling even slightly imbalanced computations is not considered in their model.

2.7 Shared Memory Machines as Parallel Memory Hierarchies

A good machine model succeeds at being a realistic representation of the machine while abstracting away the low level details. A realistic model for a variety of shared memory machines is the multi-level Parallel Memory Hierarchy [13, 38, 40, 62, 63, 64, 164] in which parallel machines are modeled using the tree-of-caches abstraction. For concreteness, we use a symmetric variant of the parallel memory hierarchy (PMH) model [13] (see Fig. 2.1), which is consistent with many other models [38, 40, 62, 63, 64].

The Parallel Memory Hierarchy (PMH) model describes the machine as a symmetric tree of caches with the memory at the root and the processors at the leaves. Each internal node is cache – the closer the cache is to the processor in the tree, the faster and smaller it is. This model captures the parallelism of the machine with multiple processors, and the hierarchical nature of locality on the machine with multiple levels of caches. Locality of a program is measured in terms of the number of cache misses at each level in the hierarchy, and parallelism in terms of program completion time.

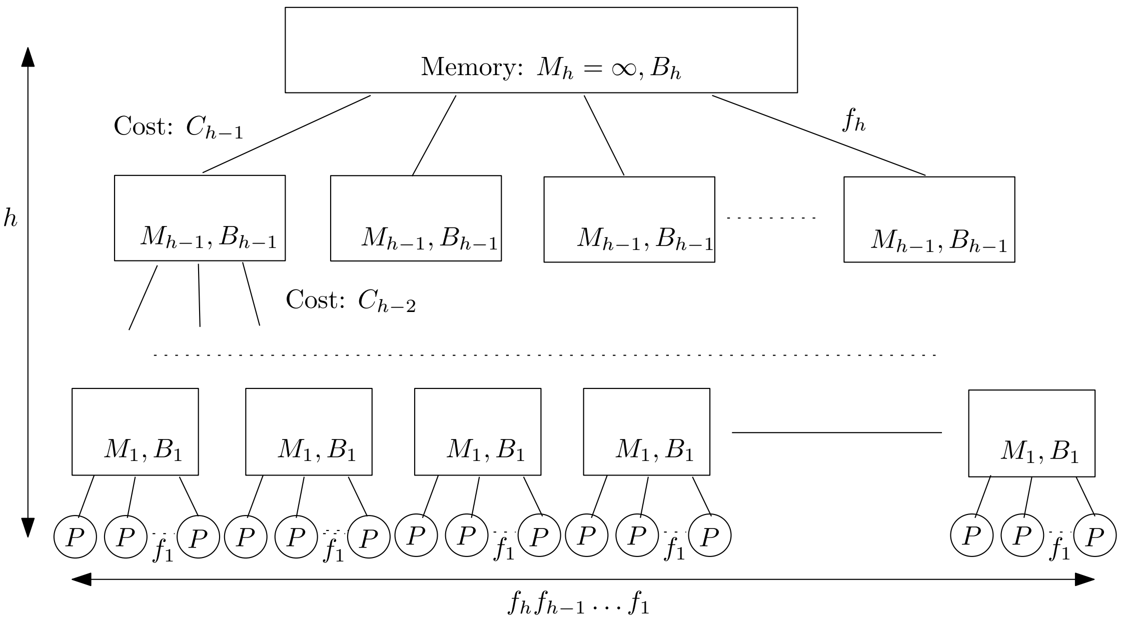


Figure 2.1: A Parallel Memory Hierarchy (replicated from figure 7.1).

This section presents the details of an adaptation of the PMH model from [13] that we refer to in the next few chapters.

2.7.1 Parameters

A PMH consists of a height-h tree of memory units, called ***caches*** which store copies of elements from the main memory. The leaves of the tree are at level-0 and any internal node has level one greater than its children. The leaves (level-0 nodes) are processors, and the level-h root corresponds to an infinitely large main memory.

Each level in the tree is parameterized by four parameters: M_i, B_i, C_i, and f_i.

- The ***capacity*** of each level-i cache is denoted by M_i.
- Memory transfers between a cache and its child occur at the granularity of ***cache lines***. We use $B_i \geq 1$ to denote the ***line size*** of a level-i cache, or the size of contiguous data transferred from a level-$(i+1)$ cache to its level-i child.
- If a processor accesses data that is not resident in its level-1 cache, a level-1 ***cache miss*** occurs. More generally, a ***level-$(i+1)$ cache miss*** occurs whenever a level-i cache miss occurs and the requested line is not resident in the parent level-$(i+1)$ cache; once the data becomes resident in the level-$(i+1)$ cache, a level-i cache request may be serviced by loading the size-B_{i+1} line into the level-i cache. The cost of a level-i cache miss is denoted by $C_i \geq 1$, where this cost represents the amount of time to load the corresponding line into the level-i cache under full load. Thus, C_i models both the latency and the bandwidth constraints of the system (whichever is worse under full load). The cost of an access at a processor that misses at all levels up to and including level-j is thus $C'_j = \sum_{i=0}^{j} C_i$.
- We use $f_i \geq 1$ to denote the number of level-$(i-1)$ caches below a single level-i cache, also called the ***fanout***.

The miss cost C_h and line size B_h are not defined for the root of the tree as there is no level-$(h+1)$ cache. The leaves (processors) have no capacity ($M_0 = 0$), and they have $B_0 = C_0 = 1$. Also, $B_i \geq B_{i-1}$ for $0 < i < h$. Finally, we call the entire subtree rooted at a level-i cache a ***level-i cluster***, and we call its child level-$(i-1)$ clusters ***subclusters***. We use $p_i = \prod_{j=1}^{i} f_j$ to denote the total number of processors in a level-i cluster.

2.7.2 Caches

2.7.2.1 Associativity

We assume that all the caches are fully associative just as in the ideal cache model [86]. In fully associative caches, a memory location from memory can be mapped to any cache line. In practice, hardware caches have limited associativity. In a 24-way associative cache such as in Intel Xeon X7560, each memory location can be mapped to one of only 24 locations. However, a fully associativity makes the machine model much simpler to work with without giving up on accuracy. Pathological access patterns may be dealt with using hashing techniques if needed.

2.7.2.2 Inclusivity

We do not assume inclusive caches, meaning that a memory location may be stored in a low-level cache without being stored at all ancestor caches. We can extend the model to support inclusive caches, but then we must assume larger cache sizes to accommodate the inclusion.

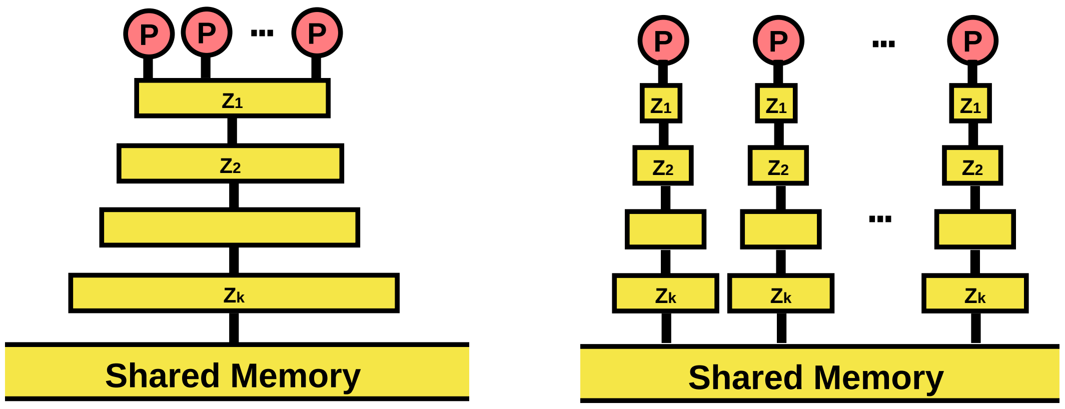


Figure 2.2: Multi-level shared and private cache machine models from [40].

We assume that the number of lines in any non-leaf cache is greater than the sums of the number of lines in all its immediate children, i.e. , $M_i/B_i \geq f_i M_{i-1}/B_{i-1}$ for $1 < i \leq h$, and $M_1/B_1 \geq f_1$. For all of our bounds to hold for inclusive caches, we would need $M_i \geq 2f_i M_{i-1}$.

2.7.2.3 Cache replacement policies

We assume that the cache has the online LRU (Least Recently Used) replacement policy. If a cache is used by only one processor, this replacement is competitive with the optimal offline strategy — furthest-in-the-future (FITF) — for any sequence of accesses [150]. However, the scenario is completely different when multiple processors share a cache [91]. Finding the optimal offline replacement cost where page misses have different costs (as in a multi-level hierarchy) is NP-complete. Moreover, LRU is not a competitive policy when a cache is shared among processors. Online replacement policies for shared caches that tend to work in practice [100, 101] and those with theoretical guarantees [107] have been studied.

Despite the lack of strong theoretical guarantees for LRU in general, it suffices for the results presented in Chapter 5.

2.7.3 Relation to other Machine Models

The PMH model directly corresponds to many contemporary machines such as workstations in the Intel Xeon series, AMD Opteron, and the IBM z and POWER series (see Figure 1.5). The PMH model generalizes other machine models. For instance the single-level shared and private cache models (Figure 1.2) are subclasses of the PMH class. In both the models, $h = 2$. In the private cache model, $f_1 = 1$ and $f_2 = p$, the number of processors. In the shared cache model, $f_1 = p$ and $f_2 = 1$.

It also generalizes other machine models including the multi-level shared and private models such as in Figurefig:multilevel. In the multi-level shared cache model, $f_h = f_{h-1} = \cdots = f_2 = 1$ and $f_1 = p$. In the multi-level private cache model, $f_h = p$ and $f_{h-1} = f_{h-2} = \cdots = f_1 = 1$.

Locality in parallel machines with topologies embeddable in finite and small number of dimensions may also be approximated with tree of caches. Loosely speaking, any arbitrary topology where the number of processors and the storage capacity of a subset of the machine depends only on the volume of the subset can be recursively bisected along each dimension, and each level of recursion mapped to a level in the hierarchy. For example, Figure 1.6 depicts a two-dimensional grid of size 4×4 and how it might be approximated with a tree of caches by recursive partitioning.

2.7.4 Memory Consistency Model

The easiest consistency model to program to is the sequential consistency model of Lamport [113]. In this memory consistency model, there exists some sequential order consistent with the execution order of instructions on each processor according to which all memory operations of the parallel program appear to take effect. However, it is inefficient for a hardware to support this. Weaker consistency models — models with fewer restrictions on what instructions observe — are necessary in practice [3].

We assume that the machine supports a relaxed program-centric consistency model called the location consistency model [83, 126]. In this consistency model, writes from a DAG to each memory location behave as if they serialized in some order consistent with the topological ordering in the DAG. This consistency model is a *stronger* and a *constructible* version of the dag-consistent models described in [49]. It is stronger in the sense that it permits fewer options for assigning writes to reads in the DAG, and constructuble in that it can supported by an online algorithm – writes can be assigned to reads in the DAG in a consistent manner irrespective of the order in which an adversary presents the nodes in the DAG. The location consistency model can be supported on shared memory with the BACKER algorithm [126] such as in the Cilk system [103]. Since our machine model involves lines with multiple locations, concurrent writes to a cache line are merged on writing back to memory thus avoiding "false sharing" issues that affect performance. We ignore the cost of maintaining memory coherence in this model.

2.8 Reuse Distance

Reuse distance analysis—also known as the LRU stack distance—was introduced as a program-centric way of measuring the temporal locality in sequential programs [30, 128]. The reuse distance of a reference r to a memory location l is the number of distinct memory locations accessed between the previous reference r' to location l and the current reference r. A histogram of reuse distances for all references in the program for a certain input can be obtained through a machine-independent program analysis. The histogram, which is program-centric, is sufficient for predicting the number of cache misses on any sequential memory hierarchy consisting of fully associative caches with LRU replacement policy and unit block size. Splitting the histogram in to two parts corresponding to reuses that are less that M apart and reuses that are greater than M apart gives the number of cache misses and cache hits the program will incur on a cache of size M. This mapping from program-centric analysis to performance on a machine is similar to mapping analyses of Cache-Oblivious algorithms to the ideal cache model [86].

In order to model spatial locality, the analysis must replace the distance metric between references by the number of memory (or cache) blocks of a certain size B instead of the number of locations [89]. However, requiring the block size parameter of a machine would make the analysis less program-centric. An alternative is to run an analysis for different values of B, say for powers of two, and store the histogram for each value of B. An approximate mapping from program analysis to performance on a machine can be done by selecting the histogram corresponding to a value of B that is most relevant to the machine. Reuse distance analysis has been used for automatically generating better loop fusions and loop tilings [77, 139].

Since reuse distance is analyzed with reference to a specific input, varying the input may not preserve the analysis. However, it may be possible to use training instances to predict data reuse pattern across the range of inputs for a family of programs [78].

Reuse distance requires non-trivial program profiling as it involves keeping track of information about the access time of every memory location. Low overhead algorithms and optimizations for efficient, yet robust, profiling of reuse distance at varying scales (loops, routines, program) has been studied in [78, 127, 175].

Other related metrics that are either equivalent or directly related to reuse distance such as miss-rate, inter-miss time, footprint, volume fill time etc. have been considered for their ease of collection or practicality. Quantitative relations between these measures were first considered in the Working Set Theory studies of Denning [76], and explored further towards building a "higher order theory" of locality [171].

A limitation of reuse distance analysis is that it does not seem to capture the parallelism of the program. Further, it is not inherently defined for parallel programs. However, it may be extended for estimating the communication costs of a parallel program by augmenting the program with extra information – imposing a sequential order such as in many shared cache models, or some notion of distance between references in parallel programs.

Chapter 3

Program Model

3.1 Directed Acyclic Graph

It is natural to represent executions of parallel programs as directed acyclic graphs. We use the representation used in section 2 of [135] by adding some additional terminology in Section 3.2. The nodes correspond to instructions and the edges correspond to precedence constraints between instructions. Each instruction operates on a constant number of variables. The mapping between variables and memory location is specified by the memory allocation policy. Section 3.3 for more details.

DAGs are allowed to unfold dynamically, i.e. parts of the graph need not be specified until their predecessors have been executed. However, they are required to be structurally deterministic for many of the results presented here — multiple executions of the DAG must unfold to the same DAG. The DAG should contain the same set of instructions (nodes) with the same precedence constraints. In case of randomized algorithms, all execution with the same input seed must unfold to the same DAG.

Other stronger forms of determinism such as internal determinism [44] have been proposed for parallel programs. These forms of determinism are useful for making it easier to reason about parallel program correctness and making programming more productive. Many of the algorithms in Chapter 6 satisfy these properties. However, such strong requirements on repeatability of variable values in each instruction are not required for the performance bounds we present for schedulers.

Throughout this thesis, when we refer to a parallel algorithm for a problem, we refer to a family of DAGs corresponding to the algorithm for different problem sizes. The DAG family is parameterized by input size. When a cost metric is described for DAGs, we often try to quantify the complexity of a DAG family with respect to the cost metric as a function of the input parameter. For example, the parallel Quicksort algorithm for the sorting problem has cache complexity $\mathbb{E}[Q^*(n; M, B)] = O(\lceil \frac{n}{B} \rceil (1 + \log \lceil \frac{n}{M+1} \rceil))$ and work complexity $W = O(n \log n)$ for an input of size n.

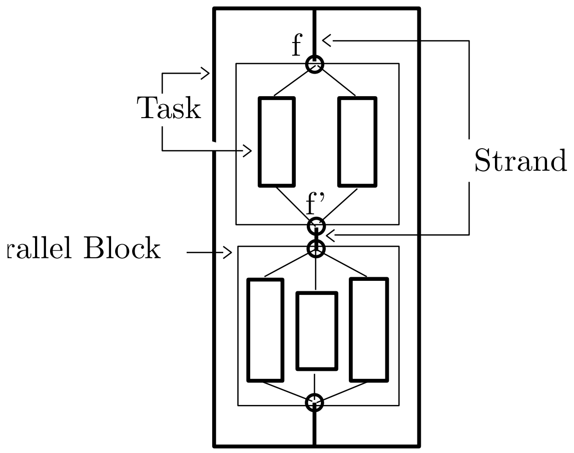


Figure 3.1: Decomposing the computation: tasks, strands and parallel blocks. The circles represent the fork and join points. f is a fork, and f' its corresponding join.

3.2 Nested-Parallel DAGs – Tasks, Parallel Blocks and Strands

Arbitrary DAGs are difficult to schedule and analyze. It is often quite difficult to reason about their properties or prove performance bounds on their executions. Therefore, this thesis considers only nested-parallel programs allowing arbitrary dynamic nesting of fork-join constructs but no other synchronizations. Fork-join constructs may be used to implement parallel loops. This corresponds to the class of algorithms with series-parallel dependence graphs (see Fig. 3.1).

Series-parallel DAGs can be decomposed into "tasks", "parallel blocks" and "strands" recursively as follows. As a base case, a ***strand*** is a serial sequence of instructions not containing any parallel constructs or subtasks. A ***task*** is formed by serially composing $k \geq 1$ strands interleaved with $(k-1)$ "parallel blocks" (denoted by $\mathsf{t} = \mathsf{s}_1; \mathsf{b}_1; \ldots; \mathsf{s}_k$). A ***parallel block*** is formed by composing in parallel one or more tasks with a fork point before all of them and a join point after (denoted by $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$). A parallel block can be, for example, a parallel loop or some constant number of recursive calls. We do allow a single task in a parallel block, and in the PCC framework, this is different from continuing with a strand. The top-level computation is a task.

The nested-parallel model assumes all strands share a single memory. We say two strands are ***concurrent*** if they are not ordered in the dependence graph. We assume that the program is free from data races [4, 87]. Concurrent reads (i.e. , concurrent strands reading the same memory location) are permitted, but not concurrent writes (i.e. , concurrent strands that read or write the same location with at least one write).

3.2.1 Properties of Nested-Parallel DAGs

Series-parallel DAGs corresponding to nested-parallel programs are simpler to analyze for several reasons. First, they can be partitioned naturally with the removal of just two nodes at various scales. Any corresponding pair of fork and join points separates the task nested within from the rest of the DAG. Apart from the small separator, there are other useful properties that makes tasks a natural point for making scheduling decisions.

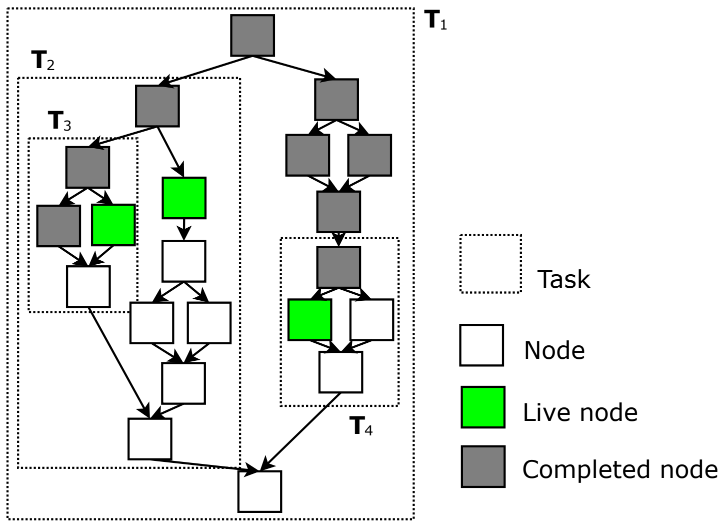


Figure 3.2: Tree-like property of live nodes: A set of live nodes in a nested-parallel DAG correspond to tasks that can be organized into a tree based on their nesting relation. In this case, there are 3 lives nodes corresponding to four tasks that can be naturally organized into the tree $(T_1\ (T_2\ (T_3)\ ())\ (T_4))$.

- **Tree-like.** Any feasible set of "live" nodes that are simultaneously executing (which forms an anti-chain of nodes in the DAG) will always be drawn from a set of tasks that can be organized into a tree based on their nesting relation (see figure 3.2.1).
- **Monotonicity.** The space footprint of task increases monotonically with levels of nesting: a task t_2 nested within a task t_1 can never access more distinct memory locations than t_1.

3.3 Memory Allocation Policy

Programs for shared memory machines are typically written in terms of variables in most high level programming models for shared memory machines. The mapping from variables to memory address space determines the memory location an instruction accesses. An extremely naive policy would map every variable to a different memory location, resulting in the program using too much space. However, variables often have limited lifetime during a program, and several variables with non-overlapping lifetimes can be mapped to the same location. Reusing memory locations appropriately for variables is essential to designing programs with good locality and low space requirements.

Consider, for example, the simple nested-parallel version of Quicksort in Figure 3.3 presented in the NESL language [33, 148]. At each level of recursion, array `A` is split into three smaller arrays `lesser`, `equal` and `greater` which are recursively sorted, and put together before returning. Note that:

```
function Quicksort(A) =
  if (#A < 2) then A
  else
    let pivot   = A[#A/2];
        lesser  = {e in A| e < pivot};
        equal   = {e in A| e == pivot};
        greater = {e in A| e > pivot};
        result  = {quicksort(v): v in [lesser,greater]};
    in result[0] ++ equal ++ result[1];
```

Figure 3.3: NESL code for nested-parallel Quicksort.

`#A = #lesser + #equal + #greater = #result[0] + #equal + #result[1].`

Since `A` is not referenced after it is split, the memory locations assigned to `A` can be recycled after the splits are computed. In fact, for an input of size n it is possible to have two memory chunks of size n and use them to support variables `A` and `lesser`, `equal`, `greater` across all levels of recursion. At each level of recursion the input variable would be mapped to one chunk, and split into the other chunk. At the recursion level beneath, the mapping of input and output variables to the memory chunks is reversed.

This scheme, in addition to saving space, provides scope for locality. In a recursive function call f with input size that is smaller than a cache size by a certain constant, the locations corresponding to the variables in f can be mapped to the cache and loaded only once into the cache. All the variables in lower levels of recursion can be mapped to the same set of locations that very mapped onto the cache. Each function call nested with in f can reuse the data in cache without incurring additional cache misses.

3.3.1 Static Allocation

A low-level approach to allocation is to provide the mapping from variables to memory locations in the program at compile time. It would be the responsibility of the programmer to ensure that no two variables used in concurrent instructions of the program are mapped to the same location. For convenience, we allow for the space required for stack frames to be allocated on demand at run time, although we restrict the size of stack frames to a pre-determined constant.

When the mapping is specified, instructions can be considered to be operating directly on memory locations instead of program variables. Let $loc(A)$ denote the set of locations that a set of nodes A can access. The space $S(A)$ of the set A (which may be a task for instance) is $|loc(A)|$, the number of distinct memory locations accessed. Note that under this allocation scheme, the space of a task is fixed at a compile time based on the mapping specified by the program, and is invariant under the scheduler.

For example, Appendix A presents an allocation scheme for Quicksort including the extra locations needed for computing the split. On an input of n elements of data-type E, the implementation uses just $2n \times sizeof(E) + 3n \times sizeof(int)$ distinct memory locations to support

```
function MM(A,B,C) =
  if (#A <= k) then return MM-Iterative(A,B)
  let
    (A_11,A_12,A_21,A_22) = split(A);
    (B_11,B_12,B_21,B_22) = split(B);
    C_11 = MM(A_11},B_11) || C_21 = MM(A_21,B_11) ||
      C_12 = MM(A_11,B_12) || C_22 = MM(A_21,B_12) ;
    C_11 += MM(A_12,B_21) || C_21 += MM(A_22,B_21) ||
      C_12 += MM(A_12,B_22) || C_22 += MM(A_22,B_22) ;
  in  join(C_{11},C_{12},C_{21},C_{22})
```

Figure 3.4: 4-way parallel matrix multiplication. Commands separated by "`||`" are executable in parallel, while "`;`" represents sequential composition.

the full parallelism of Quicksort, as opposed to the $O(n \log n)$ locations that a naive allocation would need. Section 4.3.1 deals with the locality of this program.

The static scheme avoids the runtime overhead of managing memory. However, it requires finding a good mapping manually at compile time. This is quite often difficult for dynamic DAGs and may force use of suboptimal mappings. However, it is possible to find a good mapping for many of the nested-parallel algorithms described in this dissertation, the statically allocated Quicksort presented in the Appendix being an example.

Another significant drawback of this approach is that it may not be possible to express the full parallelism of a program without using too much space or losing locality. Consider the matrix multiplication in Figure 3.4 for example. The eight recursive calls to matrix multiplication can be invoked in parallel. However, a static allocation would need $O(n^3)$ space for multiplying two $n \times n$ matrices, as there are $O(n^3)$ concurrent leaves in this recursion each of which would require a distinct memory location. To avoid this, the program in Figure 3.4 restricts the parallelism of the program. Four recursive calls are invoked in parallel followed by the other four. This allows the static mapping to reuse memory locations better and requires only $O(n^2)$ memory locations.

To simplify presentation, the bounds on communication costs and running times of schedulers we present in Section 5.3.1 and Section 5.3.2 are for statically allocated programs.

3.3.2 Dynamic allocation

A more flexible approach to memory allocation is to let programs allocate and free variables on demand. The run-time system would handle mapping from variables to locations while the program executes. Memory locations are provided on demand from a pool of free locations and recycled back when not necessary. Figure 3.5 illustrates a dynamically allocated version of Quicksort. Certain natural restrictions on the relative positions of operations to a variable v — the node that allocates v: $(alloc(v))$, the node that deallocates v: $(free(v))$ and nodes that reference v — are necessary:

- $free(v)$ must succeed $alloc(v)$.

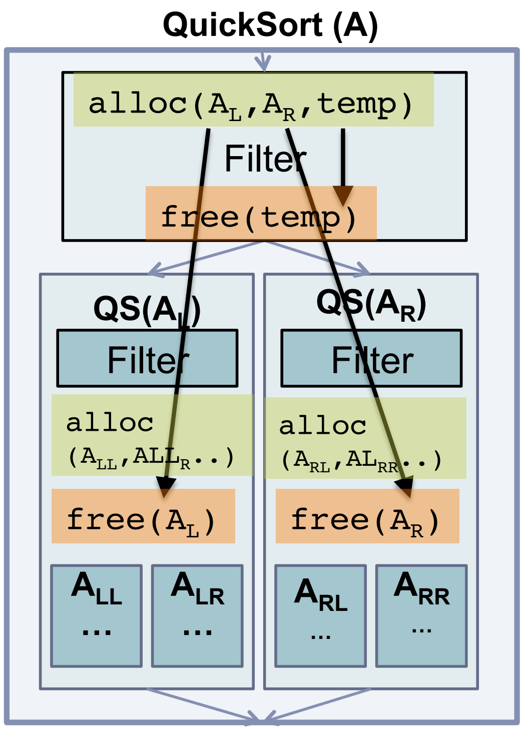


Figure 3.5: Dynamically allocated Quicksort

- v can not be referenced by a node n that precedes or is concurrent with $alloc(v)$.
- v can not be referenced by a node n that succeeds or is concurrent with $free(v)$.

The number of active locations at an instant during execution – locations to which some program variable is mapped to – depends on the execution order of the instructions in the program. The high water mark for the number of active locations during an execution represents the **space requirement of the schedule** for the program. For example, in an 8-way parallel recursive matrix multiplication, a schedule based on a depth-first schedule of the DAG would need only $O(n^2)$ space, while a breadth-first schedule would need $O(n^3)$ space. In the Quicksort presented in Figure 3.5, a depth-first schedule would need $O(n)$ space, while a bread-first schedule would need $O(n \log n)$ space. A typical baseline used for space of a dynamically allocated program is space under the sequential depth-first schedule [37, 47] which we refer to as sequential space, S_1.

An easier way to visualize space of an execution is to annotate the DAG with the number of variables allocated or deallocated at each node. The space requirement of an execution at an instant is the sum of annotations of all nodes that have been executed before the instant. It is convenient to assume that each node can allocate or deallocate at most a constant number of variables. An instruction that allocates n variables at once could be replaced with a parallel block of n tasks each of which allocated 1 variable each (see Figure 3.6) . This would not change the depth of the DAG significantly, but reflects the fact that allocating n might need n units of *work*.

The **space of a subtask** at a certain point t in an execution can be similarly defined – it is the sum of all previously allocated variables referenced by the subtask until t plus the difference

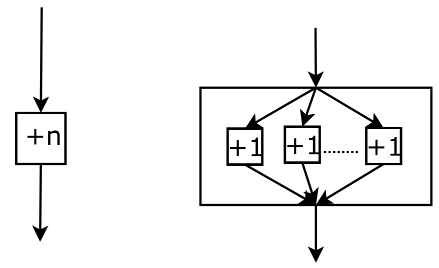


Figure 3.6: Replacing a node with $+n$ allocation with a parallel block with n parallel tasks each with $+1$ allocation.

between allocations and deallocations made in the subtask until t.

Parallel schedules have greater space requirement than S_1. Constructing a parallel schedule suitable for particular machine models while retaining space bounds close to S_1 [37, 134] and preserving locality [2, 40, 47] has been the focus of several studies. For example, Blumofe and Leiserson [47] present a scheduler for p processors with space requirement at most pS_1 in a slightly more restrictive allocation model than described here (an allocation can only be freed by a descendant node at the same nesting level). Some of these schedulers in these works are surveyed in Section 5.2. In Section 5.3.3, we present a class of schedulers for PMH machine model with provable bounds on parallel space requirements.

Ideally, in addition to good bounds on space, a dynamic memory allocation scheme would also achieve good locality on the machine the program is being mapped to. A locality preserving allocation policy has been presented in [36] for sequential programs. Research on constructing a parallel locality-preserving memory allocator is ongoing. This dissertation not explore this direction and it is left to future work. Other related work includes the relation between certain pebbling games and scheduling under dynamic allocation [37] and formal models and profiling tools to investigate space requirements of schedulers [152, 153].

Chapter 4

Cost Models for Locality and Parallelism

4.1 Program-Centric Cost Models

A program-centric cost model estimates the cost of the program itself without reference to machine attributes such as processors, caches, connections etc. Such models have the advantage of portability. The cost model can only use the description of the program such as the DAG including the instructions in each node. Examples of program-centric cost model include ***work*** W – the number of instructions and ***depth*** (or span) D – the length of the longest path in the DAG. The ratio W/D is a good metric for the parallelism of the DAG when every memory access has unit cost.

Work and Depth do not capture locality (communication costs), but several program-centric models for locality exist. For example, the Cache-Oblivious framework is a good program-cenrtic model for sequential programs. The CO framework is program-centric as it does not allow the programs to use any machine parameters unlike the External-Memory model [6] or its parallel extension, the Parallel External-Memory model [20]. The analysis in the CO framework is done with respect to couple of abstract parameters M, B which may be understand to correspond to cache dimensions that the program might be mapped on to later on.

Program-centric cost models have little utility if they do not represent costs on real machines. Therefore, in addition to the model itself, it is critical to show general bounds on performance when mapping the program onto particular machine organizations such as the PMH. Ideally, one high-level program-centric model for locality can replace many machine-centric models, allowing portability of programs and the analysis of their locality across machines via machine-specific mappings. Such mappings make use of a scheduler designed for a particular cache organization. Schedulers should come with performance guarantees that are based solely on program-centric metrics (e.g., locality, work, depth) of the program and parameters of the machine.

The definition of program-centric cost models does not restrict analysis made with respect to a specific ordering of instructions (schedule), although it may not be easy to realize a schedule-specific costs with a different scheduler. Since different machine models need different schedules, the usefulness of a cost model based on a specific schedule might be limited. It is better to construct a cost model that can realized across many schedules.

Surprisingly, cache complexity under sequential depth-first order in the CO framework (Q_1, defined in Section 4.2) can be relevant on simple machine models with one level of cache. Work, depth and cs alone can predict performance on these simple machine models as we will see in Section 5.2. However, it may not be easy to generalize such results to parallel machines with a hierarchy of caches because of the schedule-specific nature of Q_1 metric. In Section 4.3, we will present a parallel cache complexity cost model for locality which is schedule-agnostic, and realizable by many practical schedules. Further, we will extend the parallel cache complexity metric to a metric called the effective cache complexity (Section 4.3.2) which brings together both locality and parallelism under one metric. The metric leads to a new definition for quantifying the parallelizability of an algorithm (Section 4.3.3) that is more practical and general than depth (span) or the ratio of work to depth. The parallel and effective cache complexity metrics together form our parallel cache complexity (PCC) framework.

4.2 The Cache-Oblivious Framework

The Cache-Oblivious framework for sequential computations was defined in Section 2.5.2. In this framework, sequential programs are run against an abstract machine which consists of one processor that is connected to an ideal cache of size M and block size B and an infinite memory. The number of transfers between the cache and memory is termed the cache complexity $Q(M, B)$ and used as measure for locality.

Parallel programs can also be analyzed in this model by imposing a sequntial order. One useful sequntial order for nested-parallel programs is the depth-first order illustrated in Figure 1.9. We refer to the cache complexity of the depth-first order of a parallel program in the cache-oblivious model as the sequential cache complexity of the program and denote it with $Q_1(M, B)$. The three program-centric metrics – work, depth, and sequntial cache complexity – taken together can quantify the performance of a program on one-level cache models. In section 5.2, we will illustrate this by presenting schedulers and proving performance bounds in terms of these program-centric metrics.

4.3 The Parallel Cache Complexity Framework

In this section, we present the parallel cache complexity framework including the parallel and effective cache complexity cost models (metrics). The parallel cache complexity cost model is a simple, high-level model for locality in nested-parallel algorithm. As in the cache-oblivious framework [86], parallel cache complexity analysis is made against an abstract machine model including a memory of unbounded size and a single cache with size M, line-size B (in words) with optimal (i.e., furthest into the future) replacement policy. The cache state κ consists of the set of cache lines resident in the cache at a given time. When a location in a non-resident line l is accessed and the cache is full, l replaces in κ the line accessed furthest into the future, incurring a *cache miss*.

To extend cache complexity analysis in the CO framework to parallel computations, one needs to define how to analyze the number of cache misses during execution of a parallel block.

Task t forks subtasks t_1 and t_2,
with $\kappa = \{l_1, l_2, l_3\}$

t_1 accesses l_1, l_4, l_5 incurring 2 misses
t_2 accesses l_2, l_4, l_6 incurring 2 misses

At the join point: $\kappa' = \{l_1, l_2, l_3, l_4, l_5, l_6\}$

Figure 4.1: Example applying the PCC cost model (Definition 2) to a parallel block. Here, $Q^*(\mathsf{t}; M, B; \kappa) = 4$.

Analyzing using a sequential ordering of the subtasks in a parallel block (as in most prior work[1]) is problematic for mapping to even a single shared cache, as the following theorem demonstrates for the CO model:

Theorem 1 *Consider a PMH comprised of a single cache shared by $p > 1$ processors, with cache-line size B, cache size $M \geq pB$, and a memory (i.e., $h = 2$). Then there exists a parallel block such that for any greedy scheduler[2] the number of cache misses is nearly a factor of p larger than the sequential cache complexity in the CO framework.*

Proof. Consider a parallel block that forks off p identical tasks, each consisting of a strand reading the same set of M memory locations from M/B blocks. In the sequential cache complexity analysis, after the first M/B misses, all other accesses are hits, yielding a total cost of M/B misses.

Any greedy schedule on p processors executes all strands at the same time, incurring simultaneous cache misses (for the same line) on each processor. Thus, the parallel block incurs $p(M/B)$ misses. $\square$

The gap arises because a sequential ordering accounts for significant reuse among the subtasks in the block, but a parallel execution cannot exploit reuse unless the line has been loaded earlier.

To overcome this difficulty, we make use of two key ideas:

1. Ignore any data reuse among the subtasks since parallel schedules might not be able to make use of such data reuse. Figure 4.3 illustrates the difference between the sequential cache complexity which accounts for data reuse between concurrent tasks and the parallel cache complexity which forwards cache state only between parts of the computation that are ordered (have edges between them).
2. Flushing the cache at each fork and join point of any task that does not fit within the cache as it is unreasonable to expect schedules to preserve locality at a scale larger than the cache.

For simplicity, we assume that program variables have been statically allocated. Therefore the set of locations that a task accesses is invariant with respect to schedule. Furthermore, we logically segment the memory space into *cache lines* which are contiguous chunks of size B. Therefore, every program variable is mapped to an unique cache line. Let $loc(\mathsf{t}; B)$ denote the

[1]Two prior works not using the sequential ordering are the *concurrent cache-oblivious model* [27] and the *ideal distributed cache model* [84], but both design directly for p processors and consider only a single level of private caches.

[2]In a *greedy* scheduler, a processor remains idle only if there is no ready-to-execute task.

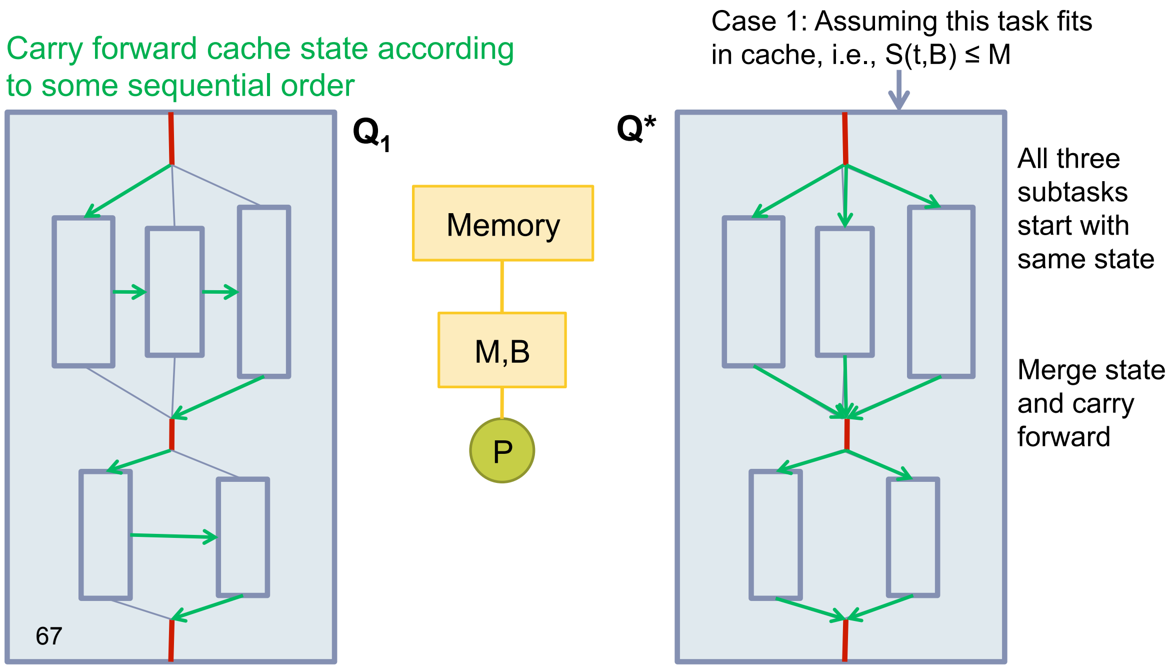

Carry forward cache state according to some sequential order
Q1
Memory
M,B
P
Q*
Case 1: Assuming this task fits in cache, i.e., S(t,B) ≤ M
All three subtasks start with same state
Merge state and carry forward
67

Figure 4.2: Difference in cache-state forwarding in the sequential cache complexity of the CO framework and the parallel cache complexity. Green arrows indicate forwarding of cache state.

set of distinct cache lines accessed by task t, and $S(\mathsf{t}; B) = |loc(\mathsf{t}; B)| \cdot B$ denote its size (also let $s(\mathsf{t}; B) = |loc(\mathsf{t}; B)|$ denote the size in terms of number of cache lines). Let $Q(\mathsf{c}; M, B; \kappa)$ be the sequential cache complexity of c in the CO framework when starting with cache state κ.

Definition 2 *[Parallel Cache Complexity] For cache parameters M and B the* ***parallel cache complexity*** *of a strand* s*, parallel block* b*, or task* t *starting at state* κ *is defined as:*

strand:

$$Q^*(\mathsf{s}; M, B; \kappa) = Q(\mathsf{s}; M, B; \kappa)$$

parallel block: *For* $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \dots \| \mathsf{t}_k$,

$$Q^*(\mathsf{b}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{t}_i; M, B; \kappa)$$

task: *For* $\mathsf{t} = \mathsf{c}_1; \mathsf{c}_2; \dots; \mathsf{c}_k$,

$$Q^*(\mathsf{t}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{c}_i; M, B; \kappa_{i-1}) \ ,$$

where $\kappa_i = \emptyset$ *if* $S(\mathsf{t}; B) > M$*, and* $\kappa_i = \kappa \cup_{j=1}^{i} loc(\mathsf{c}_j; B)$ *if* $S(\mathsf{t}; B) \leq M$.

Problem	Span	Cache Complexity Q^*
Scan (prefix sums, etc.)	$O(\log n)$	$O(\lceil \frac{n}{B} \rceil)$
Matrix Transpose ($n \times m$ matrix) [86]	$O(\log(n+m))$	$O(\lceil \frac{nm}{B} \rceil)$
Matrix Multiplication ($\sqrt{n} \times \sqrt{n}$ matrix) [86]	$O(\sqrt{n})$	$O(\left\lceil \frac{n^{1.5}}{B} \right\rceil / \sqrt{M+1})$
Matrix Inversion ($\sqrt{n} \times \sqrt{n}$ matrix)	$O(\sqrt{n})$	$O(\left\lceil \frac{n^{1.5}}{B} \right\rceil / \sqrt{M+1})$
Quicksort [109]	$O(\log^2 n)$	$O(\lceil \frac{n}{B} \rceil (1 + \log \lceil \frac{n}{M+1} \rceil))$
Sample Sort [40]	$O(\log^2 n)$	$O(\lceil \frac{n}{B} \rceil \lceil \log_{M+2} n \rceil)$
Sparse-Matrix Vector Multiply [40] (m nonzeros, n^ϵ edge separators)	$O(\log^2 n)$	$O(\left\lceil \frac{m}{B} + \frac{n}{(M+1)^{1-\epsilon}} \right\rceil)$
Convex Hull (e.g., see [39])	$O(\log^2 n)$	$O(\lceil \frac{n}{B} \rceil \lceil \log_{M+2} n \rceil)$
Barnes Hut tree (e.g., see [39])	$O(\log^2 n)$	$O(\lceil \frac{n}{B} \rceil (1 + \log \lceil \frac{n}{M+1} \rceil))$

Table 4.1: Parallel cache complexities (Q^*) of some algorithms. The bounds assume $M = \Omega(B^2)$. All algorithms are work optimal and their cache complexities match the best sequential algorithms.

We use $Q^*(\mathsf{c}; M, B)$ to denote a computation c starting with an empty cache, $Q^*(n; M, B)$ when n is a parameter of the computation, and $Q^*(\mathsf{c}; 0, 1)$ to denote the computational work. Note that by setting M to 0, we force the analysis to count every instruction that touches even a register and hence effectively corresponds to instruction count.

Comments on the definition: Since a task t alternates between strands and parallel blocks the definition effectively clears the cache at every fork and join point in t when $S(\mathsf{t}; B) > M$. This is perhaps more conservative than required but leads to a simple model and does not seem to affect bounds. Since in a parallel block all subtasks start with the same cache state, no sharing is assumed among parallel blocks. If an algorithms wants to share a value loaded from memory, then the load should occur before the fork. The notion of furthest in the future for Q in a strand might seem ill-defined since the future might entail parallel tasks. However, all future references fit into cache until reaching a supertask that does not fit in cache, at which point the cache is assumed to be flushed. Thus, there is no need to choose cache lines to evict. For a single strand the model is equivalent to the cache-oblivious framework.

We believe that the parallel cache complexity cost model is a simple, effective model for the cache analysis of parallel algorithms. It retains much of the simplicity of the ideal cache model, such as analyzing using only one level of cache. It ignores the complexities of artificial locality among parallel subtasks. Thus, it is relatively easy to analyze algorithms in this cost model (examples are given in Section 4.3.1). Moreover, as we will show in Section 5.3.1, parallel cache complexity bounds optimally map to cache miss bounds on each level of a PMH. Finally, although the complexity bounds are upper bounds, for many fundamental algorithms, they are tight: they asymptotically match the bounds given by the sequential ideal cache model, which are asymptotically optimal. Table 4.3 presents the parallel cache complexities of a few such algorithms, including both algorithms with polynomial span (matrix inversion) and highly imbalanced algorithms (the block transpose used in sample sort).

4.3.1 Example Parallel Cache Complexity Analysis

It is relatively easy to analyze algorithms in the parallel cache complexity cost model. Let us consider first a simple map over an array which touches each element by recursively splitting the array in half until reaching a single element. If the algorithm for performing the map does not touch any array elements until recursing down to a single element, then each recursive task begins with an empty cache state, and hence the cache performance is $Q^*(n; M, B) = n$. An efficient implementation would instead load the middle element of the array before recursing, thus guaranteeing that a size-$\Theta(B)$ recursive subcomputation begins with a cache state containing the relevant line. We thus have the recurrence

$$Q^*(n; M, B) \quad = \quad \begin{cases} 2Q^*(\frac{n}{2}; M, B) + O(1) & n > B \\ O(1) & n \leq B\ , \end{cases}$$

which implies $Q^*(n; M, B) = O(n/B)$, matching the sequential cache complexity.

Quicksort is another algorithm that is easy to analyze in this model. A standard quicksort analysis for work [69] observes that all work can be amortized against comparisons of keys with a pivot. The probability of comparing keys of rank i and $j > i$ is at most $2/(j - i)$, i.e. , the probability of selecting i or j as a pivot before any element in between. The expected work is thus $\sum_{i=1}^{n} \sum_{j>i}^{n} 2/(j - i) = \Theta(n \log n)$. Extending this analysis to either the sequential or parallel cache complexity cost models is nearly identical—comparisons become free once the corresponding subarray fits in memory. Specifically, for nearby keys i and $j < i + M/3$, no paid comparison occurs if a key between $i - M/3$ and $i - 1$ is chosen before i and if $j + 1$ to $j + M/3$ is chosen before j. Summing over all keys gives expected number of paid comparisons

$$\sum_{i=1}^{n} \sum_{j>i+M/3}^{n} \frac{2}{j - i} + \sum_{i=1}^{n} \sum_{j>i}^{j \leq i+M/3} \frac{6}{M} = \Theta(n \log \left\lceil \frac{n}{M + 1} \right\rceil + n).$$

Completing the analysis (dividing this cost by B) entails observing that each recursive quicksort scans the subarray in order, and thus whenever a comparison causes a cache miss, we can charge $\Theta(B)$ comparisons against the same cache miss.

The rest of the algorithms in Table 4.3 can be similarly analyzed without difficulty, observing that for the original analyses in the CO framework, the cache complexities of the parallel subtasks were already analyzed independently assuming no data reuse.

4.3.2 Extended Parallel Cache Complexity – Effective Cache Complexity

On any machine with shared caches such as PMH, all caches are associated with a set of processors. It therefore stands to reason that if a task needs memory M but does not have sufficient parallelism to make use of a cache of appropriate size, that either processors will sit idle or additional misses will be required. This might be true even if there is plenty of parallelism on average in the computation. The following lower-bound makes this intuition more concrete.

Theorem 3 *(Lower Bound) Consider a PMH comprised of a single cache shared by $p > 1$ processors with parameters $B = 1$, M and C, and a memory (i.e. , $h = 2$). Then for all $r \geq 1$, there exists a computation with $n = rpM$ memory accesses, $\Theta(n/p)$ span, and $Q^*(M, B) = pM$, such that for any scheduler, the runtime on the PMH is at least $nC/(C + p) \geq (1/2) \min(n, nC/p)$.*

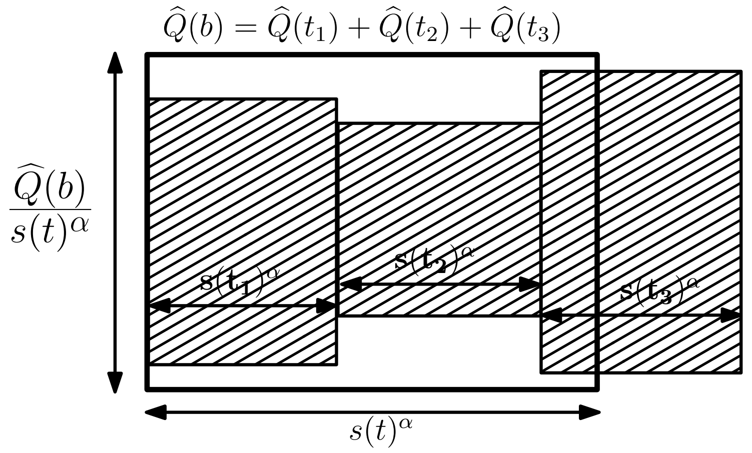
Proof. Consider a computation that forks off $p > 1$ parallel tasks. Each task is sequential (a single strand) and loops over touching M locations, distinct from any other task (i.e. , a total of Mp locations are touched). Each task then repeats touching the same M locations in the same order a total of r times, for a total of $n = rMp$ accesses. Because M fits within the cache, only a task's first M accesses are misses and the rest are hits in the parallel cache complexity cost model. The total cache complexity is thus only $Q^*(M, B) = Mp$ for $B = 1$ and any $r \geq 1$.

Now consider an execution (schedule) of this computation on a shared cache of size M with p processors and a miss cost of C. Divide the execution into consecutive sequences of M time steps, called ***rounds***. Because it takes 1 (on a hit) or $C \geq 1$ (on a miss) units of time for a task to access a location, no task reads the same memory location twice in the same round. Thus, a memory access costs 1 only if it is to a location in memory at the start of the round and C otherwise. Because a round begins with at most M locations in memory, the total number of accesses during a round is at most $(Mp - M)/C + M$ by a packing argument. Equivalently, in a full round, M processor steps execute at a rate of 1 access per step, and the remaining $Mp - M$ processor steps complete $1/C$ accesses per step, for an average "speed" of $1/p + (1 - 1/p)/C < 1/p + 1/C$ accesses per step. This bound holds for all rounds except the first and last. In the first round, the cache is empty, so the processor speed is $1/C$. The final round may include at most M fast steps, and the remaining steps are slow. Charging the last round's fast steps to the first round's slow steps proves an average "speed" of at most $1/p + 1/C$ accesses per processor time step. Thus, the computation requires at least $n/(p(1/p+1/C)) = nC/(C+p)$ time to complete all accesses. When $C \geq p$, this time is at least $nC/(2C) = n/2$. When $C \leq p$, this time is at least $nC/(2p)$. □

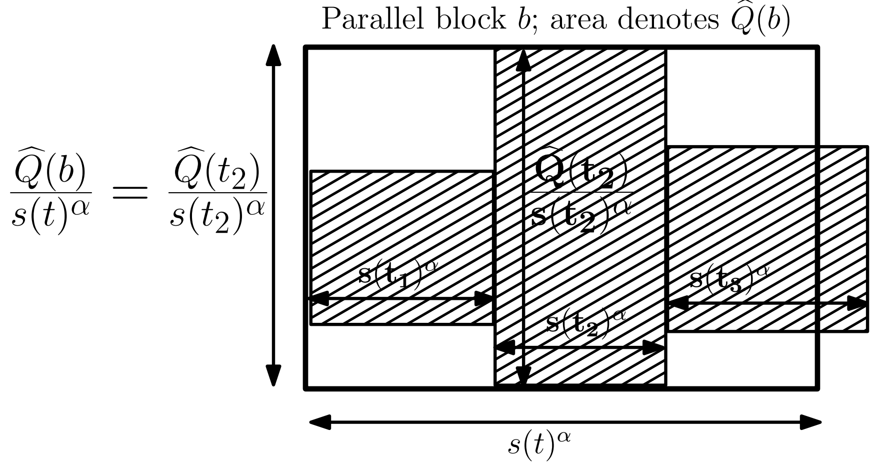
The proof shows that even though there is plenty of parallelism overall and a fraction of at most $1/r$ of the accesses are misses in Q^*, an optimal scheduler either executes tasks (nearly) sequentially (if $C \geq p$) or incurs a cache miss on (nearly) every access (if $C \leq p$).

This indicates that some cost must be charged to account for the space-parallelism imbalance. We extend parallel cache complexity with a cost metric that charges for such imbalance, but does not charge for imbalance in subtask size. When coupled with our scheduler in Section 5.3, the metric enables bounds in the PCC framework to effectively map to PMH at runtime (Section 5.3.2), even for highly-irregular computations.

The metric aims to estimate the degree of parallelism that can be utilized by a symmetric hierarchy as a function of the size of the computation. Intuitively, a computation of size S with "parallelism" $\alpha \geq 0$ should be able to use $p = O(S^\alpha)$ processors effectively. This intuition works well for algorithms whose extent of parallelism is polynomial in the size of the problem. For example, in a regular recursive algorithms where a is the number of parallel recursive subproblems, n/b is the size of subproblems for a size-n problem, and $\tilde{O}(n^c)$ is the span (depth) of



$\widehat{Q}(b) = \widehat{Q}(t_1) + \widehat{Q}(t_2) + \widehat{Q}(t_3)$
$\frac{\widehat{Q}(b)}{s(t)^\alpha}$
$s(t_1)^\alpha$
$s(t_2)^\alpha$
$s(t_3)^\alpha$
$s(t)^\alpha$



Parallel block b; area denotes $\widehat{Q}(b)$
$\frac{\widehat{Q}(b)}{s(t)^\alpha} = \frac{\widehat{Q}(t_2)}{s(t_2)^\alpha}$
$\frac{\widehat{Q}(t_2)}{s(t_2)^\alpha}$
$s(t_1)^\alpha$
$s(t_2)^\alpha$
$s(t_3)^\alpha$
$s(t)^\alpha$

Figure 4.3: Two examples of Definition 4 applied to a parallel block $\mathsf{b} = \mathsf{t}_1 \parallel \mathsf{t}_2 \parallel \mathsf{t}_3$ belonging to task t. The shaded rectangles represent the subtasks and the white rectangle represents the parallel block b. Subtask rectangles have fixed area ($\widehat{Q}_\alpha(\mathsf{t}_i)$, determined recursively) and *maximum* width $s(t_i)^\alpha$. The left example is work dominated: the total area of b's subtasks is larger than any depth subterms, and determines the area $\widehat{Q}_\alpha(\mathsf{b}) = \widehat{Q}_\alpha(\mathsf{t}_1) + \widehat{Q}_\alpha(\mathsf{t}_2) + \widehat{Q}_\alpha(\mathsf{t}_3)$. The right example is depth dominated: the height of a subtask t_2 determines the height of b and hence the area is $(s(t)/s(t_2))^\alpha \widehat{Q}_\alpha(\mathsf{t}_2)$.

an algorithm with input size n, the α here roughly corresponds to $(\log_b a) - c$.

More formally, we define a notion of ***effective cache complexity*** $\widehat{Q}_\alpha(\mathsf{c})$ for a computation c based on the definition of Q^*. Just as for Q^*, $\widehat{Q}_\alpha$ for tasks, parallel blocks and strands is defined inductively based on the composition rules described in Section 3.2 for building a computation. The definition makes use of the space of a task. We use assume that the computation is statically allocated so that the space is properly defined. Further, we use the convention that in a task $\mathsf{t} = \mathsf{s}_1; \mathsf{b}_1; \ldots; \mathsf{s}_k$, the space of parallel blocks nested directly in task t ($\mathsf{b}_1, \mathsf{b}_2, \ldots$) is equal to the space $s(\mathsf{t};)$ (this is purely for convenience of definition and does not restrict the program in any way).

Definition 4 *[parallel cache complexity extended for imbalance] For cache parameters M and B and parallelism α, the* ***effective cache complexity*** *of a strand* s*, parallel block* b*, or task* t *starting at cache state κ is defined as:*
strand: *Let* t *be the nearest containing task of strand* s

$$\widehat{Q}_\alpha(\mathsf{s}; M, B; \kappa) = Q^*(\mathsf{s}; M, B; \kappa) \times s(\mathsf{t}; B)^\alpha$$

parallel block: *For* $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$ *in task* t,

$$\widehat{Q}_\alpha(\mathsf{b}; M, B; \kappa) = \max \begin{cases} s(\mathsf{t}; B)^\alpha \max_i \left\{ \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i; M, B; \kappa)}{s(\mathsf{t}_i; B)^\alpha} \right\rceil \right\} & \textit{(depth dominated)} \\ \sum_i \widehat{Q}_\alpha(\mathsf{t}_i; M, B; \kappa) & \textit{(work dominated)} \end{cases}$$

task: *For* $\mathsf{t} = c_1; c_2; \ldots; c_k$,

$$\widehat{Q}_\alpha(\mathsf{t}; M, B; \kappa) = \sum_{i=1}^{k} \widehat{Q}_\alpha(c_i; M, B; \kappa_i) \ ,$$

where κ_i is defined as in Definition 2.

In the rule for parallel block, the *depth dominated* term corresponds to limiting the number of processors available to do the work on each subtask t_i to $s(\mathsf{t}_i)^\alpha$. This throttling yields an effective depth of $\left\lceil \widehat{Q}_\alpha(\mathsf{t}_i)/s(\mathsf{t}_i)^\alpha \right\rceil$ for subtask t_i. The effective cache complexity of the parallel block b nested directly inside task t is the maximum effective depth of its subtasks multiplied by the number of processors for the parallel block b, which is $s(\mathsf{t}; B)^\alpha$ (see Fig. 4.3).

$\widehat{Q}_\alpha$ is an attribute of an algorithm, and as such can be analyzed irrespective of the machine and the scheduler. Section 4.3.4 illustrates the analysis for $\widehat{Q}_\alpha(\cdot)$ and effective parallelism for several algorithms. Note that, as illustrated in Fig. 4.3(top) and the analysis of algorithms in the report, good effective parallelism can be achieved even when there is significant work imbalance among subtasks.

Finally, the depth dominated term implicitly includes the span so we do not need a separate depth (span) cost in our model. The term $\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}; M, B; \kappa)}{s(\mathsf{t}; B)^\alpha} \right\rceil$ behaves like the *effective depth* in that for a task $\mathsf{t} = \mathsf{s}_1; \mathsf{b}_1; \mathsf{s}_2; \mathsf{b}_2; \ldots; \mathsf{s}_k$, the effective depth of task t is the sum of the effective depths of s_is and b_i.

Note that since work is just a special case of Q^*, obtained by substituting $M = 0$, the effective cache complexity metric can be used to compute effective work just like effective cache complexity.

4.3.3 Parallelizability of an Algorithm

We say that an algorithm is *α-efficient* for a parameter $\alpha \geq 0$ if $Q^*(n; M, B) = O(\widehat{Q}_\alpha(n; M, B))$, i.e. , for any values of $M > B > 0$ there exists a constant $c_{M,B}$ such that

$$\lim_{n\to\infty} \frac{\widehat{Q}_\alpha(n; M, B)}{Q^*(n; M, B)} \leq c_{M,B},$$

where n denotes the input size. This α-efficiency occurs trivially if the work term always dominates, but can also happen if sometimes the depth term dominates. The least upper bound on the set of α for which an algorithm is α-efficient specifies the ***parallelizability*** of the algorithm. For example, the Map algorithm in Section 4.3.1 is α-efficient for all values in the range $[0, 1)$ and not for any values greater than or equal to 1. Therefore, the map algorithm has parallelizabe to degree 1.

The lower the α, the more work-space imbalance the effective complexity can absorb and still be work efficient, and in particular when $\alpha = 0$ the balance term disappears. Therefore, a sequential algorithm, i.e. a program with a trivial chain for DAG, will not be α-efficient for any $\alpha > 0$. Its parallelizability would be 0.

4.3.4 Examples of Algorithms and their Complexity

In this section, we will illustrate how to estimate the effective parallelism of an algorithm by computing the effective cache complexity $\widehat{Q}_\alpha$ as a function of α with the help of a few examples. We will start by computing the effective cache complexity of a simple map over an array, as introduced in section 4.3.1, which touches each element by touching the middle element and then recursively splitting the array in half around this element. For an input of size n, the space of the algorithm is cn for some constant c. Therefore,

$$\widehat{Q}_\alpha(n; M, B; \kappa) = \begin{cases} O(1) & cn \leq B, M > 0 \\ \max\left\{ \left\lceil \frac{cn}{B} \right\rceil^\alpha \frac{\widehat{Q}_\alpha(\frac{n}{2}; M, B; \kappa)}{\lceil cn/2B \rceil^\alpha},\ 2\widehat{Q}_\alpha(\frac{n}{2}; M, B; \kappa) \right\} + O(\left\lceil \frac{cn}{B} \right\rceil^\alpha), & otherwise. \end{cases} \tag{4.1}$$

$\widehat{Q}_\alpha(n) = O(1)$ for $cn < B, M > 0$ because only the first level of recursion after $cn < B$ (and one other node in the recursion tree rooted at this level, in case this call accesses locations across a boundary) incurs at most one cache miss (which costs $O(1^\alpha)$) and the cache state, which includes this one cache line, is carried forward to lower levels preventing them from any more cache misses. For $\alpha < 1, M > 0$, the summation term dominates (second term in the max) and the recursion solves to $O(\lceil n/B \rceil)$, which is optimal. If $M = 0$, then $O(n)$, for $\alpha < 1$.

We will see the analysis for some matrix operations. Consider a recursive version of matrix addition where each quadrant of the matrix is added in parallel. Here, a task (corresponding to a recursive addition) comprises a strand for the fork and join points and a parallel block consisting of four smaller tasks on a matrix one fourth the size. For an input of size n, the space of the algorithm is cn for some constant c. Therefore,

$$\widehat{Q}_\alpha(n; M, B) = \begin{cases} O(1), & cn \leq B, M > 0 \\ O(\lceil cn/B \rceil^\alpha) + 4\widehat{Q}_\alpha(n/4; M, B), & B < s \end{cases} \tag{4.2}$$

which implies $\widehat{Q}_\alpha(n; M, B) = O(\lceil n/B \rceil)$ if $\alpha < 1, M > 0$, and $\widehat{Q}_\alpha(n) = O(\lceil n/B \rceil^\alpha)$ if $\alpha \geq 1, M > 0$. These bounds imply that matrix addition is work efficient only when parallelism is limited to $\alpha < 1$. Further, when $M = 0$, $\widehat{Q}_\alpha(n; M, B) = O(n)$ for $\alpha < 1$.

The matrix multiplication described in Figure 4.4 consists of fork and join points and two parallel blocks. Each parallel block is a set of 4 parallel tasks each, each of which is a matrix multiplication on matrices four times smaller. Note that if at some level of recursion, the first parallel block fits in cache, then the next parallel block can reuse the same set of locations. For an input of size n, the space of the algorithm is cn for some constant c. Therefore, when $M > 0$,

$$\widehat{Q}_\alpha^{Mult}(n; M, B; \kappa) = \begin{cases} 0, \quad \text{if } \kappa \text{ includes all relevant locations} \\ O(1) \quad \text{if } \kappa \text{ does not include the relevant cache lines}, & cn \leq B \\ O(\lceil \frac{cn}{B} \rceil^\alpha) + 4\widehat{Q}_\alpha^{Mult}(\frac{n}{4}; M, B; \kappa), \kappa_p + 4\widehat{Q}_\alpha^{Mult}(\frac{n}{4}; M, B; \kappa_p), & B < cn \leq M \\ O(\lceil \frac{cn}{B} \rceil^\alpha) + 8\widehat{Q}_\alpha^{Mult}(\frac{n}{4}; M, B; \emptyset), & otherwise, \end{cases} \tag{4.3}$$

which implies $\widehat{Q}_\alpha^{Mult}(s) = O(\lceil n/M \rceil^{3/2} \lceil M/B \rceil)$ for all $\alpha < 1$. Further, for $M = 0$, $\widehat{Q}_\alpha^{Mult}(s) = O(n^{3/2})$ when $\alpha < 1$.

Note that a different version of matrix multiplication with eight parallel recursive calls has greater parallelism, but may require super-linear stack space. If we assume that the space is proportional to input size through a dynamic allocation scheme with good reuse, the effective cache complexity of 8-way matrix multiplication is

$$\widehat{Q}_\alpha^{Mult}(n; M, B; \kappa) = \begin{cases} 0, \quad \text{if } \kappa \text{ includes all relevant locations} \\ O(1) \quad \text{if } \kappa \text{ does not include the relevant cache lines}, & cn \leq B \\ O(\lceil \frac{cn}{B} \rceil^\alpha) + \max\left\{8\widehat{Q}_\alpha^{Mult}(\frac{n}{4}; M, B; \kappa), \lceil \frac{cn}{B} \rceil^\alpha\right\} & B < cn \leq M \\ O(\lceil \frac{cn}{B} \rceil^\alpha) + \max\left\{8\widehat{Q}_\alpha^{Mult}(\frac{n}{4}; M, B; \emptyset), \lceil \frac{n}{B} \rceil^\alpha\right\}, & otherwise, \end{cases} \tag{4.4}$$

which implies $\widehat{Q}_\alpha^{Mult}(n) = O(\lceil n/M \rceil^{3/2} \lceil M/B \rceil)$ for all $\alpha < \frac{3}{2} - \frac{1}{2p}$, where $p = \log_{\lceil M/B \rceil} \lceil n/B \rceil$. If M is considered a constant, then the parallelizability of this algorithm is $3/2$, consistent with the exponent in the work and cache complexities of matrix-multiplication.

Matrix Inversion, as described in Figure 4.4, consists of fork and join points, and 10 parallel blocks (each containing only on subtask). Two of these blocks are matrix inversions on matrices four times smaller, six are multiplications on matrices of the same size and two are additions on matrices of the same size. The effective cache complexity of the parallel blocks is dominated by the balance term only in the case of the matrix inversion (because they operate on much smaller matrices). The effective cache complexity is

$$\widehat{Q}_\alpha^{Inv}(n; M, B) = O\left(\left\lceil \frac{cn}{B} \right\rceil^\alpha\right) + 2\left\lceil \frac{cn}{B} \right\rceil^\alpha \times \frac{\widehat{Q}_\alpha^{Inv}(n/4; M, B; \emptyset)}{(\lceil cn/4B \rceil)^\alpha} \tag{4.5}$$

$$+ 6\widehat{Q}_\alpha^{Mult}(\frac{n}{4}; M, B; \emptyset) + 2\widehat{Q}_\alpha^{Add}(\frac{n}{4}; M, B; \emptyset), \tag{4.6}$$

when $cn > M > 0$ (other cases have been left out for brevity). It can be shown that $\widehat{Q}_\alpha^{Inv}(n) = O(\lceil n/M \rceil^{3/2} \lceil M/B \rceil)$ when $\alpha < 1, M > 0$, indicating that inversion has a parallelizability of 1. For $M = 0$, $\widehat{Q}_\alpha^{Inv}(n; M, B) = O(n^{3/2})$ for $\alpha < 1$. The gap of about $s^{1/2}$ between effective work and parallelism as indicated by the maximum α is consistent with the fact that under the conventional definitions, this inversion algorithm has a depth of about $s^{1/2}$.

In all the above examples, calls to individual tasks in a parallel block are relatively balanced. An example where this is not the case is a parallel deterministic cache oblivious sorting from [40] outlined as COSORT(i, n) Figure 4.4. This sorting uses five primitives: prefix sums, matrix transpose, bucket transpose, merging and simple merge sort. Theorem 25 will present an analysis to show that COSORT($cdot$, n) has $\widehat{Q}_\alpha(n; M, B) = \left\lceil \frac{s}{B} \right\rceil \left\lceil \log_{M+2} s \right\rceil$ for $\alpha < 1 - \Theta(1/\log n)$ demonstrating that the algorithm has a parallelizability of 1.

For a version of Quicksort in which a random pivot is repeatedly drawn until it partitions the array into two parts none of which is smaller than a third of the array, the algorithm is α-efficient for $\alpha < \log_{1.5}\left(1 + \frac{\log_2 \lceil n/3M \rceil}{2 \log_2 \lceil 2n/3M \rceil}\right)$.

```
function **QuickSort**($A$)
  if ($\#A \leq 1$) then return $A$
  p = $A$[rand($\#A$)] ;
  Les = **QuickSort**($\{a \in A | a < p\}$) ||
  Eql = $\{a \in A | a = p\}$ ||
  Grt = **QuickSort**($\{a \in A | a > p\}$) ;
  return Les ++ Eql ++ Grt
```

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n/B)\log(n/(M+1)))$$

In the code both || and { } indicate parallelism. The bounds are expected case.

```
function **SampleSort**($A$)
  if ($\#A \leq 1$) then return $A$
  parallel for $i \in [0, \sqrt{n}, \ldots, n)$
    **SampleSort**($A[i, ..., i + \sqrt{n})$) ;
  $P[0, 1, \ldots, \sqrt{n}) =$ findPivots($A$) ;
  $B[0, 1, \ldots, \sqrt{n}) =$ bucketTrans($A, P$) ;
  parallel for $i \in [0, 1, \ldots, \sqrt{n})$
    **SampleSort**($B_i$) ;
  return flattened $B$
```

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n/B)\log_{M+2} n)$$

This version is asymptotically optimal for cache misses. The pivots partition the keys into buckets and bucketTrans places the keys from each sorted subset in A into the buckets B. Each bucket ends up with about $\sqrt{n}$ elements [40].

```
function **MM**(A,B,C)
  if ($\#A \leq k$) then return MMsmall(A,B)
  $(A_{11}, A_{12}, A_{21}, A_{22}) =$ split($A$) ;
  $(B_{11}, B_{12}, B_{21}, B_{22}) =$ split($B$) ;
  $C_{11} =$ **MM**$(A_{11}, B_{11})$ ||  $C_{21} =$ **MM**$(A_{21}, B_{11})$ ||
  $C_{12} =$ **MM**$(A_{11}, B_{12})$ ||  $C_{22} =$ **MM**$(A_{21}, B_{12})$ ;
  $C_{11} += $ **MM**$(A_{12}, B_{21})$ ||  $C_{21} += $ **MM**$(A_{22}, B_{21})$ ||
  $C_{12} += $ **MM**$(A_{12}, B_{22})$ ||  $C_{22} += $ **MM**$(A_{22}, B_{22})$ ;
  return join($C_{11}, C_{12}, C_{21}, C_{22}$)
```

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n^{1.5}/B)/\sqrt{M+1})$$

Multiplying two $\sqrt{n} \times \sqrt{n}$ matrices. The split operation has to touch any of the four elements at the center of the matrices A, B. The eight recursive subtasks are divided in to two parallel blocks of four tasks each. Can easily be converted to Strassen with $\widehat{Q}_\alpha(n; M, B) = (n^{(\log_2 7)/2}/B)/\sqrt{M+1}$ and the same depth.

```
function **MatInv**(A)
  if ($\#a \leq k$) then InvertSmall(A)
  $(A_{11}, A_{12}, A_{21}, A_{22}) =$ split($A$) ;
  $A_{22}^{-1} =$ **MatInv**$(A_{22})$ ;
  $S = A_{11} - A_{12}A_{22}^{-1}A_{21}$ ;
  $C_{11} =$ **MatInv**$(S)$ ;
  $C_{12} = C_{11}A_{12}A_{22}^{-1}$ ; $C_{21} = -A_{22}^{-1}A_{21}C_{11}$ ;
  $C_{22} = A_{22}^{-1} + A_{22}^{-1}A_{21}C_{11}A_{12}A_{22}^{-1}$ ;
  return join($C_{11}, C_{12}, C_{21}, C_{22}$)
```

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n^{1.5}/B)/\sqrt{M+1})$$

Inverting a $\sqrt{n} \times \sqrt{n}$ matrix using the Schur complement (S). The split operation has to touch any of the four elements at the center of matrix A. The depth is $O(\sqrt{n})$ since the two recursive calls cannot be made in parallel. The parallelism comes from the matrix multiplies.

Figure 4.4: Examples of quicksort, sample sort, matrix multiplication and matrix inversion. All the results in this table are for $0 < \alpha < 1$. Therefore, these algorithms have parallelizability of 1.

```
function **BHT**$(P, (x_0, y_0, s))$
  if $(\#P = 0)$ then return EMPTY
  if $(\#P = 1)$ then return LEAF$(P[0])$
  $x_m = x_0 + s/2$ ; $y_m = y_0 + s/2$ ;
  $P_1 = \{(x, y, w) \in P | x < x_m \wedge y < y_m\}$ ||
  $P_2 = \{(x, y, w) \in P | x < x_m \wedge y \geq y_m\}$ ||
  $P_3 = \{(x, y, w) \in P | x \geq x_m \wedge y < y_m\}$ ||
  $P_4 = \{(x, y, w) \in P | x \geq x_m \wedge y \geq y_m\}$ ||
  $T_1 = $ **BHT**$(P_1, (x_0, y_0, s/2))$ ||
  $T_2 = $ **BHT**$(P_2, (x_0, y_0 + y_m, s/2))$ ||
  $T_3 = $ **BHT**$(P_3, (x_0 + x_m, y_0, s/2))$ ||
  $T_4 = $ **BHT**$(P_4, (x_0 + x_m, y_0 + y_m, s/2))$ ||
  $C = $ CenterOfMass$(T_1, T_2, T_3, T_4)$ ;
  return NODE$(C, T_1, T_2, T_3, T_4)$
```

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n/B) \log(n/(M+1)))$$

The two dimensional Barnes Hut code for constructing the quadtree. The bounds make some (reasonable) assumptions about the balance of the tree.

```
function **ConvexHull**$(P)$
  $P'$ =**SampleSort**($P$ by $x$ coordinate) ;
  return **MergeHull**$(P')$

function **MergeHull**$(P)$
  if $(\#P = 0)$ then return EMPTY
  Touch $P[n/2]$;
  $H_L = $ **MergeHull**$(P[0, \ldots, n/2])$ ||
  $H_R = $ **MergeHull**$(P[n/2, \ldots, n])$ ;
  return bridgeHulls$(H_L, H_R)$
```

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n/B) \log_{M+2} n)$$

The two dimensional Convex Hull code (for finding the upper convex hull). The bridgeHulls routine joins two adjacent hulls by doing a dual binary search and only requires $O(\log n)$ work. The cost is dominated by the sort.

```
function **SparseMxV**$(A, x)$
  parallel for $i \in [0, \ldots, n)$
    $r_i = $ sum$(\{v \times x_j | (v, j) \in A_i\})$
  return $r$
```

$$\widehat{Q}_\alpha(n, m; M, B) \quad = \quad O(m/B + n/(M+1)^{1-\gamma})$$

Sparse vector matrix multiply on a matrix with n rows and m non-zeros. We assume the matrix is in compressed sparse row format and A_i indicates the i^{th} row of A. The bound assumes the matrix has been diagonalized using recursive separators and the edge-separator size is $O(n^\gamma)$ [38]. The parallel for loop at the top should be done in a divide and conquer fashion by cutting the index space in half. At each such fork, the algorithm should touch at least one element from the middle row. The sum inside the loop can also be executed in parallel.

Figure 4.5: Examples of Barnes Hut tree construction, convex hull, and a sparse matrix by dense vector multiply. All the results in this table are for $0 < \alpha < 1$. Therefore, these algorithms have parallelizability of 1.

Chapter 5

Schedulers

A schedule specifies the location and order of execution for each instruction in the DAG. Since DAGs are allowed to be dynamic, schedulers should be able to make online decisions. A scheduler is designed for each class of machines, or a machine model keeping in the mind the constraints and the features of the machine. A good scheduler makes the right trade-off between load balancing the processors and preserving the locality of an algorithm. The quality of a schedule for a given machine and algorithm is measured in terms of running time, the maximum space it requires, and the amount of communication it generates.

As proposed in our solution approach, we separate the task of algorithm and scheduler design. It is the task of the algorithm designer to design algorithms that are good in the program-centric cost model. On the other hand, the scheduler is designed per machine model and is required to translate the cost in the program-centric model to good performance on the machine. Performance bounds are typically provided in terms of the program-centric cost model and the parameters that characterize the machine.

This chapter starts with a definition of schedulers (Section 5.1) and surveys previous work on scheduling for simple machine models (Section 5.2). We will then present new results on scheduling for the PMH machine model starting with a definition for space-bounded schedulers and proceeding through the construction of a specific class of space-bounded schedulers (Section 5.3 with performance guarantees on communication costs in terms of the parallel cache complexity (Section 5.3.1) and bounds on running time based on the extended cache complexity metric (Section 5.3.2). We will end this chapter with bounds on the space requirements of dynamically allocated programs in terms of their effective cache complexity (Section 5.3.3). By demonstrating that program-centric metrics of the PCC framework can be translated into provable performance guarantees on realistic machine models, this chapter validates the claim that such cost models are relevant and useful.

5.1 Definitions

A scheduler defines where and when each node in a DAG is executed on a certain machine. We differentiate between preemptive and non-preemptive schedulers based on whether a strand is executed on the same processor or is allowed to migrate for execution on multiple processors.

We consider only non-preemptive schedulers.

A non-preemptive schedule defines three functions for each strand ℓ. Here, we use P to denote the set of processors on the machine, and L to denote the set of strands in the computation.

- **Start time:** $start : \mathsf{L} \to \mathbb{Z}$, where $start(\ell)$ denotes the time the first instruction of ℓ begins executing,
- **End time:** $end : \mathsf{L} \to \mathbb{Z}$, where $end(\ell)$ denotes the time the last instruction of ℓ finishes – this depends on both the schedule and the machine, and
- **Location:** $proc : \mathsf{L} \to P$, where $proc(\ell)$ denotes the processor on which the strand is executed.

A non-preemptive schedule cannot migrate strands across processors once they begin executing, so $proc$ is well-defined. We say that a strand ℓ is ***live*** at any time τ with $start(\ell) \leq \tau < end(\ell)$.

A non-preemptive schedule must also obey the following constraints on ordering of strands and timing:

- (*ordering*): For any strands $\ell_1 \prec \ell_2$, $start(\ell_2) \geq end(\ell_1)$.
- (*processing time*): For any strand ℓ, $end(\ell) = start(\ell) + \gamma_{\langle \text{schedule,machine} \rangle}(\ell)$. Here γ denotes the processing time of the strand, which may vary depending on the specifics of the machine and the history of the schedule. The point is simply that the schedule has no control over this value.
- (*non-preemptive execution*): No two strands may run on the same processor at the same time, i.e., $\ell_1 \neq \ell_2, proc(\ell_1) = proc(\ell_2) \implies [start(\ell_1), end(\ell_1)) \cap [start(\ell_2), end(\ell_2)) = \emptyset$.

We extend the same notation and terminology to tasks. The start time $start(\mathsf{t})$ of a task t is a shorthand for $start(\mathsf{t}) = start(\ell_s)$, where ℓ_s is the first strand in t. Similarly $end(\mathsf{t})$ denotes the end time of the last strand in t. The location function $proc$, however, is not defined for a task as each strand inside the task may execute on a different processor.

When discussing specific schedulers, it is convenient to consider the time a task or strand first becomes available to execute. We use the term ***spawn time*** to refer to this time, which is exactly the instant at which the preceding fork or join finishes. Naturally, the spawn time is no later than the start time, but schedule may choose not to execute the task or strand immediately. We say that the task or strand is ***queued*** during the time between its spawn time and start time and *live* during the time between its start time and finish time. Figure 5.1 illustrates the spawn, start and end times of a task its initial strand. They are spawned and start at the same time by definition. The strand is continuously executed till it ends, while a task goes through several phases of execution and idling before it ends.

5.2 Mapping Sequential Cache Complexity – Scheduling on One-level Cache Models

This section describes two schedulers for the one-level cache models in Figure 1.2 — the work-stealing scheduler for the private cache mdoel and the PDF scheduler for the shared cache model

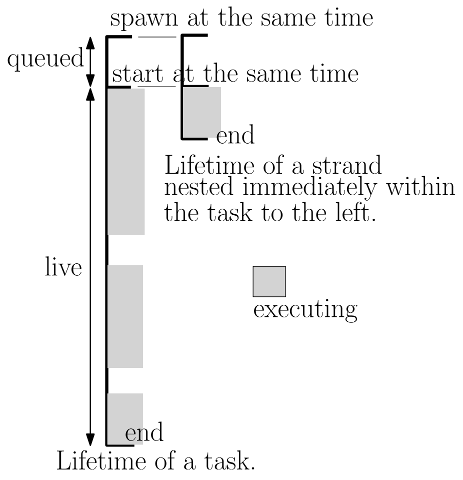


Figure 5.1: Lifetime of a task, and of a strand nested directly beneath it. Note that the task can go through multiple execution and idle phases corresponding to the execution of its fragments, but the strand executes continuously in one stretch.

— and presents performance guarantees on communication costs, time and space in terms of the sequential cache complexity Q_1, work W and depth D. Note that W is superfluous as Q_1 subsumes work. It can be obtained by setting $M = 0, B = 1$ in the sequential complexity: $W = Q_1(0, 1)$.

5.2.1 Work-Stealing Scheduler

A good choice of scheduler for the private-cache model in Figure 1.2 is the Work-Stealing scheduler. The Work-Stealing scheduler assigns one double-ended queue (dequeue) for each processor. The top-level task to be executed is introduced into one of the queues. Each processor then proceeds as follows:

- If a processor spawns tasks at a fork, it continues execution with one of the spawned subtasks, and queues the rest at the front of the queue.
- If a processor runs completes a task and has no instructions left to execute, it tries to pull a task from the front of its queue.
- If a processor does not find tasks in its own queue, it randomly selects a victim queue from other processors, and steals a task from the end of the victim queue. If that fails, it retries till it succeeds.

Work-stealing schedules are very good at load balancing because they are greedy. Since the scheduler differentiates between “local” and “remote” work based on queues, it can also be shown to preserve locality well on single-level private caches, especially if the depth of the program is small . Intuitively, a low-depth program does not present many opportunities for stealing [48]. Since each steal involves a context switch and deviation from the sequential schedule, the fewer the number of steals, the smaller the difference between sequential cache complexity Q_1 and the total number of cache misses (see [2] for an analysis). More precisely, the total number of cache misses across all caches Q_P for a p processor machine processor machines is bounded by

$$\text{WS: } Q_P \leq Q_1(M, B) + O(pD_l M/B) \tag{5.1}$$

in expectation. Assuming that each cache miss costs C cycles and cache hits cost 1 cycle, the schedule completes in time

$$\text{WS: } T \leq (W + C \cdot Q_P)/p + D_l \tag{5.2}$$

in expectation because it is greedy.

To prove bounds on the space requirement of dynamically allocated program under parallel execution required the following assumption. A location assigned to a variable at a node n_a in the DAG can only be freed by node n_f if n_a and n_f are at the same nesting depth in the DAG. Under this assumption, it can be shown [47] the work-stealing executions have space requirement at most

$$\text{WS: } S_P \leq pS_1. \tag{5.3}$$

Note that these performance bounds involve purely program-centric metrics – work, depth, and sequential cache complexity.

These performance bounds ignore the overhead of the scheduler itself. It is essential to engineer and integrate the scheduler well into the run-time system. The simplicity of the Work-Stealing scheduler allows this – many parallel programming systems, including Intel Cilk+, Microsoft TPL, Intel Thread-Building Blocks, and X10, use this system. Specific examples of engineering for efficiency include the work-first policy [85, 90], the low-overhead dequeue protocol *THE* [85] that optimizes the common-case local dequeuing operation, and memory mapping support for cactus stacks [115].

Work-Stealing can be shown to work well on the multi-level private cache model in Figure 2.2 with k levels of cache and one large memory unit at level $k+1$. Let M_i and B_i respectively denote the cache size and cache line size of a level i cache. Given a particular execution X of a computation on some parallel machine M, let $W_M^{lat}(X)$ be $W + \sum_i (C_i \cdot F_{M,i}(X))$, where $F_{M,i}(X)$ is the number of cache misses at level i and C_i the cost of a cache miss. The *latency-added work*, W_M^{lat}, of a computation is the maximum of $W_M^{lat}(X)$ over all executions X. The *latency added depth* D_M^{lat} can be defined similarly. We note that $W \cdot C_k$ and $D \cdot C_k$ are pessimistic upper bounds on the latency-added work and depth for any machine.

5.2.1.1 Bounds for the multi-level Private-Cache model

Theorem 5 (Upper Bounds) *For any $\delta > 0$, when a cache-oblivious nested-parallel computation with binary forking, sequential cache complexity $Q_1(M, B)$, work W, and depth D is scheduled on a multi-level private-cache machine P of p processors using work stealing:*

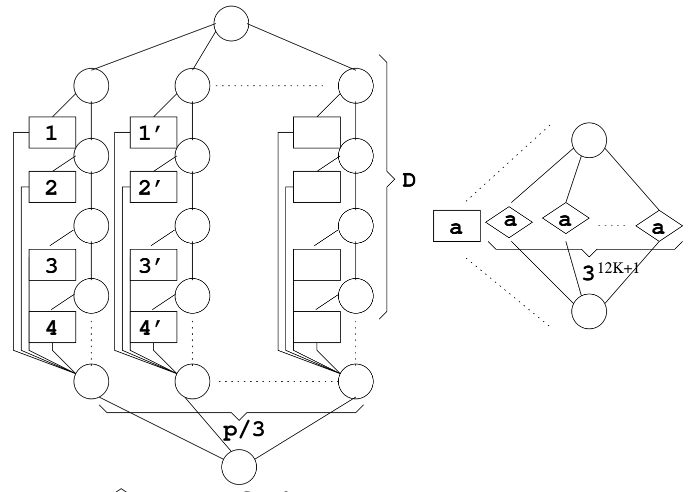
- *The number of steals is $O(p(D_P^{lat} + \log 1/\delta))$ with probability at least $1 - \delta$.*
- *All the caches at level i incur a total of less than $Q_1(M_i, B_i) + O(p(D_P^{lat} + \log 1/\delta)M_i/B_i)$ cache misses with probability at least $1 - \delta$.*

- *The computation completes in time not more than $W_P^{lat}/p + D_P^{lat} < W/p + O((DC_k + \log 1/\delta)C_k M_k/B_k) + O(\sum_i C_i(Q_1(B_{i-1}, B_{i-1}) - Q_1(M_i, B_i)))/p$ with probability at least $1 - \delta$.*

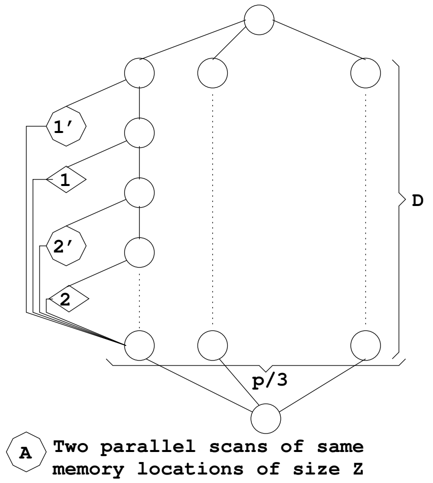
Proof. Results from [22] imply the statement about the number of steals. Because the schedule involves not more than $O(p(D_P^{lat} + \log 1/\delta))$ steals with probability at least $1 - \delta$, all the caches at level i incur a total of at most $Q_1(M_i, B_i) + O(p(D_P^{lat} + \log 1/\delta)M_i/B_i)$ cache misses with probability at least $1 - \delta$. To compute the running time of the algorithm, we count the time spent by a processor waiting upon a cache miss towards the work and use the same proof as in [22]. In other words, we use the notion of latency-added work (W_P^{lat}) defined above. Because this is not more than $W + O(p(D_P^{lat} + \log 1/\delta)C_k M_k/B_k + \sum_i C_i(Q_1(M_{i-1}, B_{i-1}) - Q_1(M_i, B_i)))$ with probability at least $1 - \delta$, the claim about the running time follows. □

Thus, for constant δ, the parallel cache complexity at level i exceeds the sequential cache complexity by $O(pD_P^{lat}M_i/B_i)$ with probability $1-\delta$. This matches the earlier bounds presented for the single-level case.

We can also consider a *centralized* work stealing scheduler. In *centralized work stealing*, processors deterministically steal a node of least depth in the DAG; this has been shown to be a good choice for reducing the number of steals [22]. The bounds in Theorem 5 carry over to centralized work stealing without the δ terms, e.g., the parallel cache complexity exceeds the sequential cache complexity by $O(pD_P^{lat}M_i/B_i)$.



(a) Randomized work stealing



(b) Centralized work stealing

Figure 5.2: DAGs used in the lower bounds for randomized and centralized work stealing.

The following lower bound shows that there exist DAGs for which Q_p does indeed exceed Q_1 by the difference stated above.

Theorem 6 (Lower Bound) *For a multi-level private-cache machine P with any given number of processors $p \geq 4$, cache sizes $M_1 < \cdots < M_k \leq M/3$ for some a priori upper bound M, cache line sizes $B_1 \leq \cdots \leq B_k$, and cache latencies $C_1 < \cdots < C_k$, and for any given depth $D' \geq 3(\log p + \log M) + C_k + O(1)$, we can construct a nested-parallel computation DAG with* *binary forking and depth D', whose (expected) parallel cache complexity on P, for all levels i, exceeds the sequential cache complexity $Q_1(M_i, B_i)$ by $\Omega(pD_P^{lat}M_i/B_i)$ when scheduled using randomized work stealing. Such a computation DAG can also be constructed for centralized work stealing.*

Proof. Such a construction is shown in Figure 5.2(a) for randomized work stealing. Based on the earlier lemma, we know that there exist a constant K such that the number of steals is at most KpD^{lat} with probability at least $1 - (1/D^{lat})$. We construct the DAG such that it consists of a binary fanout to $p/3$ spines of length $D = D' - 2(K + \log(p/3) + \log M)$ each. Each of the first $D/2$ nodes on the spine forks off a subdag that consists of $3^{(12K+1)}$ identical parallel scan structures of length M each. A scan structure is a binary tree forking out to M parallel nodes that collectively read a block of M consecutive locations. The remaining $D/2$ nodes on the spine are the joins back to the spine of these forked off subdags. Note that $D_P^{lat} = D' + C_k$ because each path in the DAG contains at most one memory request.

For any M_i and B_i, the sequential cache complexity $Q_1(M_i, B_i) = (p/3)(M_i + L + i + 3^{(12K+1)}(M - M_i + B_i))(D/2)/B_i$ because the sequential execution executes the subdags one by one and can reuse a scan segment of length M_i/B_i for all the identical scans to avoid repeated misses on a set of locations. In other words, sequential execution gets $(p/3)(3^{(12K+1)} - 1)(M_i - B_i))(D/2)/B_i$ cache hits because it executes identical scans one after the other.

We argue that in the case of randomized work stealing, there are a large number of subdags such that the probability that at least two scans from the subdag are repeated are executed by disjoint set of processors is greater than some positive constant. This implies that the cache complexity is $\Theta(pDM_i/B_i)$ higher that the sequential cache complexity.

1. Once the $p/3$ spines have been forked, each spine is occupied by at least one processor till the stage where work along a spine has been exhausted. This property follows directly from the nature of the work stealing protocol.
2. In the early stages of computation after spines have been forked, but before the computation enters the join phase on the spines, exactly $p/3$ processors have a spine node on the head of their work queue. Therefore, the probability that a random steal with get a spine node and hence a fresh subdag is $1/3$.
3. At any moment during the computation, the probability that more than $p/2$ of the latest steals of the p processors found fresh spine nodes is exponentially small in terms of p and therefore less than $1/2$. This follows from the last observation.
4. If processor p stole a fresh subdag A and started the scans in it, the probability that work from the subdag A is not stolen by some other processor before p executes the first $2/3$-rd of the scan is at most a constant $c_h \in (0, 1)$. This is because, the event that p currently chose a fresh subdag is not correlated with any event in the history, and therefore, with probability at least $1/2$, more than $p/2$ processors did not steal a fresh subdag in the latest steal. This means that these processors which stole a stale subdag (a subdag already being worked on by other processors) got less than $2/3$-rd fraction of the subdag to work on before they need to steal again. Therefore, by the time p finishes $2/3$-rd of the work, there would have been at least $p/2$ steal attempts. Since these steals are randomly distributed over all processors, there is a probability of at least $1/16$ that two of these steals where

from p. Two steals from p would cause p to lose work from it's fresh subdag.

5. Since there are at most KpD steals with high probability, there no more $pD/6$ subdags which incur more than $12K$ steals. Among nodes with fewer than $12K$ steals, consider those described in the previous scenario where the processor p that started the subdag has work stolen from it before p executes 2/3-rd of the subdag. At least $(1/16)(5pD/6)$ such subdags are expected in a run. Because there are 3^{12K+1} identical scans in each such subdag, at least one processor apart from p that has stolen work from this subdag gets to execute one complete scan. This means that the combined cache complexity at the i-th cache level for each subdag is at least M_i/B_i greater than the sequential cache complexity, proving the lemma.

The construction for lower bounds on centralized work stealing is shown in Figure 5.2(b). The DAG ensures that the least-depth node at each steal causes a scan of a completely different set of memory locations. The bound follows from the fact that unlike the case in sequential computation, cache access overlap in the pairs of parallel scans are never exploited. □

5.2.1.2 Implications for the Parallel Tree-of-Caches

The bounds in Theorem 5 hold for any tree of caches: simply view the tree of caches as a multi-level private-cache model by partitioning each proportionate cache shared by $p_{i,j}$ processors into $p_{i,j}$ disjoint caches of size M_i. However, the performance bounds are suboptimal since shared caches are partitioned and not constructively used by the WS scheduler.

5.2.2 PDF Scheduler

The shared-cache model (Figure 1.2) requires a schedule that is completely different from the work-stealing scheduler. The WS scheduler is a coarse-grained scheduler that tries to partition the DAG into large chunks just enough to load balance the processors. It is bad at constructively sharing a cache among processors as it works with too many different parts of the computation at once, and thus overloading the cache. A good scheduler for the the shared-cache model should be able to schedule instructions that are close together in the DAG simultaneously to use the locality designed into the DAG.

One strategy is to design a parallel scheduler that mimics a sequential depth-first schedule so as to capture the locality built into that order by the programmer. The Parallel Depth-First scheduler [37, 134] is one such schedule. It first prioritizes the nodes in the DAG according to their order in some sequential execution. A parallel schedule then tries to follow the priority as closely as possible while being greedy. It is possible to show that the number of nodes that have been executed out-of-order from the priority dictated by the depth-first order is bounded by the product of the number of processors and the depth of the computation. If this number is small, the number of cache misses of the sequential schedule on cache size M bounds the number of cache misses incurred by this schedule on a shared cache of slighlty larger size (larger by the number of premature nodes):

$$\text{PDF: } Q_P(M + pBD_l; B) \leq Q_1(M; B), \tag{5.4}$$

where $D_l = DC$, C being the cost of a cache miss. Since, the PDF schedule is greedy, its running time is bounded by

$$T \leq (W + C \cdot Q_P)/p + D_l. \tag{5.5}$$

Further, when space is dynamically allocated, the space requirement of the PDF schedule does not exceed the space requirement of the sequential schedule it is based on by more than the number of premature nodes:

$$S_p \leq S_1 + pD_l. \tag{5.6}$$

Since the PDF scheduler makes decisions after every node, its overhead can be very high. Some coarsening of the granularity at which scheduling decisions are made such as in AsyncDF (Section 3 of [134]) is needed to bound scheduler overheads while trading off slightly on space requirements and locality. Further loosening of the scheduling policy to allow for Work-Stealing at the lowest granularities while using PDF at higher levels (DFDequeues, Section 6 of [134]) may be also be useful.

5.2.2.1 Extending Shared Cache Results to Multiple Levels

Finally, we note that the bounds for single level of shared cache can be extended to machines with multiple levels of shared caches like Figure 2.2.

Theorem 7 *When a DAG with sequential cache complexity $Q_1(M, L)$, work W, and depth D is scheduled on a Multi-level Shared-Cache machine P of p processors using a PDF scheduler, then the cache at each level i incurs fewer than $Q_1(M_i - pB_iD_P^{lat}, B_i)$ cache misses. Moreover, the computation completes in time not more than $W_P^{lat}/p + D_P^{lat}$.*

Proof. The cache bound follows because (i) inclusion implies that hits/misses/evictions at levels $< i$ do not alter the number of misses at level i, (ii) caches sized for inclusion imply that all words in a line evicted at level $> i$ will have already been evicted at level i, and hence (iii) the key property of PDF schedulers, $Q_P(M + pLD_P^{lat}, B) \leq Q_1(M, B)$ holds at each level i cache. The time bound follows because the schedule is greedy. □

5.3 Mapping PCC Cost Model – Scheduling on PMH

A tree of caches can be viewed as an interleaving of the private and shared cache models at every level. The two previously described schedulers are not suitable for preserving locality at each level on a tree of caches. A new class of schedulers called space-bounded scheduler were proposed by Chowdhury et al. [64] for tree of caches. Roughly speaking, a space-bounded scheduler accepts dynamic DAGs that have been annotated with space requirements for each task. These schedulers run every task in a cache that just fits it (i.e., no lower cache will fit it), and once assigned, tasks are not migrated across caches. But their paper does not use the PCC framework, can not handle *irregular* tasks and hence cannot show the same kind of optimal bounds on running time such as in Theorem 8. Related to their earlier work, Cole and Ramachandran [68] describe the Priority Work-Stealing scheduler with strong asymptotic bounds on cache misses and runtime, but only for highly balanced computations.

In this section we formally define the class of space-bounded schedulers, and show (Theorem 8) that such schedulers have cache complexity on the PMH machine model that matches the parallel cache complexity. For simplify of presentation, we assume that program are stitcally allocated and therefore the space of a program is scheduler independent. We identify, by construction, specific schedules within that class that have provable guarantees on time and space based the effective cache complexity.

Informally, a space-bounded schedule satisfies two properties:

- *Anchored*: Each task is anchored to a smallest possible cache that is bigger than the task—strands within the task can only be scheduled on processors in the tree rooted at the cache.
- *Bounded*: A "maximal" live task "occupies" a cache X if it is either (i) anchored to X, or (ii) anchored to a cache in a subcluster below X while its parent is anchored above X. A live strand occupies cache X if it is live on a processor beneath cache X and the strand's task is anchored to an ancestor cache of X. The sum of sizes of live tasks and strands that occupy a cache is restricted by the scheduler to be less than the size of the cache.

These two conditions are sufficient to imply good bounds on the number of cache misses at every level in the tree of caches. A good space-bounded scheduler would also handle load balancing subject to anchoring constraints to quickly complete execution.

Space-Bounded Schedulers. A "space-bounded scheduler" is parameterized by a global ***dilation*** parameter $0 < \sigma \leq 1$ and machine parameters $\{M_i, B_i, C_i, f_i\}$. We will need the following terminology for the definition.

Cluster: For any cache X_i, recall that its cluster is the set of caches and processors nested below X_i. We use $P(X_i)$ to denote the set of processors in X_i's cluster and $\$(X_i)$ to denote the set of caches in X_i's cluster. We use $X_i \subseteq X_j$ to mean that cache X_i is in cache X_j's cluster.

Befitting Cache: Given a particular cache hierarchy and dilation parameter $\sigma \in (0, 1]$, we say that a level-i cache ***befits*** a task t if $\sigma M_{i-1} < S(\mathsf{t}, B_i) \leq \sigma M_i$.

Maximal Task: We say that a task t with parent task t' is ***level-i maximal*** if and only if a level-i cache befits t but not t', i.e., $\sigma M_{i-1} < S(\mathsf{t}, B_i) \leq \sigma M_i < S(\mathsf{t}', B_i)$.

Anchored: A task t with strand set $\mathsf{L}(t)$ is said to be ***anchored*** to level-i cache X_i (or equivalently to X_i's cluster), if and only if a) it runs entirely in the cluster, i.e., $\{proc(\ell) | \ell \in \mathsf{L}(t)\} \subseteq P(X_i)$, and b) the cache befits the task. Anchoring prevents the migration of tasks to a different cluster or cache. The advantage of anchoring a task to a befitting cache is that once it load its working set, it can reuse it without the risk of losing it from the cache. If a task is not anchored anywhere, for notational convenience we assume it is anchored at the root of the tree.

As a corollary, suppose t' is a subtask of t. If t is anchored to X and t' is anchored to X', then $X' \subseteq X$. Moreover, $X \neq X'$ if and only if t is maximal. In general, we are only concerned about where maximal tasks are anchored.

Cache occupying tasks: For a level-i cache X_i and time τ, the set of live tasks ***occupying cache X_i at time τ***, denoted by $Ot(X_i, \tau)$, is the union of (a) maximal tasks anchored to X_i that are live at time τ, and (b) maximal tasks anchored to any cache in $\$(X_i) \setminus \{X_i\}$, live at time τ, with their immediate parents anchored to a cache above X_i in the hierarchy. The tasks in (b) are called "skip level" tasks. Tasks described in (a) and (b) are the tasks that consume space in the cache at time τ. Note that the set of tasks in (b) can be dropped from the Ot without an asymptotic change in cache miss bounds if the cache hierarchy is "strongly" inclusive,i.e. , $M_i > cf_iM_{i-1}$ at all levels i of PMH for some constant $c > 1$

Cache occupying strands: The set of live strands ***occupying cache X_i at time τ***, denoted by $Ol(X_i, \tau)$, is the set of strands $\{\ell\}$ such that (a) ℓ is live at time τ, i.e., $start(\ell) \leq \tau < end(\ell)$, (b) ℓ is processed below X_i, i.e., $proc(\ell) \in P(X_i)$, and (c) ℓ's task is anchored strictly above X_i.

A ***space-bounded scheduler*** for a particular cache hierarchy is a scheduler parameterized by $\sigma \in (0, 1]$ that satisfies the two following properties:

- *Anchored* Every subtask t of the root task with is anchored to a **befitting** cache.
- *Bounded*: At every instant τ, for every level-i cache X_i, the sum of sizes of cache occupying tasks and strands is less then M_i:

$$\sum_{\mathsf{t} \in Ot(X_i,\tau)} S(\mathsf{t}, B_i) + \sum_{\ell \in Ol(X_i,\tau)} S(\ell, B_i) \leq M_i.$$

We relax the usual definition of greedy scheduler in the following: A ***greedy-space-bounded scheduler*** is a space-bounded scheduler in which a processor remains idle only if there is no ready-to-execute strand that can be anchored to the processor (and appropriate ancestor caches) without violating the space-bounded constraints.

5.3.1 Communication Cost Bounds

In this section, we will show that parallel cache complexity is realized by Space-Bounded Schedulers. The following theorem implies that a nested-parallel computation scheduled with any space-bounded scheduler achieves optimal cache performance, with respect to the parallel cache complexity. The main idea of the proof is that each task reserves sufficient cache space and hence never needs to evict a previously loaded cache line.

Theorem 8 *Consider a PMH and any dilation parameter $0 < \sigma \leq 1$. Let t be a level-i task. Then for all memory-hierarchy levels $j \leq i$, the number of level-j cache misses incurred by executing t with any space-bounded scheduler is at most $Q^*(\mathsf{t}; \sigma M_j, B_j)$.*

Proof. Let U_i be the level-i cache to which t is assigned. Observe that t uses space at most σM_i. Moreover, by definition of the space-bounded scheduler, the total space needed for tasks assigned to U_i is at most M_i, and hence no line from t need ever be evicted from U's level-i cache. Thus, an instruction x in t accessing a line ℓ does not exhibit a level-i cache miss if there is an earlier-executing instruction in t that also accesses ℓ. Any instruction serially preceding x must execute earlier than x. Hence, the parallel cache complexity $Q^*(\mathsf{t}; \sigma M_i, B_i)$ is an upper bound on the actual number of level-i cache misses.

We next extend the proof for lower-level caches. First, let us consider a level-i strand s belonging to task t. The definition of the parallel cache complexity cost model states that for any $M_{j<i}$, the cache complexity of a level-i strand matches the serial cache complexity of the strand beginning from an initially empty state. Consider each cache partitioned such that a level-i

strand can use only the σM_j capacity of a level-$(j < i)$ cache awarded to it by the space-bounded scheduler. Then the number of misses is indeed as though the strand executed on a serial level-$(i-1)$ memory hierarchy with σM_j cache capacity at each level j. Hence, $Q^*(\mathsf{s}; \sigma M_j, B_j)$ is an upper bound on the actual number of level-j cache misses incurred while executing the strand s. (The actual number may be less because an optimal replacement policy may not partition the caches and the cache state is not initially empty.)

Finally, to complete the proof for all memory-hierarchy levels j, we assume inductively that the theorem holds for all maximal subtasks of t. The defintion of parallel cache complexity assumes an empty initial level-j cache state for any maximal level-j subtask of t, as $S(\mathsf{t}; B_j) > \sigma M_j$. Thus, the level-$j$ cache complexity for t is defined as $Q^*(\mathsf{t}; \sigma M_j, B_j) = \sum_{\mathsf{t}' \in A(\mathsf{t})} Q^*(\mathsf{t}'; \sigma M_j, B_j, \emptyset)$, where $A(\mathsf{t})$ is the set of all level-i strands and nearest maximal subtasks of t. Since the theorem holds inductively for those tasks and strands in $A(\mathsf{t})$, it holds for t. □

In contrast, there is no such optimality result based on sequential cache complexity – Theorem 1 (showing a factor of p gap) readily extends to any greedy-space-bounded scheduler, using the same proof.

5.3.2 Time bounds

While all space-bounded schedulers achieve optimal cache complexity, they vary in total running time. We first describe a simple "greedy-space-bounded" scheduler that performs very well in terms of runtime on very balanced computations, and use it to highlight some of the difficulties in designing a scheduler that permits imbalance. The analysis for this scheduler is not based on the PCC framework and does not separate algorithm and scheduler. To fix this, we will modify the simple scheduler to handle more "irregularities" as quantified in the effectice cache complexity cost model. Finally, we will present a general theorem for the running time of this class of modified space-bounded schedulers in terms of the effectice cache complexity (Theorem 10).

5.3.2.1 A Simple Space-Bounded Scheduler and its Limitations.

Greedy-space-bounded schedulers, like the scheduler in [64], perform well for computations that are very well balanced. At a high level, a greedy-space-bounded scheduler operates on tasks anchored at each cache. These tasks are "unrolled" to produce maximal tasks, which are in turn anchored at descendant caches. If a processor P becomes idle and a strand is ready, we assume P begins working on a strand immediately (i.e., we ignore scheduler overheads). If multiple strands are available, one is chosen arbitrarily.

5.3.2.2 Operational details of the simple scheduler.

Before discussing how the scheduler operates, we address state maintained by the scheduler. We maintain for each level-i cache U the total space used by maximal tasks anchored to U.

We maintain for each anchored maximal level-i task t two lists. The ***ready-strand list*** $\mathcal{S}(\mathsf{t})$ contains the ready level-i strands of t. The ***ready-task list*** $\mathcal{R}(\mathsf{t})$ contains the ready, unexecuted (and hence unanchored), maximal level-$(j < i)$ subtasks of t. The ready-task list contains only maximal tasks, as tasks which are not maximal are implicitly anchored to the same cache as t.[1]

A strand or task is assigned to the appropriate list when it first becomes ready. We use $parent[\mathsf{t}]$ to denote the nearest containing task of task/strand t, and $maximal[\mathsf{t}]$ to denote the nearest maximal task containing task/strand t ($maximal[\mathsf{t}] = \mathsf{t}$ if t is maximal). When the execution of a strand reaches a fork, we also keep a count on the number of outstanding subtasks, called the join counter.

Initially, the main task t is anchored at the smallest cache U in which it fits, both $\mathcal{R}(\mathsf{t})$ and $\mathcal{S}(\mathsf{t})$ are created, and t is allocated all subclusters of U (ignore for the greedy scheduler). The leading strand of the task is added to the ready-strand list $\mathcal{S}(\mathsf{t})$, and all other lists are empty.

The scheduler operates in parallel, invoking either of the two following scheduling rules when appropriate. We ignore the precise implementation of the scheduler, and hence merely assume that the invocation of each rule is atomic. For both rule descriptions, let t be a maximal level-i task, let U be the cache to which it is anchored, and let $\mathcal{U}_t$ be all the children of U for the greedy scheduler.

strands We say that a cache U_j is ***strand-ready*** if there exists a cache-to-processor path of caches $U_j, U_{j-1}, \ldots, U_1, p$ descending from U_j down to processor P such that: 1) each $U_k \in \{U_j, \ldots, U_1\}$ has σM_k space available, and 2) P is idle. We call $U_j, U_{j-1}, \ldots, U_1, p$ the ***ready path***. If $\mathcal{S}(\mathsf{t})$ is not empty and $V \in \mathcal{U}_t$ is strand-ready, then remove a strand s from $\mathcal{S}(\mathsf{t})$. Anchor s at all caches along the ready path descending from V, and decrease the remaining capacity of each U_k on the ready path by σM_k. Execute the strand on the ready path's processor.

tasks Suppose there exists some level-$(j-1) < (i-1)$ strand-ready descendant cache U_{j-1} of $V \in \mathcal{U}_t$, and let U_j be its parent cache. If there exists level-j subtask $\mathsf{t}' \in \mathcal{R}(\mathsf{t})$ such that U_j has $S(\mathsf{t}'; B_j)$ space available, then remove t' from $\mathcal{R}(\mathsf{t})$. Anchor t' at U_j, create $\mathcal{R}(\mathsf{t}')$ and $\mathcal{S}(\mathsf{t}')$, and reduce U_j's remaining capacity appropriately. Then schedule and execute the first strand of t' as above.

Observe that the implementation of these rules can either be top-down with caches pushing tasks down or bottom up with idle processors "stealing" and pulling tasks down. To be precise about where pieces of the computation execute, we've chosen both scheduling rules to cause a strand to execute on a processor. This choice can be relaxed without affecting the bounds.

When P has been given a strand s to execute (by invoking either scheduling rule), it executes the strand to completion. Let $\mathsf{t} = parent[\mathsf{s}]$ be the containing task. When s completes, there are two cases:

Case 1 (fork). When P completes the strand and reaches a fork point, t's join counter is set to the number of forked tasks. Any maximal subtasks are inserted into the ready-task list $\mathcal{R}(maximal[\mathsf{t}])$, and the lead strand of any other (non maximal) subtask is inserted into the ready-strand list $\mathcal{S}(maximal[\mathsf{t}])$.

Case 2 (task end). If P reaches the end of the task t, then the containing parent task $parent[\mathsf{t}]$'s join counter is decremented. If $parent[\mathsf{t}]$'s join counter reaches zero, then the subsequent strand in $parent[\mathsf{t}]$ is inserted into $maximal[parent[\mathsf{t}]]$'s ready-strand list. Moreover, if t is a maximal

[1]For this greedy scheduler, a pair of lists for each cache is sufficient, i.e. , $\mathcal{R}(U) = \bigcup_{\mathsf{t} \text{ anchored to } U} \mathcal{R}(\mathsf{t})$, and similarly for $\mathcal{S}$, but for consistency of notation with later schedulers we continue with a pair of lists per task here.

level-i task, the total space used in by the cache to which t was anchored is decreased by $S(\mathsf{t}; B_i)$ to reflect t's completion.
In either of these cases, P becomes idle again, and scheduling rules may be invoked.

Space-bounded schedulers like this greedy variant perform well for computations that are very balanced. Chowdhury et al. [64] present analyses of a similar space-bounded scheduler (that includes minor enhancements violating the greedy principle). These analyses are algorithm specific and rely on the balance of the underlying computation.

We prove a theorem (Th. 9) to provides run time bounds of the same flavor for the greedy space-bounded scheduler, leveraging many of the same assumptions on the underlying algorithms. Along with our main theorem (Theorem 10), these analyses all use recursive application of Brent's theorem to obtain a total running time: small recursive tasks are assumed inductively to execute quickly, and the larger tasks are analyzed using Brent's theorem with respect to a single-level machine of coarser granularity.

5.3.2.3 Structural Restrictions on the Simple Space-Bounded Scheduler

The following are the types of informal structural restrictions imposed on the underlying algorithms to guarantee efficient scheduling with a greedy-space-bounded scheduler and previous work. For more precise, sufficient restrictions, see the technical report [39].

1. *When multiple tasks are anchored at the same cache, they should have similar structure and work. Moreover, none of them should fall on a much longer path through the computation.* If this condition is relaxed, then some anchored task may fall on the critical path. It is important to guarantee each task a fair share of processing resources without leaving many processors idle.
2. *Tasks of the same size should have the same parallelism.*
3. *The nearest maximal descendant tasks of a given task should have roughly the same size.* Relaxing this condition allows two or more tasks at different levels of the memory hierarchy to compete for the same resources. Guaranteeing that each of these tasks gets enough processing resources becomes a challenge.

In addition to these balance conditions, the previous analyses exploit preloading of tasks: the memory used by a task is assumed to be loaded (quickly) into the cache before executing the task. For array-based algorithms preloading is a reasonable requirement. When the blocks to be loaded are not contiguous, however, it may be computationally challenging to determine which blocks should be loaded. Removing the preloading requirement complicates the analysis, which then must account for high-level cache misses that may occur as a result of tasks anchored at lower-level caches.

We now formalize the structural restrictions described above and analyze a greedy space-bounded scheduler with ideal dilation $\sigma = 1$. This analysis is intended both as an outline for the main theorem in Section 5.3.2.5 and also to understand the limitations of the simple scheduler that we are trying to ameliorate. These balance restrictions are necessary in order to prove good bounds for the greed scheduler, and that are relaxed in Section 5.3.2.5.

For conciseness, define the ***total work*** of a maximal level-i task t for the given memory hierarchy as $TW(\mathsf{t}) = \sum_{j=0}^{i} C_j Q^*(\mathsf{t}; M_j, B_j)$. For s a level-i strand or task that is not maximal,

$TW(\mathsf{s}) = \sum_{j=0}^{i-1} C_j Q^*(\mathsf{s}; M_j, B_j)$, i.e. , assume all memory locations already reside in the level-i cache.

We say that a task t is ***recursively large*** if any child subtask t' of t (those nested directly within t) uses at least half the space used by t, i.e. , $S(\mathsf{t}'; B) \geq S(\mathsf{t}; B)/2$ for all B, and t' is also recursively large. Recursively large is quite restrictive, but it greatly simplifies both the analysis and further definitions here, as we can assume all subtasks of a level-i task are at least level-$(i-1)$.

Consider a level-i task t. We say that t is ***γ_i parallel at level-i*** if no more than a $1/\gamma_i$ fraction of the total work from all descendant maximal level-i strands of t fall along a single path in t's subdag, and no more than a $1/\gamma_i$ fraction of total work from all descendant level-$(i-1)$ maximal subtasks falls along a single path among level-$(i-1)$ tasks[2] in t's subdag. Moreover, we say that t is ***λ_i task heavy*** if the total work from strands comprises at most a $1/\lambda_i$ factor of the total work from level-$(i-1)$ subtasks.

5.3.2.4 Running time of the simple scheduler

The following theorem bounds the total running time of an algorithm using the greedy space-bounded scheduler.

Theorem 9 *Consider a PMH and dilation parameter $\sigma = 1$. Let t be a maximal level-i task, and suppose that*

- t *is recursively large, and*
- *all memory used by each maximal level-j (sub)task t' can be identified and preloaded into the level-j cache in parallel (in time $C_j S(\mathsf{t}'; B_j)/p_j B_j$) before executing any strands in t'.*

For each level-j, let γ_j and λ_j be values such that all maximal level-j descendent subtasks are γ_j parallel at level j and λ_j task heavy, respectively. Then t executes in

$$\frac{TW(\mathsf{t})}{p_i} \prod_{j=1}^{i} \left(1 + \frac{f_j}{\gamma_j}\right)\left(1 + \frac{p_{j-1}}{\lambda_j}\right)$$

time on a greedy space-bounded scheduler.

Proof. To start with, observe that any maximal level-k task t_k uses at least $M_k/2$ space, and hence the scheduler assigns at most one level-k task to each level-k cache at any time. This fact follows because t_k's parent task t_{k+1} uses strictly more than M_k space, (as otherwise t_k is not maximal), and hence $S(\mathsf{t}_k; B_k) \geq S(\mathsf{t}_{k+1}; B_k)/2 > M_k/2$.

Suppose that the theorem holds for all maximal level-$(k-1)$ tasks, and inductively prove for a maximal level-k task t_k.

Since each level-$(k-1)$ cache is working on at most one subtask or thread at any time, we instead interpret our level-k computation as running at a courser granularity on a machine consisting of only a single level-k cache with f_k processors (in place of the level-$(k-1)$ caches). We then interpret each level-$(k-1)$ task t_{k-1} as a serial computation that has work

[2]That is, ignore the parallelism within the level-$(i-1)$ subtasks — treat each subtask as an indivisible entity, forming a dag over level-$(i-1)$ subtasks.

$(TW(\mathsf{t}_{k-1})/p_{k-1})\prod_{j=1}^{k-1}(1+f_j/\gamma_j)(1+p_{j-1}/\lambda_j)$ (i.e. , takes this amount of time when run on a single "processor"). Similarly, each level-k strand s is a serial computation having $TW(\mathsf{s})$ work. We thus interpret t_k as a computation having

$$\begin{aligned}
&\sum_{\text{subtasks } \mathsf{t}_{k-1}} \frac{TW(\mathsf{t}_{k-1})}{p_{k-1}} \prod_{j=1}^{k-1}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right) + \sum_{\text{level-}k \text{ strands } \mathsf{s}} TW(\mathsf{s}) \\
\leq\ & \left(\frac{TW(\mathsf{t}_k) - C_k S(\mathsf{t}_k;B_k)/B_k}{p_{k-1}}\right)\prod_{j=1}^{k-1}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right) + \sum_{\text{level-}k \text{ strands } \mathsf{s}} TW(\mathsf{s}) \\
\leq\ & \left(1+\frac{p_{k-1}}{\lambda_k}\right)\left(\frac{TW(\mathsf{t}_k) - C_k S(\mathsf{t}_k;B_k)/B_k}{p_{k-1}}\right)\prod_{j=1}^{k-1}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right)
\end{aligned}$$

work after preloading the cache, where the first step of the derivation follows from the definition of Q^* and TW, and the second follows from t_k being λ_k task heavy. Since t_k is also γ_k parallel, it has a critical-path length that is at most a $1/\gamma_k$ factor of the work. Observing that the space-bounded scheduler operates as a greedy scheduler with respect to the level-k cache, we apply Brent's theorem to conclude that the space bounded scheduler executes this computation in

$$\begin{aligned}
&\left(\frac{1}{f_k}+\frac{1}{\gamma_k}\right)\left(1+\frac{p_{k-1}}{\lambda_k}\right)\left(\frac{TW(\mathsf{t}_k) - C_k S(\mathsf{t}_k;B_k)/B_k}{p_{k-1}}\right)\prod_{j=1}^{k-1}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right) \\
=\ & \left(\frac{TW(\mathsf{t}_k) - C_k S(\mathsf{t}_k;B_k)/B_k}{p_k}\right)\prod_{j=1}^{k}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right)
\end{aligned}$$

time on the f_k processors. Adding the $C_k S(\mathsf{t}_k;B_k)/p_k B_k$ time necessary to load the level-k cache completes the proof □

The $(1+f_j/\gamma_j)$ overheads arise from load imbalance and the recursive application of Brent's theorem, whereas the $(1+p_{j-1}/\lambda_j)$ overheads stem from the fact that strands block other tasks. For sufficiently parallel algorithms with short enough strands, the product of these overheads reduces to $O(1)$. This bound is then optimal whenever $Q^*(\mathsf{t};M_i,B_i) = O(Q(\mathsf{t};M_i,B_i))$ for all i.

Our new scheduler in the next section relaxes all of these balance conditions, allowing for more asymmetric computations. Moreover, we do not assume preloading. We use the effective cache complexity analysis (Section 4.3.2) to facilitate analysis of less regular computations and prove our performance bounds with respect to the effective cache complexity metric.

5.3.2.5 A general scheduler with provably good time-bounds

This section modifies the space-bounded scheduler to address some of the balance concerns discussed in previous section. These modification restrict the space bounded scheduler, potentially forcing more processors to remain idle. These restriction, nevertheless, allow nicer provable performance guarantees.

The main performance theorem for our scheduler is the following, which is proven in Section 5.3.3. This theorem does not assume any preloading of the caches, but we do assume that all block sizes are the same (except at level 0). Here, the machine parallelism β is defined as the minimum value such that for all hierarchy levels $i > 1$, we have $f_i \leq (M_i/M_{i-1})^{\beta}$, and $f_1 \leq (M_1/3B_1)^{\beta}$. Aside from the overhead v_h (defined in the theorem), this bound is optimal in the PCC framework for a PMH with $1/3$-rd the given memory sizes. Here, k is a tunable constant scheduler parameter with $0 < k < 1$, discussed later in this section. Observe that the v_h overhead reduces significantly (even down to a constant) if the ratio of memory sizes is large but the fanout is small (as in the machines in Figure 7.1), or if $\alpha \gg \beta$.[3]

Theorem 10 *Consider an h-level PMH with $B = B_j$ for all $1 \leq j \leq h$, and let* t *be a task such that $S(\mathsf{t}; B) > f_h M_{h-1}/3$ (the desire function allocates the entire hierarchy to such a task) with effective parallelism $\alpha \geq \beta$, and let $\alpha' = \min\{\alpha, 1\}$. The runtime of* t *is no more than:*

$$\frac{\sum_{j=0}^{h-1} \widehat{Q}_{\alpha}(\mathsf{t}; M_j/3, B_j) \cdot C_j}{p_h} \cdot v_h, \quad \textit{where overhead } v_h \textit{ is}$$

$$v_h = 2 \prod_{j=1}^{h-1} \left(\frac{1}{k} + \frac{f_j}{(1-k)(M_j/M_{j-1})^{\alpha'}} \right).$$

Since much of the scheduler matches the greedy-space-bounded scheduler from Section 5.3.2.1, only the differences are highlighted here. An operational description of the scheduler can be found in the associated technical report [39].

There are three main differences between this scheduler and greedy-space-bounded scheduler from Section 5.3.2.1. First, we fix the dilation to $\sigma = 1/3$ instead of $\sigma = 1$. Whereas reducing σ worsens the bound in Theorem 8 (only by a constant factor for cache-oblivious algorithms), this factor of $1/3$ allows us more flexibility in scheduling.

Second, to cope with tasks that may skip levels in the memory hierarchy, we associate with each cache a notion of how busy the descending cluster is, to be described more fully later. For now, we say that a cluster is ***saturated*** if it is "too busy" to accept new tasks, and ***unsaturated*** otherwise. The modification to the scheduler here is then restricting it to anchor maximal tasks only at *unsaturated* caches.

Third, to allow multiple differently sized tasks to share a cache and still guarantee fairness, we partition each of the caches, awarding ownership of specific subclusters to each task. Specifically, whenever a task t is anchored at U, t is also ***allocated*** some subset $\mathcal{U}_t$ of U's level-$(i-1)$ subclusters, essentially granting ownership of the clusters to t. This allocation restricts the scheduler further in that now t may execute only on $\mathcal{U}_t$ instead of all of U. This allocation is exclusive in that a cluster may be allocated to only one task at a time, and no new tasks may be anchored at any cluster $V \in \mathcal{U}_t$ except descendent tasks of t. Moreover, tasks may not skip levels through V, i.e. , a new level-$(j < i-1)$ subtask of a level-$k > i$ task may not be anchored at any descendent cache of V. Tasks that skipped levels in the hierarchy before V was allocated may have already been anchored at or below V — these tasks continue running as normal, and they are the main reason for our notion of saturation.

[3]For example, $v_h < 10$ on the Xeon 7500 as $\alpha \to 1$.

A level-i strand is allocated every cache to which it is anchored, i.e. , exactly one cache at every level below i. In contrast, a level-i task t is anchored only to a level-i cache and allocated potentially many level-$(i-1)$ subclusters, depending on its size. We say that the size-$s = S(\mathsf{t}; B_i)$ task t ***desires*** $g_i(s)$ level-$(i-1)$ clusters, g_i to be specified later. When anchoring t to a level-i cache U, let q be the number of unsaturated and unallocated subclusters of U. Select the most unsaturated $\min\{q, g_i(s)\}$ of these subclusters and allocate them to t.

For each cache, there may be one anchored maximal task that is ***underallocated***, meaning that it receives fewer subclusters than it desires. The only underallocated task is the most recent task that caused the cache to transition from being unsaturated to saturated. Whenever a subcluster frees up, allocate it to the underallocated task. If assigning a subcluster causes the underallocated task to achieve its desire, it is no longer underallocated, and future free subclusters become available to other tasks.

Scheduler details. We now describe the two missing details of the scheduler, namely the notion of saturation, as well as the desire function g_i, which specifies for a particular task size the number of desired subclusters.

One difficulty is trying to schedule tasks with large desires on partially assigned clusters. We continue assigning tasks below a cluster until that cluster becomes saturated. But what if the last job has large desire? To compensate, our notion of saturation leaves a bit of slack, guaranteeing that the last task scheduled can get some minimum amount of computing power. Roughly speaking, we set aside a constant fraction of the subclusters at each level as a reserve. The cluster becomes saturated when all other subclusters have been allocated. The last task scheduled, the one that causes the cluster to become saturated, may be allocated subclusters from the reserve.

There is some tradeoff in selecting the reserve constant here. If a large constant is reserved, we may only allocate a small fraction of clusters at each level, thereby wasting a large fraction of all processing power at each level. If, on the other hand, the constant is small, then the last task scheduled may run too slowly. Our analysis will count the first against the work of the computation and the second against the depth.

Designing a good function to describe saturation and the reserved subclusters is complicated by the fact that task assignments may skip levels in the hierarchy. The notion of saturation thus cannot just count the number of saturated or allocated subclusters — instead, we consider the degree to which a subcluster is utilized. For a cluster U with subclusters $V_1, V_2, \ldots, V_{f_i}$ ($f_i > 1$), define the utilization function $\mu(U)$ as follows:

$$\mu(U) = \begin{cases} \min\left\{1, \frac{1}{kf_i} \sum_{i=1}^{f_i} \mu'(V_i)\right\} & \text{if } U \text{ is a level-}(\geq 2) \text{ cluster} \\ \min\{1, \frac{x}{f_1 k}\} & \text{if } U \text{ is a level-1 cluster with } x \text{ allocated processors} \end{cases}$$

and

$$\mu'(V) = \begin{cases} 1 & \text{if } V \text{ is allocated} \\ \mu(V) & \text{otherwise} \end{cases},$$

where $k \in (0, 1)$, the value $(1-k)$ specifying the fraction of processors to reserve. For a cluster U with just one subcluster V, $\mu(U) = \mu(V)$. To understand the remainder of this section, it is sufficient to think of k as $1/2$. We say that U is saturated when $\mu(U) = 1$ and unsaturated otherwise.

It remains to define the desire function g_i for level i in the hierarchy. A natural choice for g_i is $g_i(S) = \lceil S/(M_i/f_i)\rceil = \lceil Sf_i/M_i\rceil$. That is, associate with each subcluster a $1/f_i$ fraction of the space in the level-i cache — if a task uses x times this fraction of total space, it should receive x subclusters. It turns out that this desire does not yield good scheduler performance with respect to our notion of balanced cache complexity. In particular it does not give enough parallel slackness to properly load-balance subtasks across subclusters.

Instead, we use $g_i(S) = \min\{f_i, \max\{1, \lfloor f(3S/M_i)^{\alpha'}\rfloor\}\}$, where $\alpha' = \min\{\alpha, 1\}$. What this says is that a maximal level-i task is allocated one subcluster when it has size $S(t; B_i) = M_i/(3f_i^{1/\alpha'})$, and the number of subclusters allocated to t increases by a factor of 2 whenever the size of t increases by a factor of $2^{1/\alpha'}$. It reaches the maximum number of subclusters when it has size $S(t; B_i) = M_{i-1}/3$. We define $g(S) = g_i(S)p_{i-1}$ if $S \in (M_{i-1}/3, M_i/3]$.

For simplicity we assumed in our model that all memory is preallocated, which includes stack space. This assumption would be problematic for algorithms with $\alpha > 1$ or for algorithms which are highly dynamic. However, it is easy to remove this restriction by allowing temporary allocation inside a task, and assume this space can be shared among parallel tasks in the analysis of Q^*. To make our bounds work this would require that for every cache we add an additional number of lines equal to the sum of the sizes of the subclusters. This augmentation would account even for the very worst case where all memory is temporarily allocated.

The analysis of this scheduler is in Section 5.3.3, summarized by Theorem 10. There are a couple of challenges that arise in the analysis. First, while it is easy to separate the run time of a task on a sequential machine in to a sum of the cache miss costs for each level, it is not as easy on a parallel machine. Periods of waiting on cache misses at several levels at multiple processors can be interleaved in a complex manner. Our **separation lemma** (lemma 13) addresses this issue by bounding the run time by the sum of its cache costs at different levels ($\widehat{Q}_\alpha(\mathsf{t}; M_, B_i) \cdot C_i$).

Second, whereas a simple greedy-space-bounded scheduler applied to *balanced* tasks lends itself to an easy analysis through an inductive application of Brent's theorem, we have to tackle the problem of subtasks skipping levels in the hierarchy and partially allocated caches. At a high level, the analysis of Theorem 10 recursively decomposes a maximal level-i task into its nearest maximal descendent level-$j < i$ tasks. By inductively assuming that these tasks finish "quickly enough," we combine the subproblems with respect to the level-i cache analogous to Brent's theorem, arguing that a) when all subclusters are busy, a large amount of productive work occurs, b) and when subclusters are idle, all tasks have been allocated sufficient resources to progress at a sufficiently quick rate. Our carefully planned allocation and reservations of clusters as described earlier in this section are critical to this proof.

5.3.2.6 Analysis

This section presents the analysis of our scheduler, proving several lemmas leading up to Theorem 10. First, the following lemma implies that the capacity restriction of each cache is subsumed by the scheduling decision of only assigning tasks to unallocated, unsaturated clusters.

Lemma 11 *Any unsaturated level-i cluster U has at least $M_i/3$ capacity available and at least one subcluster that is both unsaturated and unallocated.*

Proof. The fact that an unsaturated cluster has an unsaturated, unallocated cluster follows from the definition. Any saturated or allocated subcluster V_i has $\mu'(V_i) = 1$. Thus, for unsaturated cluster U with subclusters $V_1, \ldots, V_{f_i}$, we have $1 > (1/kf_i)\sum_{j=1}^{f_i} \mu'(V_i) \geq (1/f_i)\sum_{j=1}^{f_i} \mu'(V_i)$, and it follows that some $\mu'(V_i) < 1$.

We now argue that if U is unsaturated, then it has at least $M_i/3$ capacity remaining. This fact is trivial for $f_i = 1$, as in that case at most one task is allocated. Suppose that tasks $\mathsf{t}_1, \mathsf{t}_2, \ldots, \mathsf{t}_k$ are anchored to an unsaturated cluster and have desires $x_1, x_2, \ldots, x_k$. Since U is unsaturated $\sum_{i=1}^{k} x_i \leq f_i - 1$, which implies $x_i \leq f_i - 1$ for all i. We will show that the ratio of space to desire, $S(\mathsf{t}_i; B)/x_i$, is at most $2M_i/3f_i$ for all tasks anchored to U, which implies $\sum_{i=1}^{k} S(\mathsf{t}_i; B) \leq 2M_i/3$.

Since a task with desire $x \in \{1, 2, \ldots, f_i - 1\}$ has size at most $(M_i/3)((x+1)/f_i)^{1/\alpha'}$, where $\alpha' = \min\{\alpha, 1\} \leq 1$, the ratio of its space to its desire x is at most $(M_i/3x)((x+1)/f_i)^{1/\alpha'}$. Letting $q = 1/\alpha' \geq 1$, we have the space-to-desire ratio r bounded by

$$r \leq \frac{M_i}{3} \cdot \frac{(x+1)^q}{x} \cdot \frac{1}{f_i^q} \leq \frac{2M_i}{3} \cdot \frac{(x+1)^q}{x+1} \cdot \frac{1}{f_i^q} \tag{5.7}$$

$$\leq \frac{2M_i}{3f_i} \cdot \frac{(x+1)^{q-1}}{f_i^{q-1}} \leq \frac{2M_i}{3f_i} \tag{5.8}$$

□

Latency added cost. Section 4.3.2 introduced effective cache complexity $\widehat{Q}_\alpha(\cdot)$, which is algorithmic measure. To analyze the scheduler, however, it is important to consider when cache misses occur. To factor in the effect of the cache miss costs, we define the latency added effective work, denoted by $\widehat{W}^*_\alpha(\cdot)$, of a computation with respect to the particular PMH. Latency added effective work is only for use in the analysis *of the scheduler*, and does not need to be analyzed by an algorithm designer.

The ***latency added effective work*** is similar to the effective cache complexity, but instead of counting just instructions, we add the cost of cache misses at each instruction. The cost $\rho(x)$ of an instruction x accessing location m is $\rho(x) = W(x) + C'_i$ if the scheduler causes the instruction x to fetch m from a level i cache on the given PMH. Using this per-instruction cost, we define effective work $\widehat{W}^*_\alpha(.)$ of a computation using structural induction in a manner that is deliberately similar to that of $\widehat{Q}_\alpha(.)$.

Definition 12 (Latency added cost) *For cost $\rho(x)$ of instruction x, the **latency added effective work** of a task* t*, or a strand* s *or parallel block* b *nested inside* t *is defined as:*
strand:

$$\widehat{W}^*_\alpha(\mathsf{s}) = s(\mathsf{t}; B)^\alpha \sum_{x \in \mathsf{s}} \rho(x). \tag{5.9}$$

parallel block: *For* $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$,

$$\widehat{W}^*_\alpha(\mathsf{b}) = \max\left\{ s(\mathsf{t}; B)^\alpha \max_i \left\{ \frac{\widehat{W}^*_\alpha(\mathsf{t}_i)}{s(\mathsf{t}_i; B)^\alpha} \right\}, \sum_i \widehat{W}^*_\alpha(\mathsf{t}_i) \right\}. \tag{5.10}$$

task: *For* $\mathsf{t} = c_1; c_2; \ldots; c_k$,

$$\widehat{W}^*_\alpha(\mathsf{t}) = \sum_{i=1}^{k} \widehat{W}^*_\alpha(c_i). \tag{5.11}$$

Because of the large number of parameters involved ($\{M_i,\ B,\ C_i\}_i$ etc.), it is undesirable to compute the latency added work directly for an algorithm. Instead, we will show a nice relationship between latency added work and effective work.

We first show that $\widehat{W}^*_\alpha(\cdot)$ (and $\rho(\cdot)$, on which it is based) can be decomposed into a per (cache) level costs $\widehat{W}^{(i)}_\alpha(\cdot)$ that can each be analyzed in terms of that level's parameters ($\{M_i, B, C_i\}$). We then show that these costs can be put together to provide an upper bound on $\widehat{W}^*_\alpha(\cdot)$. For $i \in [h-1]$, $\widehat{W}^{(i)}_\alpha(\mathsf{c})$ of a computation c is computed exactly like $\widehat{W}^*_\alpha(\mathsf{c})$ using a different base case: for each instruction x in c, if the memory access at x costs at least C'_i, assign a cost of $\rho_i(x) = C_i$ to that node. Else, assign a cost of $\rho_i(x) = 0$. Further, we set $\rho_0(x) = W(x)$, and define $\widehat{W}^{(0)}_\alpha(\mathsf{c})$ in terms of $\rho_o(\cdot)$. It also follows from these definitions that $\rho(x) = \sum_{i=0}^{h-1} \rho_i(x)$ for all instructions x.

Lemma 13 ***Separation Lemma***: *For an h-level PMH with $B = B_j$ for all $1 \le j \le h$ and computation A, we have*

$$\widehat{W}^*_\alpha(A) \le \sum_{i=0}^{h-1} \widehat{W}^{(i)}_\alpha(A).$$

Proof. The proof is based on induction on the structure of the computation (in terms of its decomposition in to block, tasks and strands). For the base case of the induction, consider the sequential thread (or strand) s at the lowest level in the call tree. If $S(\mathsf{s})$ denotes the space of task immediately enclosing s, then by definition

$$\widehat{W}^*_\alpha(\mathsf{s}) = \left(\sum_{x\in\mathsf{s}} \rho(x)\right) \cdot s(\mathsf{s}; B)^\alpha \le \left(\sum_{x\in\mathsf{s}} \sum_{i=0}^{h-1} \rho_i(x)\right) \cdot s(\mathsf{s}; B)^\alpha \tag{5.12}$$

$$= \sum_{i=0}^{h-1} \left(\sum_{x\in\mathsf{s}} \rho_i(x) \cdot s(\mathsf{s}; B)^\alpha\right) = \sum_{i=0}^{h-1} \widehat{W}^{(i)}_\alpha(\mathsf{s}). \tag{5.13}$$

For a series composition of strands and blocks with in a task $\mathsf{t} = x_1; x_2; \ldots; x_k$,

$$\widehat{W}^*_\alpha(\mathsf{t}) \ = \ \sum_{i=1}^{k} \widehat{W}^*_\alpha(x_i) \le \sum_{i=1}^{k} \sum_{l=0}^{h} \widehat{W}^{(h)}_\alpha(x_i) \ = \ \sum_{l=0}^{h} \widehat{W}^{(l)}_\alpha(x)$$

For a parallel block b inside task t consisting of tasks $\{\mathsf{t}_i\}_{i=1}^m$, consider the equation 5.10 for $\widehat{W}^*_\alpha(\mathsf{b})$ which is the maximum of $m+1$ terms, the $(m+1)$-th term being a summation. Suppose that of these terms, the term that determines $\widehat{W}^*_\alpha(\mathsf{b})$ is the k-th term (denote this by T_k). Similarly, consider the equation 5.10 for evaluating each of $\widehat{W}^{(l)}_\alpha(\mathsf{b})$ and suppose that the k_l-th term (denoted by $T^{(l)}_{k_l}$) on the right hand side determines the value of $\widehat{W}^{(l)}_\alpha(\mathsf{b})$. Then,

$$\frac{\widehat{W}^*_\alpha(\mathsf{b})}{s(\mathsf{t}; B)^\alpha} = T_k \le \sum_{l=0}^{h-1} T^{(l)}_k \le \sum_{l=0}^{h-1} T^{(l)}_{k_l} = \frac{\sum_{l=0}^{h-1} \widehat{W}^{(l)}_\alpha(\mathsf{b})}{s(\mathsf{t}; B)^\alpha}, \tag{5.14}$$

which completes the proof. Note that we did not use the fact that some of the components were work or cache complexities. The proof only depended on the fact that $\rho(x) = \sum_{i=0}^{h-1} \rho_i(x)$ and the structure of the composition rules given by equations 5.11, 5.10. ρ could have been replaced with any other kind of work and ρ_i with its decomposition. $\square$

The previous lemma indicates that the latency added work can be separated into costs per cache level. The following lemma then relates these separated costs to effective cache complexity $\widehat{Q}_\alpha(\cdot)$.

Lemma 14 *Consider an h-level PMH with $B = B_j$ for all $1 \leq j \leq h$ and a computation* $\mathtt{c}$. *If* $\mathtt{c}$ *is scheduled on this PMH using a space-bounded scheduler with dilation $\sigma = 1/3$, then* $\widehat{W}^*_\alpha(\mathtt{c}) \leq \sum_{i=0}^{h-1} \widehat{Q}_\alpha(\mathtt{c}; M_i/3, B) \cdot C_i$.

Proof. (*Sketch*) The function $\widehat{W}^{(i)}_\alpha(\cdot)$ is monotonic in that if it is computed based on function $\rho'_i(\cdot)$ instead of $\rho_i(x)$, where $\rho'_i(x) \leq \rho_i(x)$ for all instructions x, then the former estimate would be no more than the latter. It then follows from the definitions of $\widehat{W}^{(i)}_\alpha(\cdot)$ and $\rho_i(\cdot)$, that $\widehat{W}^{(i)}_\alpha(\mathtt{c}) \leq \widehat{Q}_\alpha(\mathtt{c}; M_i/3, B) \cdot C_i$ for all computations $\mathtt{c}$, $i \in \{0, 1, \ldots, h-1\}$. Lemma 13 then implies that for any computation $\mathtt{c}$: $\widehat{W}^*_\alpha(\mathtt{c}) \leq \sum_{i=0}^{h-1} \widehat{Q}_\alpha(\mathtt{c}; M_i/3, B) \cdot C_i$. $\square$

Finally, we prove the main lemma, bounding the running time of a task with respect to the remaining utilization the clusters it has been allocated. At a high level, the analysis recursively decomposes a maximal level-i task into its nearest maximal descendent level-$j < i$ tasks. We assume inductively that these tasks finish "quickly enough." Finally, we combine the subproblems with respect to the level-i cache analogous to Brent's theorem, arguing that a) when all subclusters are busy, a large amount of productive work occurs, b) and when subclusters are idle, all tasks make sufficient progress. Whereas this analysis outline is consistent with a simple analysis of the greedy scheduler and that in [64], here we address complications that arise due to partially allocated caches and subtasks skipping levels in the hierarchy.

Lemma 15 *Consider an h-level PMH with $B = B_j$ for all $1 \leq j \leq h$ and a computation to schedule with $\alpha \geq \beta$, and let $\alpha' = \min\{\alpha, 1\}$. Let N_i be a task or strand which has been assigned a set $\mathcal{U}_t$ of $q \leq g_i(S(N_i; B))$ level-$(i-1)$ subclusters by the scheduler. Letting $\sum_{V \in \mathcal{U}_t}(1 - \mu(V)) = r$ (by definition, $r \leq |\mathcal{U}_t| = q$), the running time of N_i is at most:*

$$\frac{\widehat{W}^*_\alpha(N_i)}{r p_{i-1}} \cdot v_i, \quad \textit{where overhead } v_i \textit{ is} \tag{5.15}$$

$$v_i = 2 \prod_{j=1}^{i-1} \left(\frac{1}{k} + \frac{f_i}{(1-k)(M_i/M_{i-1})^{\alpha'}} \right). \tag{5.16}$$

Proof. We prove the claim on run time using induction on the levels.
Induction: Assume that all child maximal tasks of N_i have run times as specified above. Now look at the set of clusters $\mathcal{U}_t$ assigned to N_i. At any point in time, either:

1. all of them are saturated.

2. at least one of the subcluster is unsaturated and there are no jobs waiting in the queue $R(N_i)$. More specifically, the job on the critical path ($\chi(N_i)$) is running. Here, critical path $\chi(N_i)$ is the set of strictly ordered immediate child subtasks that have the largest sum of effective depths. We would argue in this case that progress is being made along the critical path at a reasonable rate.

Assuming $q > 1$, we will now bound the run time required to complete N_i by bounding the number of cycles the above two phases use. Consider the first phase. A job $x \in C(N_i)$ (subtasks of N_i) when given an appropriate number of processors (as specified by the function g) can not have an overhead of more than v_{i-1}, i.e., it uses at most $\widehat{W}^*_\alpha(x)v_{i-1}$ individual processor clock cycles. Since in the first phase, at least k fraction of available subclusters under $\mathcal{U}_t$ are always allocated (at least rp_{i-1} clock cycles put together) to some subtask of N_i, it can not last for more than

$$\sum_{x \in C(N_i)} \frac{1}{k} \frac{\widehat{W}^*_\alpha(x)}{rp_{i-1}} \cdot v_{i-1} < \frac{1}{k} \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot v_{i-1} \text{ number of cycles.}$$

For the second phase, we argue that the critical path runs fast enough because we do not underallocate processing resources for any subtask by more than a factor of $(1-k)$ as against that indicated by the g function. Specifically, consider a job x along the critical path $\chi(N_i)$. Suppose x is a maximal level-$j(x)$ task, $j(x) < i$. If the job is allocated subclusters below a level-$j(x)$ subcluster V, then V was unsaturated at the time of allocation. Therefore, when the scheduler picked the $g_{j(x)}(S(x;B))$ most unsaturated subclusters under V (call this set $\mathcal{V}$), $\sum_{v \in \mathcal{V}} \mu(v) \geq (1-k) g_{j(x)}(S(x;B))$. When we run x on V using the subclusters $\mathcal{V}$, its run time is at most

$$\frac{\widehat{W}^*_\alpha(x)}{(\sum_{v \in \mathcal{V}} \mu(v)) p_{j(x)-1}} \cdot v_{j(x)-1} < \frac{\widehat{W}^*_\alpha(x) \quad \cdot \quad v_{j(x)-1}}{(1-k) g(S(x;B_{j(x)}))} \tag{5.17}$$

$$= \frac{\widehat{W}^*_\alpha(x)}{s(x;B)^\alpha} \frac{s(x;B)^\alpha}{g(S(x;B_{j(x)}))} \frac{v_{j(x)-1}}{1-k} \tag{5.18}$$

time. Amongst all subtasks x of N_i, the ratio $\frac{s(x;B)^\alpha}{g(S(x;B_{j(x)}))}$ is maximum when when $S(x;B) = M_{i-1}/3$, where the ratio is $(M_{i-1}/3B)^\alpha / p_{i-1}$. Summing the run times of all jobs along the critical path would give us an upper bound for time spent in phase two. This would be at most

$$
\begin{aligned}
& \sum_{x \in \chi(N_i)} \frac{\widehat{W}^*_\alpha(x)}{(1-k)g(S(x;B))} \cdot v_{i-1} && (5.19)\\
= & \sum_{x \in \chi(N_i)} \frac{\widehat{W}^*_\alpha(x)}{s(x;B)^\alpha} \cdot \frac{s(x;B)^\alpha}{g(S(x;B))} \cdot \frac{v_{i-1}}{1-k} && (5.20)\\
\leq & \left(\sum_{x \in \chi(N_i)} \frac{\widehat{W}^*_\alpha(x)}{s(x;B)^\alpha} \right) \cdot \frac{(M_{i-1}/3B)^\alpha}{p_{i-1}} \cdot \frac{v_{i-1}}{1-k} && (5.21)\\
\leq & \frac{\widehat{W}^*_\alpha(N_i)}{s(N_i;B)^\alpha} \cdot \frac{(M_{i-1}/3B)^\alpha}{p_{i-1}} \cdot \frac{v_{i-1}}{1-k} \quad \text{(by defn. of } \widehat{W}^*_\alpha()) && (5.22)\\
= & \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot \frac{r(M_{i-1}/3B)^\alpha}{s(N_i;B)^\alpha} \cdot \frac{v_{i-1}}{1-k} && (5.23)\\
\leq & \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot \frac{q(M_{i-1}/3B)^\alpha}{s(N_i;B)^\alpha} \cdot \frac{v_{i-1}}{1-k} && (5.24)\\
\leq & \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot \frac{f_i}{(1-k)(M_i/M_{i-1})^{\alpha'}} \cdot v_{i-1} \quad \text{(by defn. of } g). && (5.25)
\end{aligned}
$$

Putting together the run times of both the phases, we have an upper bound of

$$
\frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} v_{i-1} \cdot \left(\frac{1}{k} + \frac{f_i}{(1-k)(M_i/M_{i-1})^{\alpha'}} \right) = \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot v_i. \tag{5.26}
$$

If $q = 1$, N_i would get allocated just one $(i-1)$-subcluster V, and of course, all the (yet unassigned) $(i-2)$ subclusters $\mathcal{V}$ below V. Then, we can view this scenario as N_i running on the $(i-1)$-level hierarchy. Memory accesses and cache latency costs are charged the same way as before with out modification so that the effective work of N_i would still be $\widehat{W}^*_\alpha(N_i)$. By inductive hypothesis, we know that the run time of N_i would be at most

$$
\frac{\widehat{W}^*_\alpha(N_i)}{(\sum_{V \in \mathcal{V}}(1-\mu(V)))p_{i-2}} \cdot v_{i-1}
$$

which is at most $\frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot v_i$ since $\sum_{V \in \mathcal{V}}(1-\mu(V)) \geq rf_{i-1}$ and $v_{i-1} < v_i$.

Base case ($i = 1$): N_1 has $q = r$ processors available, all under a shared cache. If $q = 1$, the claim is clearly true. If $q > 1$, since there is no further anchoring beneath the level-1 cache (since $M_0 = 0$), we can use Brent's theorem on the latency added effective work to bound the run time:

$\frac{\widehat{W}_\alpha^*(N_1)}{r}$ added to the critical path length, which is at most $\frac{\widehat{W}_\alpha^*(N_1)}{s(N_1;B)^\alpha}$. This sum is at most

$$\frac{\widehat{W}_\alpha^*(N_1)}{r}\left(1+\frac{q}{s(N_1;B)^\alpha}\right) \le \frac{\widehat{W}_\alpha^*(N_1)}{r}\left(1+\frac{g(S(N_1;B))}{s(N_1;B)^\alpha}\right) \tag{5.27}$$

$$\le \frac{\widehat{W}_\alpha^*(N_1)}{r}\left(1+\frac{S(N_1;B)^\alpha}{s(N_1;B)^\alpha}\cdot\frac{f_1}{(M_1/3)^\alpha}\right) \tag{5.28}$$

$$\le \frac{\widehat{W}_\alpha^*(N_1)}{r}\times 2. \tag{5.29}$$

□

Theorem 10 follows from Lemmas 14 and 15, starting on a system with no utilization.

5.3.3 Space Bounds

Under static allocation schemes, the choice of schedules does not affect the maximum space requirement. Therefore, any space bounded schedule, including those specific schedules described in 5.3.2 take up the same space. However, when variables are dynamically allocated, it is necessary to sub-select from the class of schedulers described above to get good space bounds.

As in the case of one-level cache models, a good baseline for comparison is the space requirement of a sequential execution. In the previous section, the metrics $\widehat{Q}_\alpha(\mathsf{t})$ (Def. 4) and $\widehat{W}_\alpha^*(\mathsf{t})$ (Def. 12) for a task t were defined in terms of its static space $s(\mathsf{t},B)$. In this section, we use these these metrics with their definitions revised by replacing $s(\mathsf{t},B)$ with $s_1(\mathsf{t},B)$, the sequential execution space. We then design a parallel schedule with bounds on the extra space requirement over S_1 for each task. By adding the extra space required by the parallel schedule for all the maximal level-i tasks to the level i cache, the bounds on running time and communication costs presented in the previous sections can be preserved.

Any space-bounded schedule that follows the anchoring and processor allocation rules specified in section has good bounds on time, irrespective of the order in which subtasks of a maximal level-i task are scheduled at level-$(i-1)$ subclusters. But to achieve good bounds on space, subtasks must be scheduled in an order that is close to the depth-first order. Therefore, we use an adaptation of the Parallel Depth-First (PDF) scheduler described in Section 5.2.2.

Recursive PDF-based space-bounded schedules. We organize each maximal level-i task t_i into a collection of maximal tasks of any level less than i, which we refer to as *super-nodes*, and nodes which do not belong to any maximal subtask, which we refer to as *glue-nodes*. When t_i is pinned to a level-i cluster, the nodes in t_i (glue-nodes and super-nodes) are assigned to the subclusters in the PDF order according to same allocation policy and desire function as in the earlier section. A subcluster that starts a super node completes it before seeking more work in the PDF order. Each super-node anchored to a subcluster is in turn recursively scheduled using the PDF order over its glue and super-nodes.

Fix α to be any constant for which the computation being mapped is α-efficient according to effective cache complexity calculations. We can define the α*-shortened DAG* and α*-effective depth* of a task as follows.

Definition 16 (**α-Shortened DAG and α-effective depth**) *The α-shortened DAG $G^x_{\alpha,y}$ of task t evaluated at point x with respect to size $y \le S_1(\mathsf{t}, B)$ is constructed based on the decomposition of the DAG G corresponding to t in to super-nodes of size at most y and glue-nodes. Replace the super-node corresponding to every maximal subtask t' of size (at most y) by a chain of $\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}';x,B)}{(y/B)^\alpha} \right\rceil$ nodes in $G^x_{\alpha,y}$. For every maximal sequential chain of glue-nodes (glue-strand) s, add $Q^*(\mathsf{s}; x, B; \emptyset)$ nodes in $G^x_{\alpha,y}$ (one for every cache miss starting from a cold state). Precedence constraints between the nodes in $G^x_{\alpha,y}$ are identical to those between the super-nodes and glue-nodes they represent in G. The α-effective depth $d^x_{\alpha,y}$ of task t evaluated at point x with respect to size $y < S_1(\mathsf{t}, B)$ is the depth of the α-shortened DAG $G^x_{\alpha,y}$.*

We can bound the shorted DAG of a task t by its effective depth.

Lemma 17 *It follows from the composition rules for effective cache complexity that the α-effective depth $d^x_{\alpha,y}$ of a task t is at most $\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t};x,B)}{s_1(\mathsf{t},B)^\alpha} \right\rceil$, when $y \le S_1(\mathsf{t}, B)$.*

Proof. We will fix y, and prove the claim by induction on the size of the task t. If $S_1(\mathsf{t}, B) = y$, the claim follows immediately.

For a task $\mathsf{t} = \mathsf{s}_1; \mathsf{b}_1; \mathsf{s}_2; \mathsf{b}_2; \ldots; \mathsf{s}_k$ with sequential space greater than y, it follows from the composition rules for effective depth that

$$\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}; x, B; \kappa)}{s(\mathsf{t}; B)^\alpha} \right\rceil \ge \sum_{i=1}^{k} Q^*(\mathsf{s}_i; x, B) + \sum_{i=i}^{k-1} \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i; x, B; \kappa)}{s(\mathsf{t}_i; B)^\alpha} \right\rceil, \tag{5.30}$$

where t_i is the parallel task in block b_i with the largest effective depth.

If we inductively assume that the α-effective depths of every task t_i with $S_1(\mathsf{t}, B) \ge y$ is at most $\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i;x,B)}{s_1(\mathsf{t}_i,B)^\alpha} \right\rceil$, then we can use the above inequality to bound the α-effective depth of t. In the α-shortened graph of t, each of the strands s_i would show up as $Q^*(\mathsf{s}_i; x, B)$ nodes corresponding to the first term on the right hand side of the inequality 5.30. For those t_i with sequential space greater than y, the second term on the RHS of the inequality dominates the α-shortened DAG of t_i (inductive assumption). For those t_i with sequential space y, their depth in the α-shortened DAG $\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i;x,B)}{(y/B)^\alpha} \right\rceil$ is smaller than $\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i;x,B)}{s_1(\mathsf{t}_i,B)^\alpha} \right\rceil$ as $s_1(\mathsf{t}_i, B) < y$. The claim then follows.

□

Just as in the case of the PDF scheduler for the shared cache (Section 5.2.2), we will demonstrate bounds on that the extra space added by premature nodes in the execution of a task anchored at level i. We assume that allocations in a maximal level-$(i-1)$ task incur the cost of cache miss to be fetched from level i cache – i.e. , they incur C_{i-1} cost. By adding the extra space to the cache at level i, we can retain the same asymptotic bounds on communication cost and time as presented in Theorems 8 and 10.

Lemma 18 *Let t_i be a level-i task. Suppose that $C_{i-1} = 1$ and $C_j = 0$ for all $j < i-1$, i.e. cache misses to level i cost one cycle and accesses to all lower level caches are free. Then the recursive PDF-based space-bounded schedule mapping t_i to a machine with parallelism $\beta < \alpha$*

requires no more than

$$2\left(\frac{1}{1-k}\right)^{i-1} \times \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i; M_i, B)}{s(\mathsf{t}_i, B)^\alpha} \right\rceil \times \left(f_i M_{i-1} + f_i \left(\frac{M_{i-1}}{B}\right)^\alpha \right)$$

additional space over $S_1(\mathsf{t}_i, B)$ at level i. When the sequential space of t_i is exactly M_i, the expression above simplifies to

$$O\left(\frac{\widehat{Q}_\alpha(\mathsf{t}_i; M_i, B)}{(M_i/M_{i-1})^\alpha} \times f_i \left(1 + M_{i-1}^{1-\alpha} \cdot B^\alpha\right) \right) \quad \textit{or} \quad O\left(\frac{\widehat{Q}_\alpha(\mathsf{t}_i; M_i, B)}{f_i^{\alpha/\beta - 1}} \left(1 + M_{i-1}^{1-\alpha} \cdot B^\alpha\right) \right).$$

Proof. Let $H_\alpha := G_{\alpha, M_{i-1}}^{M_{i-1}}$ be the α-shortened DAG of t_i at point M_{i-1} with respect to size M_{i-1}. Denote its effective depth by d_α. The PDF schedule on H_α allows us to adapt the arguments in the proof of Theorem 2.3 in [37] which bounds the number of premature nodes in a PDF schedule for H_α. To do this, we adopt the following convention: for a maximal task t' replaced by $l > 1$ nodes in H_α, we say that a node at depth $i < l$ of t' in H_α is completed in a space-bounded schedule if there have been at least $i \times (M_{i-1}/B)^\alpha$ cache misses from instructions in t'. t' is completed to depth l if it has been completed. We partition the execution of t' into phases that can related to execution of levels in H_α.

Partition wall-clock time into continuous phases of $\left\lceil \left(\frac{1}{1-k}\right)^{i-1} \times \frac{(M_{i-1}/B)^\alpha}{p_{i-1}} \right\rceil$ clock cycles, where $k \in (0,1)$ is the reservation parameter described in the previous section. If a super-node is executed to depth i at the beginning of a phase and yet to be completed, it will be executed to depth $i+1$ by the end of the phase because of the allocation policy. Put another way, each phase provides ample time for a $(i-1)$-level subcluster to completely execute all the instructions in any set of nodes from H_α corresponding to maximal tasks or glue nodes that fit in M_{i-1} space.

Consider a snapshot of the execution and let $\mathcal{C}$ denote the set of nodes of H_α that have been executed and let the longest sequential prefix of the sequential execution of t_i contained in $\mathcal{C}$ be $\mathcal{C}_1$. A phase is said to complete level l in the H_α if it is the earliest phase in which all nodes at depth l in $\mathcal{C}_1$ have been completed. We will bound the number of phases that complete a level $l \leq d_\alpha$ by arguing that if a *new* premature node is started in phase i, either phase i or $i+1$ completes a level.

Suppose that a node of H_α premature with respect to $\mathcal{C}_1$ was started in phase i. Let l be the lowest level in H_α completed in $\mathcal{C}_1$ at the start of phase i. Then, at the beginning of phase i, all nodes in $\mathcal{C}_1$ at level $l+1$ are either executed, under execution or ready to be executed (denote these three sets by $\mathcal{C}_{l+1,i,d}$, $\mathcal{C}_{l+1,i,e}$, and $\mathcal{C}_{l+1,i,r}$ respectively). A premature node with respect to $\mathcal{C}_1$ can not be started unless all of the nodes in $\mathcal{C}_{l+1,i,r}$ have been started as they are ready for execution and have a higher priority in the PDF order on H_α. Therefore, if a premature node is started in phase i, all nodes in $\mathcal{C}_{l+1,i,r}$ have been started by the end of phase i, which implies they will be completed by phase $i+1$. Nodes in $\mathcal{C}_{l+1,i,e}$ will be completed by phase i. This proves our claim that if a premature node is started in phase i, a new level of H_α in phase i or $i+1$.

There are at most d_α nodes in which a level of $\mathcal{C}_1$ in H_α is completed. Since a premature node can be executed only in a phase that completes a level or in the phase before it, the number of phases that start a new premature node with respect to $\mathcal{C}_1$ is at most $2d_\alpha$. We will bound the additional space that premature nodes take up in each such phase. Note that there will be phases in which premature nodes are executed but not started. We account for the space added by each such premature node in the phase that started it.

Suppose a phase contained new premature nodes. A premature super-node in the decomposition of t_i that increases the space requirement by M units at some point during its execution must have at least M cache misses or allocations. It costs at least M processor cycles since $C_{i-1} = 1$, i.e. every unit of extra space is paid for with a processor cycle. Therefore, the worst case scenario in terms of extra space added by premature nodes is the following. Every processor allocates an unit of space in a premature node every cycle until the last cycle. In the last cycle of the phase, an additional set of premature nodes of the largest possible size are started. The contribution of all but the last cycle of the phase to extra space is at most number of cycles per phase multiplied by the number of processors, i.e.

$$\left(\left(\frac{1}{1-k}\right)^{i-1} \times \frac{(M_{i-1}/B)^{\alpha}}{p_{i-1}}\right) \times p_i = \left(\frac{1}{1-k}\right)^{i-1} \times f_i(M_{i-1}/B)^{\alpha}.$$

In the last cycle of phase, the costliest way to schedule premature nodes is to schedule a M_{i-1} size super-node at each level-$(i-1)$ subcluster for a total of $f_i M_{i-1}$ space. Adding together the extra space contributed by premature nodes across all phases gives an upper bound of

$$\begin{aligned}
&2d_{\alpha} \times \left(\left(\frac{1}{1-k}\right)^{i-1} \times f_i(M_{i-1}/B)^{\alpha} + f_i M_{i-1}\right) \\
&\leq 2\left\lceil \frac{\widehat{Q}_{\alpha}(\mathsf{t}_i; M_i, B)}{s(\mathsf{t}_i, B)^{\alpha}} \right\rceil \times \left(\left(\frac{1}{1-k}\right)^{i-1} \times f_i(M_{i-1}/B)^{\alpha} + f_i M_{i-1}\right) \\
&\leq 2\left(\frac{1}{1-k}\right)^{i-1} \times \left\lceil \frac{\widehat{Q}_{\alpha}(\mathsf{t}_i; M_i, B)}{s(\mathsf{t}_i, B)^{\alpha}} \right\rceil \times \left(f_i(M_{i-1}/B)^{\alpha} + f_i M_{i-1}\right).
\end{aligned}$$

When the space of t_i is M_i, the above expression simplifies to

$$\begin{aligned}
&O\left(\frac{\widehat{Q}_{\alpha}(\mathsf{t}_i; M_i, B)}{(M_i/M_{i-1})^{\alpha}} \times f_i + \frac{\widehat{Q}_{\alpha}(\mathsf{t}_i; M_i, B)}{(M_i/M_{i-1})^{\alpha}} \times f_i M_{i-1}^{1-\alpha} \times B^{\alpha}\right) \\
&= O\left(\frac{\widehat{Q}_{\alpha}(\mathsf{t}_i; M_i, B)}{f_i^{\alpha/\beta - 1}} \left(1 + M_{i-1}^{1-\alpha} \cdot B^{\alpha}\right)\right),
\end{aligned}$$

concluding the proof.

□

The requirement that $C_{i-1} = 1$ in the above lemma can be easily relaxed by letting phases be longer by a factor of C_{i-1}. However, allowing C_j to be non-trivial for $j < i-1$ requires a slightly different argument and results in a different bound.

To interpret this lemma, consider matrix multiplication with parallelizability 1.5. A bad schedule might need superlinear space to execute this algorithm. However, the recursive PDF schedule executes a matrix multiplication with inputs size M_i with the same guarantees on communication costs and running time as Theorems 8 and 10 using only $o(M_i)$ extra space at level-i cache M_i when the machine parallelism β is less than 1.

We now relax the condition that $C_j = 0$ for all $j < i-1$ and bound the amount of extra space needed to maintain communication and running time bounds. The bounds in the next lemma are relatively weak compared to the previous lemma. However, we conjecture that the bound can be improved.

Lemma 19 *The recursive PDF-based space-bounded schedule that maps a level-i task t_i on to a machine with parallelism $\beta < \alpha$ would require no more than*

$$2\left(\frac{1}{1-k}\right)^{i-1} \times \left(\sum_{j=0}^{i-1} \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i; M_j, B)}{s(\mathsf{t}_i, B)^\alpha} \right\rceil \right) \times \left(f_i M_{i-1} + f_i \sum_{j=0}^{i-1} \left\lceil \frac{C_j}{C_{i-1}} \right\rceil \left(\frac{M_j}{B}\right)^\alpha \right)$$

additional space over $S_1(\mathsf{t}, B)$ a level-i.

Proof. There will be two main differences from the arguments in the previous proof. First, we increase the duration of each phase to

$$\left\lceil \left(\frac{1}{1-k}\right)^{i-1} \times \frac{1}{p_i} \sum_{j=0}^{i-1} C_j \left(\frac{M_j}{B}\right)^\alpha \right\rceil$$

wall clock cycles. The second difference is that we have to relate the start of new premature nodes to completion of levels not just in $H_\alpha^{i-1} := G_{\alpha, M_{i-1}}^{M_{i-1}}$ but also in each of $H_\alpha^j := G_{\alpha, M_{i-1}}^{M_j}$ for $j \leq i-1$. Further, we change the convention used in the previous proof: for a maximal task t' replaced by $l > 1$ nodes in H_α^j, we say that a node at depth $i < l$ of t' in H_α^j is completed in a space-bounded schedule if there have been at least $i \times (M_{i-1}/B)^\alpha$ cache misses from instructions in t' to a level j cache. t' is completed to depth l if it has been completed.

Let t' be a maximal task in the decomposition of t_i into maximal level-$\leq (i-1)$ subtasks and glue nodes. If the super-node corresponding to t' is under execution at the beginning of phase i, then with the new phase duration and conventions, the super-node completes and another level of t' in at least one of $\{H_\alpha^j\}_{j \leq i-1}$ is completed by the end of phase i. A similar statement can be made of the all the super-nodes under execution at the beginning of phase i — the maximal tasks that these super-nodes correspond to will progress by at least one level in one of $\{H_\alpha^j\}_{j \leq i-1}$.

In parallel with the previous proof, it can be argued that if phase i starts a new premature node with respect to an execution snapshot $\mathcal{C}$ and its sequential prefixes $\mathcal{C}_1^j$ in each of H_α^j, then phase i or $i+1$ completes another level in one of $\mathcal{C}_1^j$. Therefore, there are at most $2\sum_{j=0}^{i-1} d_{\alpha, M_{i-1}}^{M_j}$ phases in which new premature nodes start execution. A phase which starts a new premature node adds at most

$$\left(\frac{1}{1-k}\right)^{i-1} \times \left(f_i M_{i-1} + f_i \sum_{j=0}^{i-1} \left\lceil \frac{C_j}{C_{i-1}} \right\rceil \left(\frac{M_j}{B}\right)^\alpha \right)$$

extra space over $S_1(\mathsf{t}, B)$ at the level-i cache. The C_{i-1} in the denominator of the expression comes because every allocation or cache miss to level i cache costs C_{i-1} wall clock cycles. The lemma follows by multiplying the number of phases that allow premature nodes by the maximum amount of extra space that can be allocated in those phases.

□

Chapter 6

Algorithms with Good Locality and Parallelism

We advocated the advantage of designing portable algorithms with provable bounds in program-centric cost models in Section 1.3. Such algorithms can be used across different machines with an appropriate scheduler with good performance results. Two closely related program-centric cost models were described as appropriate choices for measuring parallelism and locality in algorithms. Good algorithms in these frameworks have the following properties.

- **Low-depth and good sequential cache complexity.** Design algorithms with low depth (polylogarithmic in input size if possible) and minimal sequential cache complexity Q_1 in the Cache-Oblivious framework according to a depth-first schedule.
- **Low effective cache complexity in the PCC framework.** Design algorithms with minimal effective cache complexity $\widehat{Q}_\alpha$ for as large a value of α as possible.

This Chapter presents several polylogarithmic depth algorithms with good locality for many common problems. The algorithms are oblivious to cache sizes, number of processors, and other machine parameters which makes them portable.

Most algorithms in this chapter (except a few graph algorithms) are optimal in both the cost models. This is not surprising because the cost models are closely related for divide and conquer algorithms. Loosely speaking, a divide and conquer algorithm designed to be optimal in the first cost model will be optimal in the second cost model if each divide step is "balanced". One trivial way to ensure balance is to divide the problem size evenly at each recursive divide. Of course, this is not always feasible and algorithms with some irregularity in the division can also be optimal in the second model as described in Section 4.3.2. For example, consider the analysis of Samplesort in Section 6.3. Despite an unequal divide, the algorithm has optimal asymptotic complexity $\widehat{Q}_\alpha$ for values of α up to 1 as a result of ample parallelism. In general, the greater the parallelism in the algorithm, the greater the imbalance it can handle without an increase in effective cache complexity. In other words, they are easier to load balance.

This chapter starts with parallel algorithms for building blocks including Scan, Pack, Merge, and Sort. Section 6.4 presents algorithms for Sparse-Matrix Vector Multipliation. Section 6.5 uses Scan and Sort to design algorithms for List-Ranking, Euler Tours and other graph algorithms. Section 6.6 presents algorithms for set cover. Tables 6.1 and 6.2 summarize these results.

Problem	Depth	Cache Complexity Q_1 and Q^*	Parallelizability
Prefix Sums (Sec.6.1)	$O(\log n)$	$O(\lceil \frac{n}{B} \rceil)$	1
Merge (Sec.6.2)	$O(\log n)$	$O(\lceil \frac{n}{B} \rceil)$	1
Sort (deterministic)* (Sec.6.3.1)	$O(\log^2 n)$	$O(\lceil \frac{n}{B} \rceil \lceil \log_{M+2} n \rceil)$	1
Sort (randomized; bounds w.h.p.)* (Sec.6.3.2)	$O(\log^{1.5} n)$	$O(\lceil \frac{n}{B} \rceil \lceil \log_{M+2} n \rceil)$	1
Sparse-Matrix Vector Multiply (m non-zeros, n^ϵ separators)* (Sec.6.4)	$O(\log^2 n)$	$O(\lceil \frac{m}{B} + \frac{n}{M^{1-\epsilon}} \rceil)$	<1
Matrix Transpose ($n \times m$ matrix) [86]	$O(\log (n+m))$	$O(\lceil \frac{nm}{B} \rceil)$	1

Table 6.1: Complexities of low-depth cache-oblivious algorithms. New algorithms are marked (*). All algorithms are work optimal and their cache complexities match the best sequential algorithms. The fourth columnd presents the parallelism of the algorithm (see Section 4.3.3), i.e. , the maximum value of α for which $\widehat{Q}_\alpha = O(Q^*)$. The bounds assume $M = \Omega(B^2)$. The parallelizability of Sparse-matrix vector multipliaction depends on the precise balance of the separators.

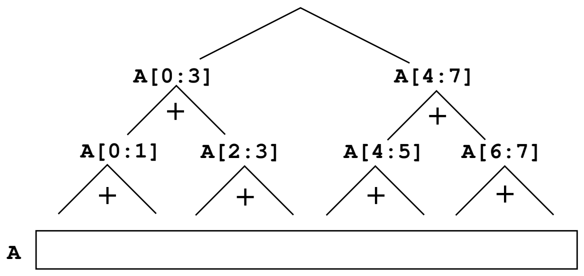


Figure 6.1: Prefix Sums. The first phase involves recursively summing up the elements and storing the intermediate values.

6.1 Prefix Sums

Consider a balanced binary tree laid over the input (see Figure 6.1). The algorithm works in two phases that traverse the tree: the first calculates for each node the sum of the left subtree and the second pushes values down the tree based on these partial sums. The recursive codes for the two phases are shown in Algorithms 1 and 2, respectively. The code uses an input array A of length n, a temporary array T of length $n-1$, and an output array R. The only adaptation to the standard prefix sums algorithm is that we lay out the tree in infix order in the array T, so as to make the algorithm cache-efficient. The down phase basically takes the input s, passes it to the left call and passes $s \oplus root(T)$ to the right call. To compute prefix-sum of A, we call the function UP$(A, T, 0, n)$ followed by DOWN$(R, T, 0, n, 0)$.

Algorithm 1 UP(A, T, i, n)

```
  **if** $n = 1$ **then**
    **return** $A[i]$
  **else**
    $k \leftarrow i + n/2$
    $T[k] \leftarrow$ UP$(A, T, i, n/2)$
    right $\leftarrow$ UP$(A, T, k, n/2)$
    **return** $T[k] \oplus$ right
  **end if**
```

Algorithm 2 DOWN(R, T, i, n, s)

```
  **if** $n = 1$ **then**
    $R[i] \leftarrow s$
  **else**
    $k \leftarrow i + n/2$
    DOWN$(R, T, i, n/2, s)$
    DOWN$(R, T, k, n/2, s \oplus T[k])$;
  **end if**
```

Lemma 20 *The prefix sum of an array of length n can be computed in $W(n) = O(n)$, $D(n) = O(\log n)$ and $Q^*(n; M, B) = Q_1(n; M, B) = O(\lceil n/B \rceil)$. Further its $\widehat{Q}_\alpha(s; M, B) = O(\lceil s/B \rceil)$ for $\alpha < 1, M > 0$.*

Proof. There exists a positive constant c such that for a call to the function UP with $n \le cM$, all the memory locations accessed by the function fit into the cache. Thus, $Q^*(n; M, B) \le M/B$ for $n \le cM$. Also $Q^*(n; M, B) = O(1) + 2 \cdot Q_1(n/2; M, B)$ for $n > cM$. Therefore, the sequential cache complexity for UP is $O(n/M + (n/(cM)) \cdot (M/B)) = O(\lceil n/B \rceil)$. A similar analysis for the cache complexity of DOWN shows that $Q_1(n; M, B) = O(\lceil n/B \rceil)$. Both the algorithms have depth $O(\log n)$ because the recursive calls can be made in parallel.

Each of these phases involves a simple balanced recursion where each task of size s adds two integers apart from two recrsive tasks on an array half the size. Therefore, the recursion for $\widehat{Q}_\alpha$ of each phase is the same as for the Map operation analyzed in Section 4.3.4. The bounds for effective cache complexity follows. □

We note that the temporary array T can also be stored in preorder or postorder with the same bounds, but that the index calculations are then a bit more complicated. What matters is that any subtree is contiguous in the array. The standard heap order (level order) does not give the same result.

Pack. The Scan operation can be used to implmenent the Pack operation. The Pack operation takes as input an array A of length n and a binary predicate that can be applied to each element in the array. It outputs an array with only those elements in array A that satisfy the predicate. To do this, the Pack operation (i) applies the predicate to each element in A independently in parallel marking a *select* bit on the element to 1 if selected for inclusion and 0 otherwise, (ii) computes a prefix sum on the *select* bit, and (iii) transfers the selected elements to their position in the output array according to the prefix sum. The works, depth and cache complexity of the Pack operation are asymptotically the same as the of the Prefix Sum operation.

6.2 Merge

To make the usual merging algorithm cache oblivious, we use divide-and-conquer with a branching factor of $n^{1/3}$. (We need a $o(\sqrt{n})$ branching factor in order to achieve optimal cache complexity.) To merge two arrays A and B of sizes l_A and l_B ($l_A + l_B = n$), conduct a dual binary search of the arrays to find the elements ranked $\{n^{2/3}, 2n^{2/3}, 3n^{2/3}, \dots\}$ among the set of keys from both arrays, and recurse on each pair of subarrays. This takes $n^{1/3} \cdot \log n$ work, $\log n$ depth and at most $O(n^{1/3} \log (n/B))$ cache misses. Once the locations of pivots have been identified, the subarrays, which are of size $n^{2/3}$ each, can be recursively merged and appended.

Lemma 21 *Two arrays of combined length n can be merged in $W(n) = O(n)$, $D(n) = O(\log n)$ and $Q^*(n; M, B) = Q_1(n; M, B) = O(\lceil n/B \rceil)$. Further its $\widehat{Q}_\alpha(s; M, B) = O(\lceil s/B \rceil)$ for $\alpha < 1$ when $M > \Omega(B^2)$.*

Proof. The cache complexity of Algorithm MERGE can be expressed using the recurrence

$$Q^*(n; M, B) \le n^{1/3}(\log (n/B) + Q^*(n^{2/3}; M, B)), \tag{6.1}$$

Algorithm 3 MERGE $((A, s_A, l_A), (B, s_B, l_B), (C, s_C))$

```
if $l_B = 0$ then
    Copy $A[s_A : s_A + l_A)$ to $C[s_C : s_C + l_A)$
else if $l_A = 0$ then
    Copy $B[s_B : s_B + l_B)$ to $C[s_C : s_C + l_B)$
else
    $\forall k \in [1 : \lfloor n^{1/3} \rfloor]$, find pivots $(a_k, b_k)$ such that $a_k + b_k = k\lceil n^{2/3} \rceil$
                  and $A[s_A + a_k] \leq B[s_B + b_k + 1]$ and $B[s_B + b_k] \leq A[s_A + a_k + 1]$.
    $\forall k \in [1 : \lfloor n^{1/3} \rfloor]$, MERGE$((A, s_A+a_k, a_{k+1}-a_k), (B, s_B+b_k, b_{k+1}-b_k), (C, s_C+a_k+b_k))$
end if
```

$\{$This function merges $A[s_A : s_A + l_A)$ and $B[s_B : s_B + l_B)$ into array $C[s_C : s_C + l_A + l_B)\}$

where the base case is $Q^*(n; M, B) = O(\lceil n/B \rceil)$ when $n \leq cM$ for some positive constant c. When $n > cM$, Equation 6.1 solves to

$$Q^*(n; M, B) = O(n/B + n^{1/3} \log (n/B)). \tag{6.2}$$

Because $M = \Omega(B^2)$ and $n > cM$, the $O(n/B)$ term in Equation 6.2 is asymptotically larger than the $n^{1/3} \log (n/B)$ term, making the second term redundant. Therefore, in all cases, $Q^*(n; M, B) = O(\lceil n/B \rceil)$. The analysis for Q_1 is similar. The bounds on $\widehat{Q}_\alpha$ follow because the recursion divides the problem into exactly equal parts.

The recurrence relation for the depth is:

$$D(n) \leq \log n + D(n^{2/3}),$$

which solves to $D(n) = O(\log n)$. It is easy to see that the work involved is linear. □

Mergesort. Using this merge algorithm in a mergesort in which the two recursive calls are parallel gives an algorithm with depth $O(\log^2 n)$ and cache complexity $\widehat{Q}_\alpha(n; M, B) = O((n/B) \log_2 (n/M))$ for $\alpha < 1$, which is not optimal. Blelloch *et al.* [38] analyze similar merge and mergesort algorithms with the same (suboptimal) cache complexities but with larger depth. Next, we present a sorting algorithm with optimal cache complexity (and low depth).

6.3 Sorting

In this section, we present the first cache-oblivious sorting algorithm that achieves optimal work, polylogarithmic depth, and good sequential cache complexity. Prior cache-oblivious algorithms with optimal cache complexity [55, 56, 57, 81, 86] have $\Omega(\sqrt{n})$ depth. In the PCC framework, the algorithm has optimal cache complexity and is parallelizable up to $\alpha < 1$.

Our sorting algorithm uses known algorithms for prefix sums and merging as subroutines. A simple variant of the standard parallel prefix-sums algorithm has logarithmic depth and cache complexity $O(n/B)$. The only adaptation is that the input has to be laid out in memory such that any subtree of the balanced binary tree over the input (representing the divide-and-conquer recursion) is contiguous. For completeness, the precise algorithm and analysis are in Section 6.1 of the appendix. Likewise, a simple variant of a standard parallel merge algorithm also has logarithmic depth and cache complexity $O(n/B)$, as described next.

6.3.1 Deterministic Sorting

Our parallel sorting algorithm is based on a version of sample sort [82, 141]. Samplesort first use a sample to select a set of pivots that partition the keys into buckets, then route all the keys to their appropriate buckets, and finally sort within the buckets. Compared to prior cache-friendly sample sort algorithms [7, 109], which incur $\Omega(\sqrt{n})$ depth, our cache-oblivious algorithm uses (and analyzes) a new parallel bucket-transpose algorithm for the key distribution phase, in order to achieve $O(\log^2 n)$ depth.

Algorithm 4 COSORT(A, n)

```
  **if** $n \leq 10$ **then**
    **return** Sort $A$ sequentially
  **end if**
  $h \leftarrow \lceil \sqrt{n} \rceil$
  $\forall i \in [1 : h]$, Let $A_i \leftarrow A[h(i-1)+1 : hi]$
  $\forall i \in [1 : h]$, $S_i \leftarrow$ COSORT($A_i$, $h$)
  **repeat**
    Pick an appropriate sorted pivot set $\mathcal{P}$ of size $h$
    $\forall i \in [1 : h]$, $M_i \leftarrow$ SPLIT$(S_i, \mathcal{P})$
    {Each array $M_i$ contains for each bucket $j$ a start location in $S_i$ for bucket $j$ and a length of
    how many entries are in that bucket, possibly 0.}
    $L \leftarrow h \times h$ matrix formed by rows $M_i$ with just the lengths
    $L^T \leftarrow$ TRANSPOSE($L$)
    $\forall i \in [1 : h]$, $O_i \leftarrow$ PREFIX-SUM($L_i^T$)
    $O^T \leftarrow$ TRANSPOSE($O$)    {$O_i$ is the $i$th row of $O$}
    $\forall i, j \in [1 : h]$, $T_{i,j} \leftarrow \langle M_{i,j}\langle 1 \rangle, O^T_{i,j}, M_{i,j}\langle 2 \rangle \rangle$
    {Each triple corresponds to an offset in row $i$ for bucket $j$, an offset in bucket $j$ for row $i$
    and the length to copy.}
  **until** No bucket is too big
  Let $B_1, B_2, \ldots, B_h$ be arrays (buckets) of sizes dictated by $T$
  B-TRANSPOSE($S$, $B$, $T$, 1, 1, $h$)
  $\forall i \in [1 : h]$, $B'_i \leftarrow$ COSORT($B_i$, length($B_i$))
  **return** $B'_1 || B'_2 || \ldots || B'_h$
```

Our sorting algorithm (COSORT in Algorithm 4) first splits the set of elements into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and recursively sorts each of the subarrays. Then, samples are chosen to determine pivots. This step can be done either deterministically or randomly. We first describe a deterministic version of the algorithm for which the **repeat** and **until** statements are not needed; Section 6.3.2 will describe a randomized version that uses these statements. For the deterministic version, we choose every $(\log n)$-th element from each of the subarrays as a sample. The sample

Algorithm 5 B-TRANSPOSE(S, B, T, i_s, i_b, n)

```
if $n = 1$ then
    Copy $S_{i_s}[T_{i_s,i_b}\langle 1\rangle : T_{i_s,i_b}\langle 1\rangle + T_{i_s,i_b}\langle 3\rangle)$
    to $B_{b_s}[T_{i_s,i_b}\langle 2\rangle : T_{i_s,i_b}\langle 2\rangle + T_{i_s,i_b}\langle 3\rangle)$
else
    B-TRANSPOSE($S$, $B$, $T$, $i_s$, $i_b$, $n/2$)
    B-TRANSPOSE($S$, $B$, $T$, $i_s$, $i_b + n/2$, $n/2$)
    B-TRANSPOSE($S$, $B$, $T$, $i_s + n/2$, $i_b$, $n/2$)
    B-TRANSPOSE($S$, $B$, $T$, $i_s + n/2$, $i_b + n/2$, $n/2$)
end if
```

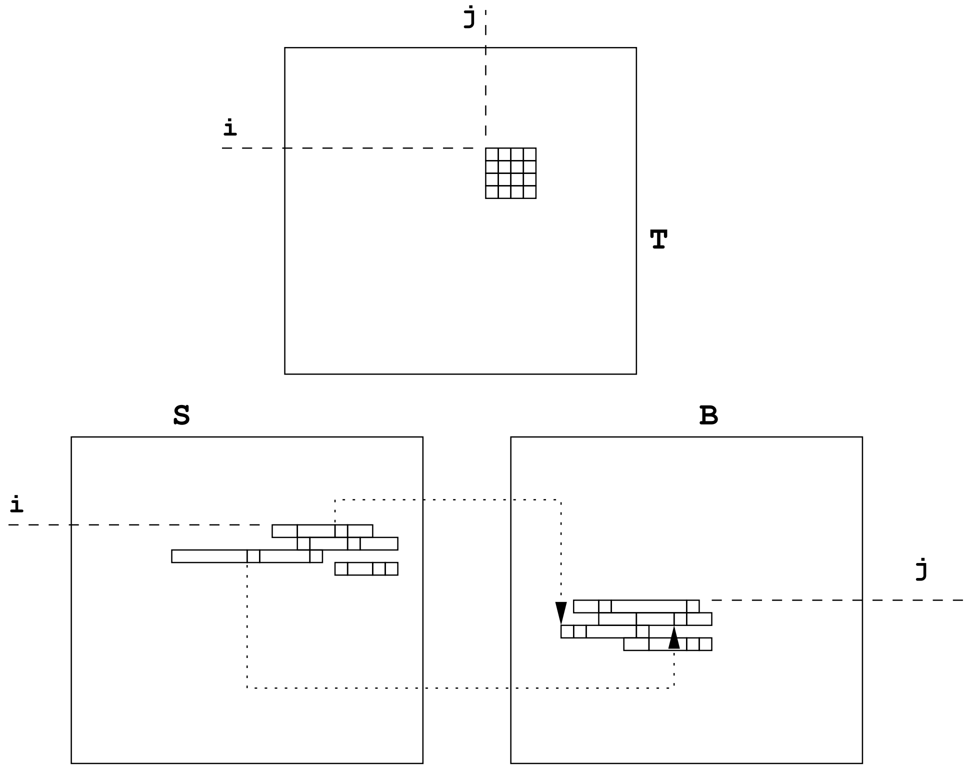


Figure 6.2: *Bucket transpose diagram:* The 4x4 entries shown for T dictate the mapping from the 16 depicted segments of S to the 16 depicted segments of B. Arrows highlight the mapping for two of the segments.

set, which is smaller than the given data set by a factor of $\log n$, is then sorted using the mergesort algorithm outlined above. Because mergesort is reasonably cache-efficient, using it on a set slightly smaller than the input set is not too costly in terms of cache complexity. More precisely, this mergesort incurs $O(\lceil n/B \rceil)$ cache misses. We can then pick $\sqrt{n}$ evenly spaced keys from the sample set $\mathcal{P}$ as pivots to determine bucket boundaries. To determine the bucket boundaries, the pivots are used to split each subarray using the cache-oblivious merge procedure. This procedure also takes no more than $O(\lceil n/B \rceil)$ cache misses.

Once the subarrays have been split, prefix sums and matrix transpose operations can be used to determine the precise location in the buckets where each segment of the subarray is to be sent. We can use the standard divide-and-conquer matrix-transpose algorithm [86], which is work optimal, has logarithmic depth and has optimal cache complexity when $M = \Omega(B^2)$ (details in Table 6.1). The mapping of segments to bucket locations is stored in a matrix T of size $\sqrt{n} \times \sqrt{n}$. Note that none of the buckets will be loaded with more than $2\sqrt{n}\log n$ keys because of the way we select pivots.

Once the bucket boundaries have been determined, the keys need to be transferred to the buckets. Although a naive algorithm to do this is not cache-efficient, we show that the bucket transpose algorithm (Algorithm B-TRANSPOSE in Algorithm 5) is. The bucket transpose is a four way divide-and-conquer procedure on the (almost) square matrix T which indicates a set of segments of subarrays (segments are contiguous in each subarray) and their target locations in the bucket. The matrix T is cut in half vertically and horizontally and separate recursive calls are assigned the responsibility of transferring the keys specified in each of the four parts. Note that ordinary matrix transpose is the special case of $T_{i,j} = \langle j, i, 1 \rangle$ for all i and j.

Lemma 22 *Algorithm B-TRANSPOSE transfers a matrix of $\sqrt{n} \times \sqrt{n}$ keys into bucket matrix B of $\sqrt{n}$ buckets according to offset matrix T in $O(n)$ work, $O(\log n)$ depth, and $Q^*(n; M, B) = O(\lceil n/B \rceil)$ cache complexity.*

Proof. It is easy to see that the work is $O(n)$ because there is only constant work for each of the $O(\log n)$ levels of recursion and each key is copied exactly once. Similarly, the depth is $O(\log n)$ because we can use prefix sums to do the copying whenever a segment is larger than $O(\log n)$.

To analyze the cache complexity, we use the following definitions. For each node v in the recursion tree of bucket transpose, we define the node's size $s(v)$ to be n^2, the size of its submatrix T, and the node's weight $w(v)$ to be the number of keys that T is responsible for transferring. We identify three classes of nodes in the recursion tree:

1. Light-1 nodes: A node v is light-1 if $s(v) < M/100$, $w(v) < M/10$, and its parent node is of size $\geq M/100$.

2. Light-2 nodes: A node v is light-2 if $s(v) < M/100$, $w(v) < M/10$, and its parent node is of weight $\geq M/10$.

3. Heavy leaves: A leaf v is heavy if $w(v) \geq M/10$.

The union of these three sets covers the responsibility for transferring all the keys, i.e., all leaves are accounted for in the subtrees of these nodes.

From the definition of a light-1 node, it can be argued that all the keys that a light-1 node is responsible for fit inside a cache, implying that the subtree rooted at a light-1 node cannot incur more than M/B cache misses. It can also be seen that light-1 nodes can not be greater than

$4n/(M/100)$ in number leading to the fact that the sum of cache complexities of all the light-1 nodes is no more than $O(\lceil n/B \rceil)$.

Light-2 nodes are similar to light-1 nodes in that their target data fits into a cache of size M. If we assume that they have a combined weight of $n - W$, then there are no more than $4(n - W)/(M/10)$ of them, putting the aggregate cache complexity for their subtrees at $40(n-W)/B$.

A heavy leaf of size w incurs $\lceil w/B \rceil$ cache misses. There are no more than $W/(M/10)$ of them, implying that their aggregate cache complexity is $W/B + 10W/M < 11W/B$. Therefore, the cache complexities of light-2 nodes and heavy leaves add up to another $O(\lceil n/B \rceil)$.

Note that the validity of this proof does not depend on the size of the individual buckets. The statement of the lemma holds even for the case where each of the buckets is as large as $O(\sqrt{n} \log n)$. □

Lemma 23 *The effective cache complexity of Algorithm B-TRANSPOSE transfers a matrix of* $\sqrt{n} \times \sqrt{n}$ *keys into bucket matrix* B *of* $\sqrt{n}$ *buckets according to offset matrix* T *if* $\widehat{Q}_\alpha(n; M, B) = O(\lceil n/B \rceil)$ *for* $\alpha < 1$.

Proof. Bucket tranpose involves work that is linear in terms of space, but the recursion is highly imbalanced. A call (task) to bucket transpose with space s invokes four parallel bucket tranposes on sizes s_1, s_2, s_3, s_4 such that $\sum_{i=1}^{4} s_i = s$. The only other work here is fork and join points. When this recursion reaches the leaf (of size s), an array copy (completely parallel) whose effective work is s (for $\alpha < 1$) is done. For an internal node of this recursion, when $s > B, M > 0$,

$$\widehat{Q}_\alpha^{BT}(s; M, B; \emptyset) = O\left(\left\lceil \frac{s}{B} \right\rceil^\alpha\right) \tag{6.3}$$

$$+ \max\left\{ \max_i \left\{ \widehat{Q}_\alpha^{BT}(s_i; M, B; \emptyset) \frac{\left\lceil \frac{s}{B} \right\rceil^\alpha}{\left\lceil \frac{s_i}{B} \right\rceil^\alpha} \right\}, \sum_{i=1}^{4} \widehat{Q}_\alpha^{BT}(s_i; M, B; \emptyset) \right\}. \tag{6.4}$$

If we inductively assume that for $\alpha < 1$, $\widehat{Q}_\alpha^{BT}(s_i; M, B; \emptyset) \le k(\lceil s_i/B \rceil - \lceil s_i/B \rceil^{(1+\alpha)/2})$, then the i-th balance term is at most $k(\lceil s/B \rceil^\alpha \lceil s_i/B \rceil^{1-\alpha} - \lceil s_i/B \rceil^{(\alpha-1)/2})$, which is dominated by the summation on the right, when $\alpha < 1$. Further the summation is also less than $k(\lceil s/B \rceil - \lceil s/B \rceil^{(1+\alpha)/2})$, when $s > k_1 B$ for some constant k_1. Filling in other cases of the recurrence will show that $\widehat{Q}_\alpha^{BT}(s; M, B) = O(\lceil s/B \rceil)$, for $\alpha < 1, M > 0$ (also $\widehat{Q}_\alpha^{BT}(s; 0, B) = O(s)$ for $\alpha < 1$). □

Like wise, matrix tranpose has $\widehat{Q}_\alpha^{MT}(s; M, B) = O(\lceil s/B \rceil)$.

Theorem 24 *On an input of size* n*, the deterministic COSORT has complexity* $Q^*(n; M, B) = O(\lceil n/B \rceil \lceil \log_{M+2} n \rceil)$*,* $O(n \log n)$ *work, and* $O(\log^2 n)$ *depth.*

Proof. All the subroutines other than recursive calls to COSORT have linear work and cache complexity $O(\lceil n/B \rceil)$. Also, the subroutine with the maximum depth is the mergesort used to find pivots; its depth is $O(\log^2 n)$. Therefore, the recurrence relations for the work, depth, and cache complexity are as follows:

$$W(n) = O(n) + \sqrt{n} W(\sqrt{n}) + \textstyle\sum_{i=1}^{\sqrt{n}} W(n_i)$$

$$D(n) = O(\log^2 n) + \max_{i=1}^{\sqrt{n}}\{D(n_i)\}$$

$$Q^*(n; M, B) = O\left(\left\lceil \tfrac{n}{B} \right\rceil\right) + \sqrt{n}Q^*(\sqrt{n}; M, B) + \textstyle\sum_{i=1}^{\sqrt{n}} Q^*(n_i; M, B),$$

where the $\{n_i\}_i$ are such that their sum is n and none individually exceed $2\sqrt{n}\log n$. The base case for the recursion for cache complexity is $Q^*(n; M, B) = O(\lceil n/B \rceil)$ for $n \leq cM$ for some constant c. Solving these recurrences proves the theorem. □

Theorem 25 *The effective cache complexity of deterministic COSORT for $\alpha < 1$ is $\widehat{Q}_\alpha(n; M, B) = O(\lceil n/B \rceil \lceil \log_{M+2} n \rceil)$.*

Proof. Each call to such a COSORT with input size s starts with a completely balanced parallel block of $\sqrt{s}$ recursive calls to COSORT with input size $\sqrt{s}$. After this, $(s/\log s)$ pivots are picked and sorted, the subarrays are split (same complexity as merging) with the pivots, offsets in to target buckets computed with prefix sum (and matrix tranpose operations) and the elements are sent in to buckets using bucket transpose. At the end, there is a parallel block sorting the $\sqrt{s}$ buckets (not all of same size) using COSORT. All the operations except the recursive COSORT have effective cache complexity that sums up $O(\lceil s/B \rceil)$ (if starting with empty cache) when $\alpha < 1$. Therefore, for $\alpha < 1$, $s > M$,

$$\widehat{Q}_\alpha^{SS}(s; M, B) = O\left(\left\lceil \frac{s}{B} \right\rceil\right) + \sqrt{s}\widehat{Q}_\alpha^{SS}(\sqrt{s}; M, B) + \max\left\{ \max_i \left\{ \widehat{Q}_\alpha^{SS}(s_i; M, B) \frac{\lceil s/B \rceil^\alpha}{\lceil s_i/B \rceil^\alpha} \right\}, \sum_{s_i \leq s\log s, \sum_i s_i = s} \widehat{Q}_\alpha^{SS}(s_i; M, B) \right\},$$

when $s \geq M/k$. The base case for this recursion is $\widehat{Q}_\alpha^{SS}(s; M, B) = O(\lceil s/B \rceil)$ for $s < M/k$ for some small constant k.

There exist constants $c_1 > c_4 > 0, c_2, c_3$ such that the cache complexity of COSORT has the bounds

$$c_4 \left\lceil \frac{s}{B} \right\rceil \lceil \log_{M+2} s \rceil \leq Q^*(s) \leq c_1 \left\lceil \frac{s}{B} \right\rceil \lceil \log_{M+2} s \rceil - c_2 \left\lceil \frac{s}{B} \right\rceil \lceil \log\log_{M+2} s \rceil - c_3 \left\lceil \frac{s}{B} \right\rceil. \quad (6.5)$$

Inductively assume that the recursion is work-dominated at every level below size s so that $\widehat{Q}_\alpha^{SS}(\cdot)$ has the same complexity as in equation 6.5. Without loss of generality, assume that $s_1 \geq s_i \; \forall i \in \{1, 2, \ldots, \lceil \sqrt{s} \rceil\}$. Then the recursion is work-dominated at this level if for some constant c',

$$\frac{\sum_i \widehat{Q}_\alpha^{SS}(s_i; M, B)}{\lceil s/B \rceil^\alpha} \geq \frac{\widehat{Q}_\alpha^{SS}(s_1; M, B)}{\lceil s_1/B \rceil^\alpha} \cdot c' \quad (6.6)$$

$$\iff \frac{\sum_i \widehat{Q}_\alpha^{SS}(s_i; M, B)}{\widehat{Q}_\alpha^{SS}(s_1; M, B)} \geq \frac{\lceil s/B \rceil^\alpha}{\lceil s_1/B \rceil^\alpha} \cdot c' \quad (6.7)$$

$$\iff 1 + \frac{\sum_{i>1} \widehat{Q}_\alpha^{SS}(s_i; M, B)}{\widehat{Q}_\alpha^{SS}(s_1; M, B)} \geq \frac{\lceil s/B \rceil^\alpha}{\lceil s_1/B \rceil^\alpha} \cdot c'. \quad (6.8)$$

Fix the value of s_1 and let $s' = (s - s_1)/\sqrt{s}$ and $0 < \alpha < 1$. The left-hand side attains minimum value when all s_i except s_1 are equal to s'. The right hand-side attains maximum value when all s_i are equal to s'. Therefore, the above inequality holds if and only if

$$1 + (\sqrt{s} - 1)\frac{\lceil \frac{s'}{B} \rceil \lceil \log_{M+2} s' \rceil}{\lceil \frac{s_1}{B} \rceil \lceil \log_{M+2} s_1 \rceil} \geq \frac{\lceil \frac{s}{B} \rceil^\alpha}{\lceil \frac{s_1}{B} \rceil^\alpha} \cdot c' \tag{6.9}$$

$$\Leftarrow 1 + (\sqrt{s} - 1)\frac{s' \lceil \log_{M+2} s' \rceil}{(s_1 + B) \lceil \log_{M+2} s_1 \rceil} \geq \left(\frac{s + B}{s_1}\right)^\alpha \cdot c' \tag{6.10}$$

$$\Leftarrow (\sqrt{s} - 1)\frac{s' \lceil \log_{M+2} s' \rceil}{s_1 \lceil \log_{M+2} s_1 \rceil} \geq \left(\frac{s}{s_1}\right)^\alpha \cdot 4c' \tag{6.11}$$

$$\Leftarrow \frac{s \lceil \log_{M+2} s' \rceil}{s_1 \lceil \log_{M+2} s_1 \rceil} \geq \left(\frac{s}{s_1}\right)^\alpha \cdot 8c' \tag{6.12}$$

$$\Leftarrow \left(\frac{s}{s_1}\right)^{1-\alpha} \geq \frac{\lceil \log_{M+2} s_1 \rceil}{\lceil \log_{M+2} s' \rceil} \cdot 8c' \tag{6.13}$$

$$\Leftarrow \left(\frac{s}{s_1}\right)^{1-\alpha} \geq \frac{\lceil \log_{M+2} s_1 \rceil}{\lceil \log_{M+2} s' \rceil} \cdot 8c' \tag{6.14}$$

$$\Leftarrow \left(\sqrt{s}/\log s\right)^{1-\alpha} \geq \frac{\lceil \log_{M+2}(\sqrt{s} \log s) \rceil}{\lceil \log_{M+2}(\sqrt{s}/2) \rceil} \cdot 8c' \tag{6.15}$$

$$\Leftarrow \left(\sqrt{s}/\log s\right)^{1-\alpha} \geq \frac{\lceil \log_{M+2}(\sqrt{s} \log s) \rceil}{\lceil \log_{M+2}(\sqrt{s}/2) \rceil} \cdot 8c', \tag{6.16}$$

which is true for $\alpha \leq 1 - \log_s c''$ for some constant c''. When all levels of recursion are work-dominated, $\widehat{Q}_\alpha^{SS}(s)$ has the same asymptotic value as cache complexity. This demonstrates that COSORT has a parallelizability of 1. □

6.3.2 Randomized Sorting

A simple randomized version of the sorting algorithm is to randomly pick $\sqrt{n}$ elements for pivots, sort them using brute force (compare every pair) and using the sorted set as the pivot set $\mathcal{P}$. This step takes $O(n)$ work, $O(\log n)$ depth and has cache complexity $O(n/B)$ and the probability that the largest of the resultant buckets are larger than $c\sqrt{n}\log n$ is not more $1 - 1/n$ for a certain constant c. When one of the buckets is too large ($> c\sqrt{n}\log n$), the process of selecting pivots and recomputing bucket boundaries is repeated. Because the probability of this happening repeatedly is low, the overall depth of the algorithm is small.

Theorem 26 *On an input of size n, the randomized version of COSORT has, with probability greater than $1 - 1/n$, $Q^*(n; M, B) = O(\lceil n/B \rceil \lceil \log_M n \rceil)$ sequential cache complexity, $O(n \log n)$ work, and $O(\log^{1.5} n)$ depth.*

Proof. In a call to randomized COSORT with input size n, the loop terminates with probability $1 - 1/n$ in each round and takes less than 2 iterations on average to terminate. Each iteration of the while loop, including the brute force sort requires $O(n)$ work and incurs at most $O(\lceil n/B \rceil)$ cache misses with high probability. Therefore,

$$\mathbb{E}[W(n)] = O(n) + \sqrt{n}\,\mathbb{E}[W(\sqrt{n})] + \sum_{i=1}^{\sqrt{n}} \mathbb{E}[W(n_i)]$$

, where each $n_i < 2\sqrt{n}\log n$ and the n_is add up to n. This implies that $\mathbb{E}[W(n)] = O(n \log n)$. Similarly for cache complexity we have

$$\mathbb{E}[Q^*(n; M, B)] = O(n/B) + \sqrt{n}\,\mathbb{E}[Q^*(\sqrt{n}; M, B)] + \sum_{i=1}^{\sqrt{n}} \mathbb{E}[Q^*(n_i; M, B)],$$

which solves to $\mathbb{E}[Q^*(n; M, B)] = O((n/B)\log_{\sqrt{M}} n) = O((n/B)\log_M n)$. To show the high probability bounds for work and cache complexity, we can use Chernoff bounds because the fan-out at each level of recursion is high.

Proving probability bounds on the depth is more involved. We prove that the sum of depths of all nodes at level d in a recursion tree is $O(\log^{1.5} n/\log\log n)$ with probability at least $1 - 1/n^{\log^2 \log n}$. We also show that the recursion tree has at most $O(\log\log n)$ levels and the number of instances of recursion tree is $n^{O(1.1 \log\log n)}$. This implies that the depth of the critical path computation is at most $O(\log^{1.5} n)$ with high probability.

To analyze the depth of the dag, we obtain high probability bounds on the depth of each level of recursion tree (we assume that the levels are numbered starting with the root at level 0). To get sufficient probability at each level we need to execute the outer loop more times toward the leaves where the size is small. Each iteration of the outer loop at node N of input size m at level k in the recursion tree has depth $\log m$ and the termination probability of the loop is $1 - 1/m$.

To prove the required bounds on depth, we prove that the sum of depths of all nodes at level d in the recursion tree is at most $O(\log^{3/2} n/\log\log n)$ with probability at least $1 - 1/n^{O(\log^2 \log n)}$ and that the recursion tree is at most $1.1\log_2\log_2 n$ levels deep. This will prove that the depth of the recursion tree is $O(\log^{3/2} n)$ with probability at least $1 - (1.1\log_2\log_2 n)/n^{O(2\log^2 \log n)}$. Since the actual depth is a maximum over all "critical" paths which we will argue are not more than $n^{O(1.1\log\log n)}$ in number (critical paths are shaped liked the recursion tree), we can conclude that the depth is $O(\log^{3/2} n)$ with high probability.

The maximum number of levels in the recursion tree can be bounded using the recurrence relation $X(n) = 1 + X(\sqrt{n}\log n)$ and $X(10) = 1$. Using induction, it is straightforward to show that this solves to $X(n) < 1.1\log_2\log_2 n$. Similarly the number of critical paths $C(n)$ can be bounded using the relation $C(n) < (\sqrt{n}X(\sqrt{n}))(\sqrt{n}X(\sqrt{n}\log n))$. Again, using induction, this relation can be used to show that $C(n) = n^{O(1.1\log\log n)}$.

To compute the sum of the depth of nodes at level d in the recursion tree, we consider two cases: (1) when $d > 10\log\log\log n$ and (2) otherwise.

Case 1: The size of a node one level deep in the recursion tree is at most $O(\sqrt{n}\log n) = O(n^{1/2+r})$ for any $r > 0$. Also, the size of a node which is d levels deep is at most $O(n^{(1/2+r)^d})$, each costing $O((1/2+r)^d \log n)$ depth per trial. Since there are 2^d nodes at level d in the recursion tree, and the failure probability of a loop in any node is no more than $1/2$, we show that the probability of having to execute more than $(2^d \cdot \log^{1/2} n)/((1+2r)^d \cdot \log\log n)$ loops is small.

Since we are estimating the sum of 2^d independent variables, we use Chernoff bounds of the form:

$$Pr[X > (1+\delta)\mu] \leq e^{-\delta^2 \mu}, \tag{6.17}$$

with $\mu = (2 \cdot 2^d)$, $\delta = (1/2)(\log^{1/2} n/((1+2r)^d \cdot \log\log n)) - 1$. The resulting probability bound is asymptotically lesser than $1/n^{O(\log^2 \log n)}$ for $d > 10 \log\log\log n$. Therefore, the contribution of nodes at level d in the recursion tree to the depth of recursion tree is at most $2^d \cdot (1/2+r)^d \log n \cdot \log^{1/2} n/((1+2r)^d \cdot \log\log n) = \log^{3/2} n/\log\log n$ with probability at least $1 - 1/n^{O(\log^2 \log n)}$.

Case 2: We classify all nodes at level d in to two kinds, the large ones with size greater than $\log^2 n$ and the smaller ones with size at most $\log^2 n$. The total number of nodes is $2^d < (\log\log n)^5$. Consider the small nodes. Each small node can contribute a depth of at most $2 \log\log n$ to the recursion tree and there are at most $(\log\log n)^1 0$ of them. Therefore, their contribution to depth of the recursion tree at level d is asymptotically lesser than $\log n$.

We use Chernoff bounds to bound the contribution of large nodes to the depth of the recursion tree. Suppose that there are j large nodes. We show that with probability not more than $1/n^{O(\log^2 \log n)}$, it takes more than $10 \cdot j$ loop iterations at depth d for j of them to succeed. For this, consider $10 \cdot j$ random independent trials with success probability at least $1 - 1/\log^2 n$ each. The expected number of failures is no more than $\mu = 10 \cdot j/\log^2 n$. We want to show that the probability that there are greater than $9 \cdot j$ failures in this experiment is tiny. Using Chernoff bounds in the form presented above with the above μ, and $\delta = (0.9 \cdot \log^2 n - 1)$, we infer that this probability is asymptotically less than $1/n^{O(\log^2 \log n)}$. Since $j < 2^d$, the depth contributed by the larger nodes is at most $2^d(1/2+r)^d \log n$, asymptotically smaller than $\log^{3/2} n/\log\log n$.

The above analysis shows that the combined depth of all nodes at level d is $O(\log^{3/2} n/\log\log n)$ with high probability. Since there are only $1.1 \log_2 \log_2 n$ levels in the recursion tree, this completes the proof.

□

6.4 Sparse-Matrix Vector Multiply

In this section, we consider the problem of multiplying a sparse $n \times n$ matrix with m non-zeros by a dense vector (SpMV-multiply). For general sparse matrices Bender *et al.* [28] show a lower bound on cache complexity, which for $m = O(n)$ matches the sorting lower bound. However, for certain matrices common in practice the cache complexity can be improved. For example, for matrices with non-zero structure corresponding to a well-shaped d-dimensional mesh, a multiply can be performed with cache complexity $O(m/B + n/M^{1/d})$ [29]. This requires pre-processing to lay out the matrix in the right form. However, for applications that run many multiplies over the same matrix, as with many iterative solvers, the cost of the pre-processing can be amortized. Note that for $M \geq B^d$ (e.g. , the tall-cache assumption in 2 dimension), the cache complexity reduces to $O(m/B)$ which is as fast as scanning memory.

The cache-efficient layout and bounds for well-shaped meshes can be extended to graphs with good edge separators [38]. The layout and algorithm, however, is not efficient for graphs such as planar graphs or graphs with constant genus that have good vertex separators but not necessarily any good edge separators. We generalize the results to graphs with good vertex

```
Algorithm BuildTree($V, E$)
if $|E| = 1$ then
    return $V$
end if
$(V_a, V_{sep}, V_b) \leftarrow$ FindSeparator($V, E$)
$E_a \leftarrow \{(u, v) \in E | u \in V_a \vee v \in V_a\}$
$E_b \leftarrow E - E_a$
$V_{a,sep} \leftarrow V_a \cup V_{sep}$
$V_{b,sep} \leftarrow V_b \cup V_{sep}$
$T_a \leftarrow$ BuildTree($V_{a,sep}, E_a$)
$T_b \leftarrow$ BuildTree($V_{b,sep}, E_b$)
return SeparatorTree($T_a, V_{sep}, T_b$)

Algorithm SparseMxV($x$,$T$)
if isLeaf($T$) then
    $T.u$.value $\leftarrow x[T.v$.index$] \otimes T.v$.weight
    $T.v$.value $\leftarrow x[T.u$.index$] \otimes T.u$.weight
    {Two statements for the two edge directions}
else
    SparseMxV($T$.left) and SparseMxV($T$.right)
    for all $v \in T$.vertices do
        $v$.value $\leftarrow$ ($v$.left$\rightarrow$value $\oplus$ $v$.right$\rightarrow$value)
    end for
end if
```

Figure 6.3: Cache-Oblivious Algorithms for Building a Separator Tree and for Sparse-Matrix Vector Multiply

separators and present the first cache-oblivious, low cache complexity algorithm for the sparse-matrix multiplication problem on such graphs. We do not analyze the cost of finding the layout, which involves the recursive application of finding vertex separators, as it can be amortized across many solver iterations. Our algorithm for matrices with n^ϵ separators is linear work, $O(\log^2 n)$ depth, and $O(m/B + n/M^{1-\epsilon})$ parallel cache complexity.

Let S be a class of graphs that is closed under the subgraph relation. We say that S satisfies a $f(n)$-*vertex separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that every graph $G = (V, E)$ in S with n vertices can be partitioned into three sets of vertices V_a, V_s, V_b such that $|V_s| \leq \beta f(n)$, $|V_a|, |V_b| \leq \alpha n$, and $\{(u, v) \in E | (u \in V_a \wedge v \in V_b) \vee (u \in V_b \wedge v \in V_a)\} = \emptyset$. In our presentation we assume the matrix has symmetric non-zero structure (but not necessarily symmetric values (weights)); if it is asymmetric we can always add zero weight reverse edges while at most doubling the number of edges.

We now describe how to build a separator tree assuming we have a good algorithm FindSeparator for finding separators. For planar graphs this can be done in linear time [125]. The algorithm for building the tree is defined by Algorithm BuildTree in Figure 6.4. At each recursive call it partitions the edges into two subsets that are passed to the left and right children. All the vertices in the separator are passed to both children. Each leaf corresponds to a single edge. We assume that FindSeparator only puts a vertex in the separator if has an edge to each side and always returns a separator with at least one vertex on each side unless the graph is a clique. If the graph is a clique, we assume the separator contains all but one of the vertices, and that the remaining vertex is on the left side of the partition.

Every vertex in our original graph of degree Δ corresponds to a binary tree embedded in the separator tree with Δ leaves, one for each of its incident edges. To see this consider a single vertex. Every time it appears in a separator, its edges are partitioned into two sets, and the vertex is copied to both recursive calls. Because the vertex will appear in Δ leaves, it must appear in $\Delta - 1$ separators, so it will appear in $\Delta - 1$ internal nodes of the separator tree. We refer to the tree for a vertex as the *vertex tree*, each appearance of a vertex in the tree as a *vertex copy*, and the root of each tree as the *vertex root*. The tree is used to sum the values for the SpMV-multiply.

We reorder the rows/columns of the matrix based on a preorder traversal of their root locations in the separator tree (i.e. , all vertices in the top separator will appear first). This is the order we will use for the input vector x and output vector y when calculating $y = Ax$. We keep a vector R in this order that points to each of the corresponding roots of the tree. The separator tree is maintained as a tree T in which each node keeps its copies of the vertices in its separator. Each of these vertex copies will point to its two children in the vertex tree. Each leaf of T is an edge and includes the indices of its two endpoints and its weight. In all internal vertex copies we keep an extra value field to store a temporary variable, and in the leaves we keep two value fields, one for each direction. Finally we note that all data for each node of the separator tree is stored adjacently (i.e. , all its vertex copies are stored one after the other), and the nodes are stored in preorder. This is important for cache efficiency.

Our algorithm for SpMV-multiply is described in Algorithm SparseMxV in Figure 6.4. This algorithm will take the input vector x and leave the results of the matrix multiplication in the root of every vertex. To gather the results up into a result vector y we simply use the root pointers R to fetch each root. The algorithm does not do any work on the way down the recursion, but when it gets to a leaf the edge multiplies its two endpoints by its weight putting the result in its temporary value. Then on the way back up the recursion the algorithms sums these values. In particular whenever it gets to an internal node of a vertex tree it adds the two children. Since the algorithm works bottom up the values of the children are always ready when the parent reads them.

Theorem 27 *Let $\mathcal{M}$ be a class of matrices for which the adjacency graphs satisfy an n^ϵ-vertex separator theorem. Algorithm SparseMxV on an $n \times n$ matrix $A \in \mathcal{M}$ with $m \geq n$ non-zeros has $O(m)$ work, $O(\log^2 n)$ depth and $O(\lceil m/B + n/M^{1-\epsilon} \rceil)$ parallel cache complexity.*

Proof. For a constant k we say a vertex copy is heavy if appears in a separator node with size (memory usage) larger than M/k. We say a vertex is heavy if it has any heavy vertex copies. We first show that the number of heavy vertex copies for any constant k is bounded by $O(n/M^{1-\epsilon})$ and then bound the number of cache misses based on the number of heavy copies.

For a node of n vertices, the size $X(n)$ of the tree rooted at the node is given by the recurrence relation:

$$X(n) = \max_{1/2 \geq \alpha' \geq \alpha} \{X(\alpha' n) + X((1-\alpha')n) + \beta n^\epsilon\}$$

This recurrence solves to $X(n) = k(n - n^\epsilon)$, $k = \beta/(\alpha^\epsilon + (1-\alpha)^\epsilon - 1)$. Therefore, there exists a positive constant c such that for $n \leq cM$, the subtree rooted at a node of n vertices fits into the cache. We use this to count the number of heavy vertex copies $H(n)$. The recurrence relation for $H(n)$ is:

$$H(n) = \begin{cases} \max_{\alpha \leq \alpha' \leq \frac{1}{2}} \{H(\alpha' n) + H((1-\alpha')n) + \beta n^{\epsilon}\} & \text{if } n > cM \\ 0 & \text{otherwise} \end{cases}$$

This recurrence relation solves to $H(n) = k(n/(cM)^{1-\epsilon} - \beta n^{\epsilon}) = O(n/M^{1-\epsilon})$.

Now we note that if a vertex is not heavy (i.e. , light) it is used only by a single subtree that fits in cache. Furthermore because of the ordering of the vertices based on where the roots appear, all light vertices that appear in the same subtree are adjacent. Therefore the total cost of cache misses for light vertices is $O(n/B)$. We note that the edges are traversed in order so they only incur $O(m/B)$ misses. Now each of the heavy vertex copies can be responsible for at most $O(1)$ cache misses. In particular reading each child can cause a miss. Furthermore, reading the value from a heavy vertex (at the leaf of the recursion) could cause a miss since it is not stored in the subtree that fits into cache. But the number of subtrees that just fit into cache (i.e. , their parents do not fit) and read a vertex u is bounded by one more than the number of heavy copies of u. Therefore we can count each of those misses against a heavy copy. We therefore have a total of $O(m/B + n/M^{1-\epsilon})$ misses.

The work is simply proportional to the number of vertex copies, which is less than twice m and hence is bounded by $O(m)$. For the depth we note that the two recursive calls can be made in parallel and furthermore the **for all** statement can be made in parallel. Furthermore the tree is depth $O(\log n)$ because of the balance condition on separators (both $|V_{a,sep}|$ and $|V_{b,sep}|$ are at most $\alpha n + \beta n^{\epsilon}$). Since the branching of the **for all** takes $O(\log n)$ depth, the total depth is bounded by $O(\log^2 n)$. □

6.5 Applications of Sorting: Graph Algorithms

PRAM algorithms for many problems such as List Ranking, Euler Tours, Tree Contraction, Least Common Ancestors, Connected Components and Minimum Spanning Forest can be decomposed into primitive operations such as scans and sorts. The algorithms for scanning and sorting presented in the previous sections can be used to construct parallel (polylogarithmic depth) and cache-oblivious algorithms for these problems. The approach in this section is similar to that of [20, 21] except that we use the cache-oblivious model instead of the parallel external memory model. Arge *et al.* [19] demonstrate a cache-oblivious algorithm for list ranking using priority queues and use it to construct other graph algorithms. But their algorithms have $\Omega(n)$ depth because their list ranking uses a serially-dependent sequence of priority queue operations to compute independent sets. The parallel algorithms derived from sorting are the same as in [60] except that we use different algorithms for the primitive operations scan and sort to have good complexity in the cost models relevant here. Moreover, a careful analysis (using standard techniques) is required to prove our sequential cache complexity and depth bounds under this framework.

6.5.1 List Ranking

A basic strategy for list ranking follows the three stage strategy proposed in [15] and adapted into the Parallel-External Memory model by [21]:

1. Shrink the list to size $O(n/\log n)$ through repeated contraction.
2. Apply Wyllie's pointer jumping [170] on this shorter list.
3. Compute the place of the origninal nodes in the list by splicing the elements thrown out in the compression and pointer jump

Stage 1 is achieved through finding independent sets in the list of size $\Theta(n)$ and removing them to yield a smaller problem. This can be done randomly using the random mate technique in which case, $O(\log\log n)$ rounds of such reduction would suffice. Each round involves a Pack operation (Section 6.1) to compute the elements still left.

Alternately, we could use the deterministic technique similar to Section III of [21]. Use two rounds of Cole and Vishkin's deterministic coin tossing [65] to find a $O(\log\log n)$-ruling set and then convert the ruling set to an independent set of size at least $n/3$ in $O(\log\log n)$ rounds. Arge *et al.* [21] showed how this conversion can be made cache-efficient, and it is straightforward to change this algorithm to a cache-oblivious one by replacing the Scan and Sort primitives. To convert the $O(\log\log n)$ ruling set into a 2- ruling set, in parallel, start at each ruler and traverse the list iteratively and assign a priority number to each node based on the distance from the ruler. Then sort the nodes based on the priority numbers so that the nodes are now in contiguous chunks based on distance to the previous ruler in the list. Choose chunks corresponding to prioirity numbers $0, 3, 6, 9, \ldots$ to get a 2-ruling set.

Stage 2 uses $O(\log n)$ rounds of pointer jumping, each round essentially involving a sort operation on $O(n/\log n)$ elements in order to figure out the next level of pointer jumping. Thus, the cache complexity of this stage is asymptotically the same as sorting and its depth is $O(\log n)$ times the depth of sorting.

Once this stage is completed, the original location of nodes in the list can be computed by tracing back through each of the compression and pointer jumping steps.

Theorem 28 *The deterministic list ranking outlined above has $Q^*(n; M, B) = O(Q^*_{sort}(n; M, B))$ sequential cache complexity, $O(n \log n)$ work, and $O(D_{sort}(n) \log n)$ depth. The effective cache complexity is $\widehat{Q}_\alpha(n; M, B) = Q^*(n; M, B)$ for $\alpha < 1$.*

Proof. Consider a single contraction step of stage one. The first part of this involving two deterministic coin tossing steps to compute $O(\log\log n)$-ruling set can be implemented with a small constant number of scans and sorts.

The second part involving iteratively creating a 2-ruling set requires that in the i-th iteration concerning the i-th group of size n_i, we use a small constant number of scans and sorts on problem of size n_i. In addition, it involves writing at most n_i elements in to $O(\log\log n)$ contiguous locations in to the memory. The cache complexity of this step is atmost

$$c\left(\sum_{i=1}^{\log\log n} Q^*_{sort}(n_i; M, B) + n_i/B + 1\right) \leq (c+1)\left(\lceil n/B\rceil \log_{M+2} n + \log\log n\right)$$

for some constant c. A closer analysis using the tall cache assumption would show that the second term $O(\log\log n)$ is asymptotically smaller than the first. If it is the case that $n < cM$ for

Problem	Depth	Cache Complexity (Q^*)
List Ranking	$O(D_{sort}(n)\log n)$	$O(Q_{sort}(n))$
Euler Tour on Trees	$O(D_{LR}(n))$	$O(Q_{sort}(n))$
Tree Contraction	$O(D_{LR}(n)\log n)$	$O(Q_{sort}(n))$
Least Common Ancestors (k queries)	$O(D_{LR}(n))$	$O(\lceil \frac{k}{n} \rceil Q_{sort}(n))$
Connected Components	$O(D_{LR}(n)\log n)$	$O(Q_{sort}(\lvert E\rvert)\log(\lvert V\rvert/\sqrt{M}))$
Minimum Spanning Forest	$O(D_{LR}(n)\log n)$	$O(Q_{sort}(\lvert E\rvert)\log(\lvert V\rvert/\sqrt{M}))$

Table 6.2: Low-depth cache-oblivious graph algorithms from Section 6.5. All algorithms are deterministic. D_{LR} represents the depth of List Ranking. The bounds assume $M = \Omega(B^2)$. D_{sort} and Q_{sort} are the depth and cache complexity of cache-oblivious sorting.

some suitable constant $0 < c < 1$, then the entire problem fits in cache and the cache complexity is simply $O(\lceil n/B \rceil)$. Suppose not, in order for $O(\log\log n)$ to be asymptotically greater than $O(n/B)$, it has to be the case that $B = \Omega(n/\log\log n)$. However, by tall cache assumption $B = o(M^{1-\epsilon})$ for some $0 < \epsilon < 1$ and consequently, $B = o(n^{1-\epsilon})$ which is a contradiction. Therefore, we can neglect the $\log\log n$ term.

Since we need $O(\log\log n)$ rounds each of such contractions each on a set of geometrically smaller size, the overall cache complexity of this first phase is just $Q^*_{Sort}(n; M, B)$. Since we have established that the second phase involving pointer jumping has the same cache complexity, the result follows. It is straightforward to verify the bounds for work and depth.

To compute the effective cache complexity, observe that all steps except the iterative coloring to find 2-ruling are existing parallel primitives with effective parallelism 1. Since the iterative coloring on n elements involves a fork with $O(n/\log\log n)$ parallel calls each with linear work on atmost $O(\log\log n)$ locations, the effective cache complexity term is work-dominated for $\alpha < 1 - o(1/\log n)$. The bounds on effective cache complexity follow. $\square$

6.5.2 Graph Algorithms

The complexity of some basic graph algorithms is tabulated in table 6.2. The algorithms that lead to these complexities are straightforward adaptations of known PRAM algorithms (as described for the external memory model in [60], and the Parallel External Memory Model [21]). For instance, finding an Euler Tour [156] involves scanning the input to compute the successor function for each edge and running a list ranking. Parallel Tree contraction involves alternating Rake and Compress operations[132]. While a Rake operation can be performed with Scan and Sort operations, the Compress operation requires constructing an Euler tour of the tree, finding an independent set on it and contracting the tour into a geometrically smaller problem. Since the size of the problem decreases geometrically after every phase of Rake and Compress operations, and each phase with the same asymptotic complexity as sorting, the bounds follow.

Finding Least Common Ancestors of a set of vertex pairs (batch version) in a tree involves computing the Euler tour and reducing the problem to a range minima query problem [26, 31]. The batch version of the range minima query problems which can be solved with cache-oblivious search trees [73].

Deterministic algorithms for Connected Components and Minimum Spanning Forest are similar and use tree contraction as their basic idea [61]. The connected components algorithm finds a forest on the graph, contracts the forest, and repeats the procedure. In each phase, every vertex picks the neighbor with the minumum label in parallel. The edges selected form a forest. Individual trees can be identified using the Euler tree technique and list ranking. Contracting the trees reduces the problem size by atleast a constant. For the Minimum Spanning Forest, each node selects the least weight edge incident on it instead of the neighbor with lowest index. The cache bounds are slightly worse than those in [60]: $\log(|V|/\sqrt{M})$ versus $\log(|V|/M)$. While [60] uses knowledge of M to transition to a different approach once the *vertices* in the contracted graph fit within the cache, cache-obliviously we need for the *edges* to fit before we stop incurring misses.

All the algorithms in this section except the LCA query problem have parallelizability 1.

6.6 Combinatorial Algorithms

The primitives described in the earlier section can be used to build good combinatorial algorithms. For instance, consider the parallel version of Maximal Independent Set problem – the Maximal Nearly Independent Set (MaNIS) described in [43]. The MaNIS primitive is exteremely useful for designing parallel approximation algorithms. It can be used to build algorithms for Set Cover, Max cover, Min-sum Set cover, Asymmetric k-center and Metric Facility Location problems [43, 154]. This section describes a parallel and cache-oblivious Set Cover algorithm and the MaNIS algorithm adapted from [43].

Theorem 29 (Parallel and I/O Efficient Set Cover) *The randomized approximate set cover algorithm on an instance of size n has expected complexity $\mathbb{E}[Q^*(n; M, B)] = O(Q^*_{sort}(n))$ and depth is $O(polylog(n))$* ***whp.*** *Furthermore, this implies an algorithm for prefix-optimal max cover and min-sum set cover in the same complexity bounds.*

Consider the BPT algorithm in Algorithm 6.6.1. At the core of it are the following 3 ingredients—prebucketing, maximal near-independent set (MaNIS), and bucket management—which we discuss in turn:

— *Prebucketing:* This component (Step ii of the algorithm) buckets the sets based on their cost. To ensure that the ratio between the costliest set and cheapest set is polynomially bounded so that the total number of buckets is kept logarithmic, as described in Lemma 4.2 of [43], sets that cost more than a threshold are discarded, and all sets cheaper than a certain threshold ($\mathcal{A}$) are included in the solution and the elements in these included sets marked as covered. Then $\mathcal{U}_0$ consists of the uncovered elements. The remaining sets are placed into $O(\log n)$ buckets $(A_0, A_1, \ldots, A_T)$ by their normalized cost (cost per element).

The algorithm then enters the main loop (step iii.), iterating over the buckets from the least to the most expensive and invoking MaNIS once in each iteration.

Algorithm 6.6.1 `SetCover` — Blelloch et al. parallel greedy set cover.

Input: a set cover instance $(\mathcal{U}, \mathcal{F}, c)$ and a parameter $\varepsilon > 0$.
Output: a *ordered* collection of sets covering the ground elements.

```
i. Let $\gamma = \max_{e \in \mathcal{U}} \min_{S \in \mathcal{F}} c(S)$, $n = \sum_{S \in \mathcal{F}} |S|$, $T = \log_{1/(1-\varepsilon)}(n^3/\varepsilon)$, and $\beta = \frac{n^2}{\varepsilon \cdot \gamma}$.
ii. Let $(\mathcal{A}; A_0, \ldots, A_T) = \texttt{Prebucket}(\mathcal{U}, \mathcal{F}, c)$ and $\mathcal{U}_0 = \mathcal{U} \setminus (\cup_{S \in \mathcal{A}} S)$.
iii. For $t = 0, \ldots, T$, perform the following steps:
    1. Remove deleted elements from sets in this bucket: $A'_t = \{S \cap \mathcal{U}_t : S \in A_t\}$
    2. Only keep sets that still belong in this bucket: $A''_t = \{S \in A'_t : c(S)/|S| > \beta \cdot (1-\varepsilon)^{t+1}\}$.
    3. Select a maximal nearly independent set from the bucket: $J_t = \texttt{MaNIS}_{(\varepsilon, 3\varepsilon)}(A''_t)$.
    4. Remove elements covered by $J_t$: $\mathcal{U}_{t+1} = \mathcal{U}_t \setminus X_t$ where $X_t = \cup_{S \in J_t} S$
    5. Move remaining sets to the next bucket: $A_{t+1} = A_{t+1} \cup (A'_t \setminus J_t)$
iv. Finally, return $\mathcal{A} \cup J_0 \cup \cdots \cup J_T$.
```

— MANIS *:* Invoked in Step iii(3) of the set cover algorithm, MANIS finds a subcollection of the sets in a bucket that are almost non-overlapping with the goal of closely mimicking the greedy behavior. Algorithm 6.6.2 shows the MANIS algorithm, reproduced from [43]. (The annotations on the side indicate which primitives in the PCC cost model we will use to implement them.) Conceptually, the input to MANIS is a bipartite graph with left vertices representing the sets and the right vertices representing the elements. The procedure starts with each left vertex picking random priorities (step 2). Then, each element identifies itself with the highest priority set containing it (step 3). If "enough" elements identify themselves with a set, the set selects itself (step 4). All selected sets and the elements they cover are eliminated (steps $5(1), 5(2)$), and the cost of remaining sets is re-evaluated based on the uncovered elements. Only sets (A') with costs low enough to belong to the correct bucket (which invoked this MANIS) are selected in step $5(3)$ and the procedure continues with another level of recursion in step 6. The net result is that we choose nearly non-overlapping sets, and the sets that are not chosen are "shrunk" by a constant factor.

— *Bucket Movement:* The remaining steps in Algorithm 6.6.1 are devoted to moving sets between buckets, ensuring the contents of the least-costly bucket contain only sets in a specific cost range. Step iii(4) removes elements covered by the sets returned by MANIS ; step iii(5) transfers the sets not selected in MANIS to the next bucket; and step iii(2) selects only those sets with costs in the current bucket's range for passing to MANIS .

6.6.1 Algorithm with good locality

We now discuss the right set of data structures and primitives to make the above algorithm both parallel and I/O efficient. In both the set cover and MANIS algorithms, the set-element instance is represented as a bipartite graph with sets on the left. A list of sets as well as a compact and contiguous adjacency list for each set is stored. The universe of elements $\mathcal{U}$ is represented as a bitmap indexed by an element identifier which is updated to indicate when element has been covered. Since we only need one bit of information per element to indicate whether it is covered or not, this can be stored in $O(\frac{|\mathcal{U}|}{\log n})$ words. Unlike in [42], we do not maintain back pointers from the elements to the sets.

— *Prebucketing:* This phase involves sorting sets based on their cost and a filter to remove the costliest and the cheapest set. Sets can then be partitioned into buckets with a merge operation. All operations have less than $Q^*_{sort}(n)$ cache complexity in the PCC framework and $O(\log^2 n)$ depth.

— MANIS *:* We invoke MANIS in step 3 of the set cover algorithm. Inside MANIS , we store the remaining elements of $\mathcal{U}$ (right vertices) as a sequence of element identifiers. To implement MANIS , in Step 3, for each left vertex, we copy its value x_a to all the edges incident on it, then sort the edges based on the right vertex so that the edges incident on the same right vertex are contiguous. For each right vertex, we can now use a prefix "sum" using maximum to find the neighbor a with the maximum x_a. In step 4, the "winning" edges (an edge connecting a right vertex with it's chosen left vertex) are marked on the edge list for each right vertex we computed in step 3. The edge list is then sorted based on the left vertex. Prefix sum can then be used to compute the number of elements each set has "won". A compact representation for J and its adjacency list can be computed with a filter operation. In step 5(1), the combined adjacency list of elements in J is sorted and duplicates removed to get $\overline{B}$. For step 5(2), we first evaluate the list $A \setminus J$. Then, we sort the edges incident on $A \setminus J$ based on their right vertices; merge with the remaining elements to identify which is contained in $\overline{B}$, marking these edges accordingly. After sorting these edges based on their left vertices, for each left vertex $a \in A \setminus J$, we filter and pack the "live edges" to compute $N'_G(a)$. Steps 5(3) and 5(4) involve simple filter and sum operations. The most I/O intensive operation (as well as the operation with maximum depth) in each round of MANIS is sorting, which has at most $O(Q^*_{sort}(|G|))$ cache complexity in the PCC framework and $O(\log^2 |G|)$ depth. As analyzed in [43], for a bucket with n_t edges to start with, MANIS runs for at most $O(\log n_t)$ rounds—and after each round, the number of edges drops by a constant factor; therefore, we have the following bounds:

Lemma 30 *The cost of running* MANIS *on a bucket with n_t edges in the parallel cache complexity model is $O(Q^*_{sort}(n_t))$, and the depth is $O(D_{sort}(n_t)\log^2 n_t)$. Its parallelizability is* 1.

— *Bucket Movement:* We assume the A_t, A'_t and A''_t are stored in the same format as the input for MANIS (see 6.6.1). The right set of vertices of the bipartite graph is now a bitmap corresponding to the elements indicating whether an element is alive or dead. Step iii(1) is similar to Step 3 of MANIS . We first sort $S \in A_t$ to order the edges by element index, then merge this representation (with a vector of length $O(|\mathcal{U}|/\log n)$) to match them with the elements bitmap, do a filter to remove deleted edges, and perform another sort to get them back ordered by set. Step iii(2) is simply a filter. The append operation in Step iii.5 is no more expensive than a scan. In the PCC models, these primitives have cache complexity at most $O(Q^*_{sort}(n_t) + Q^*_{scan}(|\mathcal{U}|/\log n))$ for a bucket with n_t edges. They all have $O(D_{sort}(n))$ depth.

To show the final cache complexity bounds, we make use of the following claim:

Claim 31 ([43]) *Let n_t be the number of edges in bucket t at the beginning of the iteration which processes this bucket. Then, $\sum n_t = O(n)$.*

Therefore, we have $O(Q^*_{sort}(n))$ from prebucketing, $O(Q^*_{sort}(n))$ from MANIS combined, and $O(Q^*_{sort}(n)+Q^*_{scan}(\mathcal{U}))$ from bucket management combined (since there are $\log(n)$ rounds).

Algorithm 6.6.2 $\texttt{MaNIS}_{(\varepsilon,3\varepsilon)}(G)$

Input: A bipartite graph $G = (A, N_G(a))$
A is a sequence of left vertices (the sets), and $N_G(a), a \in A$ are the neighbors of each left vertex on the right.
These are represented as contiguous arrays. The right vertices are represented implicitly as $B = N_G(A)$.
Output: $J \subseteq A$ of chosen sets.

```
1. If $A$ is empty, return the empty set.
2. For $a \in A$, randomly pick $x_a \in_R \{0, \ldots, n_G^7 - 1\}$.                //map
3. For $b \in B$, let $\varphi$ be $b$'s neighbor with maximum $x_a$     // sort and prefix sum
4. Pick vertices of $A$ "chosen" by sufficiently many in $B$:   // sort, prefix sum, sort and filter

        $J = \{a \in A | \#\{b : \varphi(b) = a\} \geq (1 - 4\varepsilon)D(a)\}.$

5. Update the graph by removing $J$ and its neighbors, and elements of $A$ with too few remaining neighbors:
   (1) $\overline{B} = N_G(J)$ (elements to remove)                                   // sort
   (2) $N'_G = \{\{b \in N_G(a) | b \notin \overline{B}\} : a \in A \setminus J\}$   // sort, merge, sort and filter
   (3) $A' = \{a \in A \setminus J : |N'_G(a)| \geq (1 - \varepsilon)D(a)\}$          // filter
   (4) $N''_G = \{\{b \in N'_G(a)\} : a \in A'\}$                                      // filter
6. Recurse on reduced graph: $J_R = \texttt{MaNIS}_{(\varepsilon,3\varepsilon)}((A', N''_G))$
7. return $J \cup J_R$
```

This simplifies to an cache complexity of $Q^*(n; M, B) = O(Q^*_{sort}(n; M, B))$ since $U \leq n$. The depth is $O(\log^4 n)$, since set cover has $O(\log n)$ iterations to go through buckets and invokes a MaNIS which has $O(\log n)$ rounds (w.h.p.) and each round is dominated by the sort. Every subroutine in the algorithm has parallelizability 1. Since the entire algorithm can be programmed with $O(n)$ statically allocated space, the parallelizability of the algorithm is 1.

6.7 Performance

The algorithms presented in this Chapter also perform and scale well in practice. The sample sort and the set cover algorithms were implemented in Intel Cilk+ and run on a 40-core consisting of 4 2.4GHZ Intel $E7 - 8870$ Xeon Processors with 256GB of RAM connected with a 1066MHz bus. Figures 6.4 and 6.5 demonstrate the scalability of the Sample Sort and MaNIS based set cover algorithms respectively.

Furthermore, the algorithms described here are widely used in the algorithms written for the problems in the PBBS benchmark suite [34]. The performance of these algorithms is tabulated in Table 2 of Shun et.al. [149] which is reproduced here in Table 6.3. The algorithms have good sequential performance and demonstrate good scalability.

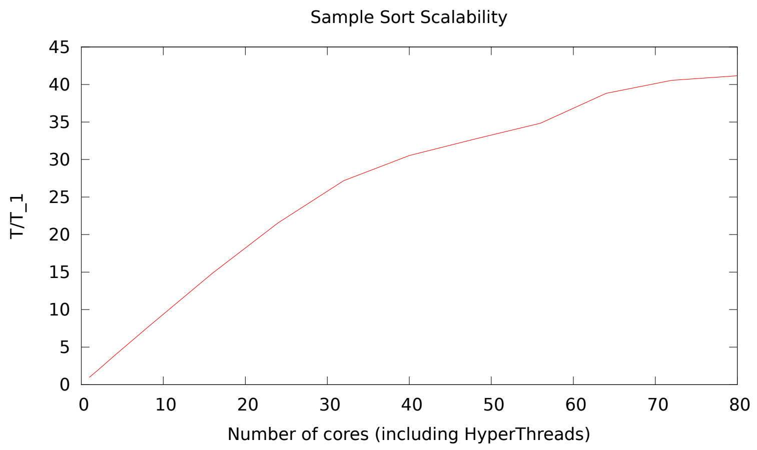


Figure 6.4: Plot of the speedup of samplesort on upto 40 cores with $2\times$ hyperthreading. The algorithm achieves over $30\times$ speedup on 40 cores without hyperthreading. The baseline for one core is 2.72 seconds and the parallel running time is 0.066 seconds.

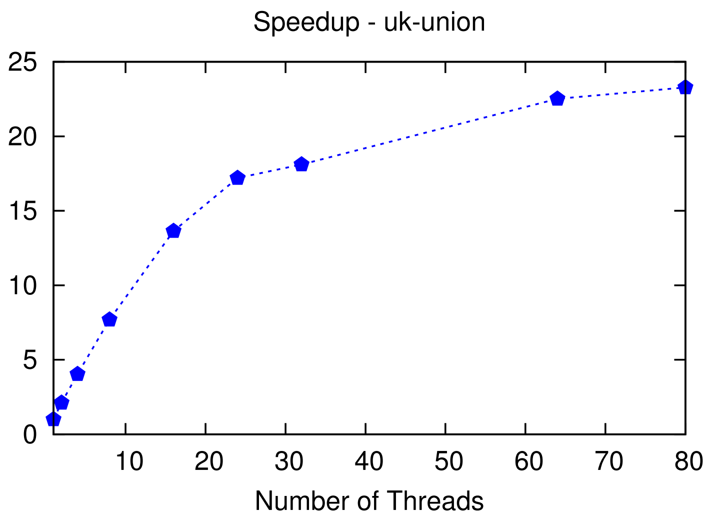


Figure 6.5: Speedup of the parallel MANIS based set cover algorithm on an instance with 121 million sets, 133 million elements and 5.5 billion edges (set-element relations). The baseline for one core is 278 seconds and the parallel running time is 11.9 seconds.

Application Algorithm	**1 thread**	**40 core**	T_1/T_{40}	T_S/T_{40}
Integer Sort				
serialRadixSort	0.48	–	–	–
parallelRadixSort	0.299	0.013	23.0	36.9
Comparison Sort				
serialSort	2.85	–	–	–
sampleSort	2.59	0.066	39.2	43.2
Spanning Forest				
serialSF	1.733	–	–	–
parallelSF	5.12	0.254	20.1	6.81
Min Spanning Forest				
serialMSF	7.04	–	–	–
parallelKruskal	14.9	0.626	23.8	11.2
Maximal Ind. Set				
serialMIS	0.405	–	–	–
parallelMIS	0.733	0.047	14.1	8.27
Maximal Matching				
serialMatching	0.84	–	–	–
parallelMatching	2.02	0.108	18.7	7.78
K-Nearest Neighbors				
octTreeNeighbors	24.9	1.16	21.5	–
Delaunay Triangulation				
serialDelaunay	56.3	–	–	–
parallelDelaunay	76.6	2.6	29.5	21.7
Convex Hull				
serialHull	1.01	–	–	–
quickHull	1.655	0.093	17.8	10.9
Suffix Array				
serialKS	17.3	–	–	–
parallelKS	11.7	0.57	20.5	30.4

Table 6.3: Weighted average of running times (seconds) over various inputs on a 40-core machine with hyper-threading (80 threads). Time of the parallel version on 40 cores (T_{40}) is shown relative to both (i) the time of the serial version (T_S) and (ii) the parallel version on one thread (T_1). In some cases our parallel version on one thread is faster than the baseline serial version. Table reproduced from [149].

Chapter 7

Experimental Analysis of Space-Bounded Schedulers

We advocated the separation of algorithm and scheduler design, and argued that schedulers are to be designed for each machine model. Robust schedulers for mapping nested parallel programs to machines with certain kinds of simple cache organizations such as single-level shared and private caches were described in 5.2. They work well both in theory [37, 48, 49] and in practice [120, 134]. Among these, the *work-stealing scheduler* is particularly appealing for private caches because of its simplicity and low overheads, and is widely deployed in various run-time systems such as Cilk++. The Parallel Depth-First scheduler [37] is suited for shared caches and practical versions of this schedule have been studied. The cost of these schedulers in terms of cache misses or running times can be bounded by the locality cost of the programs as measured in certain abstract program-centric cost models [2, 41, 46, 49].

However, modern parallel machines have multiple levels of cache, with each cache shared amongst a subset of cores (e.g. , see Fig. 1(a)). We used the parallel memory hierarchy model, as represented by a tree of caches [13] (Fig. 1(b)), as a reasonably accurate and tractable model for such machines [41, 80]. Previously studied schedulers for simple machine models may not be optimal for these complex machines. Therefore, a variety of hierarchy-aware schedulers have been proposed recently [41, 64, 68, 80, 140] for use on such machines.

Among these are the class of space-bounded schedulers [41, 64, 68]. We described a particular class of space-bounded schedeulers in Section 5.3 along with performance bounds. To use space-bounded schedulers, the computation needs to annotate each function call with the size of its memory footprint. Roughly speaking, the scheduler tries to match the memory footprint of a subcomputation to a cache of appropriate size in the hierarchy and then run the subcomputation fully on the cores associated with that cache. Note that although space annotations are required, the computation can be oblivious to the size of the caches and hence is portable across machines. Under certain conditions these schedulers can guarantee good bounds on cache misses at every level of the hierarchy and running time in terms of some intuitive program-centric metrics. Cole and Ramachandran [68] describe such schedulers with strong asymptotic bounds on cache misses and runtime for highly balanced computations. Recent work [41] describes schedulers that generalizes to unbalanced computations as well.

In parallel with this line of work, certain variants of the work-stealing scheduler such as the

PWS and HWS schedulers [140] that exploit knowledge of memory hierarchy to a certain extent have been proposed, but no theoretical bounds are known.

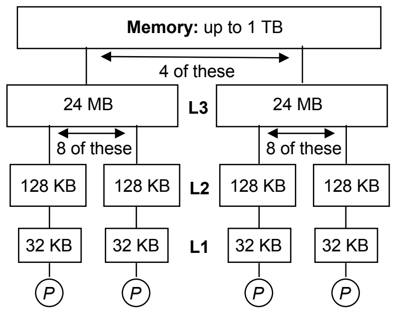
While space-bounded schedulers have good theoretical guarantees on the PMH model, there has been no extensive experimental evidence to suggest that these (asymptotic) guarantees translate into good performance on real machines with multi-level caches. Existing analyses of these schedulers ignore the overhead costs of the scheduler itself and account only for the program run time. Intuitively, given the low overheads and highly-adaptive load balancing of work-stealing in practice, space-bounded schedulers would seem to be inferior on both accounts, but superior in terms of cache misses. This raises the question as to the relative effectiveness of the two types of schedulers in practice.

Experimental Framework and Results. We presents the first experimental study aimed at addressing this question through a head-to-head comparison of work-stealing and space-bounded schedulers. To facilitate a fair comparison of the schedulers on various benchmarks, it is necessary to have a framework that provides separate modular interfaces for writing portable nested parallel programs and specifying schedulers. The framework should be light-weight, flexible, provide fine-grained timers, and enable access to various hardware counters for cache misses, clock cycles, etc. Prior scheduler frameworks, such as the Sequoia framework [80] which implements a scheduler that closely resembles a space-bounded scheduler, fall short of these goals by (i) forcing a program to specify the specific sizes of the levels of the hierarchy it is intended for, making it non-portable, and (ii) lacking the flexibility to readily support work-stealing or its variants.

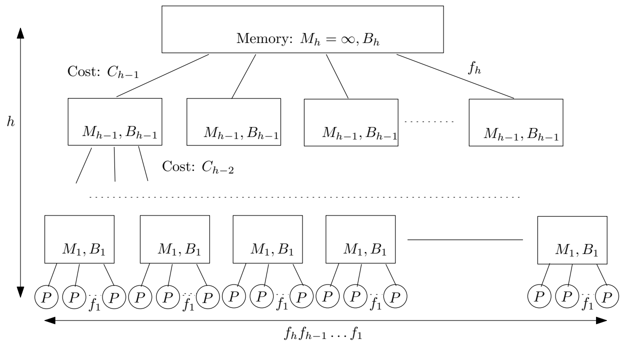
This chapter describes a scheduler framework that we designed and implemented, which achieves these goals. To specify a program in the framework, the programmer uses the Fork-Join (and Parallel-For built on top of Fork-Join) primitive. To specify the scheduler, one needs to implement just three primitives describing the management of tasks at Fork and Join points: `add`, `get`, and `done`. Any scheduler can be described in this framework as long as the schedule does not require the preemption of sequential segments of the program. A simple work-stealing scheduler, for example, can be described with only 10s of lines of code in this framework. Furthermore, in this framework, program tasks are completely managed by the schedulers, allowing them full control of the execution.

The framework enables a head-to-head comparison of the relative strengths of schedulers in terms of running times and cache miss counts across a range of benchmarks. (The framework is validated by comparisons with the commercial CilkPlus work-stealing scheduler.) We present experimental results on a 32-core Intel Nehalem series Xeon 7560 multi-core with 3 levels of cache. As depicted in Figure 7.1(a)), each L3 cache is shared (among the cores on a socket) while the L1 and L2 caches are exclusive to cores. We compare four schedulers—work-stealing, priority work-stealing (PWS) [140], and two variants of space-bounded schedulers—on both divide-and-conquer micro-benchmarks (scan-based and gather-based) and popular algorithmic kernels such as quicksort, sample sort, matrix multiplication, and quad trees.

Our results indicate that space-bounded schedulers reduce the number of L3 cache misses compared to work-stealing schedulers by 25–50% for most of the benchmarks, while incurring up to 6% additional overhead. For memory-intensive benchmarks, the reduction in cache misses



(a) 32-core Intel Xeon 7560



(b) PMH model of [13]

Figure 7.1: Memory hierarchy of a current generation architecture from Intel, plus an example abstract parallel hierarchy model. Each cache (rectangle) is shared by all cores (circles) in its subtree.

overcomes the added overhead, resulting in up to a 25% improvement in running time for synthetic benchmarks and about 5% improvement for algorithmic kernels. To better understand the impact of memory bandwidth on scheduler performance, we quantify runtime improvements over a 4-fold range in the available bandwidth per core and show further improvements in the running times of kernels (up to 25%) when the bandwidth is reduced.

Finally, as part of our study, we generalize prior definitions of space-bounded schedulers to allow for more practical variants, and explore the tradeoffs in a key parameter of such schedulers. This is useful for engineering space-bounded schedulers, which were previously described only at a high level suitable for theoretical analyses, into a form suitable for real machines.

Contributions. The contributions of this chapter are as follows.

- A modular framework for describing schedulers, machines as tree of caches, and nested parallel programs (Section 7.1). The framework is equipped with timers and counters. Schedulers that are expected to work well on tree of cache models such space-bounded schedulers and certain work-stealing schedulers are implemented.
- A precise definition for the class of space-bounded schedulers that retains the competitive cache miss bounds expected for this class, but also allows more schedulers than previous definitions (which were motivated mainly by theoretical guarantees [41, 68]) (Section 7.2). We describe two variants, highlighting the engineering details that allow for low overhead.
- The first experimental study of space-bounded schedulers, and the first head-to-head comparison with work-stealing schedulers (Section 7.3). On a common multi-core machine configuration (4 sockets, 32 cores, 3 levels of caches), we quantify the reduction in L3 cache misses incurred by space-bounded schedulers relative to both work-stealing variants on synthetic and non-synthetic benchmarks. On bandwidth-bound benchmarks, an improvement in cache misses translates to improvement in running times, although some of the improvement is eroded by the greater overhead of the space-bounded scheduler.

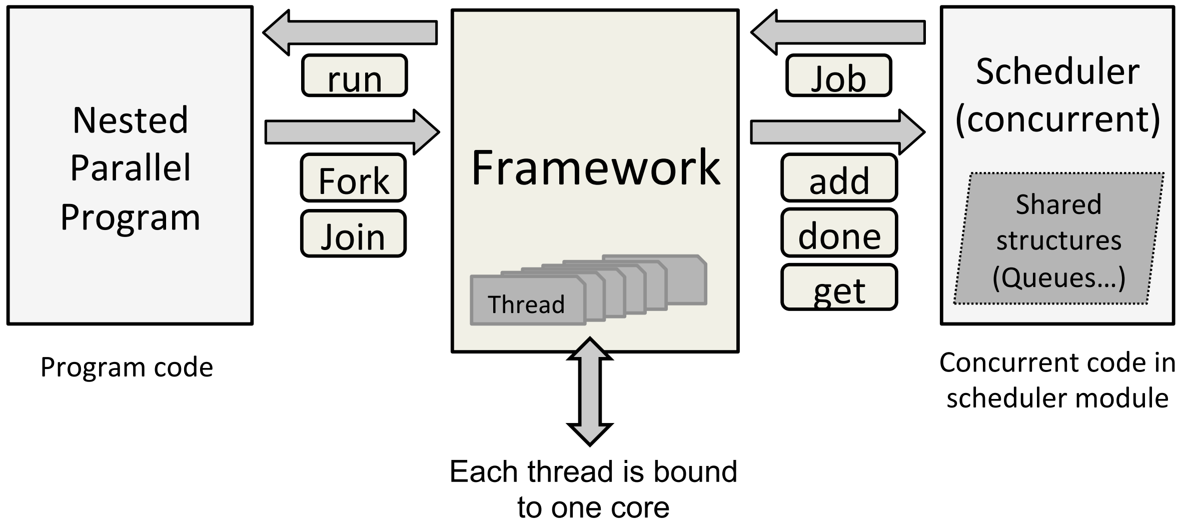
7.1 Experimental Framework

We implemented a C++ based framework with the following design objectives in which nested parallel programs and schedulers can be built for shared memory multi-core machines. Some of the code for threadpool has been adapted from an earlier implementation [108].
Modularity: The framework completely separates the specification of three components—programs, schedulers, and description of machine parameters—for portability and fairness. The user can choose any of the candidates from these three categories. Note, however, some schedulers may not be able to execute programs without scheduler-specific hints (such as space annotations).
Clean Interface: The interface for specifying the components should be clean, composable, and the specification built on the interface should be easy to reason about.
Hint Passing: While it is important to separate program and schedulers, it is necessary to allow the program to pass useful hints to the scheduler. Because scheduling decisions are made "online", i.e. on the fly, the scheduler may not have a complete picture of the program until its execution is complete. However, carefully designed algorithms allow static analyses that provide estimates on various complexity metrics such as work, size, and cache complexity. Annotating the program with these analyses and passing them to the scheduler at run time may help the scheduler make better decisions.
Minimal Overhead: The framework itself should be light-weight with minimal system calls, locking and code complexity. The control flow should pass between the functional modules (program, scheduler) with negligible time spent outside. The framework should avoid generating background memory traffic and interrupts.
Timing and Measurement: It should enable fine-grained measurements of the various modules. Measurements include not only clock time, but also insightful hardware counters such as cache and memory traffic statistics. In light of the earlier objective, the framework should avoid OS system calls for these, and should use direct assembly instructions.

7.1.1 Interface

The framework has separate interfaces for the program and the scheduler.
Programs: Nested parallel programs, with no other synchronization primitives, are composed from tasks using `fork` and `join` constructs. A `parallel_for` primitive built with `fork` and `join` is also provided. Tasks are implemented as instances of classes that inherit from the `Job` class. Different kinds of tasks are specified as classes with a method that specifies the code to be executed. An instance of a class derived from `Job` is a task containing a pointer to a strand nested immediately within the task. The control flow of this function is sequential with a terminal `fork` or `join` call. To adapt this interface for non-nested parallel programs such as futures [153], we would need to add other primitives to the interface that the program uses to call the framework beyond `fork` and `join`.

The interface allows extra annotations on a task such as its size, which is required by space-bounded schedulers. Such tasks inherit a derived class of `Job` class, the extensions in the derived class specifying the annotations. For example, the class `SBJob` suited for space-bounded schedulers is derived from `Job` by adding two functions—`size(uint block_size)` and `strand_size(uint block_size)`—that allow the annotations of the job size.



Nested Parallel Program
run
Fork
Join
Framework
Thread
Job
add
done
get
Scheduler (concurrent)
Shared structures (Queues...)
Program code
Concurrent code in scheduler module
Each thread is bound to one core

Figure 7.2: Interface for the program and scheduling modules

Scheduler: The scheduler is a concurrent module that handles queued and live tasks (as defined in Section 2) and is responsible for maintaining its own queues and other internal shared data structures. The module interacts with the framework that consists of a thread attached to each processing core on the machine, through an interface with three call-back functions.

- `Job* get (ThreadIdType)`: This is called by the framework on behalf of a thread attached to a core when the core is ready to execute a new strand, after completing of a previously live strand. The function may change the internal state of the scheduler module and return a (possibly null) `Job` so that the core may immediately being execution on the strand. This function specifies *proc* for the strand.
- `void done(Job*,ThreadIdType)` This is called when a core completes executing a strand. The scheduler is allowed to update its internal state to reflect this completion.
- `void add(Job*,ThreadIdType)`: This is called when a `fork` or `join` is encountered. In case of a `fork`, this call-back is invoked once for each of the newly spawned tasks. For a `join`, it is invoked for the continuation task of the `join`. This function decides where to enqueue the job.

Other auxiliary parameters to these call-backs have been dropped from the above description for clarity and brevity. Fig. B.2 provides an example of a work-stealing scheduler. The `Job*` argument passed to these functions may be instances of one of the derived classes of `Job*` and carry additional information that is helpful to the scheduler.

Machine configuration: The interface for specifying machine descriptions accepts a description of the cache hierarchy: number of levels, fanout at each level, and cache and cache-line size at each level. In addition, a mapping between the logical numbering of cores on the system to their left-to-right position as a leaf in the tree of caches must be specified. For example, Fig. B.3

```
void
WS_Scheduler::add (Job *job, int thread_id) {
  _local_lock[thread_id].lock();
  _job_queues[thread_id].push_back(job);
  _local_lock[thread_id].unlock();
}
int
WS_Scheduler::steal_choice (int thread_id) {
  return (int)((((double)rand())/((double)RAND_MAX))
               *_num_threads);
}
Job*
WS_Scheduler::get (int thread_id) {
  _local_lock[thread_id].lock();
  if (_job_queues[thread_id].size() > 0) {
    Job * ret = _job_queues[thread_id].back();
    _job_queues[thread_id].pop_back();
    _local_lock[thread_id].unlock();
    return ret;
  } else {
    _local_lock[thread_id].unlock();
    int choice = steal_choice(thread_id);
    _steal_lock[choice].lock();
    _local_lock[choice].lock();
    if (_job_queues[choice].size() > 0) {
      Job * ret = _job_queues[choice].front();
      _job_queues[choice].erase(_job_queues[choice].begin());
      ++_num_steals[thread_id];
      _local_lock[choice].unlock();
      _steal_lock[choice].unlock();
      return ret;
    }
    _local_lock[choice].unlock();
    _steal_lock[choice].unlock();
  }
  return NULL;
}
void
WS_Scheduler::done (Job *job, int thread_id,
                    bool deactivate) {}
```

Figure 7.3: WS scheduler implemented in scheduler interface

is a description of one Nehalem-EX series 4-socket $\times$ 8-core machine (32 physical cores) with 3 levels of caches as depicted in Fig. 7.1(a).

7.1.2 Implementation

The runtime system initially fixes a POSIX thread to each core. Each thread then repeatedly performs a call (`get`) to the scheduler module to ask for work. Once assigned a task and a specific strand inside it, it completes the strand and asks for more work. Each strand either ends in a `fork` or a `join`. In either scenario, the framework invokes the `done` call back to allow the scheduler to clean up. In addition, when a task invokes a `fork`, the `add` call-back is invoked to let the scheduler add new tasks to its data structures.

All specifics of how the scheduler operates (e.g. , how the scheduler handles work requests, whether it is distributed or centralized, internal data structures, where mutual exclusion occurs, etc.) are relegated to scheduler implementations. Outside the scheduling modules, the runtime system includes no locks, synchronization, or system calls except during the initialization and cleanup of the thread pool.

7.1.3 Measurements

Active time and overheads: Control flow on each thread moves between the program and the scheduler modules. Fine-grained timers in the framework break down the execution time into five components: (i) active time—the time spent executing the program, (ii) `add` overhead, (iii) `done` overhead, (iv) `get` overhead, and (v) empty queue overhead. While active time depends on the number of instructions and the communication costs of the program, `add`, `done` and `get` overheads depend on the complexity of the scheduler, and the number of times the scheduler code is invoked by Forks and Joins. The empty queue overhead is the amount of time the scheduler fails to assign work to a thread (`get` returns null), and reflects on the load balancing capability of the scheduler. In most of the results in Section 7.3, we usually report two numbers: active time averaged over all threads and the average overhead, which includes measures (ii)–(v). Note that while we might expect this partition of time to be independent, it is not so in practice—the background coherence traffic generated by the scheduler's bookkeeping may adversely affect active time. The timers have very little overhead in practice—less than 1% in most of our experiments.
Measuring hardware counters: Multi-core processors based on newer architectures like Intel Nehalem-EX and Sandybridge contain numerous functional components such as cores (which includes the CPU and lower level caches), DRAM controllers, bridges to the inter-socket interconnect (QPI) and higher level cache units (L3). Each component is provided with a performance monitoring unit (PMU)—a collection of hardware registers that can track statistics of events relevant to the component.

For instance, while the core PMU on Xeon 7500 series (our experimental setup, see Fig. 7.1(a)) is capable of providing statistics such as the number of instructions, L1 and L2 cache hit/miss statistics, and traffic going in and out, it is unable to monitor L3 cache misses which constitute a significant portion of active time. This is because L3 cache is a separate unit with its own PMU(s). In fact, each Xeon 7560 die has eight L3 cache banks on a bus that also connects

```
int num_procs=32;
int num_levels = 4;
int fan_outs[4] = {4,8,1,1};
long long int sizes[4] = {0, 3*(1<<22), 1<<18, 1<<15};
int block_sizes[4] = {64,64,64,64};
int map[32] = {0,4,8,12,16,20,24,28,
               2,6,10,14,18,22,26,30,
               1,5,9,13,17,21,25,29,
               3,7,11,15,19,23,27,31};
```

Figure 7.4: Specification entry for a 32-core Xeon machine depicted in Fig. 7.1(a)

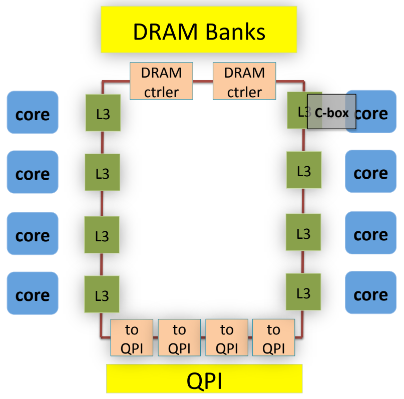


Figure 7.5: Layout of 8 cores and L3 cache banks on a bidirectional ring in Xeon 7560. Each L3 bank hosts a performance monitoring unit called C-box that measures traffic into and out of the L3 bank.

DRAM and QPI controllers (see Fig. 7.5). Each L3 bank is connected to a core via buffered queues. The address space is hashed onto the L3 banks so that a unique bank is responsible for each address. To collect L3 statistics such as L3 misses, we monitor PMUs (called C-Boxes on Nehalem-EX) on all L3 banks and aggregate the numbers in our results.

Software access to core PMUs on most Intel architectures is well supported by several tools including the Linux kernel, the Linux perf tool, and higher level APIs such as libpfm [137]. We use libpfm library to provide fine-grained access to the core PMU. However, access to *uncore* PMUs—complex architecture-specific components like the C-Box—is not supported by most tools. Newer Linux kernels (3.7+) are incrementally adding software interfaces to these PMUs at the time of this writing, but we are only able to make program-wide measurements using this interface rather than the fine-grained measurements. For accessing uncore counters, we adapt the Intel PCM 2.4 tool [98].

7.2 Schedulers Implemented

In this section, we will define the class of space-bounded schedulers and describe the schedulers we compare in our framework.

7.2.1 Space-bounded Schedulers

Space-Bounded schedulers were described in Section 5.3. We make a minor modification to the boundedness property introducing a new parameter μ that results in better results in practice.

A ***space-bounded scheduler*** for a particular cache hierarchy is a scheduler parameterized by $\sigma \in (0, 1]$ that satisfies the two following properties:

- *Anchored* Every subtask t of the root task is anchored to a **befitting** cache.
- *Bounded*: At every instant τ, for every level-i cache X_i, the sum of sizes of cache occupying tasks and strands is less than M_i:

$$\sum_{\mathsf{t} \in Ot(X_i,\tau)} S(\mathsf{t}, B_i) + \sum_{\ell \in Ol(X_i,\tau)} S(\ell, B_i) \leq M_i.$$

Well behaved: We say that a scheduler is well behaved if the total number of cache misses at level-i caches of a PMH incurred by scheduling a task t anchored to the root of the tree is at most $O(Q^*(\mathsf{t}; \sigma M_j, B_j))$, where Q^* is the parallel cache complexity as defined in the PCC model (Section 4.3). We assume that each cache uses the optimal offline replacement policy. Roughly speaking, the cache complexity Q^* of a task t in terms of a cache of size M and line size B is defined as follows. Decompose the task into a collection of maximal subtasks that fit in M space, and "glue nodes" – instructions outside these subtasks. For a maximal size M task t', the parallel cache complexity $Q^*(\mathsf{t}'; M; B)$ is defined to be the number of distinct cache lines it accesses, counting accesses to a cache line from unordered instructions multiple times. The model then pessimistically counts all memory instructions that fall outside of a maximal subtask (i.e., glue nodes) as cache misses. The total cache complexity of a program is the sum of the complexities of the maximal subtasks, and the memory accesses outside of maximal subtasks.

All space-bounded schedulers are well behaved. To see this, use the following replacement policy at each cache X_i in the hierarchy. Let cache occupying tasks at X_i cache their working sets at X_i. Allow a maximum of μ fraction of cache for each cache occupying strand at X_i. The cache occupying tasks bring in their working sets exactly once because the boundedness property prevents cache overflows. Cache misses for each cache occupying strand ℓ at cache X_i with task $\mathsf{t}(\ell)$ (anchored above level-i) is at most $1/\mu$ times $Q^*(\mathsf{t}(\ell); \sigma M_i, B_i)$. Since every subtask of the root task is anchored at a befitting cache, the cache misses are bounded by $O(Q^*(\cdot; \sigma M_i, B_i))$ at every level-i under this replacement policy. An ideal replacement policy at X_i would perform at least as well as such a policy. Note that while setting σ to 1 yields the best bounds on cache misses, it also makes load balancing harder. As we will see later, a lower value for σ like 0.5 allows greater scheduling flexibility.

7.2.2 Schedulers implemented

Space-bounded schedulers: SB and SB-D We implemented a space-bounded scheduler by constructing a tree of caches based on the specification of the target machine. Each cache is assigned one logical queue, a counter to keep track of "occupied" space and a lock to protect updates to the counter and queue. Cores can be considered to be leaves of the tree; when a scheduler call-back is issued to a thread, that thread can modify an internal node of the tree after gathering all locks on the path to the node from the core it is mapped onto. This scheduler accepts `Job`s which are annotated with task and strand sizes. When a new `Job` is spawned at a fork, the `add` call-back enqueues it at the cluster where it's parent was anchored. For a new `Job` spawned at a join, `add` enqueues it at the cluster where the `Job` that called the corresponding fork of this join was anchored.

A basic version of such a scheduler would implement logical queues at each cache as one queue. However, this presents two problems: (i) It is difficult to separate tasks in queues by the level of cache that befits it, and (ii) a single queue might be a contention hotspot. To solve problem (i), behind each logical queue, we use separate "buckets" for each level of cache below to hold tasks that befit those levels. Cores looking for a task at a cache go through these buckets from the top (heaviest tasks) to bottom. We refer to this variant as the **SB** scheduler. To solve problem (ii) involving queueing hotspots, we replace the top bucket with a distributed queue — one queue for each child cache — like in the work-stealing scheduler. We refer to the **SB** scheduler with this modification as the **SB-D** scheduler.

In the course of our experimental study, we found that the following minor modification to the boundedness property of space-bounded schedulers improves their performance. Namely, we introduce a new parameter $\mu \in (0,1]$ and modify the boundedness property to be that at every instant τ, for every level-i cache X_i:

$$\sum_{\mathsf{t} \in Ot(X_i,\tau)} S(\mathsf{t}, B_i) + \sum_{\ell \in Ol(X_i,\tau)} \min\{\mu M_i, S(\ell, B_i)\} \le M_i,$$

The minimum term with μM_i is to allow several large strands to be explored simultaneously without their space measure taking too much of the space bound. This helps the scheduler to quickly traverse the higher levels of recursion in the DAG and reveal parallelism so that the scheduler can achieve better load balance.

Work-Stealing scheduler: WS A basic work-stealing scheduler based on Cilk++ [48] is implemented and is referred to as the **WS** scheduler. Since the Cilk++ runtime system is built around work-stealing and deeply integrated and optimized exclusively for it, we focus our comparison on the **WS** implementation in our framework for fairness and to allow us to implement variants. The head-to-head micro-benchmark study in Section 7.3 between our **WS** implementation and Cilk++ suggests that, for these benchmarks, **WS** well-represents the performance of Cilk++'s work-stealing. We associate a double-ended queue (dequeue) of ready tasks with each core. The function `add` enqueues new tasks (or strands) spawned on a core to the bottom of its dequeue. When in need of work, the core uses `get` to remove a task from the bottom of its dequeue. If its dequeue is empty, it chooses another dequeue uniformly at random, and *steals* the work by removing a task from the *top* of that core's dequeue. The only contention in this type of scheduler is on the distributed dequeues—there is no other centralized data structure.

To implement the dequeues, we employed a simple two-locks-per-dequeue approach, one associated with the owning core, and the second associated with all cores currently attempting to steal. Remote cores need to obtain the second lock before they attempt to lock the first. Contention is thus minimized for the common case where the core needs to obtain only the first lock before it asks for work from its own dequeue.

Priority Work-Stealing scheduler: PWS Unlike in the basic **WS** scheduler, cores in the **PWS** scheduler [140] choose victims of their steals according to the "closeness" of the victims in the socket layout. Dequeues at cores that are closer in the cache hierarchy are chosen with a higher probability than those that are farther away to improve scheduling locality while retaining the load balancing properties of **WS** scheduler. On our 4 socket machines, we set the probability of an intra-socket steal to be 10 times that of an inter-socket steal.

7.3 Experiments

We describe experiments on two types of benchmarks. The first are a pair of micro-benchmarks designed specifically to enable us to measure various characteristics of the scheduler. We use these benchmarks to verify that the scheduler is getting the performance we expect and to see how the performance scales with bandwidth and number of cores per processor(socket). The second set of benchmarks are a set of divide-and-conquer algorithms for real problems (sorting, matrix-multiply and nearest-neighbors). These are used to better understand the performance in a more realistic setting. In all cases we measure both time and level 3 (last level) cache misses. We have found the cache misses on other levels do not vary significantly among the schedulers.

7.3.1 Benchmarks

Synthetic Benchmarks. We use two synthetic micro-benchmarks that mimic the behavior of memory-intensive divide-and-conquer algorithms. Because of their simplicity, we use these benchmarks to closely analyze the behavior of the schedulers under various conditions and verify that we get the expected cache behavior on a real machine.

- **Recursive Repeated Map (RRM).** This benchmark takes two n-length arrays A and B and a point-wise map function that maps elements of A to B. In our experiments the each element of the arrays is a double and the function simply adds one. RRM first does a parallel point-wise map from A to B, and repeats the same operation multiple times. It then divides A and B into two by some ratio (e.g. 50/50) and recursively calls the same operation on each of the two parts. The base case of the recursion is set to some constant at which point the recursion terminates. The input parameters are the size of the arrays n, number of repeats r, the cut ratio f, and the base-case size. We set $r = 3,\ f = 50\%$ in the experiments here unless mentioned otherwise. RRM is a memory intensive benchmark since there is very little work done per memory operation. However, once a recursive call fits in a cache (the cache size is at least $16n$ bytes) accesses are no longer required past that cache.
- **Recursive Repeated Gather (RRG).** This benchmark is similar to RRM but instead of doing a simple map it does a gather. In particular at any given level of the recursion it takes three n-length arrays A, B and I and for each location i sets $B[i] = A[I[i] \mod n]$. The values in I are random integers. As with RRM after repeating r times it splits the arrays in two and repeats on each part. RRG is even more memory intensive than RRM since its accesses are random instead of linear. Again, however, once a recursive call fits in a cache accesses are no longer required past the cache.

Algorithms. Our other benchmarks are a set of algorithms for some full, albeit simple, problems. These include costs beyond just memory accesses and are therefore more representative of real-world workloads.

- **Matrix Multiplication.** This benchmark is an 8-way recursive matrix multiplication. Matrix multiplication has cache complexity $Q(n) = \lceil 2n^2/B \rceil \times \left\lceil n/\sqrt{M} \right\rceil$. The ratio of instructions to cache misses is therefore very high, about $B\sqrt{M}$, making this is a very compute-intensive benchmark. We switch to a serial quicksort for sizes of 30×30, which fit within the L1 cache.
- **Quicksort.** This is a parallel quicksort algorithm that both parallelizes the partitioning and the recursive calls. It switches to a version which only parallelizes the recursive calls for $n < 128K$ and a serial version for $n < 16K$. These parameters worked well for all the schedulers. We note that our quicksort is about 2x faster than the Cilk code found in the Cilk+ guide [120]. This is because it does the partitioning in parallel. It is also the case that the divide does not exactly partition the data evenly, since it depends on how well the pivot divides the data. For an input of size n the program has cache complexity $Q(n; M, B) = O(\lceil n/B \rceil \log_2(n/M))$ and therefore is reasonably memory intensive.
- **Samplesort.** This is cache-optimal parallel Sample Sort algorithm described in [40]. The algorithm splits the input of size n into $\sqrt{n}$ subarrays, recursively sorts each subarray, "block transposes" them into $\sqrt{n}$ buckets and recursively sorts these buckets. For this algorithm, $W(n) = O(n \log n)$ and $Q^*(n; M, B) = O(\lceil n/B \rceil \log_{2+\frac{M}{B}} n/B)$ making it relatively cache friendly, and optimally cache oblivious even.
- **Aware Samplesort.** This is a variant of sample-sort that is aware of the cache sizes. In particular it moves elements into buckets that fit into the L3 cache and then runs quicksort on the buckets. This is the fastest sort we implemented and is in fact faster than any other sort we found for this machine. In particular it is about 10% faster than the PBBS sample sort [149].
- **Quad-Tree.** This generates a quad tree for a set of n points in two dimensions. This is implemented by recursively partitioning the points into four sets along the mid line of each of two dimensions. When the number of points is less than 16K we revert to a sequential algorithm.

7.3.2 Experimental Setup

The benchmarks were run against each of the four schedulers on the 4-socket 32 core Nehalem-EX architecture based Xeon 7560 machine described in Figs. 7.1(a) and B.3. The highest-level cache on each socket is L3 which is 24MB in size and shared by eight processors.

Controlling bandwidth. Each of the four socket on the machine has memory links to distinct DRAM modules. The sockets are connected with the Intel QPI interconnect which has a very high bandwidth. Memory requests from a socket to a DRAM module connected to another socket pass through the QPI, the remote socket and finally the remote memory link. Since the QPI has high bandwidth, if we treat the RAM as a single functional module, the L3-RAM bandwidth depends on the number of the memory links used which in turn depends on the mapping of pages to the DRAM modules. If all pages used by a program are mapped to DRAM modules connected to one socket, the program effectively utilizes one-fourth of the memory bandwidth. On the other hand, an even distribution of pages to DRAM modules across the sockets provides full bandwidth to the program. By controlling the mapping of pages, we can control the bandwidth available to the program. We report numbers with different bandwidth values in our results to highlight the sensitivity of running time to L3 cache misses.

Monitoring L3 cache. We picked the L3 cache for reporting as it is the only cache level that is shared and is consequently the only level where space-bounded schedulers can be expected to outperform work-stealing scheduler. It is also the highest and the most expensive cache level before DRAM on our machine. Unlike total execution time, partitioning L3 misses between application code and scheduler code was not possible due to software limitations. Even if it were possible to count them separately, it would difficult to interpret it because of the non-trivial interference between the data cached by program and the scheduler.

To count L3 cache misses, the uncore counters in the C-boxes were programmed using the Intel PCM tool to count misses that occur due to any reason (`LLC_MISSES - event code: 0x14, umask: 0b111`) and L3 cache fills in any coherence state (`LLC_S_FILLS - event code: 0x16, umask: 0b1111`). Both the numbers concur up to three significant digits in most cases. Therefore, only the L3 cache miss numbers are reported here.

To prevent excessive TLB cache misses, we use Linux hugepages of size 2MB to pre-allocate the space required by the algorithms. We configured the system to have a pool of $10,000$ huge pages by setting `vm.nr_hugepages` to that value using `sysctl`. We used the `hugectl` tool to supper memory allocations with hugepages.

7.3.3 Results

We set the values of σ and μ in **SB** and **SB-D** schedulers to 0.5 and 0.2 respectively after some experimentation with the parameters. All numbers reported in this paper are the average on at least 5 runs with the smallest and largest readings across runs removed.

Synthetic benchmarks. The number of L3 cache misses of RRM and RRG, along with their active times and overheads at different bandwidth values are plotted in Figs. 7.6 and 7.7 respectively. In addition to the four schedulers discussed in Section 7.2.2 (WS, PWS, SB and SB-D), we ran the experiments using the Cilk Plus work stealing scheduler to show that our scheduler performs reasonably compared to Cilk Plus. We could not separate overhead from time in Cilk Plus since it does not supply such information.

First note that, as expected, the number of L3 misses of a scheduler does not significantly depend on the bandwidth. The space-bounded schedulers perform significantly better in terms of L3 cache misses. The active time is influenced by the number of instructions in the benchmark as well as the number of L3 misses. Since the number of instructions is constant across schedulers, the number of L3 cache misses is the primary factor influencing active time. The extent to

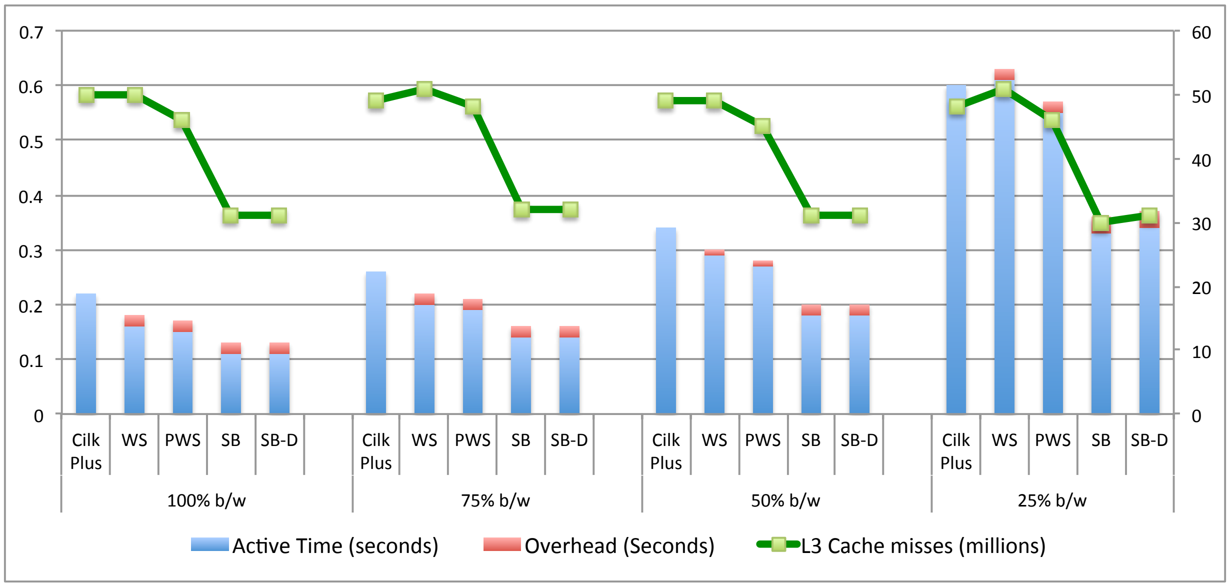


Figure 7.6: RRM on 10 million `double` elements running time and L3 misses for different bandwidth values. Left axis is time, right is cache misses.

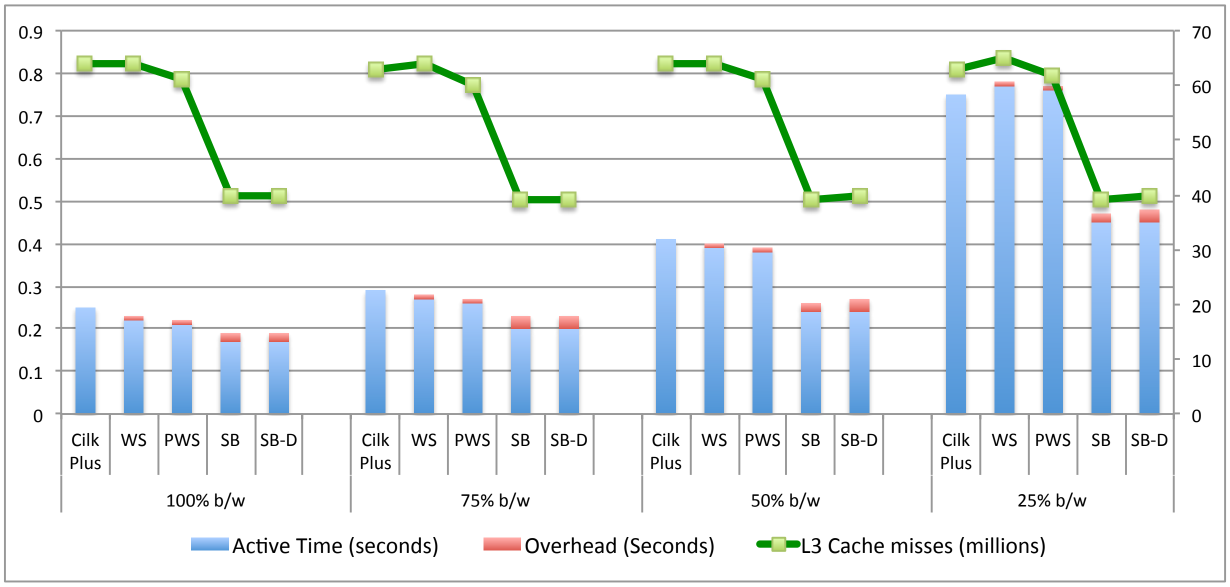


Figure 7.7: RRG on 10 million `double` elements running time and L3 misses for different bandwidth values.

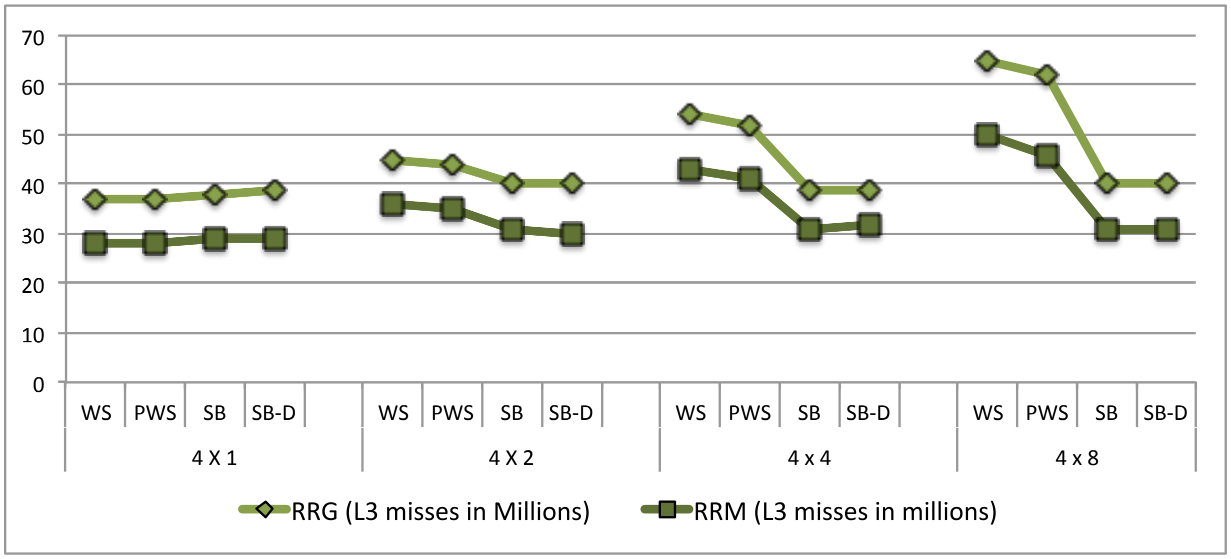


Figure 7.8: L3 cache misses for RRM and RRG with varying number of processors per socket.

which improvements in L3 misses translates to an improvement in active time depends on the bandwidth given to the program. When the bandwidth is low (25%), the active times are almost directly proportional to the number of L3 cache misses. At full bandwidth, the active time is less sensitive to misses.

The difference between L3 cache costs of space-bounded schedulers and **WS** scheduler roughly corresponds to the difference between cache complexity of the program with a cache of size σM_3 (M_3 being the size of L3) and a cache of size $M_3/8$ (eight processors share an L3 cache). In other words, space-bounded schedulers share the cache constructively while the work-stealing scheduler effectively splits the cache between processors. To see this, consider the RRM benchmark with double-precision arrays of size 10 million with 3 repeated Map operations and an evenly split recursion. Each Map operation that does not fit in L3 touches 160 million bytes of data, and RRM has to unfold four levels of recursion before it fits in σM_3 space ($= 0.5 \times 24MB = 12MB$) with space-bounded schedulers. Therefore, space-bounded schedulers incur about $(160 \times 10^6 Bytes) \times 3 \times 4/(64 Bytes) = 30 \times 10^6$ cache misses which matches closely with the experiment. On the other hand, the number of cache misses of the work-stealing scheduler (48 million) corresponds to unfolding about 6 levels of recursions, two more than with space-bounded schedulers. Loosely speaking, this means that the recursion has to unravel to one-eight the size of L3 before work-stealing scheduler starts preserving locality.

To support this observation, we ran these benchmarks with varying number of processors per socket and plotted them in Fig. 7.8. The number of L3 cache misses of the **SB** and **SB-D** scheduler do not change with the number of processors as processors constructively share the L3 cache independent of their numbers. However, in the **WS** scheduler, when fewer processors are active on each socket, there is lesser contention for the shared L3 cache which reflects in the smaller number of cache misses.

We note that although the priority work stealing scheduler works better in terms of cache misses than standard work stealing, it does not perform nearly as well as the space bounded schedulers. We also note that we found little difference between our two versions of space bounded schedulers.

These experiments indicate that the advantages of space bounded schedulers over work-stealing schedulers improve as (1) the number of cores per processor goes up, and (2) as the bandwidth per core goes down. At 8 cores there is about a 35% reduction in L3 cache misses. At 64 cores we would expect over a 60% reduction.

Algorithms. We measured the performance of the algorithms at two different bandwidths—100% and 25%—and plotted them in Figs. 7.9 and 7.10 respectively. The space-bounded schedulers **SB** and **SB-D** incur significantly fewer L3 cache misses across the benchmarks, up to 50% on matrix multiply. This difference is significantly larger than the difference between the cache costs of **WS** and **PWS** schedulers. The cache-oblivious sample sort is the only benchmark in which there is little difference in L3 cache misses across schedulers. Due to its a $\sqrt{n}$-way recursion, all subtasks after one level of recursion are much smaller than L3 cache for the problem sizes we consider. Both **SB** and **WS** schedulers are thus capable of preserving locality on these small subtasks equally well.

The sensitivity of active time to L3 cache misses depends on whether the algorithm is memory-intensive enough to stress the bandwidth. Marix Multiplication, although benifitting from space-bounded schedulers in terms of cache misses, shows no improvement in run times even at 25% bandwidth as it is very compute intensive. The other three benchmarks—Quicksort, Aware Samplesort and Quad-tree are memory intensive and see significant improvements in running time (up to 25%) at limited bandwidth (Figure 7.10). Some of the improvement in running time is lost at full bandwidth on this machine.

Load balance and dilation parameters σ. The choice of σ, determining which tasks are maximal, is an important parameter affecting the performance of space-bounded schedulers, especially the load balance. If σ is set to 1, it is likely that one task that is about the size of cache gets anchored to the cache leaving little room for other tasks or strands. This adversely affects load balance, and we might expect to see greater empty queue times (see Figure 7.11 for Quad-Tree sort). If σ is set to a lower value like 0.5, then each cache can allow more than one task or strand to be simultaneously anchored leading to better load balance. If σ is too low (closer to 0), the recursion level at which space-bounded scheduler preserves locality is lowered resulting in less effective use of shared caches.

7.4 Conclusion

We developed a framework for comparing schedulers, and deployed it on a 32-core machine with 3 levels of caches. We used it to compare four schedulers, two each of work stealing and space-bounded types. As predicted by theory, we did notice that space-bounded schedulers demonstrate some, or even significant, improvement over work-stealing schedulers in terms of cache miss counts on shared caches for most benchmarks. In memory-intensive benchmarks with low instruction count to cache miss count ratios, an improvement in L3 miss count because

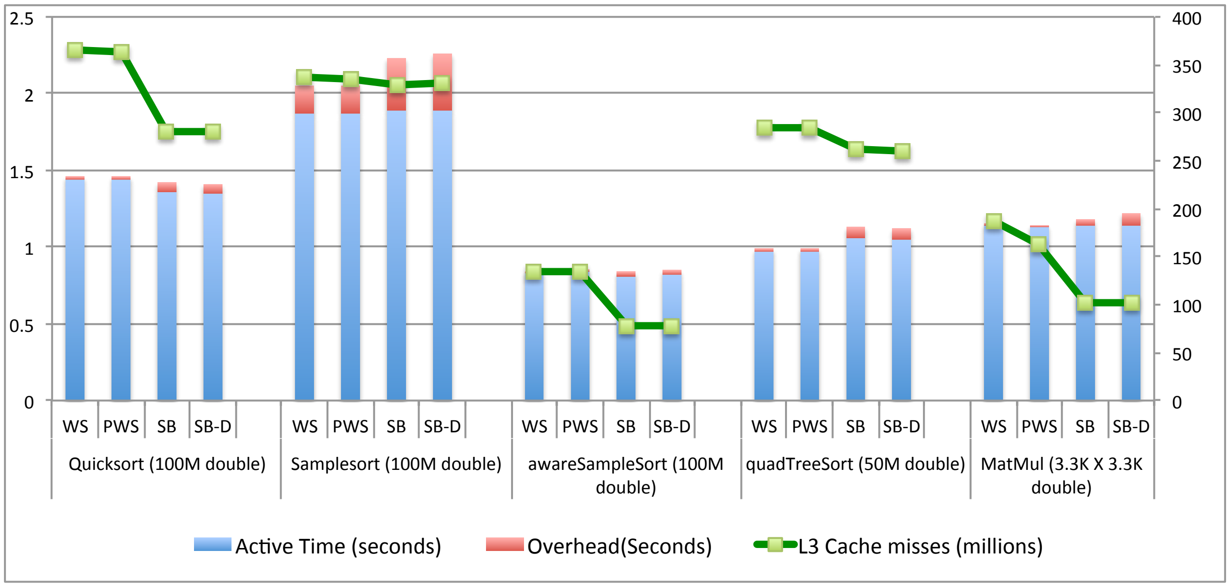


Figure 7.9: Active times, overheads and L3 cache misses for algorithms at full bandwidth. L3 misses for MatMul are multiplied by 10.

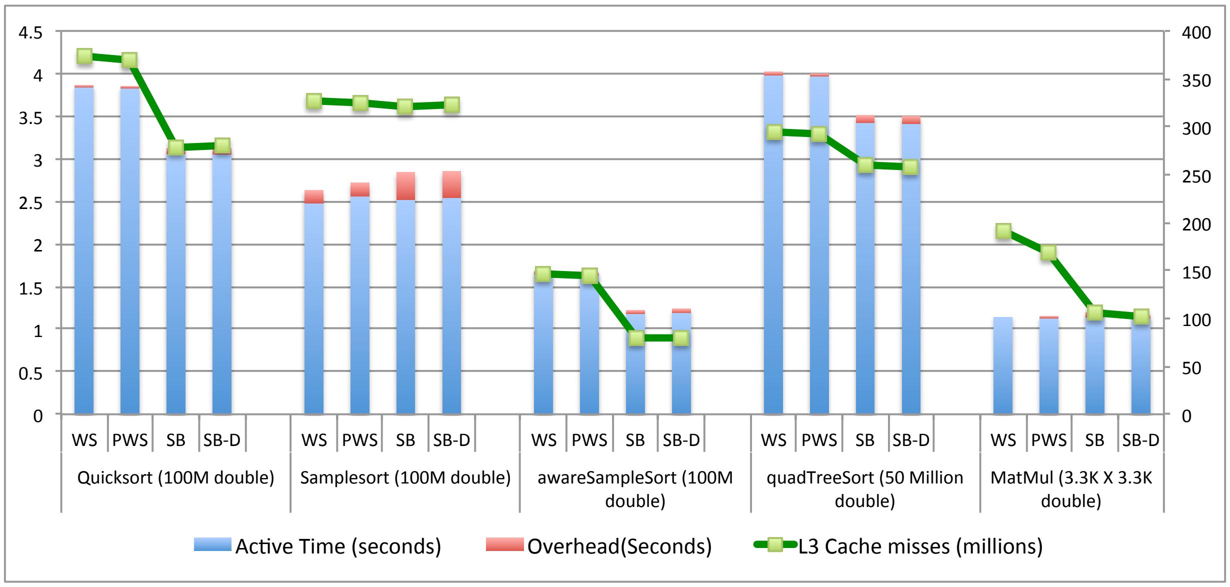


Figure 7.10: Active times, overheads and L3 cache misses for algorithms at 25% bandwidth. L3 misses for MatMul are multiplied by 10.

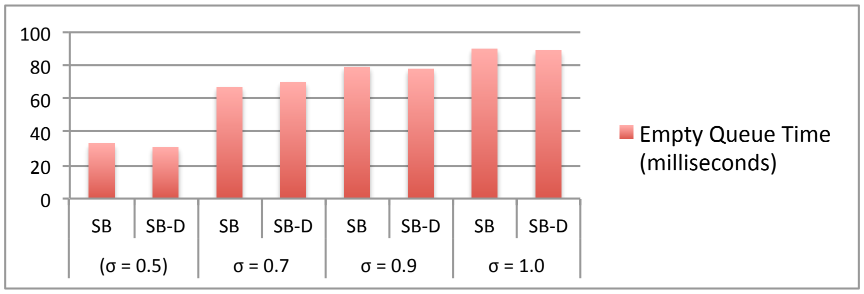


Figure 7.11: Empty queue times for Quad-Tree Sort.

of space-bounded schedulers can improve running time, despite their added overheads. On the other hand, for compute-intensive benchmarks or benchmarks with highly optimized cache complexities, space-bounded schedulers do not yield much advantage over WS schedulers in terms of active time and most of the advantage is lost due to greater scheduler overheads. Improving the overhead of space-bounded schedulers further could make the case for space-bounded schedulers stronger and is an important direction for future work.

Our experiments were run on a current generation Intel multicore with only 8 cores per socket, only 32 cores total, and only one level of shared cache (the L3). The experiments make it clear that as the core count per chip goes up the advantages of space-bounded schedulers significantly increases due to the increased benefit of avoiding conflicts among the many unrelated threads sharing the limited memory (cache) on the chip. A secondary benefit is that it is likely that the bandwidth per core will go down with an increased number of cores, again giving an advantage to space-bounded schedulers. We also anticipate a greater advantage for space-bounded schedulers over work-stealing schedulers when more cache levels are shared and when the caches are shared amongst a greater number of cores. Such studies are left to future work as and when the necessary hardware becomes available.

Chapter 8

Conclusions

We made a case for designing program-centric models for locality and parallelism and presented the PCC framework as a suitable candidate. We designed schedulers that mapped program costs in the PCC framework to good performance bounds on communication costs, time and space on a Parallel Memory Hierarchy. This demonstrates that the PCC framework captures the cost of parallel algorithms on shared memory machines since the parallel memory hierarchy model is a good approximation for a broad class of shared memory machines. Therefore, we claim that the program-centric cost model for quantifying locality and parallelism is feasible.

The usefulness of the model is demonstrated through the design of algorithms for many common problems that are not only optimal according to the metrics in the PCC framework, but also perform extremely well in practice. We also demonstrated that practical adaptations of our theoretical designs for schedulers perform better than the state of the art for a certain class of algorithms, and anticipate their increasing relevance on larger scale machine where the gap between computation and communication speed will widen.

8.1 Future Work

Many cost models covering different models of computation (circuits, machines and programs) have been proposed for quantifying locality and parallelism. However, quantitative relations between these cost models are not well understood. The most important direction for future work would be to understand the connections between them, and establish a more "unified" theory of locality for computation. Such a theory would hopefully allow designers to formally think about "software topology" in conjunction with the commonly understood notion of hardware topology; schedulers could then be thought of as "embeddings" of the software topology onto the hardware topology. A unified theory might also enable significant reuse of ideas for algorithm design and lower bounds across cost models.

Here are a few other possible directions for further exploration.

1. Space-bounded schedulers require the programmer to annotate every task with its size. While this is possible for many common algorithms, it may be tedious for some programs. One way to gain information about the size of tasks is to use hardware cache counters. Is it possible to use feedback from these counters to guide scheduling decisions without

requiring program annotations?

2. Are there problems for which it is not possible to design an algorithm that is both optimally parallel and cache-efficient?
3. Are there schedulers capable of load-balancing irregularities beyond what is characterized by effective cache complexity.
4. How accurate are program-centric models at characterizing locality and parallelism in non-nested parallel programs?
5. Writing algorithms with static allocation while retaining locality is a tedious task. Dynamic memory allocation is a much more elegant model for writing parallel program. However, we do not completely understand the design of an allocation scheme along with a scheduler that preserves locality at every level in the hierarchy while also retaining time and space bounds. Designing such an allocator would add weight to our program-centric models.
6. Is there a quantitative relation between the effective cache complexity $\widehat{Q}_\alpha$ of an algorithm and the AT^α complexity of its corresponding VLSI circuit?
7. The cost models and schedulers in this thesis have been constructed with shared memory machines in mind. Is it possible to adapt and generalize these ideas to distributed memory machines.

Appendix A

Statically Allocated QuickSort

A statically allocated quick sort algorithm writen in Cilk. The `map scan, filterLR` are all completely parallel and take pre-allocated memory chunks from `quickSort`. The total amount of space required for quickSort is about $2*n*sizeof(E)+3*n*sizeof(int)$ in addition to the $< n*sizeof(int)$ space required for stack variables. In this version stack frames are allocated automatically by the cilk runtime system.

```
void quickSort (E* A, int n, BinPred f,
                int* compared, int* less_pos, int* more_pos,
                E* B, void *stack_var_space, bool invert=0)
{
    if (n < QSORT_PAR_THRESHOLD) {
        seqQuickSort (A,n,f,B,invert);
    } else {
        E pivot = A[n/2];
        map<E,int,CompareWithPivot<E> >(A,compared,n,
                                CompareWithPivot<E>(pivot));

        PositionScanPlus lessScanPlus(LESS);
        PositionScanPlus moreScanPlus(MORE);

        less_pos_n = stack_var_space;
        more_pos_n = 1+stack_var_space;
        scan<int,PositionScanPlus>(less_pos_n,compared,less_pos,
                                   n,lessScanPlus,0);
        scan<int,PositionScanPlus>(more_pos_n,compared,more_pos,
                                   n,moreScanPlus,0);

        filterLR<E> (A,B,B+*less_pos_n,B+n-*more_pos_n,
                     less_pos,more_pos,compared,0,n);

        cilk_spawn
        quickSort<E,BinPred>(B,*less_pos_n,f,
                             compared,less_pos,more_pos,
                             A,stack_var_space+2,!invert);
        quickSort<E,BinPred>(B+*more_pos_n,*more_pos_n,f,
                             compared+*more_pos_n,
                             less_pos+*more_pos_n,
                             more_pos+*more_pos_n,
                             A+*more_pos_n,
                             stack_var_space+2
                           +2*(*more_pos_n)/QSORT_PAR_THRESHOLD,
                             !invert)
        cilk_sync;

        if (!invert)
            map<E,E,Id<E> >(B+*less_pos_n,A+*less_pos_n,
                            n-*less_pos_n-*more_pos_n,Id<E>());
    }
}
```

The top-level call with pre-allocated array `A` of length `LEN` is:

```
quickSort<E, std::less<E> > (A,LEN,std::less<E>(),
                            new(int,LEN+1),new(int,LEN+1),
                            new(int,LEN+1),new(E,LEN+1)
                            new(int,4*n/QSORT_PAR_THRESHOLD,0);
```

Appendix B

Code fragments from the experimental framework

```
template <class AT, class BT, class F>
class Map : public SBJob {
  AT* A; BT* B; int n; F f; int stage;
public:
  Map (AT *A_, BT *B_, int n_, F f_, int stage_=0,
       bool del=true)
    : SBJob (del),  A(A_), B(B_), n(n_), f(f_), stage(stage_) {}
  lluint size (const int block_size) {
    return round_up (n*sizeof(AT), block_size)
         + round_up (n*sizeof(BT), block_size);
  }
  lluint strand_size (const int block_size) {
      if (n < _SCAN_BSIZE && stage==0)
        return  size (block_size);
      else return STRAND_SIZE;  // A small constant ~100B
  }
  void function () {
    if (stage == 0) {
      if (n<_SCAN_BSIZE) {
        for (int i=0; i<n; ++i) B[i] = f(A[i]);  //Sequential map
        join ();
      } else {
        binary_fork(new Map<AT,BT,F>(A,B,n/2,f),          // Left half
                    new Map<AT,BT,F>(A+n/2,B+n/2,n-n/2,f), //Right half
                    new Map<AT,BT,F>(A,B,n,f,1));        //Continuation
      }
    } else {
      join ();
    }
  }
};
```

Figure B.1: The specification of the map operation in programming interface

```
void
WS_Scheduler::add (Job *job, int thread_id) {
  _local_lock[thread_id].lock();
  _job_queues[thread_id].push_back(job);
  _local_lock[thread_id].unlock();
}
int
WS_Scheduler::steal_choice (int thread_id) {
  return  (int) ((((double)rand())/((double)RAND_MAX))*_num_threads);
}
Job*
WS_Scheduler::get (int thread_id) {
  _local_lock[thread_id].lock();
  if (_job_queues[thread_id].size() > 0) {
    Job * ret = _job_queues[thread_id].back();
    _job_queues[thread_id].pop_back();
    _local_lock[thread_id].unlock();
    return ret;
  } else {
    _local_lock[thread_id].unlock();
    int choice = steal_choice(thread_id);
    _steal_lock[choice].lock();
    _local_lock[choice].lock();
    if (_job_queues[choice].size() > 0) {
      Job * ret = _job_queues[choice].front();
      _job_queues[choice].erase(_job_queues[choice].begin());
      ++_num_steals[thread_id];
      _local_lock[choice].unlock();
      _steal_lock[choice].unlock();
      return ret;
    }
    _local_lock[choice].unlock();
    _steal_lock[choice].unlock();
  }
  return NULL;
}
void
WS_Scheduler::done (Job *job, int thread_id, bool deactivate) {
  //Nothing to be done
}
```

Figure B.2: Work-stealing scheduler implemented in scheduler interface

```
/******  32 Core Teach ******/
int num_procs=32;
int num_levels = 4;
int fan_outs[4] = {4,8,1,1};
long long int sizes[4] = {0, 3*(1<<22), 1<<18, 1<<15};
int block_sizes[4] = {64,64,64,64};
int map[32] = {0,4,8,12,16,20,24,28,
               2,6,10,14,18,22,26,30,
               1,5,9,13,17,21,25,29,
               3,7,11,15,19,23,27,31};
```

Figure B.3: Specification entry for a 32-core Xeon machine sketched in 7.1(a)

Bibliography

[1] Harold Abelson and Peter Andreae. Information transfer and area-time tradeoffs for VLSI multiplication. *Communications of the ACM*, 23(1):20–23, January 1980. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/358808.358814`.

[2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[3] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1995.

[4] Sarita V Adve and Mark D Hill. Weak ordering – a new definition. *ACM SIGARCH Computer Architecture News*, 18(3a):2–14, 1990.

[5] Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *CoRR*, abs/1206.4377, 2012.

[6] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/48529.48535`.

[7] Alok Aggarwal, Bowen Alpern, Ashok Chandra, and Marc Snir. A model for hierarchical memory. In *Proceedings of the 19th annual ACM Symposium on Theory of Computing*, STOC '87, 1987. ISBN 0-89791-221-7.

[8] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 204–216, 1987.

[9] Alfred V. Aho, Jeffrey D. Ullman, and Mihalis Yannakakis. On notions of information transfer in VLSI circuits. In *Proceedings of the 15th annual ACM Symposium on Theory of Computing*, STOC '83, pages 133–139, New York, NY, USA, 1983. ACM. ISBN 0-89791-099-0. URL `http://doi.acm.org/10.1145/800061.808742`.

[10] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM. ISBN 0-89791-099-0. URL `http://doi.acm.org/10.1145/800061.808726`.

[11] Romas Aleliunas. Randomized parallel communication (preliminary version). In *Proceedings of the first ACM SIGACT-SIGOPS Symposium on Principles of distributed computing*, PODC '82, pages 60–72, New York, NY, USA, 1982. ACM. ISBN 0-89791-081-8. URL

http://doi.acm.org/10.1145/800220.806683.

[12] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71 – 79, 1997. ISSN 0743-7315. URL http://www.sciencedirect.com/science/article/pii/S0743731597913460.

[13] Bowen Alpern, Larry Carter, and Jeanne Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers*, 1993.

[14] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12, 1994.

[15] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–869, 1991.

[16] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[17] Lars Arge. External geometric data structures. In Kyung-Yong Chwa and J.IanJ. Munro, editors, *Computing and Combinatorics*, volume 3106 of *Lecture Notes in Computer Science*, pages 1–1. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22856-1. URL http://dx.doi.org/10.1007/978-3-540-27798-9_1.

[18] Lars Arge and Norbert Zeh. *Algorithms and Theory of Computation Handbook*, chapter External-memory algorithms and data structures, pages 10–10. Chapman & Hall/CRC, 2010. ISBN 978-1-58488-822-2. URL http://dl.acm.org/citation.cfm?id=1882757.1882767.

[19] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, STOC '02, 2002.

[20] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA '08: Proceedings of the 20th annual Symposium on Parallelism in algorithms and architectures*, 2008.

[21] Lars Arge, Michael T. Goodrich, and Nodari Sitchinava. Parallel external memory graph algorithms. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11, 2010.

[22] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures , Puerto Vallarta*, 1998.

[23] Grey Ballard. *Avoiding Communication in Dense Linear Algebra*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2013. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-151.html.

[24] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM*, 59(6):32:1–32:23, January 2013. ISSN 0004-5411. URL http://doi.acm.org/10.1145/

2395116.2395121.

[25] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the spring joint computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM. URL http://doi.acm.org/10.1145/1468075.1468121.

[26] Michael A. Bender and Martn Farach-Colton. The LCA problem revisited. In GastonH. Gonnet and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67306-4. URL http://dx.doi.org/10.1007/10719839_9.

[27] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *SPAA '05: Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005.

[28] Michael A. Bender, Gerth S. Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *SPAA '07: Proceedings of the 16th annual ACM Symposium on Parallelism in algorithms and architectures*, 2007.

[29] Michael A. Bender, Bradley C. Kuszmaul, Shang-Hua Teng, and Kebin Wang. Optimal cache-oblivious mesh layout. Computing Research Repository (CoRR) abs/0705.1033, 2007.

[30] Bryan T. Bennett and Vincent J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

[31] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993. URL http://epubs.siam.org/doi/abs/10.1137/0222017, arXiv:http://epubs.siam.org/doi/pdf/10.1137/0222017.

[32] Gianfranco Bilardi, Andrea Pietracaprina, Geppino Pucci, and Francesco Silvestri. Network-oblivious algorithms. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[33] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, 1992.

[34] Guy E. Blelloch. Problem based benchmark suite. www.cs.cmu.edu/~guyb/pbbs/Numbers.html, 2011.

[35] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proceedings of the sixteenth annual ACM Symposium on Parallelism in algorithms and architectures*. ACM, 2004. ISBN 1-58113-840-7.

[36] Guy E. Blelloch and Robert Harper. Cache and I/O efficent functional algorithms. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 39–50, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. URL http://doi.acm.org/10.1145/2429069.2429077.

[37] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[38] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA '08: Proceedings of the 19th annual ACM-SIAM Symposium on Discrete algorithms*, 2008.

[39] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. A cache-oblivious model for parallel memory hierarchies. Technical Report CMU-CS-10-154, Computer Science Department, Carnegie Mellon University, 2010.

[40] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low-depth cache oblivious algorithms. In *SPAA '10: Proceedings of the 22th annusl Symposium on Parallelism in Algorithms and Architectures*, pages 189–199. ACM, 2010.

[41] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Symposium on Parallelism in Algorithms and Architectures*, pages 355–366, 2011.

[42] Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan. Near linear-work parallel sdd solvers, low-diameter decomposition, and low-stretch subgraphs. In *Proceedings of the 23rd ACM Symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 13–22, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. URL `http://doi.acm.org/10.1145/1989493.1989496`.

[43] Guy E. Blelloch, Richard Peng, and Kanat Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *Proceedings of the 23rd ACM Symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 23–32, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. URL `http://doi.acm.org/10.1145/1989493.1989497`.

[44] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. *SIGPLAN Not.*, 47(8):181–192, February 2012. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/2370036.2145840`.

[45] Guy E. Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Parallel and i/o efficient set covering algorithms. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 82–90, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4. URL `http://doi.acm.org/10.1145/2312005.2312024`.

[46] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Program-centric cost models for locality. In *MSPC '13*, 2013.

[47] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the twenty-fifth annual ACM Symposium on Theory of Computing*, STOC '93, pages 362–371, New York, NY, USA, 1993. ACM. ISBN 0-89791-591-7. URL `http://doi.acm.org/10.1145/167088.167196`.

[48] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5), 1999.

[49] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *IPPS*, 1996.

[50] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, SPAA '96, pages 297–308, New York, NY, USA, 1996. ACM. ISBN 0-89791-809-6. URL `http://doi.acm.org/10.1145/237502.237574`.

[51] OpenMP Architecture Review Board. OpenMP application program interface. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008. version 3.0.

[52] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[53] Riachrd P. Brent and H. T. Kung. The Area-Time complexity of binary multiplication. *Journal of the ACM*, 28:521–534, 1981.

[54] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974. ISSN 0004-5411. URL `http://doi.acm.org/10.1145/321812.321815`.

[55] Gerth S. Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages, and Programming*, 2002.

[56] Gerth S. Brodal, Rolf Fagerberg, and G. Moruz. Cache-aware and cache-oblivious adaptive sorting. In *ICALP '05: Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming*, 2005.

[57] Gerth S. Brodal, Rolf Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics*, 12, 2008. Article No. 2.2.

[58] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the 35th annual ACM Symposium on Theory of Computing*, STOC '03, pages 307–315, New York, NY, USA, 2003. ACM. ISBN 1-58113-674-9. URL `http://doi.acm.org/10.1145/780542.780589`.

[59] Bernard Chazelle and Louis Monier. A model of computation for VLSI with related complexity results. *Journal of the ACM*, 32:573–588, 1985.

[60] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *SODA '95: Proceedings of the 6th annual ACM-SIAM Symposium on Discrete algorithms*, 1995. ISBN 0-89871-349-8.

[61] Francis Y. Chin, John Lam, and I-Ngo Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, September 1982. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/358628.358650`.

[62] Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *SPAA '07: Proceedings of the 19th annual ACM Symposium on Parallel algorithms and*

architectures, 2007. ISBN 978-1-59593-667-7.

[63] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *SPAA '08: Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, 2008. ISBN 978-1-59593-973-9.

[64] Rezaul Alam Chowdhury, Francesco Silvestri, Brandon Blakeley, and Vijaya Ramachandran. Oblivious algorithms for multicores and network of processors. In *IPDPS '10: Proceedings of the IEEE 24th International Parallel and Distributed Processing Symposium*, pages 1–12, 2010.

[65] R Cole and U Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *STOC '86: Proceedings of the 18th annual ACM Symposium on Theory of Computing*, 1986. ISBN 0-89791-193-8.

[66] Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. In *Proceedings of the 37th international colloquium Conference on Automata, languages and programming*, ICALP'10, pages 226–237, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14164-1, 978-3-642-14164-5. URL `http://dl.acm.org/citation.cfm?id=1880918.1880944`.

[67] Richard Cole and Vijaya Ramachandran. Efficient resource oblivious scheduling of multicore algorithms. manuscript, 2010.

[68] Richard Cole and Vijaya Ramachandran. Efficient resource oblivious algorithms for multicores. *Arxiv preprint arXiv.11034071*, 2011. URL `http://arxiv.org/abs/1103.4071`.

[69] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms, 2nd Edition.* MIT Press, 2001.

[70] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM. ISBN 0-89791-589-5. URL `http://doi.acm.org/10.1145/155332.155333`.

[71] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 137–150, 2004. URL `http://www.usenix.org/events/osdi04/tech/dean.html`.

[72] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, 2002.

[73] Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming*, volume 5555 of *Lecture Notes in Computer Science*, pages 341–353. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02926-4. URL `http://dx.doi.org/10.1007/978-3-642-02927-1_29`.

[74] James Demmel and Oded Schwartz. Communication avoiding algorithms – course website, Fall 2011. URL `http://www.cs.berkeley.edu/~odedsc/CS294/`.

[75] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *ArXiv e-prints*, August 2008, `arXiv:0808.2664`.

[76] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/363095.363141`.

[77] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, January 2004. ISSN 0743-7315. URL `http://dx.doi.org/10.1016/j.jpdc.2003.09.005`.

[78] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 245–257, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. URL `http://doi.acm.org/10.1145/781131.781159`.

[79] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on*, 38(3):408–423, 1989.

[80] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 83. ACM, 2006.

[81] G. Franceschini. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[82] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496–507, July 1970. ISSN 0004-5411. URL `http://doi.acm.org/10.1145/321592.321600`.

[83] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 240–249, New York, NY, USA, 1998. ACM. ISBN 0-89791-989-0. URL `http://doi.acm.org/10.1145/277651.277690`.

[84] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA '06: Proceedings of the 18th annual ACM Symposium on Parallelism in algorithms and architectures*, 2006. ISBN 1-59593-452-9.

[85] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[86] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, 1999.

[87] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(3a):15–26, May 1990. ISSN 0163-5964. URL http://doi.acm.org/10.1145/325096.325102.

[88] R. L. Graham. Bounds for certain multiprocessing anomalies pdf. *The Bell System Technical Journal*, 45, November 1966.

[89] Xiaoming Gu, Ian Christopher, Tongxin Bai, Chengliang Zhang, and Chen Ding. A component model of spatial locality. In *Proceedings of the 2009 International Symposium on Memory management*, ISMM '09, pages 99–108, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-347-1. URL http://doi.acm.org/10.1145/1542431.1542446.

[90] Yi Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS '09: IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, 2009.

[91] Avinatan Hassidim. Cache replacement policies for multicore processors. In *ICS*, pages 501–509, 2010.

[92] F. C. Hennie. One-tape, off-line turing machine computations. *Information and Control*, 8(6):553–578, Dec 1965.

[93] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape turing machines. *Journal of the ACM*, 13(4):533–546, October 1966. ISSN 0004-5411. URL http://doi.acm.org/10.1145/321356.321362.

[94] Jia-Wei Hong. On similarity and duality of computation (i). *Information and Control*, 62 (23):109 – 128, 1984. ISSN 0019-9958. URL http://www.sciencedirect.com/science/article/pii/S0019995884800303.

[95] John Hopcroft, Wolfgang Paul, and Leslie Valiant. On time versus space. *Journal of the ACM*, 24(2):332–337, April 1977. ISSN 0004-5411. URL http://doi.acm.org/10.1145/322003.322015.

[96] Intel. Intel Cilk++ SDK programmer's guide. https://www.clear.rice.edu/comp422/resources/Intel_Cilk++_Programmers_Guide.pdf, 2009.

[97] Intel. Intel Thread Building Blocks reference manual. http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm#reference/reference.htm, 2013. Version 4.1.

[98] Intel. Performance counter monitor (PCM). http://www.intel.com/software/pcm, 2013. Version 2.4.

[99] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, September 2004. ISSN 0743-7315. URL http://dx.doi.

org/10.1016/j.jpdc.2004.03.021.

[100] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, 38(3):60–71, June 2010. ISSN 0163-5964. URL http://doi.acm.org/10.1145/1816038.1815971.

[101] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. Cruise: cache replacement and utility-aware scheduling. In *Proceedings of the seventeenth international Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 249–260, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. URL http://doi.acm.org/10.1145/2150976.2151003.

[102] Hong Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the 13th annual ACM Symposium on Theory of Computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM. URL http://doi.acm.org/10.1145/800076.802486.

[103] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1996. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-701.

[104] T. S. Jung. Memory technology and solutions roadmap. http://www.sec.co.kr/images/corp/ir/irevent/techforum_01.pdf, 2005.

[105] Anna R. Karlin and Eli Upfal. Parallel hashing: an efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, 1988.

[106] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics. ISBN 978-0-898716-98-6. URL http://dl.acm.org/citation.cfm?id=1873601.1873677.

[107] Anil Kumar Katti and Vijaya Ramachandran. Competitive cache replacement strategies for shared cache environments. In *IPDPS*, pages 215–226, 2012.

[108] Ronald Kriemann. Implementation and usage of a thread pool based on posix threads. http://www.hlnum.org/english/projects/tools/threadpool/doc.html.

[109] Piyush Kumar. Cache oblivious algorithms. In Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors, *Algorithms for Memory Hierarchies*. Springer, 2003.

[110] H. T. Kung. Let's design algorithms for VLSI systems. In *Proceedings of the Caltech Conference On Very Large Scale Integration*, pages 65–90. California Institute of Technology, 1979.

[111] Hsiang Tsung Kung and Charles E Leiserson. Algorithms for VLSI processor arrays. *Introduction to VLSI systems*, pages 271–292, 1980.

[112] Edya Ladan-Mozes and Charles E. Leiserson. A consistency architecture for hierarchi-

cal shared caches. In *SPAA '08: Proceedings of the 20th ACM Symposium on Parallel Algorithms and Architectures*, pages 11–22, Munich, Germany, June 2008.

[113] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46, 1997.

[114] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-latency dram: A low latency and low cost dram architecture. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 0: 615–626, 2013. ISSN 1530-0897.

[115] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 411–420, Vienna, Austria, September 2010. ACM.

[116] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. ISBN 1-55860-117-1.

[117] Frank T. Leighton, Bruce M. Maggs, Abhiram G. Ranade, and Satish B. Rao. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17(1):157 – 205, 1994. URL `http://www.sciencedirect.com/science/article/pii/S0196677484710303`.

[118] Frank T. Leighton, Bruce M. Maggs, and Satish B. Rao. Packet routing and job-shop scheduling ino(congestion+dilation) steps. *Combinatorica*, 14(2):167–186, 1994. ISSN 0209-9683. URL `http://dx.doi.org/10.1007/BF01215349`.

[119] Tom Leighton. Tight bounds on the complexity of parallel sorting. In *Proceedings of the 16th annual ACM Symposium on Theory of Computing*, STOC '84, pages 71–80, New York, NY, USA, 1984. ACM. ISBN 0-89791-133-4. URL `http://doi.acm.org/10.1145/800057.808667`.

[120] Charles Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51, 2010.

[121] Charles E. Leiserson. Fat-Trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C–34(10), 1985.

[122] Charles E. Leiserson and Bruce M. Maggs. Communication-efficient parallel graph algorithms for distributed random-access machines. *Algorithmica*, 3(1):53–77, 1988.

[123] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. URL `http://doi.acm.org/10.1145/1555754.1555789`.

[124] Richard J. Lipton and Robert Sedgewick. Lower bounds for VLSI. In *Proceedings of the 13th annual ACM Symposium on Theory of Computing*, STOC '81, pages 300–307, New York, NY, USA, 1981. ACM. URL `http://doi.acm.org/10.1145/800076.`

802482.

[125] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36, 1979.

[126] Victor Luchangco. Precedence-based memory models. In Marios Mavronicolas and Philippas Tsigas, editors, *Distributed Algorithms*, volume 1320 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63575-8. URL `http://dx.doi.org/10.1007/BFb0030686`.

[127] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. *SIGMETRICS Performance Evaluation Review*, 32(1):2–13, June 2004. ISSN 0163-5999. URL `http://doi.acm.org/10.1145/1012888.1005691`.

[128] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[129] C.A. Mead and M. Rem. Cost and performance of VLSI computing structures. *IEEE Transactions on Electron Devices*, 26(4):533–540, 1979.

[130] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979. ISBN 0201043580.

[131] Microsoft. Task Parallel Library. `http://msdn.microsoft.com/en-us/library/dd460717.aspx`, 2013. .NET version 4.5.

[132] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. In Silvio Micali, editor, *Randomness and Computation*, pages 47–72. JAI Press, Greenwich, Connecticut, 1989. Vol. 5.

[133] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 261 –270, sept. 2009.

[134] Girija J. Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1999.

[135] Girija J. Narlikar. *Space-Efficient Scheduling for Parallel, Multithreaded Computations*. PhD thesis, Carnegie Mellon University, May 1999. Available as CMU-CS-99-119.

[136] David A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10): 71–75, October 2004. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/1022594.1022596`.

[137] Perfmon2. libpfm. `http://perfmon2.sourceforge.net/`, 2012.

[138] N. Pippenger. Parallel communication with limited buffers. In *Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984*, SFCS '84, pages 127–136, Washington, DC, USA, 1984. IEEE Computer Society. ISBN 0-8186-0591-X. URL `http://dx.doi.org/10.1109/SFCS.1984.715909`.

[139] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th annual International Conference on Supercomputing*, ICS '06, pages 249–258, New York, NY, USA, 2006. ACM. ISBN 1-59593-282-8. URL `http://doi.acm.org/10.1145/1183401.1183437`.

[140] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Proceedings of the 16th international Euro-Par Conference on Parallel processing: Part I*, EuroPar'10, pages 217–229, Berlin, Heidelberg, 2010. Springer-Verlag.

[141] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989. ISSN 0097-5397.

[142] Abhiram G. Ranade. How to emulate shared memory. *Journal of Computing and Systems Sciences*, 42(3):307–326, 1991.

[143] Arnold L. Rosenberg. The VLSI revolution in theoretical circles. In Jan Paredaens, editor, *Automata, Languages and Programming*, volume 172 of *Lecture Notes in Computer Science*, pages 23–40. Springer Berlin Heidelberg, 1984. ISBN 978-3-540-13345-2. URL `http://dx.doi.org/10.1007/3-540-13345-3_2`.

[144] Samsung. DRAM Data Sheet. `http://www.samsung.com/global/business/semiconductor/prod\uct`.

[145] John E. Savage. Areatime tradeoffs for matrix multiplication and related problems in VLSI models. *Journal of Computer and System Sciences*, 22(2):230 – 242, 1981. ISSN 0022-0000. URL `http://www.sciencedirect.com/science/article/pii/0022000081900295`.

[146] John E. Savage. Extending the hong-kung model to memory hierarchies. In *Proceedings of the First Annual International Conference on Computing and Combinatorics*, COCOON '95, pages 270–281, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-60216-X. URL `http://dl.acm.org/citation.cfm?id=646714.701412`.

[147] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997. ISBN 0201895390.

[148] Scandal. Irregular algorithms in NESL language. `http://www.cs.cmu.edu/~scandal/alg/algs.html`, July 1994.

[149] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedinbgs of the 24th ACM Symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4.

[150] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2), 1985.

[151] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. Technical Report UCB/EECS-2011-72, EECS Department, University of California, Berkeley, Jun 2011. URL `http://www.eecs.`

berkeley.edu/Pubs/TechRpts/2011/EECS-2011-72.html.

[152] Daniel Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, May 2009. Available as CMU-CS-09-126.

[153] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *SPAA '09: Proceedings of the 21st annual Symposium on Parallelism in Algorithms and Architectures*, pages 91–100, 2009.

[154] Kanat Tangwongsan. *Efficient Parallel Approximation Algorithms*. PhD thesis, Carnegie Mellon University, 2011.

[155] Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal mapreduce algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 529–540, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. URL http://doi.acm.org/10.1145/2463676.2463719.

[156] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science, 1984*, SFCS '84, pages 12–20, Washington, DC, USA, 1984. IEEE Computer Society. ISBN 0-8186-0591-X. URL http://dx.doi.org/10.1109/SFCS.1984.715896.

[157] C. D. Thompson. Area-time complexity for VLSI. In *Proceedings of the 11th annual ACM Symposium on Theory of computing*, STOC '79, pages 81–88, New York, NY, USA, 1979. ACM. URL http://doi.acm.org/10.1145/800135.804401.

[158] C.D. Thompson. The VLSI complexity of sorting. *IEEE Transactions on Computers*, 32(12):1171–1184, 1983. ISSN 0018-9340. URL http://doi.ieeecomputersociety.org/10.1109/TC.1983.1676178.

[159] C.D. Thompson. Fourier transforms in VLSI. *IEEE Transactions on Computers*, 32(11): 1047–1057, 1983. ISSN 0018-9340. URL http://doi.ieeecomputersociety.org/10.1109/TC.1983.1676155.

[160] Clark Thompson. *A Complexity Theory of VLSI*. PhD thesis, Carnegie Mellon University, 1980. available as CMU-CS-80-140.

[161] Pilar Torre and ClydeP. Kruskal. Submachine locality in the bulk synchronous setting. In Luc Boug, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 352–358. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61627-6. URL http://dx.doi.org/10.1007/BFb0024723.

[162] Eli Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31(3):507–517, June 1984. ISSN 0004-5411. URL http://doi.acm.org/10.1145/828.1892.

[163] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990. ISSN 0001-0782.

[164] Leslie G. Valiant. A bridging model for multi-core computing. In *ESA '08: Proceedings*

of the 16th European Symposium on Algorithms, 2008.

[165] Leslie G. Valiant and Gordon J. Brebner. Universal schemes for parallel communication. In *Proceedings of the thirteenth annual ACM Symposium on Theory of Computing*, STOC '81, pages 263–277, New York, NY, USA, 1981. ACM. URL `http://doi.acm.org/10.1145/800076.802479`.

[166] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2), 2001. ISSN 0360-0300.

[167] J. Vuillemin. A combinatorial limit to the computing power of VLSI circuits. *IEEE Transactions on Computers*, 32(3):294–300, 1983. ISSN 0018-9340. URL `http://doi.ieeecomputersociety.org/10.1109/TC.1983.1676221`.

[168] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1 edition, June 2009. ISBN 0596521979.

[169] Andrew W Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th annual International symposium on Computer architecture*, pages 244–252. ACM, 1987.

[170] J. C. Wyllie. *The Complexity of Parallel Computation*. PhD thesis, Cornell University, 1979.

[171] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *Proceedings of the eighteenth International Conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 343–356, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. URL `http://doi.acm.org/10.1145/2451116.2451153`.

[172] Andrew C. Yao. The entropic limitations on VLSI computations(extended abstract). In *Proceedings of the 13th annual ACM Symposium on Theory of Computing*, STOC '81, pages 308–311, New York, NY, USA, 1981. ACM. URL `http://doi.acm.org/10.1145/800076.802483`.

[173] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing(preliminary report). In *Proceedings of the 11th annual ACM Symposium on Theory of Computing*, STOC '79, pages 209–213, New York, NY, USA, 1979. ACM. URL `http://doi.acm.org/10.1145/800135.804414`.

[174] Norbert Zeh. I/O-efficient graph algorithms. `http://web.cs.dal.ca/~nzeh/Publications/summer_school_2002.pdf`, 2002.

[175] Yutao Zhong, Steven G. Dropsho, Xipeng Shen, Ahren Studer, and Chen Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, 56(3):328–343, March 2007. ISSN 0018-9340.