Write-efficient Algorithms

Yan Gu

CMU-CS-18-113

September, 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch, Chair
Phillip B. Gibbons
Anupam Gupta
Jeremy T. Fineman (Georgetown University)
Julian Shun (MIT)

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

This research was sponsored by Intel and the National Science Foundation under grant numbers CCF-1018188, CCF-1314590, and CCF-1533858. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Write-Efficient Algorithms, Non-Volatile Memory, Asymmetric Read and Write Cost, Computational Model, Parallel Algorithms

To my family.

Abstract

New non-volatile memory (NVM) technologies are projected to become the dominant type of main memory in the near future. They promise byte-addressability, good read latencies, and significantly lower energy and higher density compared to DRAM. However, a key property of NVMs is the asymmetric read and write cost: write operations are much more expensive than reads regarding energy, bandwidth, and latency. This property contradicts fifty years of classic algorithms research that has focused on the setting in which reads and writes have similar cost, and poses the need to develop *write-efficient algorithms* that use fewer writes than the classic approaches.

This thesis provides a comprehensive study of the design and analysis of write-efficient algorithms, which includes computational models, lower bounds, algorithms, and experimental validations. More specifically, this thesis first presents and studies several models that account for read-write asymmetry in different settings (sequential, parallel, I/O models, etc.). Then, a number of lower bounds are shown, which indicate the hardness of asymptotically improving some classic algorithms under certain assumptions.

The main contribution of this thesis consists of new write-efficient algorithms for fundamental algorithms in Computer Science, from basic algorithmic building blocks (sorting, reduce, filter, etc.), to graph algorithms ((bi)connectivity, shortest paths, MST, BFS, etc.), geometric algorithms and data structures (convex hull, Delaunay triangulation, k-d trees, augmented trees, etc.), as well as many cache-oblivious algorithms for dynamic programming and linear algebra problems. The numbers of writes in all algorithms studied in this thesis are significantly reduced asymptotically. Furthermore, most of these algorithms are also highly parallel. Many techniques used to obtain these results are of independent interest, since they are applicable to many other problems outside those studied in this thesis, and lead to improved algorithms in the classic setting without read-write asymmetry.

This thesis also proposes the first experimental framework to analyze the performance of write-efficient algorithms in practice, and conducts experiments on a variety of algorithms. The experimental results show the effectiveness of write-efficient algorithms, and suggest that the algorithms developed in this thesis may be of both theoretical and practical interest.

Acknowledgement

It is surprising that I had not known my advisor Guy Blelloch before accepting the offer to become a graduate student at CMU. However, when I first met him during his IC class where he was talking about his parallel MIS algorithm, I discovered the kind of research I had sought for years. I consider myself lucky to be Guy's student without a related research background, despite the false perception of his unpopularity and lack of attraction for other students.

Jokes apart, I owe my deepest gratitude to Guy, the best advisor that one could imagine. He has always been a great mentor and an extremely patient teacher. I especially thank him for letting me have the freedom of exploring my true research interests over the course of these six years, and supporting me no matter what I was working on, from computer graphics in the first years, several random topics in the middle, and eventually this thesis. I could not have asked for a better advisor.

I would also like to thank the rest of my thesis committee members. Other than helping me throughout my 6-year graduate studies and providing me with useful feedback to improve this thesis, they have all influenced me significantly with regard to both research and career paths.

Phil Gibbons is an encyclopedia of computer science. He can always propose insightful thoughts and long-term pictures of research work, and novel ideas for research challenges. He provided me thorough explanations of my trivial and non-trivial questions. I have learned a lot from Phil during our weekly meetings, and without him I would not have this thesis.

I have taken, audited and helped with (as a TA) many of Anupam Gupta's courses. I was impressed by his expertise in mathematics and thorough understanding of algorithms, and thereby realizing how attractive and sexy an algorithm researcher can be. He is one of the key factors for my switch to algorithm design as my research topic.

Jeremy Fineman is an encyclopedia of algorithms. In our discussions, he seems to understand the details of all the algorithms that were proposed in the last decade or back in the 1980s. His erudite knowledge in algorithms and quick reaction helped in developing many of the results in this thesis. From Jeremy and my other committee members, I comprehended that I needed to put in a lifelong effort to become a great researcher like them in the future.

Unlike other committee members, Julian Shun had partial overlap with me as a graduate student at CMU. I am fortunate to have collaborated with Julian on about half of my papers at CMU, and I have learned so many useful skills from him, such as conducting research, background study, paper writing, and even the spirit of diligence. The only pity I nurse is that I could have collaborated with and learned from him earlier, instead of my third year as he was graduating.

Research is a team effort, and I was fortunate to have my collaborators and our research group. I would like to thank my other co-authors, Naama Ben-David, Kayvon Fatahalian, Yong He, Charles McGuffey, Yihan Sun, and Kanat Tangwongsan for contributing their ideas and effort in all the papers I have worked on at CMU. I am thankful to Umut Acar, Laxman Dhulipala, Yuanhao Wei, Sam Westrick, Shen Chen Xu and Yu Zhao for the valuable discussions and

feedback of my research work and talks, and I am looking forward to the chance to collaborate with all of them. I specially thank Charles and Laxman for their help on the writing of this thesis. I also thank all those who were previously mentioned and my other PhD friends at CMU. Although we work on different areas, our frequent chats help me in figuring the entire computer science world more (and gossip), which is very beneficial for me.

During my graduate studies, I was lucky to have had the opportunity to interact with many other faculty members at CMU. I would like to give my special thanks to Kayvon Fatahalian and Danny Sleator, who granted me lots of assistance with regard to both school work and daily issues during the first several years when I was still a helpless foreigner. I had the opportunity to work as a teaching assistant for Anupam, Danny, Guy, and Kayvon, and from them I learned how to become more effective at teaching. I also thank Kayvon, Danny, Phil, and Bob Harper for helping me with my speaking and writing skills requirements, and Kayvon and Guy for trusting me to give several lectures in their classes.

I spent my first two years at CMU as a member of the Graphics lab. I thank the faculty members Kayvon, Alexei Efros, Jessica Hodgins, Nancy Pollard, Yaser Sheikh, Adrien Treuille, and the students for their help on my research and presentation skills. I would like to give my special thanks to Nancy. She was the first professor I met at CMU as an undergrad. She also kindly provided me a considerable amount of assistance on my application, helped me settle down in Pittsburgh, and conducted mock interview for my practice. I have nothing to offer in return other than my gratitude.

My journey at CMU would be a dull one without my friends. Our team Stuck in Mitosis participated in many intramural sports for years, and I also enjoyed the table tennis club in our department. As a Chinese, the Chinese cuisine parties were always a kind of expectancy for my busy PhD life. I thank Yu for his once-a-semester hotpot parties and many other hosts in Pittsburgh, and my friends regularly coming to my house to enjoy our cooking, including Fan Yang, Fei Fang, Haoxian Chen, Linxi Zou, Junxing Wang, Kairui Zhang, Nan Lin, Wenting Shao, Shen-Chen Xu, Yan Wang, Yanzhe Yang, Yining Wang, Yixin Luo, Yong He, Yu Zhao, Zhanpeng Fang, and Zhou Su. Our annual travel group comprising Yihan's high school classmates Fan Gao, Lu Zheng, Tao Yu, Wanjian Tang, and Yanzhe Yin explored many places of interests in the US, which is an excellent memory of my PhD journey.

I would like to express my earnest gratitude to my parents. Naming me Yan which means research, they cultivated my interests in science and exploring this world. It would have been impossible for me to be at the position without their inexorable love, support, and encouragement.

Finally, I am indebted to the support of my beloved wife Yihan Sun. Her unconditional and constant love, patience, support and encouragement made my PhD career extremely enjoyable. Yihan is also a wonderful co-author, a 24/7 collaborator with whom I can discuss my research ideas and progress, a great audience who consistently provided valuable feedbacks, and a mentor who forces me to think about long-term scenarios regarding both research and life. I cannot imagine how my life would be like without her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Striving for the efficient performance of algorithms, both in theory and in practice, is one of the ultimate goals in computer science. Since people started writing computer programs, the cost of memory accesses has always been one of the major determining factors of the efficiency of an algorithm (the other factor being the number of algorithmic operations). Optimizing the memory access costs in practice, however, is highly dependent on the storage medium. For example, B-trees [38] and their variants with larger branching factors were widely used when the storage medium was magnetic tapes in the early years of computing, and hard disks later, because of the expensive random-access cost on these devices and the mechanism of the data storage incorporated into the computer system. On the other hand, if the data are stored and organized on a DRAM, which supports relatively cheap-random access cost, then usually a B-tree with a small constant branching factor, or even a binary tree is preferred for better practical performance.

To formally conduct a scientific study regarding the efficiency of an algorithm, mathematical tools are required for capturing the performance measurement in various settings. Computer scientists use different computational models to measure the costs of algorithms, show lower and upper bounds on the algorithmic costs of problems, and engineer implementations of efficient algorithms based on the models. In addition to basic models like the random-access machine (RAM) model, many models have also been proposed to optimize memory-access cost (e.g., the external-memory or I/O model [7] and the ideal-cache model [131]), based on which a great number of practically efficient algorithms have been designed and engineered.

Almost all previous models and research have focused on settings in which reads and writes (to the memory) have similar (symmetric) cost. This symmetric assumption simplifies the model, and roughly fits the current architecture. However, is the assumption always valid? Or, what if reads and writes to memory have significantly ***different*** (asymmetric) costs? This challenge of asymmetric read and write costs has been posed recently due to the arrival of non-volatile memory (NVM) technologies that are projected to

become the dominant type of main memory in the near future [171, 177, 210, 283]. These memories offer the promise of significantly lower energy and higher density (bits per area) than DRAM, with byte-addressability and read latencies approaching or improving on DRAM speeds. Despite these useful properties, one characteristic of these new memory technologies is that reading from memory is significantly cheaper than writing to it, with regards to latency, memory bandwidth, and especially energy consumption. Roughly speaking, the reason for this asymmetry is that writing to memory requires a change to the state of the material, while reading only requires detecting the current state. Based on the information currently available to us, the cost ratio between writes and reads is between 5 to 40 for the next-generation memories. This asymmetry poses the need to develop ***write-efficient algorithms*** that use significantly fewer writes than their classic counterparts.

The sequential read/write cost happens to be roughly symmetric for DRAMs and hard disks, but this symmetry is not intrinsic and does not hold for many other cases (e.g., solid-state disks, or concurrent accesses to shared memory). As a result, it is fundamental to understand such asymmetry in algorithm design. Studying the read-write asymmetry is not only motivated by the new hardware technologies, but also out of intellectual curiosity. More motivation for studying read-write asymmetry is discussed in Section 1.1.

The motivations and challenges of asymmetric read and write costs inevitably lead to the following new questions in algorithm design, which are crucial for understanding both the theory and practice in the asymmetric setting.

- How should existing computational models be modified to account for asymmetry between reads and writes, and how should such new memory be modeled?
- How does this asymmetry impact the design of algorithms?
- What new techniques are useful for trading off expensive operations (fewer writes) for cheap operations (more reads)?
- What are the fundamental limitations on such trade-offs (lower bounds)?
- Can write-efficient algorithms also be parallel?
- Do new algorithms actually reduce the number of writes in practice, and if so, by how much?

The aim of this thesis is to provide a comprehensive study of the read-write asymmetry in response to these questions.

The first contribution of this thesis is to introduce several computation and cost models to address read-write asymmetry. These models represent the asymmetry with a parameter ω, indicating the cost of a write relative to a read. They consider various settings including sequential vs. parallel, explicit vs. implicit memory management, etc. System-level support for these models has also been discussed in this thesis (e.g., asymmetry-aware cache-

replacement policies and scheduling parallel algorithms in asymmetric memory). More detailed results about these models are provided in Section 1.2 and Chapter 2.

Based on the models, the second contribution of this thesis is lower bounds for many fundamental problems (e.g., sorting, fast Fourier transform, and many dynamic programming and linear algebra problems), indicating the (asymptotical) required number of writes under certain assumptions. These results are summarized in Section 1.3, and discussed in detail in Chapter 3.

The third and main contribution of this thesis is a wide range of write-efficient algorithms and data structures for many fundamental problems, including sorting, graph processing, computational geometry, dynamic programming, linear algebra, etc. Each algorithm performs (asymptotically) fewer writes to memory than the best algorithm classic algorithm for the problem. Furthermore, most of these algorithms are highly parallel. This is because new NVMs provides a much larger (terabyte-level) capacity than the systems they replace, so good parallelism is necessary to process data with such sizes in a timely manner. The algorithms are introduced in categories in Chapters 4-7. A high-level overview of the results is listed here, and a more detailed summary is provided in Section 1.4.

- Algorithmic building blocks (Chapter 4) include reduce, filter, list/tree contraction, sorting, matrix multiplications, etc. These algorithms are fundamental and widely used in designing other algorithms in the later chapters of this thesis.
- Graph algorithms (Chapter 5) are further categorized into connectivity algorithms (connectivity and biconnectivity) and distance-based algorithms (single-source shortest-paths, minimum spanning trees, breadth-first search, etc.). For the connectivity problems, this thesis proposes a new graph partitioning methodology as a subroutine, referred to as the *implicit decomposition* of a graph. This decomposition is the first approach that can preserve the (bi)connectivity information of a graph with a sub-linear size (thus requires fewer writes to generate). (Bi)connectivity queries can be answered from this representation with a small extra cost. This thesis also discusses sequential or parallel distance-based algorithms for single-source shortest-paths, minimum spanning tree, and breadth-first search.
- Geometry algorithms (Chapter 6) include convex hull, planar Delaunay triangulation, low-dimensional LP-style algorithms, etc., as well as algorithms for data structures including k-d trees and augmented trees (e.g., interval trees, priority search trees). Here, write optimality indicates that the number of writes the algorithm or data structure construction performs asymptotically equals the output size. The algorithmic technique in many of these algorithms is to use randomized incremental algorithms to achieve write-efficiency. Unfortunately, theoretically-efficient parallel algorithms for many problems were unknown, even in the symmetric setting. To solve this problem, this thesis first describes a generic framework to analyze the parallelism of the incremental algorithms, and shows new algorithms

with improved parallelism. For example, this thesis shows the first work-efficient polylogarithmic-depth algorithms for planar Delaunay triangulation, which has been open for 25 years. Then the write optimality of many algorithms in this framework can be achieved using another generic approach. This framework is used to generate many write-optimal parallel geometric algorithms. Another general technique in these algorithms is the α-labeling with reconstruction-based updates for augmented trees, which trades off extra reads during queries for fewer writes during updates.

- Cache-oblivious algorithms for dynamic programming include LWS/GAP/RNA/-Parenthesis and other recurrences, and linear algebra problems include matrix multiplication, Gaussian elimination, and triangular system solver (Chapter 7). Classic solutions to these problems are based on divide-and-conquer schemes, and use asymptotically the same numbers of reads and writes to the main memory. Meanwhile, given the varied combinations of computations and data dependencies, designing individual algorithms for each specific problem can take significant effort. To overcome these challenges, this thesis proposes a level of abstraction for such problems, which is referred to as the k-d grid computation structure. By analyzing the lower and upper bounds of the cost to compute such grids, not only can the write-efficiency (write-optimality) of these algorithms be achieved under certain assumptions, but the sequential cost and parallelism of many problems in the symmetric setting can also be improved.

The last contribution of this thesis is the first experimental framework to evaluate and analyze the performance of write-efficient algorithms in practice. This framework is simple, clean and hardware-independent. Within the framework, a variety of different algorithms and data structures and their write-efficient implementations are discussed in this thesis. Many of them are non-trivial and require careful algorithmic design, analysis, and coding. Under the new asymmetric cost measure, this thesis proposes better implementations on all problems that were evaluated, compared to their traditional and commonly-used counterparts in the symmetric setting. This thesis summarizes many interesting algorithmic techniques, which can be useful in designing and engineering write-efficient algorithms in the future. More results are summarized in Section 1.5 and the full experimental evaluation is presented in Chapter 8.

1.1 Motivations for Write-Efficient Algorithms

This section discusses the motivations for write-efficient algorithms in greater depth, and argues that there is a timely need and importance of studying them. The motivations are presented below in three categories.

1.1.1 Read-Write Asymmetry in Emerging Non-Volatile Memories

Emerging non-volatile/persistent memory (NVM) technologies offer the promise of significantly lower energy and higher density (bits per area) than DRAM. With byte-addressability and read latencies approaching or improving on DRAM speeds, these NVM technologies are projected to become the dominant memory within the decade [171, 177, 210, 283], as manufacturing improves and costs decrease.

Despite the advantages of the new memory as compared to DRAM, there is an important distinction: writes are significantly more costly than reads, suffering from higher latency, lower per-chip bandwidth, higher energy costs, and endurance problems (a cell wears out after 10^8–10^{12} writes [210]). Thus, unlike DRAM, there is a significant (often an order of magnitude or more) asymmetry between read and write costs [14, 32, 117, 118, 176, 187, 234, 280], for which more technical details are provided in Appendix A. Motivated by these techniques, the study of *write-efficient* algorithms, which reduce the number of writes relative to existing algorithms, is of significant and lasting importance.

Read-write asymmetry has been the focus of many system efforts [91, 196, 281, 284]. Reducing the number of writes has long been a goal in disk arrays, distributed systems, cache-coherent multiprocessors, and the like. However, these works do not focus on NVMs and the solutions are not suitable for the properties of the new NVMs. Several papers [41, 123, 133, 220, 226, 273] have looked at read-write asymmetries in the context of flash memory. However, due to the different physical properties between main memory and flash memory (large block vs. byte addressability), these results cannot directly be applied to designing faster algorithms for new NVMs. Some prior work [90, 272, 273] has also looked at algorithms for asymmetric read-write costs in emerging NVMs, in the context of databases. However, these papers are focused on the empirical performance of specific algorithms. The new results in this thesis extend far beyond these papers, and lay the foundation for studying write-efficient algorithms both in theory and practice. In particular, this thesis provides a systematic and comprehensive study of models, lower bounds, dozens of new write-efficient algorithms, and runtime systems considering the asymmetric read-write costs. Furthermore, this thesis also conducts thorough experiments on new NVMs.

The work of Carson et al. [84] also considers asymmetric costs in reads and writes, and presents upper and lower bounds for various linear algebra problems and direct N-body methods under a similar model with read-write asymmetry. However, their focus is restricted to the class of "communication-avoiding" algorithms, i.e., parallel algorithms that minimize the (unweighted) sum of reads and writes, instead of the overall cost. Their results are more useful in the distributed or external memory setting, while this thesis focuses on sequential and shared-memory parallel algorithms. Further discussion is provided in Section 7.2.

1.1.2 Persistency and Other System-Level Considerations

The previous section introduced the hardware-level asymmetry between read and write costs. Another major property of these new main memories are their non-volatility, or persistency: unlike DRAM, they have the capability of surviving power outages and other failures without losing data. As a result, it is possible to design programming models and algorithms that are resilient to either processor faults or power outages. Persistence can be useful since in current and upcoming large parallel systems, the probability that an individual processor faults is not negligible [83].

To achieve fault-tolerant programming, one has to guarantee that at some certain stages in the execution of the program, the intermediate data stored in the persistent main memory are in some consistent states, such that either the computation of a single processor or the entire program can be restarted from these states. This step is achieved by either marking check pointers or encapsulating updates (via transactions, various atomic sections, etc.). On the other hand, standard caches are write-back (write-behind), meaning that a write to a memory location will make it as far as the cache, until at some later point the updated cache line gets flushed out to the persistent memory. Programmers usually have no control of this process.

A simple algorithmic solution to this problem is to explicitly flush the cache lines immediately for writes to the persistent memory. This can guarantee the desired states of the data in the persistent memory (more details discussed in Chapter 9.2), using instructions (such as Intel's `CLFLUSH` instruction) supported by various programming models (e.g., [178, 240, 241]). Compared to more general system-level support, such an algorithmic solution can be easier to implement (by simply adding a few lines in the code) and can handle multiple types of faults.

On the other hand, the flush instructions require memory barriers to enforce the ordering in the execution, and may also cost extra to interfere the system buffers or synchronize the processors. As a result, the drawback of this simple solution for fault-tolerance is the additional cost for the writes.

Write-efficient algorithms can be useful in this solution. At a high level, a write-efficient algorithm generally requires fewer writes to the main memory. Since these write operations are now flushed explicitly and are expensive, either the reduced number of writes can make such bottleneck less severe, or the cost of these operations will not dominate the overall cost. In either case, write-efficient algorithms alleviate or diminish the extra cost of making the algorithms or programs fault-tolerant.

There are other system-level or architecture-level considerations that cause writes to be more expensive than reads. For example, in general multicore programming, algorithms can make concurrent reads and writes to memory locations. In practice, concurrent reads do not have much overhead, but concurrent writes are more costly due to the cache coherency traffic over a shared bus [140, 166]. Another example can be a cloud computing

framework, where the read-only data is kept in DRAM, but intermediate results need to be written to disk for reliability [84, 237].

1.1.3 Intellectual Curiosity

Many architecture-related and system-related considerations that cause an asymmetry between reads and writes have been discussed in previous sections. At a high level, these reasons all produce asymmetry because of different mechanisms between load and store data: reads only check the data, but writes change the data. The reason symmetric algorithms worked well is that the read and write costs happen to be roughly the same on DRAM and hard disks (but they do not have to always be in other cases). As mentioned in previous sections, there are many scenarios where read-write asymmetry exists. It is intellectually interesting to understand how such asymmetry can impact algorithm design.

There are two possibilities for such asymmetry: reads are more expensive, or writes are more expensive. The first case does not make much sense for an algorithm since utilizing cheaper writes indicates that many intermediate results are written out and never read back again.[1] The second case, which is the setting discussed in this thesis, is more general and shares a high-level similarity to many other settings, such as the streaming setting (space-limited computations) or the distributed setting. These settings study whether less communication to the data carrier for certain algorithms is possible, at the cost of possible extra reads or lower accuracy of the solutions.

For example, the streaming model [19] usually assumes a memory size that is a (poly)logarithmic function of the input stream size. In practice, caches in this day and age can hold dozens of megabytes of data, but the results under this model show many intrinsic properties of the problems that would not be considered in other settings. (Of course, there are other assumptions and special applications for streaming algorithms.) Similarly, many distributed algorithms are designed based on a message-passing model that synchronizes at every single round [228]. Although this assumption is less practical, these results can usually be used in designing practical distributed algorithms, or demonstrate lower and upper bounds of many other problems in other settings as long as the problems can be modeled similarly.

Therefore, we argue that the study of write-efficient algorithms is beneficial even without considering the emerging hardware technologies, since it provides a new angle to rethink whether the operations in previous approaches are necessary or not. In many cases, such studies help us find bottlenecks in the algorithms which were hidden or have previously gone unnoticed. Furthermore, this thesis also contains many results that are interesting even without considering the read-write asymmetry, and some of these results

[1] It may facilitate some applications that require preprocessing and each query only reads a small portion of the computed values.

were obtained by designing algorithms in the context of write-efficiency. Two examples are shown here.

- Chapter 7 studies cache-oblivious dynamic programming algorithms, including both lower and upper bounds. When studying the lower bounds, we observed that some commonly-seen recurrences which were previously believed to have the same complexity [92, 94, 263, 270] appear to be different in the asymmetric setting. By checking the results carefully, we found that these recurrences have different asymptotical complexities even in the classical (symmetric) setting, and that it is too subtle to be noticed in the classic setting. To address such relationship, we propose the k-d grid computation structure, which abstracts these computations. By showing both computational lower bounds and efficient parallel algorithms for computing k-d grids, we improved the results for dozens of problems on both the asymmetric setting and the classic symmetric setting.
- In designing write-efficient planar Delaunay triangulation, the most promising candidate is the incremental construction algorithm, which also runs faster than other approaches in practice (in parallel) [60, 150]. However, all algorithms of the incremental construction do not have polylogarithmic depth for the worst-cases and are not write-efficient. After thoroughly investigating this algorithm, a new algorithm is designed which is highly parallel and write-optimal, and at the same time it is still work-optimal. As a result, we believe that the new algorithm in this thesis has the potential to outperform the existing state-of-the-art implementations even on the symmetric memories, because of the better parallelism and reduced writes. This approach is abstracted as a framework which is applied to many other incremental algorithms, and these parallel write-efficient algorithms are introduced in Chapter 6.

In conclusion, the write-efficient algorithms in this thesis can improve the existing results in the symmetric setting, and we believe there can be more in the future. The outcome of such study is beyond the boundary of efficient algorithms on the future NVMs, and the techniques proposed in thesis can be generalized to a broader extent.

1.2 Computational Models

To systematically study algorithms under asymmetric read and write cost, computational models are needed to measure the runtime of algorithm in different settings. The full details of the models considered in this thesis are shown in Chapter 2. They are briefly summarized here for the convenience of overviewing the results of the write-efficient algorithms in this thesis later in this chapter.

(M, ω)-ARAM: the sequential model.

The simplest model considered in this thesis consists of an asymmetric random-access memory such that reads cost 1 and writes cost $\omega > 1$, as well as a small number of

symmetric "registers" that can be read or written at unit cost. These registers are crucial, since otherwise the computed result of any operation needs to be written out.

More generally, this thesis considers settings in which the size of the symmetric memory is $M \ll n$, where n is the input size. This defines the (M, ω)-*Asymmetric RAM* ((M, ω)-ARAM), comprised of a symmetric small-memory of size M, an asymmetric large-memory of unbounded size, and an integer ω representing the relative cost of a write to a read. Note that the use of small amounts of symmetric memory along with the large asymmetric memory matches the expected reality of real machines.

The *ARAM cost* Q is the number of reads from large-memory plus ω times the number of writes to large-memory. The *work* W is Q plus the number of reads and writes to small-memory. Ideally, a write-efficient algorithm is also work-efficient if $W = Q + W_{OPT}$, where W_{OPT} is the optimal time complexity of this algorithm on the RAM model (i.e., without considering the I/O issues).

In most of the cases in this thesis, the block size B for each memory access is ignored for the sake of simplicity in algorithm designs. If necessary, it can be considered straightforwardly and in this case each read (write) loads (stores) a memory block of B words. This variant is referred to as the (M, B, ω)-*ARAM*, which can be viewed as the extension of the external-memory model [7] on the asymmetric setting. We only measure the ARAM cost Q on this model since it seems unreasonable to define the corresponding work when considering this block size.

Asymmetric NP model: the parallel model, and the scheduling theorem.

The next step is to consider modeling parallel computations with asymmetric read and write costs. To address the read-write asymmetry, this thesis defines the *Asymmetric Nested-Parallel (NP)* model, which combines the features of the sequential (M, ω)-Asymmetric RAM model and the popular nested-parallel model but with a distinctive memory allocation scheme. Such a scheme is necessary in order to allow the computations to be scheduled effectively on asymmetric memories.

Specifically, the Asymmetric NP model is comprised of small stack-allocated memories with symmetric read-write costs, and an unbounded heap-allocated shared memory with asymmetric read-write costs. Stack-allocated memory is allocated by a task, available to the task and any children it forks, but becomes invalid when the task finishes. This thesis shows that the model, with its costs analyzed based on the computation DAG (with no notions of processors or scheduling) maps efficiently onto a more concrete machine model, when using a work-stealing scheduler [73]. In particular, the model's careful accounting for task memory usage yields good bounds on the number of writes incurred during a steal, because it can accurately capture the true working set sizes that need to be transferred.

More accurately, in this model, the cost measures of a computation are the ***depth*** D which is the length of the longest (unweighted) path in the DAG, and the ***work*** W which

is the sum of the (weighted) costs of all operations in its DAG. When the algorithm is executed sequentially, the work W in Asymmetric NP model matches the work W in the (M, ω)-ARAM model. This thesis shows that under mild assumptions, a work-stealing scheduler can execute an algorithm with work W and depth D in $O(W/P + \omega D)$ expected time on P processors. This bound indicates that the classic nested-parallel computation can be scheduled efficiently on the asymmetric memory when $P = O(W/(\omega D))$, which holds for the parallel algorithms in this thesis under most practical situations.

The asymmetric ideal-cache model and cache-oblivious paradigm.

The *ideal-cache model* [131] is widely used in designing algorithms that optimize the communication between the CPU and the memory. Similar to the external-memory model, the ideal-cache model is comprised of an unbounded large-memory and a small-memory (cache) of size M. Data are transferred between the two levels using cache lines of size B, and all computation occurs on the data in the cache. An algorithm is *cache-oblivious* if it is unaware of both M and B. The goal in designing these algorithms is to reduce the *cache complexity* of an algorithm, which is the number of cache lines transferred between the cache and the main memory, assuming an optimal (offline) cache replacement policy. This optimal replacement means that the analysis of algorithm cost may assume any replacement policy, even defining an arbitrary strategy for selecting which blocks to load or evict. The optimal cache replacement strategy always requires no more than such cost, and a more practical LRU policy uses no more than twice of this (ideal) cost. The advantage of cache-oblivious algorithms is that they are flexible and portable, and adapt to all cache parameters and all levels of a multi-level memory hierarchy.

The asymmetric variant of this model distinguishes reads from writes as follows. A cache block is *dirty* if the version in the cache has been modified since it was brought into the cache, and *clean* otherwise. When a cache miss evicts a clean block the cost is 1, but when evicting a dirty block the cost is $1 + \omega$, where 1 for the read and ω for the write. Again, an ideal offline cache replacement policy is assumed—i.e., minimizing the total cache complexity. The cost of an algorithm Q or Q_I is the overall cost for moving the cache lines. Under this model however, the LRU policy is no longer 2-competitive, and can be as worse as a factor ω. To overcome it, this thesis discusses a variant of the LRU policy (the read-write LRU policy), which is competitive within a constant factor. That is, for any sequence S of instructions, if it has cost $Q_I(S)$ on the asymmetric ideal-cache model with cache size M_I, then it will have cost

$$Q_L(S) \leq \frac{M_L}{M_L - 2M_I} Q_I(S) + (1 + \omega)M_I/B$$

on an asymmetric cache with read-write LRU policy and cache size $M_L > 2M_I$.

To read the result, the last term $(1 + \omega)M_I/B$ accounts for some initialization of the cache, which can be ignored asymptotically. Assuming that $M_L = 3M_I$, the multiplicative term before Q_I is three, which indicates that the read-write LRU policy requires no more

than three times more memory transfers than the optimal offline policy when the cache size of the LRU policy is three times larger. More details about this cache policy and the proof can be found in Section 2.3.4.

Regarding designing parallel cache-oblivious algorithms, known scheduling results [59] indicate that the depth, D, and the sequential cache complexity of a computation, Q_1, are sufficient for deriving bounds on parallel cache complexity [59]. Let D and Q_1 be given. Then for a p-processor shared-memory machine with private caches (each processor has its own cache) using a work-stealing scheduler, the total number of misses Q_p across all processors is at most $Q_1 + O(pDM/B)$ with high probability [2]. For a p-processor shared-memory machine with a shared cache of size $M + pBD$ using a parallel-depth-first (PDF) scheduler, $Q_p \leq Q_1$ [50]. These bounds can also be extended to multi-level hierarchies of private or shared caches respectively [59].

1.3 Overview of Lower Bounds

This thesis shows a variety of lower bounds of the ARAM cost Q of some fundamental problems including Fast Fourier Transform (FFT), sorting network, diamond DAGs, permuting, and sorting. From a theoretical view, these lower bounds indicate the amount of improvement that can be made to classic algorithms when considering the asymmetry between reads and writes. The lower bounds lead us to design algorithms that are asymptotically optimal on (M, ω)-ARAM and close the gaps. On the other hand, these lower bounds show the hardness of these problems, indicating that practically (i.e., considering the actual values of ω and M), we can only achieve a constant improvement unless $M = o(\omega)$, which seems to be unrealistic. A list of results is shown in Table 1.1.

To interpret the results, for FFT and sorting network that have a fixed computational DAG, the improvement is limited to $\log(\omega M)/\log M$ (the bounds in the symmetric setting are listed in Table 2.1 and $B = 1$). The result for Diamond DAG is interesting: there is no asymptotic improvement without allowing redundant computation, even if the reads are free! The general proof techniques are partitioning a computation into subcomputations that each have a lower bound on cost but an upper bound on the number of inputs and outputs, which lower bound the costs to finish the computations of these problems.

Based on the model proposed in this thesis, Jacob and Sitchinava [182] recently show lower bounds on permuting, sorting and sparse-matrix vector multiplication. These results are also summarized in Table 1.1(b). In this case, the block size B is also considered since the bounds are trivial otherwise.

(a). The results in this thesis based on the (M, ω)-ARAM.

Problems	ARAM Cost (Q)	Work (W)	Remarks
FFT	$Q(n) = \Omega\left(\frac{\omega n \log n}{\log(\omega M)}\right)$	$W(n) = Q(n) + \Omega(n \log n)$	Section 3.2
Sorting Network	$Q(n) = \Omega\left(\frac{\omega n \log n}{\log(\omega M)}\right)$	$W(n) = Q(n) + \Omega(n \log n)$	Section 3.3
Diamond DAG	$Q(n) = \Omega\left(\frac{\omega n^2}{M}\right)$	$W(n) = Q(n) + \Omega(n^2)$	Section 3.4

(b). Later results from Jacob and Sitchinava's 2017 SPAA paper based on the (M, ω)-ARAM. On these problems, the block size B (full definition given in Section 2.1.2) is considered since the bounds are trivial otherwise.

Problems	ARAM Cost (Q)	Remarks
Permuting and sorting	$Q(n) = \Omega\left(\min\left\{n, \frac{\omega n \log(n/B)}{B \log(\omega M/B)}\right\}\right)$	Section 3.5
SparseMxV	$Q(n) = \Omega\left(\min\left\{h, \frac{\omega h \log(n/\max\{h/n, M\})}{B \log(\omega M/B)}\right\}\right)$	Section 3.5

Table 1.1: Summary of lower bounds on the (M, ω)-ARAM. In all cases n is the input size. In sparse-matrix vector multiplication (referred to as "SparseMxV" in the table), h is the number of non-zero elements in the matrix.

1.4 Overview of Write-Efficient Algorithms

1.4.1 Basic Algorithmic Building Blocks

To start the discussion of write-efficient algorithms, this thesis first introduces the basic building blocks in Chapter 4 that are widely used in designing other algorithms. Therefore, when designing more complicated and advanced algorithms, these building blocks can be directly used, just like in the symmetric setting.

Most of the building blocks can be write-efficient sequentially using some known techniques. Their cost bounds on (M, ω)-ARAM are listed on the top part in Table 1.2, which include search tree and priority queue, FFT, and some simple computational geometry and graph algorithms. The exception is algorithm for edit distance (or LCS, in Section 4.5), which requires delicate design and improves the ARAM cost by a factor of $\Omega(\omega^{1/3})$ at the cost of a factor of $O(\omega^{2/3})$ work. It remains an interesting open problem on whether such tradeoff is optimal.

Problem	ARAM Cost ($Q(n)$ or $Q(n,m)$)	Work ($W(n)$ or $W(n,m)$)
Search Tree, Priority Queue	$O(\omega + \log n)$ per update	$O(\omega + \log n)$ per update
FFT	$\Theta(\omega n \log n / \log(\omega M))$	$\Theta(Q(n) + n \log n)$
Diamond DAG	$\Theta(n^2\omega/M)$	$\Theta(Q(n) + n^2)$
Longest Common Subsequence, Edit Distance	$O\left(\frac{\omega n^2}{\min(\omega^{1/3}M, M^{3/2})}\right)$	$O\left(n^2 + \frac{\omega n^2}{\min(\omega^{1/3}M^{2/3}, M^{3/2})}\right)$
Planar Convex Hull, Triangulation	$O(n(\log n + \omega))$	$\Theta(n(\log n + \omega))$
BFS, DFS, Topological Sort, SCC	$\Theta(\omega n + m)$	$\Theta(\omega n + m)$
Minimum Spanning Tree	$O(m \min(n/M, \log n) + \omega n)$	$O(Q(n,m) + n \log n)$
Single-source Shortest Paths	$O(\min(\omega(m + n\log n), m(\omega + \log n), n(\omega + m/M)))$	$O(Q(n,m) + n \log n)$

Table 1.2: Summary of the upper bounds on the (M, ω)-ARAM. For all cases, n is the input size. For graph algorithms, m is the number of edges in the graph (n is the number of vertices). All algorithms with parallel versions are later shown in Table 1.3.

Then the thesis focuses on the parallel primitives, which are more challenging in general. Parallel algorithms on reduce (generalized sum), ordered filter, and list and tree contraction are discussed in Section 4.2 and 4.3, and summarized in Table 1.3. All of these algorithms have low depth and are optimal in terms of both I/Os and the number of arithmetic operations. Particularly, for list and tree contraction, the parallel write-efficient algorithms reduce the number of writes by a factor ω for any value of ω, and remain highly parallel. This approach is discussed in Section 4.3.

Sorting is one of the most important algorithmic primitives and is extensively applied in other algorithms in this thesis. Write-efficient algorithms for sorting are discussed in Section 4.4. These algorithms have optimal bounds on (M, ω)-ARAM, (M, B, ω)-ARAM, and the asymmetric ideal-cache model.

1.4.2 Graph Algorithms

This thesis also introduces several write-efficient graph algorithms. Many previous algorithms partition the computations so that each piece can fit into the small-memory to avoid reading and writing to the large-memory. However, due to the lack of efficient graph partitioners for general graphs, designing write-efficient graph algorithms is generally hard. In most cases, our goal is to fundamentally change the algorithmic design in order to trade fewer writes from more reads (and other operations).

Problems	Work W	Depth D
Primitives		
Reduce	$\Theta(n+\omega)$	$\Theta(\log n)$
Ordered Filter	$\Theta(n+\omega k)^{\dagger}$	$O(\log n)^{\dagger}$
List Contraction	$\Theta(n)$	$O(\omega \log n)^{\dagger}$
Tree Contraction	$\Theta(n)$	$O(\omega \log n)^{\dagger}$
Comparison Sort	$\Theta(n \log n + n\omega)$	$O(\log n)^{\dagger}$
Graph Algorithms		
Construction of (bi)connectivity oracles	$O(m+\omega n)^{*}$ $O(\sqrt{\omega}m)^{*}$	$O(\omega \log^2 n)^{\dagger}$ $O(\omega^{3/2} \log^3 n)^{\dagger}$
MST	$O(\alpha(n)m + \omega n \log(\min(m/n, \omega)))$	$O(\omega \operatorname{polylog}(n))^{\dagger}$
Breadth-first search	$O(m+\omega n)^{*}$	$O(\Delta \log^2 n)^{\dagger}$
Geometric Algorithms		
Planar convex hull and Delaunay triangulation	$O(n \log n + \omega n)^{*}$	$\tilde{O}(\log^2 n)^{\dagger}$
The construction of k-d tree, interval tree, priority search tree	$O(n \log n + \omega n)^{*}$	$O(\log^2 n)^{\dagger}$
Output-sensitive planar convex hull	$O(n(\log k + \omega \log\log k))^{*}$ $O(n(\min(k, \log n) + \omega))^{*}$	$\tilde{O}(\log^2 n \log k)^{\dagger}$ $\tilde{O}(\log^2 n \log k)^{\dagger}$

Table 1.3: The work and depth of some write-efficient algorithms in this thesis on the Asymmetric NP Model. For graph algorithms, n and m are the number of vertices and edges. In all other cases, n is the input size. Δ is the diameter of a graph. polylog means polylogarithmic. k is the output size. $(*)$ indicates in expectation; $(\dagger)$ indicates with high probability.

This thesis first studies undirected graph connectivity and biconnectivity in Section 5.3. The proposed sequential and parallel algorithms solve these connectivity problems using significantly fewer writes than the conventional algorithms.

There are two key techniques in these algorithms to achieve write-efficiency. The first technique is the parallel algorithms to generate the (bi)connectivity oracles using linear writes proportional to the number of vertices, which further consists of two components. The first component is the ***BC (biconnected-component) labeling*** of a graph, which is a compact representation of the biconnectivity information of a graph using $O(n)$ space, where n is the number of vertices. Then a query for biconnected component,

	Connectivity		Biconnectivity		Best choice
	Sequential	**Parallel**	**Sequential**	**Parallel**	**when**
Best prior results	$O(m+\omega n)$	$O(\omega m)^*$	$O(\omega m)$	$O(\omega m)^*$	–
This thesis (§5.3.4.2,§5.3.5.2)	$O(m+\omega n)^*$	$O(m+\omega n)^*$	$O(m+\omega n)^*$	$O(m+\omega n)^*$	$m \in \Omega(\sqrt{\omega}n)$
This thesis §5.3.4.3,§5.3.5.3	$O(\sqrt{\omega}m)^*$	$O(\sqrt{\omega}m)^*$	$O(\sqrt{\omega}m)^*$	$O(\sqrt{\omega}m)^*$	$m \in o(\sqrt{\omega}n)$

Table 1.4: Summary of main results for constructing connectivity oracles (n nodes, m edges, $*$=expected), where $\omega > 1$ is the cost of writes to the asymmetric memory. Here "Sequential" indicates the ARAM work, and "Parallel" indicates the work on Asymmetric NP. All parallel algorithms have depth polynomial in $\omega \log n$. For prior results, this table shows the work of the best prior sequential algorithm and parallel algorithm respectively. Compared to prior work, asymmetric memory writes are reduced by up to a factor of ω, yielding improvements in both sequential and parallel settings. Query times are $O(\sqrt{\omega})^*$ (connectivity) and $O(\omega)^*$ (biconnectivity) for the last row and $O(1)$ for the rest. For all algorithms the small symmetric memory is only $O(\omega \log n)$ words.

which originally requires an output size of $O(m)$ where m is the number of edges, can be answered in constant time from the BC labeling. Such representations have been adopted in future research work [112]. The second component is the approach to compute the connectivity and biconnectivity labeling using $O(m)$ work and polylogarithmic depth.

The second and primary technique is the construction of an $o(n)$-sized ***implicit decomposition*** of a sparse graph G on n nodes, which partitions the graph into connected clusters with sub-linear space to be stored. Using an implicit decomposition, a connectivity or biconnectivity query only requires read-only access to G and a small cost. The construction breaks the linear-write "barrier", resulting in costs that are asymptotically lower than conventional algorithms while adding only a modest cost to the querying time.

To be more specific, the new results are summarized in Table 1.4. Denote n to be the number of vertices and m to be the number of edges. This thesis provides (bi)connectivity oracles that can be preprocessed in either $O(m+\omega n)$ or $O(\sqrt{\omega}m)$ work with small depth. Then each connectivity query can be answered in $O(1)$ or $O(\sqrt{\omega})$ work, and each biconnectivity query can be answered in $O(1)$ or $O(\omega)$ work, respectively.

Another important class of graph algorithms are distance-based graph algorithms, which are considered in this thesis in Section 5.4. Since most of these distance-based problems are notoriously hard to solve in parallel, they are mainly studied in the sequential setting based on (M, ω)-ARAM model. This thesis covers single source shortest paths (SSSP) using Dijkstra in Section 5.4.1 and minimum spanning trees (MST) in Section 5.4.2, and their costs are summarized in Table 5.2. In the parallel setting, this thesis also discusses

the minimum spanning trees in Section 5.4.2.2, and BFS in Section 5.4.3. The bounds of these algorithms are summarized in Table 5.3. The improvements of these algorithms are decided by the numbers of vertices and edges of the input instance, and the hardware parameters M (symmetric memory size) and ω (read-write asymmetry).

1.4.3 Geometric Algorithms

Achieving parallelism (polylogarithmic depth) and optimal write-efficiency simultaneously seems generally hard for many algorithms and data structures in computational geometry. Here, optimal write-efficiency means that the number of writes that the algorithm or data structure construction performs is asymptotically equal to the output size.

This thesis mainly consists of two general frameworks and shows how they can be used to design algorithms and data structures from geometry with high parallelism as well as optimal write-efficiency. The first framework is designed for randomized incremental algorithms. Randomized incremental algorithms are relatively easy to implement in practice, and the challenge is in simultaneously achieving high parallelism and write-efficiency. There are several technical parts in this framework. The first part includes several new incremental algorithms, which is the first step of the parallel and write-efficient algorithms. The difficulty of this step is usually in showing the parallelism, and in this approach the new results are based on analyzing the dependence graph of these algorithms. This technique is used in other problems in [61, 164, 225, 254]. The second part is for the write-efficiency (while maintaining parallelism), which further consists of two components: a DAG-tracing algorithm and a prefix doubling technique. The write-efficiency is from the DAG-tracing algorithm, that given a current configuration of a set of objects and a new object, finds the part of the configuration that "conflicts" with the new object. Finding n objects in a configuration of size n requires $O(n \log n)$ reads but only $O(n)$ writes. Once the conflicts have been found, then previous and new parallel incremental algorithms can be used to resolve the conflicts among objects taking linear reads and writes. This allows for a prefix doubling approach in which the number of objects inserted in each round is doubled until all objects are inserted.

This framework obtains parallel write-efficient algorithms for comparison sort, planar Delaunay triangulation, and k-d trees, all requiring optimal work, linear writes, and polylogarithmic depth. The most interesting result is for Delaunay triangulation (DT). Although DT can be solved in optimal time and linear writes sequentially using the plane sweep method, previous parallel DT algorithms seem hard to make write efficient. Most are based on divide-and-conquer, and seem to inherently require $\Theta(n \log n)$ writes. The DT algorithm in this thesis requires delicate above-mentioned design and analysis in order to achieve parallelism and write-efficiency. For k-d trees, the p-batched incremental construction technique is introduced that maintains the balance of the tree while asymptotically reducing the number of writes.

	Construction	Query	Update
Classic interval tree	$O(\omega n \log n)$	$O(\omega k + \log n)$	$O(\omega \log n)$
WE interval tree	$O(\omega n + n \log n)$	$O(\omega k + \alpha \log_\alpha n)$	$O((\omega + \alpha) \log_\alpha n)$
Classic priority search tree	$O(\omega n \log n)$	$O(\omega k + \log n)$	$O(\omega \log n)$
WE priority search tree	$O(\omega n + n \log n)$	$O(\omega k + \alpha \log_\alpha n)$	$O((\omega + \alpha) \log_\alpha n)$
Classic range Tree	$O(\omega n \log n)$	$O(\omega k + \log^2 n)$	$O((\log n + \omega) \log n)$
WE range tree	$O((\alpha + \omega)\, n \log_\alpha n)$	$O(\omega k + \alpha \log_\alpha n \log n)$	$O((\alpha \log n + \omega) \log_\alpha n)$

Table 1.5: A summary of the work cost of the data structures discussed in Section 6.7. In all cases, it is assumed that the tree contains n objects (intervals or points). For interval trees and priority search trees, the number of writes in the construction can be reduced from $O(\log n)$ per element to $O(1)$. For dynamic updates, the number of writes per update can be reduced by a factor of $\Theta(\log \alpha)$ at the cost of increasing the number of reads in update and queries by a factor of α for any $\alpha \geq 2$.

The second framework is designed for augmented trees, including interval trees, range trees, and priority search trees. The goal is to achieve write-efficiency for both the initial construction as well as future dynamic updates. The framework consists of two techniques. The first technique is to decouple the tree construction from sorting, and introduce parallel algorithms to construct the trees in linear reads and writes after the objects are sorted (the sorting can be done with linear writes). Such algorithms provide write-efficient constructions of these data structures, but can also be applied in the rebalancing scheme for dynamic updates—once a subtree is reconstructed once it is unbalanced. The second technique is the α-labeling. Some tree nodes are subselected as critical nodes, and the augmentation is only maintained on these nodes. By doing so the number of tree nodes that need to be written on each update is limited, at the cost of having to read more nodes.

This framework obtains efficient augmented trees in the asymmetric setting. In particular, the trees can be constructed in optimal work and writes, and polylogarithmic depth. For dynamic updates, a trade-off is provided between performing extra reads in queries and updates, while doing fewer writes on updates. A standard algorithm uses $O(\log n)$ reads and writes per update ($O(\log^2 n)$ reads on a 2D range tree). The number of writes can be reduced by a factor of $\Theta(\log \alpha)$ for $\alpha \geq 2$, at a cost of increasing reads by at most a factor of $O(\alpha)$ in the worst case. These results are shown in Table 1.5. For example, when the number of queries and updates are about equal, we can improve the work by a factor of $\Theta(\log \omega)$, which is significant given that the update and query costs are only logarithmic.

The previous two frameworks introduce new parallel write-efficient algorithms for comparison sorting, planar Delaunay triangulation, k-d trees, and static and dynamic augmented trees (including interval trees, range trees and priority search trees). We believe the techniques in these frameworks will be useful for designing other algorithms

in both the symmetric and asymmetric settings. Also, new parallel write-efficient algorithms for write-sensitive hash tables, and sequential write-efficient algorithms for linear programming-style algorithms are also discussed in this thesis.

1.4.4 Cache-Oblivious Algorithms for Dynamic Programming and Linear Algebra

Cache-oblivious algorithms [131] are widely used in designing algorithms that optimize the communication between CPU and memory. They are flexible and portable, and adapt to all levels of a multi-level memory hierarchy. As a result, cache-oblivious algorithms are used for engineering efficient implementations [245], especially for applications in linear algebra and dynamic programming.

This thesis focuses on a class of cache-oblivious algorithms that have a similar computation structure as matrix multiplication and can be coded up as nested for-loops. Such algorithms are in the scope of dynamic programming (e.g., the LWS/GAP/RNA/Parenthsis problems, and definitions given in Section 7.8) and linear algebra (e.g., matrix multiplication, Gaussian elimination, LU decomposition) [59, 92, 94, 96, 131, 181, 263, 270, 271, 279].

To improve the performance of these algorithms in the asymmetric setting, this thesis proposes a level of abstraction of the computation in these cache-oblivious algorithms. This abstraction is referred to as the k-d grid computation structure (short for the k-d grid). Unlike previous methods that consider the number of nested loops, we observe that the key underlying factor in determining the cache complexity of these computations is the **number of input entries** involved in each basic computation cell (e.g., two input values for the multiplication in matrix product), which is captured by the k-d grid. Interestingly, this observation and the associated solution also improves the bounds of these algorithms in the symmetric setting.

Then this thesis discusses the lower bounds to compute these k-d grids with and without considering the asymmetric cost between writes and reads. Based on the analysis of the lower bounds, highly-parallelized algorithms with the matching bound are also proposed to compute a k-d grid assuming no data dependency within it.

All the cache-oblivious computations considered in this thesis can all be abstracted as or decomposed into multiple k-d grids without any local dependencies within each of them. Due to this reason, the analysis of the lower and upper bounds on the k-d grid can be applied to these computations, yielding the new results as shown in Table 1.6. For the same reason, the better parallelism in computing k-d grid can also be applied to all these computations.

We believe that the framework for analyzing cache-oblivious algorithms based on k-d grids provides a better understanding of these algorithms. In particular, the contributions include:

Dimension	Problems	Cache Complexity	
		Symmetric	Asymmetric
$k = 2$	LWS/GAP*/RNA/knapsack recurrences	$\Theta\left(\frac{C}{BM}\right)$	$\Theta\left(\frac{\omega^{1/2}C}{BM}\right)$
$k = 3$	Combinatorial matmul, LU-decomp, Kleene's algorithm, Parenthesis recurrence	$\Theta\left(\frac{C}{B\sqrt{M}}\right)$	$\Theta\left(\frac{\omega^{1/3}C}{B\sqrt{M}}\right)$

Table 1.6: I/O costs of cache-oblivious algorithms based on the k-d grid computation structures. Here C is the number of algorithmic instructions in the corresponding computation. (*) For the GAP recurrence, the upper bounds have addition terms that can be found in Section 7.8.2.

- Algorithms with improved cache complexity on many problems in the symmetric setting (without considering more expensive writes), including the GAP recurrence, protein accordion folding, and RNA recurrence.
- Improved cache-oblivious algorithms with linear parallel depth for solving all-pair shortest-paths, Gaussian elimination, triangular system solver, LWS recurrences and protein accordion folding.
- Write-efficient cache-oblivious algorithms, including problems in matrix multiplication, many linear algebra algorithms, all-pair shortest-paths, and a number of dynamic programming recurrences. The asymmetric cache complexity is improved by a factor of $\Theta(\omega^{1/2})$ or $\Theta(\omega^{2/3})$ on each problem compared to the previous results [66]. This improvement is optimal under certain assumptions (the CBCO paradigm, defined in Section 7.4.2).
- The analytical framework is concise and includes about a dozen algorithms, while each is discussed about its lower and upper cost bound and parallelism in both the symmetric and the asymmetric settings.

1.5 Experimental Validation

This thesis also conducts experiments to analyze whether the theoretically write-efficient algorithms can lead to good performance in practice, and which algorithmic techniques are useful. As the first research of this kind, the experiment section focuses more on the experimental framework. Then several of the most commonly-used algorithmic building blocks in modern programming are tested in this framework, which includes: unordered set/map implemented using **hash tables**, set/map implemented using **balanced binary search trees**, **comparison sort**, and graph traversal algorithms: **breadth-first search** for unweighted graphs and **Dijkstra's algorithm** for weighted

graphs. An interesting direction for future work is to examine other algorithms proposed in this thesis using the experimental framework.

At this moment, no non-volatile main memory is currently available, making it impossible to get real timings. Since the details about the parameters of the memory remain unknown, performing detailed cycle-level simulation (by using e.g., PTLsim [232], MARSSx86 [214] or ZSim [244]) is of questionable utility. Hence, our framework is based on a software simulator that can efficiently and precisely measure the number of read and write transfers of an algorithm using different caching policies. These numbers can be used as reasonable proxies for both runtime (especially when implemented in parallel) and energy consumption for I/O-bounded algorithms. Moreover, conclusions drawn from these numbers can likely give insights into tradeoffs between reads and writes among different algorithms.

We also note that designing write-efficient algorithms falls in a multi-dimensional parameter space since the asymmetries on latency, bandwidth, and energy consumption between reads and writes are different. In our experimental framework, the cost of these different asymmetries is abstracted as a single value, ω. This value together with the cache size M and cache-line size B (set to be 64 bytes in this paper) form the parameter space of an algorithm.

Our framework provides a simple and hardware-independent method to analyze and experiment the performance on the asymmetric memory. We investigate the algorithmic techniques and learn lessons from the experiments that generally apply for a reasonably large parameter space of ω, M, and B. This framework also allows monitoring, reasoning, and debugging the code easily, and it can remain useful even after the new hardware is available.

Along with the framework, many different algorithms and data structures and their write-efficient implementations are designed, implemented, and discussed in this thesis. Many of them are non-trivial and require careful algorithmic design, analysis, and implementation. Under our cost measure, which is the asymmetric I/O cost, this thesis shows better approaches on all problems that are studied in this chapter, compared to the most basic and commonly-used ones on symmetric memories.

Also, it is interesting to point out that in the experiments, many interesting algorithmic strategies that improve the performance are unintuitive in the symmetric setting. For example, indirect addressing should be avoided in the symmetric setting if possible, but this cost is less problematic in the asymmetric setting. Similarly, strict balancing (e.g., in AVL or red-black trees) is less critical in the asymmetric setting because they only lead to more reads, which is a small cost compared to the costly writes. More details about our experiment are elaborated in Chapter 8, and we believe that these results can be valuable in designing and engineering write-efficient algorithms in the future.

1.6 Organization of this Thesis

The results in this thesis are primarily based on several previous publications. They are listed here for reference and are the result of the collaboration with my co-authors. In this thesis, most chapters are notably rewritten and more detailed than the original papers. Also, there are several new results in this thesis (mostly in Chapter 2, 4, and 7), which may show up in the future publications.

- Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. "Sorting with asymmetric read and write costs." *In Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015, [64]. (Chapter 2 and 4.)
- Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. "Efficient Algorithms with asymmetric read and write costs." *In Proceedings of the 24th Annual European Symposium on Algorithms (ESA)*, 2016, [67]. (Chapter 2, 3, 4 and 5.)
- Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. "Parallel algorithms for asymmetric read and write costs." *In Proceedings of the 28th ACM symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016, [42]. (Chapter 2, 4, 5 and 6.)
- Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. "Implicit decomposition for write-efficient connectivity algorithms." *In Proceedings of the 32nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2018, [43]. (Chapter 5.)
- Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. "Parallelism in randomized incremental algorithms." *In Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016, [68]. (Chapter 6.)
- Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. "Parallel Write-Efficient Algorithms and Data Structures for Computational Geometry." *In Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018, [72]. (Chapter 6.)
- Yan Gu. "Improved parallel cache-oblivious algorithms for dynamic programming and linear algebra." *ArXiv:1809.09330*, [151]. (Chapter 7.)
- Yan Gu, Guy E. Blelloch, and Yihan Sun. "Algorithmic building blocks for asymmetric memories." *In Proceedings of the 26th Annual European Symposium on Algorithms (ESA)*, 2018, [155]. (Chapter 8.)

As mentioned in Section 1.1.2, the persistency property of the new non-volatile main memory is one of the motivation to investigate write-efficient algorithms. The following paper defines a programming model or an algorithmic cost model that ensures fault-

tolerant programming and algorithm design on the new persistent main memories, or provides machanism to support parallel algorithms to be resilient to faults. Based on this model, we can design write-efficient fault-tolerant algorithms.

- Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. "The Parallel Persistent Memory Model." *In Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018, [71].

Since the content of this paper does not completely overlap with the main theme of this thesis, the summary of this paper is shown in Section 9.2 as for future work.

Many other papers by the thesis author in graduate school are not included in this thesis, They include distance-based graph algorithms [69, 70], parallel sorting and other basic building blocks [154, 254], efficient low-dimension search structures, and other graphics applications [152, 153, 165].

1.7 Related Work

The related work of this thesis are spread out in different places. Previous work on hardware-related read-write asymmetry is discussed in Section 1.1. Existing computational models are reviewed in Section 2.1 and 2.2. Related papers for each new algorithm in this thesis are provided in each associated chapters and sections for easier referencing.

Chapter 2

Models of Computation

To study algorithms systematically with asymmetric read and write cost we need computation models to measure the running time of an algorithm on different settings.

In this chapter, we first review some existing models that are most commonly-used in analyzing the performance of an algorithm in the real world. These models consider the number of CPU operations and memory accesses, parallelism, etc.

We then discuss how to extend the existing models with the asymmetry in read and write costs in the next section (Section 2.3). Some of the extensions are straightforward, but others require careful designing and analysis.

Lastly, some related issues about cache policies and parallel scheduler, are also discussed, which guarantee the performance of an algorithm in a real machine. Directly applying the classic approaches in the symmetric setting may lead to significant inefficient.

To be more specific, a ***machine model*** is the set of allowable operations used in computation and their respective costs. A ***programming model*** defines the basic primitives to form the computation. A ***cost model*** or a ***cost measure*** is an objective function that is based on a specific machine model, and takes input as an algorithm.

2.1 Existing Sequential Computational Models

2.1.1 Random-Access Machine Model and Time Complexity

A random-access machine (RAM) is an abstract machine model. Each memory location holds a w-bit word, and usually $w = \Theta(\log n)$. These words can be operated as a whole by the CPU instructions. The cost under this model of computation is as follows.

1. Each basic operation takes exactly one step.
2. Loops and subroutines are not considered simple operations. Instead, they are the composition of many single-step operations.

3. Each memory access takes exactly one step, and there is an infinite amount of memory space.

For the third point, the RAM model takes no distinguish of whether an item is in the cache, in the main memory or on the disk, which largely simplifies the analysis.

The ***time complexity*** is the cost measure on the RAM model. It is measured by counting up the number of time steps it takes on a given problem instance on the RAM model. The worst-case, average, expected or high-probability cost bound can be analyzed on a given set of input instances (and their distribution).

The most significant advantages and disadvantages of RAM are its simplicity. The cost measurement enables analyzing algorithms in a machine-independent way, and thus time complexity is extensively used and taught in the computer-science world. However, the main concern of RAM model and time complexity is the cost of memory access. The latency of memory access differs significantly depending on whether data sits in the cache, in the main memory or on the disk, thus violating the third assumption. For example, accessing a memory location in the main memory is two orders of magnitude slower than a normal CPU operation. This gap is increasing in the modern multi-core era: with the increasing of parallel processing units, basic operations become cheaper and the resource (bandwidth) of each CPU processor to access the memory is even more limited. As a result, the time complexity on RAM model can provide a relatively good estimation of the execution time of an algorithm when the cost of computation is the bottleneck. However when an algorithm intensively accesses the memory, we need the following models to get a more accurate estimation of running time.

2.1.2 External-Memory (EM) Model and I/O Complexity

The external-memory (EM) model is a machine model introduced by Aggarwal and Vitter in 1988 [7] (aka. the "I/O model" or the "disk access model"). The external-memory model simplifies the memory hierarchy to just two levels.

1. The CPU is connected to a small-memory (e.g., cache) of size M; this small-memory in turn is connected to a large-memory of effectively infinite size.
2. Both small-memory and large-memory are divided into blocks of size B, so there are M/B blocks in the small-memory.
3. The CPU can only access the memory on blocks resident in the small-memory and it is free of charge.
4. Transferring one block between the small-memory and the large-memory of a block with size B costs 1 unit.

Thus, the natural goal is to minimize the number of transfers between two memories. The ***I/O complexity*** of an algorithm on the EM model is defined as the minimum amount of memory transfers that are required to finish the computation on a given input instance.

The reason that a 2-level hierarchy is sufficient for analyzing the performance of an algorithm is because: (1) it is simpler than considering the multilevel hierarchy; (2) a single level may dominate the runtime of the application, so designing an algorithm for that level may be sufficient; (3) considering a single level exposes the algorithmic difficulties and generalizing to a multilevel is often straightforward.

The I/O complexity can usually provide a reasonable estimation of the running time of an algorithm when the algorithm is I/O bottlenecked. It can also be used as guidance for algorithm design and a cost measure to optimize. An algorithm with low or asymptotically optimal cost on I/O complexity is called an ***I/O-efficient algorithm***. A list of cost bounds of the problems can be found in Table 2.1.

2.1.3 Ideal-Cache Model, Cache-Oblivious (CO) Algorithms, and Cache Complexity

Although the EM model and I/O complexity is a useful measurement of the cost of memory access, the model itself is low-level and machine-centric, requiring the algorithm to be implemented in a careful way such that the I/O complexity matches the theoretical analysis.[1] The parameter M and B usually show up in the algorithms, so that the implementation for a specific machine may require significant effort to run on other machines, or the performance can significantly deteriorate.

In 1999, Frigo, Leiserson, Prokop and Ramachandran [131] proposed concepts of ***ideal-cache model***, which is a variation of the EM model, and ***cache-oblivious (CO) algorithms*** that are algorithms designed based on the model. In this model, the cache-oblivious algorithm is unaware of the block size B or the fast memory size M. In particular, cache-oblivious algorithms will look like normal RAM algorithms, but the cost is measured by the number of memory transfers between the fast and slow memories.

The ideal-cache model is the same as the EM model expect that it assumes an optimal replacement policy between the two memories (in the offline sense), and the caching is done automatically. This optimal replacement means proofs of the algorithm cost may posit any replacement policy, even defining an arbitrary strategy for selecting which blocks to load or evict. The optimal cache replacement always requires no more than the cost analyzed in this way. The ***cache complexity*** of an algorithm is the number of memory transfers of a cache-oblivious algorithm required on the ideal-cache model.

In practice, the block replacement can be achieved with either LRU (Least-Recently Used) or FIFO (First-In First-Out) strategy since they are $O(1)$-competitive with the optimal offline algorithm if they have a cache with twice the size. Since the cost of none of the existing CO algorithms changes by replacing the cache size of M with $2M$, this does not change the asymptotic bounds of the algorithms.

[1] In most programming languages, programmers cannot explicitly control the cache and memory transfers.

Table 2.1: Summary of some basic results on various existing models (CO algorithms requires $M = \Omega(B^2)$)

Problems	Classic RAM algorithms	I/O-efficient algorithms	Cache-oblivious algorithms
Scanning	$\Theta(n)$	$\Theta(n/B)$	$\Theta(n/B)$
Sorting, FFT	$\Theta(n \log_2 n)$	$\Theta((n/B) \log_{M/B}(n/B))$	$O((n/B) \log_M n)$
Searching	$\Theta(\log_2 n)$	$\Theta(\log_B n)$	$\Theta(\log_B n)$
Matrix multiply	$\Theta(n^3)$	$\Theta(n^3/B\sqrt{M})$	$\Theta(n^3/B\sqrt{M})$
Diamond DAG	$\Theta(n^2)$	$\Theta(n^2/M)$	$\Theta(n^2/M)$

There are many advantages of cache-oblivious algorithms. First, cache-oblivious algorithms do not depend on memory parameters, so that the cost bounds generalize to multilevel hierarchies. Algorithms are platform independent (i.e., one algorithm works on all different machines or architectures). Lastly, algorithms are efficient even when B and M are not static in some runtime environments.

A list of existing results of cache-oblivious algorithms are generalized in Table 2.1.

2.2 Existing Models for Parallel Algorithms

2.2.1 Parallel Random-Access Machine (PRAM) Model

A ***parallel random-access machine (PRAM)*** is a shared-memory abstract machine. As its name indicates, the PRAM is viewed as the parallel analogy to the random-access machine (RAM). On this model, there are a problem-size-dependent number of processors, and each can run a basic operation or read and write to a specific memory location with unit cost. All processors are synchronized, which means that in every time stamp, every processor proceeds one unit of work simultaneously. The cost of an algorithm is estimated using two parameters: the overall time to finish, and the product of this time and the number of processors.

Since multiple processors may access the same memory location, there are several assumptions to handle contentions: (1) the exclusive-read exclusive-write (EREW) model, which does not allow for concurrent reads or writes; (2) the concurrent-read exclusive-write (CREW) model, which allows for concurrent reads but not concurrent writes; (3) the concurrent-read concurrent-write (CRCW) model, which allows for both concurrent reads and writes. For the CRCW model, concurrent writes to a shared location results in either an arbitrary write being recorded (arbitrary CRCW), or the minimum (or maximum) value being recorded (priority CRCW).

Despite the simplicity of PRAM, several practical issues prevent the actual implementation of PRAM algorithms. The largest concern is the cost of synchronization and

communication, which is neglected in the model. For example, PRAM does not distinguish the time for accessing the data at different levels of the memory hierarchy on real-world machines, as well as a basic algebraic operation. The latencies among these operations can differ by up to a few hundred, so the synchronization after each step can therefore lead to significant overhead. Also in practice, explicitly controlling the operations on each physical core on most nowadays programming languages is messy and tedious. As a result, although abundant prior research on PRAM algorithms provides insightful inspiration on designing practical parallel algorithms, more recent parallel algorithm research is mostly based on the following programming and cost models, which narrow the gap between theory and practice (programming), and at the same time remain to be simple.

2.2.2 Nested-Parallel Model

The ***nested-parallel (NP) model*** (or nested fork-join parallelism) is a programming model of shared-memory parallel algorithms, in which a FORK specifies procedures that can be called in parallel, and a JOIN specifies a synchronization point among procedures. The FORK and JOIN constructs can be nested.

More formally, nested parallel computations can be defined inductively in terms of the composition of sequential and parallel components. A ***strand*** is a sequential computation at the base case. A ***task*** is then a sequential composition of strands and parallel blocks, where a ***parallel block*** is a parallel composition of tasks starting with a FORK and ending with a JOIN. The FORK instruction takes an integer n and creates n child tasks, which can run in parallel. Child tasks get a copy of the parent's register values, with one special register getting an integer from 1 to n indicating which child it is. The parent task suspends until all its children finish at the point of JOIN, and it continues with the registers in the same state as when it suspended, except the program counter advanced by one. We say that a computation has ***binary branching*** if $n = 2$.

A nested parallel computation can be viewed as a series-parallel ***computation DAG*** over the operations of the computation: the tasks in a parallel block are composed in parallel, and the operations within a strand as well as the strands and parallel blocks of a task are composed in series in the order they are executed.

Nested parallelism is supported by many parallel languages including NESL [52], Cilk [130], the Java fork-join framework [184], OpenMP [223], X10 [87], Habanero [80], Intel Threading Building Blocks [179], and the Task Parallel Library [268].

The theoretical and practical advantages of nested parallelism including: simple schedulers for dynamically allocating tasks to cores, compositional analysis of work and depth, and good space and cache behavior, which will be discussed in the next sections.

2.2.3 Work-Depth Model

The work-depth model is a cost model to analyze the cost of an algorithm on the nested-parallel (NP) model. As just described, the computation of an algorithm on the NP model can be viewed as a computation DAG.

- The ***work*** W of this algorithm is equal to the number of operations the algorithm performs, or the costs of all tasks in the computation DAG. This work W can be (asymptotically) viewed as the time complexity of running this algorithm on the RAM model.
- The ***depth*** D of this algorithm is equal to the maximum sum of costs of tasks over all directed paths in the computation DAG. It can also be viewed as the length of the longest series of operations that have to be performed sequentially due to data dependencies (the critical path). The depth may also be called the critical path length or the span of the computation.

2.2.4 Scheduling the Computation of Nested Parallelism

Brent's scheduling theorem [78, 183] indicate that using a greedy offline scheduler with p cores, the running time is bounded between W/p and $W/p + D$. When $D \ll W/p$, this running time is almost optimal. However, the offline scheduler has to be aware of the computation DAG, which is impossible in most algorithms.

In practice, the languages and libraries that support nested parallelism use the ***randomized work-stealing scheduler*** to actually schedule the computation. The scheduler assigns one double-ended queue (dequeue) for each thread. The root node is assigned to 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 (continuation), and queues the rest at the front of the queue (spawned child).
- If a processor completes a task, it tries to pull a task from the front of its own queue.
- If a processor finishes all 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.

More details of the work-stealing scheduler can be found in [54].

Work-stealing schedules are very good at load balancing because they are greedy. It can guarantee the running time to be $W/p + O(D)$ with high probability on a PRAM model with p cores [74]. Again when $D \ll W/p$, this running time is almost as good as the greedy offline scheduler. Here we will show yet another proof of this theorem, which we believe is much simpler, and we will adapt the proof to the asymmetric setting in Section 2.3.3 to show the schedule guarantee.

Theorem 2.2.1. *A work-stealing schedule with p processors on a binary DAG of size W and depth D will take at most $W/p + O(D)$ time on a CRCW PRAM* whp[2].

The main step to prove the theorem is show that the overall number of steals for the entire computation is upper bounded by $O(pD)$ *whp*. This result is of an independent interest as a step in the proof of Theorem 2.2.1, since such steals are the only scheduling overhead other than the necessary computation which is captured by the work term W that we have to pay even sequentially. When a parallel algorithm has a small depth D (e.g., $O(\text{polylog}(n))$), such overhead is always negligible.

To prove the theorem, we first show the following lemma.

Lemma 2.2.2. *It takes $O((p-1)(D + \log(1/\epsilon))$ steals from $p-1$ processors to steal D tasks from one queue with probability at least $1-\epsilon$.*

Proof. The most optimistic situation is that, no two steals process in the same timestep (i.e., each steal does not affect each other). In this case, each steal has $1/(p-1)$ probability to hit the queue and steal one task. The probability that $S = 2(p-1)(D + \log(1/\epsilon))$ steals incur less than D hits is no more than $e^{-\delta^2\mu/2}$ by Chernoff bound where $\mu = 2D + 2\log(1/\epsilon)$ and $\delta = (D + \log(1-\epsilon))/\mu$. Plugging in the values leads to the probability to be less than ϵ.

However, it is possible that multiple steals occur simultaneously in one timestep and only one of them succeeds while others fail even though they hit the queue. The most pessimistic situation is that $p-1$ steals always happen together, which maximize the chance that a steal hits the queue but fails to get the task since it is taken by another simultaneous steal. The possibility that at least one of the $p-1$ steals hit the queue in one timestep (so that at least one of the task is stolen) is at least $1-(1-1/(p-1))^{p-1} > 1-1/e$.

We can then apply a similar analysis to show that the probability that less than D tasks are stolen after $S' = 2(D + \log(1/\epsilon))/(1-1/e)$ steps is small. Similar to the previous case, such probability is no more than $e^{-\delta^2\mu/2}$ by Chernoff bound where $\mu = 2D + 2\log(1/\epsilon)$ and $\delta = (D + \log(1-\epsilon))/\mu$. This gives the same probability of less than ϵ, and S' timesteps contain $(p-1)S' = O(p(D + \log(1/\epsilon)))$ steals. □

With Lemma 2.2.2, we now prove Theorem 2.2.1.

Proof of Theorem 2.2.1. We consider each path in the DAG, and show that it can be finished with $O(pD)$ steals *whp*. For each specific path, the length is no more than D. Each node on this path is either a spawned child or executed directly after the previous node by the same processor. We now show that this path can be finished using no more than $O(pD)$ steals. It is easy to see that the later case (i.e., the next node will always be executed in the next timestep) is strictly better than the first case when a steal is required to continue

[2]We say a result holds ***with high probability (whp)*** for an input of size n if it holds with probability at least $1 - n^{-c}$, for any constant $c > 0$, over all possible random choices made by the algorithm.

the execution of this path. It is possible that the spawned child is not stolen until the continuation finishes and the processor takes over this branch, but this will only help.

As a result, the worst case is when all nodes on the path are spawned children. In this case, since we assume that a FORK spawns a task in a unit time, at any synchronized step there is always a task available to be stolen. Since the length of the path is no more than D, Lemma 2.2.2 can upper bound the number of steals to finish the execution of this path (here the D tasks waiting to be stolen may not necessarily in the same queue, but this does not affect the analysis). Given a DAG with depth D, there are at most 2^D paths in the DAG, so if we set $\epsilon = 2^D \cdot n^c$ for any constant $c \geq 1$, $O(p(D + \log n))$ steals are sufficient for executing all existing paths. We note that since the input size is at least n and the DAG needs to have $D = \Omega(\log n)$ depth to contains $\Omega(n)$ nodes, the $\log n$ term will not dominate.

In every timestep, each processor is either making some progress on processing a node, or trying for a steal attempt. The DAG contains W nodes and we have upper bounded the number of steal attempts. Since we have p processors, the entire computation can be executed in $(W + O(pD))/p = W/p + O(D)$ time. Namely, other than the W/p term to actually execute the computations in the DAG, $O(pD)$ steals are the upper bound for the additional cost for the work-stealing scheduler, which is an additional $O(D)$ cost when executing on a p-processor machine. □

This proof can be viewed as a simplified version of the previous proofs in [2, 30, 54, 74], and we show how it can be adapted into the asymmetric setting in Section 2.3.2.

Meanwhile, since the scheduler differentiates between "local" and "remote" work based on queues, it also preserves locality well, especially if the depth of the program is small. Intuitively, a low-depth program does not present many opportunities for stealing [74]. If the sequential I/O or cache complexity is $Q_1(n; M, B)$ with private caches, then the total number of cache misses for all caches $Q_p(n; M, B)$ is bounded by $Q_1(n; M, B) + O(pDM/B)$ in expectation [2]. On a parallel machine with a share cache, using the parallel depth-first (PDF) scheduler [57], then $Q_p(n; M + pMB, B) \leq Q_1(n; M, B)$. Most parallel algorithms throughout this thesis have only polylogarithmic depth, so that the parallel I/O or cache complexity is asymptotically bounded by Q_1, when plugging in the parameters from the real-world settings. As a result, it is usually enough to just analyze the sequential I/O or cache complexity, and the overhead caused by parallelism is negligible.

2.3 Models Accounting for Asymmetry

In Section 2.1 we reviewed the existing computational models that not only measure the costs of computations, memory access, and parallelism of an algorithm, but also provide an objective to optimize when designing algorithms for a specific problem. These cost models are therefore heavily used in algorithm research and conducting practical implementations. Unfortunately, none of these models have considered the asymmetric

cost between reads and writes, so directly applying algorithms designed to optimize the existing measurements may not be efficient on the future hardware. We will now introduce how these models can be extended to the asymmetric setting.

2.3.1 (M, ω)-Asymmetric RAM: the Sequential Model

To start with, the first machine model that will be introduced is the (M, ω)-***Asymmetric RAM***, which is used to analyze the sequential costs of computations, memory access.

Similar to the external-memory and cache-oblivious model, (M, ω)-ARAM assumes a symmetric small-memory of size $M \geq 1$ [3], an asymmetric large-memory of unbounded size, and a *write cost* $\omega \geq 1$, which is assumed to be an integer without loss of generality. Typically, we are interested in the setting where $n \gg M$, where n is the input size, and $\omega \gg 1$.

Table 2.2: The cost of a single access to the two memories for ARAM cost and work on (M, ω)-ARAM.

Measure	small-memory	large-memory
ARAM cost Q	0	read: 1, write: ω
work W	1	read: 1, write: ω

The model assumes standard random access machine (RAM) instructions. Two cost measures for computations are considered in the model. The (asymmetric) ***ARAM cost*** Q is defined as the total number of reads from large-memory plus ω times the number of writes to large-memory. The (asymmetric) ***work*** W is defined as the ARAM cost plus the number of reads from and writes to small-memory.[4] Because all instructions are from memory, this includes any cost of computation. In the thesis, the results are presented using both cost measures.

The model contrasts with the external-memory model [7] in the asymmetry of the read and write costs. For simplicity, as the very first research on write-efficient algorithms, the simple version of the memory is not partitioned into blocks of size B to keep the algorithms simpler. It is worth to mention that even the case of $M = O(1)$ is interesting. An example of this is the asymmetric sorting algorithms introduced in Section 4.4.1.

This blocking can easily be considered if necessary. In this case each read/write loads/stores a memory block of B words. This is referred to the (M, B, ω)-ARAM. We only measure the ARAM cost Q on this model since it seems unreasonable to define the

[3] The small-memory contains M words, each of size $\Theta(\log n)$ for input size n.

[4] The work metric models the fact that reads to certain emerging asymmetric memories are projected to be roughly as fast as reads to symmetric memory (DRAM). The ARAM cost metric Q does not make this assumption and hence is more generally applicable.

corresponding work when considering this block size. An example of algorithms analyzed using the (M, B, ω)-ARAM is given in Section 4.4.3.

The work W in the (M, ω)-ARAM is the analogy of the time complexity of the symmetric case. Each basic operation or a read is also counted as a unit cost, and a write to the large-memory costs ω. Since there is usually a return value for each operation, the small-memory is necessary in the asymmetric setting (otherwise each operation will cost ω). Here the reason we use the word "work" instead of "time" is to be consistent with the parallel setting.

2.3.2 Asymmetric NP Model: the Parallel Model

We now introduce the cost model of nested-parallel computations. The model is named as the ***Asymmetric Nested-Parallel*** (NP) model and can be viewed as the asymmetric extension of the nested-parallel model.

The ***Asymmetric Nested-Parallel*** (NP) model assumes a stack allocated symmetric small-memory, and a heap allocated asymmetric large-memory. ***Stack allocated*** memory is memory allocated by a task, available to the task and its children, but invalid when the task finishes. ***Heap allocated*** memory is allocated by a task and can be accessed by any other task, including ancestor tasks (it is completely shared memory). Each instruction has weight one, except a write to the heap memory, which has weight $\omega \geq 1$ (in practice, $\omega \gg 1$). When analyzing an algorithm, the term "writes" will be used to refer to the number of writes to heap allocated memory. We assume the amount of stack memory allocated by all but the leaf tasks (tasks with no forks) is constant. The amount of symmetric stack memory a leaf task can allocate is bounded by a parameter M_l. This separation into stack and heap allocated memory, and the distinction between leaf and non-leaf tasks for stack memory size, is made both because it is a convenient model for using the different memories, and also because it enables an efficient mapping onto a fixed number of processors, as justified below.

Note that the Asymmetric NP is an algorithmic cost model, as opposed to a machine model, enabling reasoning about nested-parallel computations without worrying about mapping the computation to machines. We address this scheduling issue next.

2.3.3 Scheduling Asymmetric NP Computations

This section shows that the algorithmic cost metrics of Asymmetric NP are sufficiently descriptive to capture the performance of Asymmetric NP computations when using good schedulers.

The Asymmetric NP model has been designed in a manner that yields an efficient mapping to an (M, ω)-***Asymmetric PRAM*** machine model. In this model, there are p processors, each running its own instructions using a small symmetric ***local*** memory of size M. The processors share an unbounded asymmetric ***global*** memory, to which

concurrent reads and writes are allowed. We also allow any processor to read the local memory of another processor (concurrently), but not to write to it. A request to read the local memory of another processor is viewed as requiring a write out to the global memory in order to enable the read, and hence is charged ω. On each processor any write to the global memory also takes ω time. All other instructions take unit time. For synchronizing we assume an atomic fetch-and-add to the global memory that can be performed in constant depth and work linear in the number of processors.

The challenge with Asymmetric NP computations is that stack variables must be written out to global memory before tasks can be migrated to a different processor from the one that forked it. The naïve approach would forego the stack and instead write all $O(1)$ stack variables for non-leaf tasks directly to global memory, making each FORK cost $\Theta(\omega)$. For fine-grained parallelism especially, where the number of FORKs for an algorithm with work W is $\Omega(W)$, this approach would yield running time no better than $\Omega(\omega W/p)$. In other words, one may as well consider every instruction a write if adopting the naïve scheduler.

Here we show that a variant of a work-stealing scheduler [74] achieves $W/p + O(\omega D)$ expected time when limited to binary forking. As discussed in Section 2.2.4, in standard (symmetric-memory) work stealing, each worker (or processor) maintains a double-ended queue called a deque of tasks that are ready to execute. Whenever a worker executes a (binary) FORK instruction, the processor continues working on the "left" child task and places the "right" child on the bottom of its deque. When a worker completes a task by executing its FINISH instruction, there are two options. If that task enabled another one, i.e., the completed task was the last outstanding child of its parent, then the worker continues working on the now-enabled parent. Otherwise, the worker removes the bottom task from its deque and executes it. If the deque is empty, the worker instead ***steals***, meaning that it chooses a random victim processor and takes the task from the top of that processor's deque if the deque is non-empty. In the event that the steal is unsuccessful, the worker will continue to attempt to steal until it successfully steals a task or the computation is finished.

In the Asymmetric NP model, working locally on a deque is cheap, and in general stack frames need not be written out. For example, if the entire computation runs sequentially on one processor, then no stack-related writes occur (assuming that the local memory is large enough to hold the stack depth).

There are still some challenges, however, most notably on steals. Because a task has access to any stack variables of its ancestor tasks, any unwritten stacks of ancestor tasks must be written out to global memory when a steal occurs. This situation is particularly challenging to analyze as it may cause steals to take time proportional to ω times the nesting depth. To cope with this challenge, Lemma 2.3.1 shows that a simple modification to work stealing results in at most a constant number of frames needing to be written. We assume that a steal request somehow interrupts its target task (e.g., all tasks can regularly

poll to check if there are any outstanding steal requests), and if work is available, the registers for the stolen task and relevant ancestors are written to the asymmetric global memory. There is a similar potential issue when a task completes: its return value must be written to its parent task, which may no longer be local, and hence a write to global memory could be required.

Lemma 2.3.1. *There exists a variant of work stealing such that on each steal (or steal attempt), only $O(1)$ stack frames are written to global memory.*

Proof. The lemma can be achieved either by modifying work stealing or by an equivalent program transformation. The program transformation is as follows. Transform every FORK into two FORKs as follows. First FORK two tasks: the left child performs the intended FORK, which we call a forking task, and the right child is a dummy task that does nothing. The dummy task is inserted onto the deque, whereas the worker continues to execute the forking task. (In Cilk-like work-stealing, e.g., [74], expressing the FORK as two "spawns" followed by a "sync" would automatically create a similar dummy task as a continuation from the second spawn.)

We claim that for the topmost task on the deque (i.e., the one that can be stolen), at most its parent and grandparent have not already been written out to global memory. In order for a task to be on the deque and hence stealable, it must be the right child of its parent. A simple induction shows that for any task on a deque, all right children of ancestors have either been stolen already or are on the same deque. (A similar claim is proven as Lemma 3 in [30].) Thus, a steal need only write-out the frames corresponding to the longest right-only path in the computation graph. The longest is three nodes, which we can show by cases. A forking task is always the left child and hence not stealable. A real task (i.e., one existing before the transformation) is always the child of a forking task, so it can have at most one unwritten ancestor frame: the parent forking task. A dummy task is always the right child of a real task, so stealing a dummy task could entail writing out three frames. □

Theorem 2.3.2. *Consider a computation in the Asymmetric NP model with binary branching factor, W work, D unweighted depth, δ nesting depth, and M_l leaf stack memory. There exists a work-stealing scheduler that executes the computation in $W/p + O(\omega D)$ expected time on a p-processor $(O(\delta) + M_l, \omega)$-Asymmetric PRAM.*

Proof. Our analysis is based on the proof of Theorem 2.2.1. But we need to account for the fact that (1) steals involve writing out to global memory, and (2) finishing tasks may also entail writing out to global memory, i.e., if the parent frame is in global memory.

We will use the similar argument from Section 2.2.4 that at most $O(pD)$ steals are sufficient for the entire computation. Note that based on our assumption and Lemma 2.3.1, each successful steal occupies two processors for at most $O(\omega)$ timesteps during the steal process and the join part, since a constant stack frames are required to communicate via the global memory.

To adapt the analysis for Lemma 2.2.2 and Theorem 2.2.1, we now require a processor to idle for $t = \Theta(\omega)$ timesteps after each unsuccessful steal, before the next steal attempt. By making this change, each successful steal can only "block" at most $O(p)$ simultaneous steal attempts instead of $O(\omega p)$ steal attemps. Therefore, the proof of Lemma 2.2.2 and Theorem 2.2.1 with minor modifications can adapt to here.

For a path in this setting, each node is either a FORK operation, which is local and assumed to use unit time, or a regular instruction which requires either unit time or ω time when it is write to the shared asymmetric large-memory. The path can always make progress on the same processor, unless the nodes that are spawn children that need to be stolen. Since the (unweighted) length of the path is upper bounded by D and the steal attempts from one processor appear at most once in every $O(\omega)$ timesteps, we can show that for every $O(\omega)$ timesteps, we can either proceed at least one step further on the path (corresponding to nodes that are not spawn children), or there is at least an task corresponding to a spawn children available to be stolen in the deque. Since there are at most D spawn children on one path, $O(pD)$ steals are sufficient to finish the computation on all paths in the DAG. The analysis in Lemma 2.2.2 and Theorem 2.2.1 can directly go through and the only difference is that at most $O(p)$ instead of $p - 1$ steals can affect each other, but such difference only influence the constant in the big-O. Other than the writes to the asymmetric large-memory during the steal, assuming local small-memory is large enough to hold the entire stack (i.e., $O(\delta) + M_l$)), the only additional writes that occur are when a child returns to a parent task that resides in large-memory, i.e., it has been stolen. This event also requires writes to the asymmetric large-memory, but the number of these writes is bounded by the number of steals.

Since we can bound the number of steals to be $O(pD)$ and each steal costs $O(\omega)$, we thus have a total running time that is $W/p + O(\omega S/p) = W/p + O(\omega D)$. □

In a very high level, each steal attempt is slowed down and charged for $O(\omega)$ time instead of a unit time. Despite that a steal is more expensive, less frequent steal is overall beneficial (each processor can execute more work) and will not increase the number of steals.

The above theorem provides justification for charging only unit cost for FORK, and for example, means that the standard reduce via a binary tree incurs only $\Theta(n + \omega)$ work instead of $\Theta(\omega n)$ work on the Asymmetric NP, as discussed in Section 4.2. Note also that our separate accounting for leaf stack memory in the Asymmetric NP model, and the observation that the non-leaf tasks of all the algorithms in this thesis each allocate only $O(1)$ stack memory, means that the bound in the lemma is only a constant number of writes per steal, whereas without the separate accounting, it would be $O(M_l)$ writes per steal.

Bulk-Synchronous Computations.

Many of the algorithms in this thesis are ***bulk-synchronous*** algorithms for which there is only one level of nesting ($\delta = 1$). The root task proceeds in a sequence of R rounds. In each round i it forks n_i child tasks (each a leaf) and waits for them to finish. The root task can run arbitrary computation between such rounds. We define the iteration count I as $\sum_{i=1}^{R} n_i$. The following lemma for scheduling bulk-synchronous algorithms provides additional support for the model.

Lemma 2.3.3. *A bulk-synchronous computation with arbitrary branching on the Asymmetric NP model with W work, D (unweighted) depth, I iteration count, and M_l leaf stack memory, can be simulated on an $(O(M_l), \omega)$-Asymmetric PRAM with P processors in $O((W + \omega I)/p + \omega D)$ time.*

Proof. The idea is that the root task runs on some processor and when it gets to a fork, it sets up the registers for the children and sets a count to n. The processors then grab tasks by decrementing this count (using the fetch-and-add). When the count reaches zero idle processors quit for that round and wait for a later round when it is again set to some non-zero number. A separate counter can be used to detect when all processors are done, and the processor that detects termination can continue with the root task. The additional work done for accessing the counter is $O(\omega I)$ and most pessimistically the charge to the weighted depth is $O(\omega D)$ so using Brent's scheduling principle [78], $(T \leq W/p + D)$, we have the given time bounds.

Each child task takes $O(M_l)$ local memory. Since the nesting depth is one, the total memory needed by a child task is $O(M_l + 1)$. This gives the memory bound. □

We note that in the realistic cases, each leaf task needs at least a write after the computation, so the ωI term will not dominate. As a result, even with a slightly different definitions for work W and depth D in the bulk-synchronous computation, we do not specifically identify this case in the rest of this thesis since the simulated running time is the same as $O(W/p + \omega D)$.

2.3.4 Asymmetric Ideal-Cache Model and Cache Policy

Recall from Section 2.1.3 the Ideal-Cache model [131] is a variant of the external-memory model. The machine model is organized in the same way with two memories each partitioned into blocks, but there are no explicit memory transfer instructions. Instead all addressable memory is in the large-memory, but any subset of up to M/B of the blocks can have a copy resident in the small-memory (cache). Any reference to a resident block is a *cache hit* and is free. Any reference to a word in a block that is not resident is a *cache miss* and requires a memory transfer from the large-memory. The cache miss can replace a block in the cache with the loaded block, which might require *evicting* a cache block. The model makes the *tall cache assumption* where $M = \Omega(B^2)$, which is easily met in practice. The I/O or *cache complexity* of an algorithm is the number of cache misses. An optimal (offline) cache eviction policy is assumed—i.e., one that minimizes the I/O

complexity. It is well known that the optimal policy can be approximated using the online least recently used (LRU) policy at a cost of at most doubling the number of misses, and doubling the cache size [257].

The main purpose of the Ideal-Cache model is for the design of *cache-oblivious algorithms*. These are algorithms that do not use the parameters M and B in their design, but for which one can still derive effective bounds on I/O complexity. This has the advantage that the algorithms work well for any cache sizes on any cache hierarchies. The I/O complexity of cache-oblivious sorting is asymptotically the same as for the EM model.

In order to distinguish the extra cost of writes from reads, the ***Asymmetric Ideal-Cache*** model is defined as follows. A cache block is *dirty* if the version in the cache has been modified since it was brought into the cache, and *clean* otherwise. When a cache miss evicts a clean block the cost is 1, but when evicting a dirty block the cost is $1 + \omega$, 1 for the read and ω for the write. Again, we assume an ideal offline cache replacement policy—i.e., minimizing the total I/O cost.

Under this model however, we note that the LRU policy is no longer 2-competitive comparing to the optimal offline policy. Consider a cache with $k = M/B$ blocks and a memory access pattern that repeatedly and sequentially writes to $k - 1$ blocks and read from other $k - 1$ blocks. An ideal cache policy will keep all $k - 1$ blocks associated to writes, so the I/O cost of each round is $k - 1$ for $k - 1$ read misses. An LRU policy however causes a cache miss for every single memory access, leading the I/O cost of each round to $\omega(k - 1) + k - 1$. This approximation is proportional to ω, which is problematic. Therefore a new variant of the cache replacement policy is required, which needs to be competitive within a constant factor.

The idea is to separately maintain two equal-sized pools of blocks in the cache (primary memory), a read pool and a write pool. When reading a location, (i) if its block is in the read pool we just read the value; (ii) if it is in the write pool we copy the block to the read pool; or (iii) if it is in neither, we read the block from secondary memory into the read pool. In the latter two cases we evict the LRU block from the read pool if it is full, with cost 1. The rules for the write pool are symmetric when writing to a memory location, but the eviction has cost $\omega + 1$ because the block is dirty. We call this the read-write LRU policy. This policy is competitive with the optimal offline policy:

Lemma 2.3.4. *For any sequence S of instructions, if it has cost $Q_I(S)$ on the Asymmetric Ideal-Cache model with cache size M_I, then it will have cost*

$$Q_L(S) \leq \frac{M_L}{M_L - M_I} Q_I(S) + (1 + \omega) M_I / B$$

on an asymmetric cache with read-write LRU policy and cache sizes (read and write pools) M_L.

Proof. Partition the sequence of instructions into regions that contain memory reads to exactly M_L/B distinct memory blocks each (except perhaps the last). Each region will

require at most M_L/B misses under LRU. Each will also require at least $(M_L - M_I)/B$ cache misses on the ideal cache since at most M_I/B blocks can be in the cache at the start of the region. The same argument can be made for writes, but in this case each operation involves evicting a dirty block. The $(1 + \omega)M_I/B$ is for the last region. To account for the last region, in the worst case at the start of the last write region the ideal cache starts with M_I/B blocks which get written to, while the LRU starts with none of those blocks. The LRU therefore invokes an addition M_I/B write misses each costing $1 + \omega$ (1 for the load and ω for the eviction). Note that if the cache starts empty then we do not have to add this term since an equal amount will be saved in the first round. □

We define the ***asymmetric I/O cost*** or ***asymmetric cache complexity*** to be the cost of an algorithm on the asymmetric ideal-cache model.

2.3.5 Asymmetric Low-depth Cache-Oblivious Paradigm

The last model considered in this thesis is based on developing low-depth cache-oblivious algorithms [58]. As discussed in Section 2.1.3, the cache complexity can be analyzed on the Ideal-Cache model under this sequential order.

Using known scheduling results the depth and sequential cache complexity of a computation are sufficient for deriving bounds on parallel cache complexity. In particular, let D be the depth and Q_1 be the sequential cache complexity. Then for a p-processor shared-memory machine with private caches (each processor has its own cache) using a work-stealing scheduler, the total number of misses Q_p across all processors is at most $Q_1 + O(pDM/B)$ with high probability [2]. For a p-processor shared-memory machine with a shared cache of size $M + pBD$ using a parallel-depth-first (PDF) scheduler, $Q_p \leq Q_1$ [50]. These bounds can be extended to multi-level hierarchies of private or shared caches, respectively [58]. Thus, algorithms with low depth have good parallel cache complexity.

The asymmetric variant of the low-depth cache-oblivious paradigm simply accounts for ω in the depth and uses the Asymmetric Ideal-Cache model for sequential cache complexity. We observe that the above scheduler bounds readily extend to this asymmetric setting. The $O(pDM/B)$ bound on the additional cache misses under work-stealing arises from an $O(pD)$ bound on the number of steals and the observation that each steal requires the stealer to incur $O(M/B)$ misses to "warm up" its cache. Pessimistically, we will charge $2M/B$ writes (and reads) for each steal, because each line may be dirty and need writing back before the stealer can read it into its cache and, once the stealer has completed the stolen work (reached the join corresponding to the fork that spawned the stolen work), the contents of its cache may need to be written back. Therefore for private caches we have $Q_P \leq Q_1 + O(\omega pDM/B)$. The PDF bounds extend because there are no additional cache misses and hence no additional reads or writes.

Chapter 3

Lower Bounds

We start by showing a variety of lower bounds on (M, ω)-ARAM of some fundamental problems including Fast Fourier Transform (FFT), sorting network, diamond DAGs, permuting and sorting. These lower bounds mainly focus on the number of I/Os to the large-memory (i.e., the ARAM cost). From a theoretical point of view, these lower bounds indicate the amount of improvement that can be made to the classic algorithms without considering the asymmetry between reads and writes. That further leads us to design algorithms that are asymptotically optimal on (M, ω)-ARAM and close the gaps. On the other hand, these lower bounds show the hardness of these problems, indicating that practically (i.e., considering the actual values of ω and M), we can only achieve a constant improvement unless $M = o(\omega)$, which seems to be unrealistic. A list of results are shown in Table 3.1.

In a high-level overview, we show that FFT and sorting network that have fixed computational DAG, the amount of improvement is limited to $\log(\omega M)/\log M$ (the previous bounds are listed in Table 2.1 and $B = 1$). For Diamond DAG the result is interesting: we show that there is no asymptotic improvement without allowing redundant computation, even if the reads are free! The proof techniques are to partition a computation into subcomputations that each have a lower bound on cost, but an upper bound on the number of inputs and outputs, which lower bound the costs to finish the computations of these problems.

Based on the model proposed in this thesis, Jacob and Sitchinava [182] recently show the lower bounds on permuting, sorting and sparse-matrix vector multiplication, and the results are also summarized in Table 3.1. In this case the block size B is also considered since the bounds are trivial otherwise. The outline of these results are overviewed in Section 3.5.

(a). The results in this thesis based on the (M, ω)-ARAM.

Problems	ARAM Cost (Q)	Work (W)	Remarks
FFT	$Q(n) = \Omega\left(\frac{\omega n \log n}{\log(\omega M)}\right)$	$W(n) = Q(n) + \Omega(n \log n)$	Section 3.2
Sorting Network	$Q(n) = \Omega\left(\frac{\omega n \log n}{\log(\omega M)}\right)$	$W(n) = Q(n) + \Omega(n \log n)$	Section 3.3
Diamond DAG	$Q(n) = \Omega\left(\frac{\omega n^2}{M}\right)$	$W(n) = Q(n) + \Omega(n^2)$	Section 3.4

(b). Later results from Jacob and Sitchinava's 2017 SPAA paper based on the (M, B, ω)-ARAM. On these problems, the block size B (full definition given in Section 2.1.2) is considered since the bounds are trivial otherwise.

Problems	ARAM Cost (Q)	Remarks
Permuting and sorting	$Q(n) = \Omega\left(\min\left\{n, \frac{\omega n \log(n/B)}{B \log(\omega M/B)}\right\}\right)$	Section 3.5
SparseMxV	$Q(n) = \Omega\left(\min\left\{h, \frac{\omega h \log(n/\max\{h/n, M\})}{B \log(\omega M/B)}\right\}\right)$	Section 3.5

Table 3.1: Summary of lower bounds on the (M, ω)-ARAM. In all cases n is the input size. In sparse-matrix vector multiplication (referred to as "SparseMxV" in the table), h is the number of non-zero elements in the matrix.

3.1 General Technique in the Proofs

To start with, we show the general techniques to show the lower bounds for Fast Fourier Transform (FFT) DAGs, sorting networks and diamond DAGs. The idea in showing the lower bounds is to partition a computation into subcomputations that each have a lower bound on cost, but an upper bound on the number of inputs and outputs they can use. Our lower bound for FFT DAGs then uses an interesting accounting technique that gives every node in the DAG a unit weight, and fractionally assigns this weight across the subcomputations. In the special case $\omega = 1$, this leads to a simpler proof for the lower bound on the I/O complexity of FFT DAGs than the well-known bound by Hong and Kung [170].

We refer to a *subcomputation* as any contiguous sequence of instructions. The *outputs* of a subcomputation are the values written by the subcomputation that are either an output of the full computation or read by a later subcomputation. Symmetrically, the *inputs* of a subcomputation are the values read by the subcomputation that are either

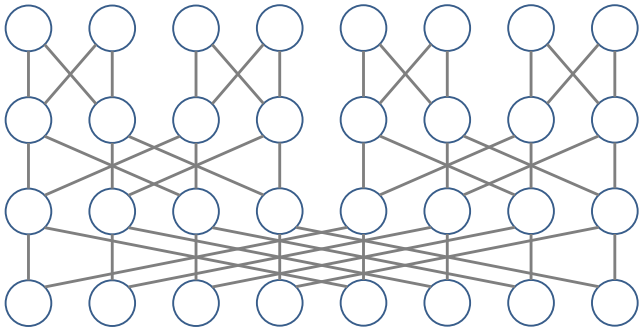


Figure 3.1: An example of the computational DAG of a FFT DAG (aka. butterfly networks) of size 8. The vertices in top row are the input and the bottom row is the output. This network has 3 levels, and in each level each vertex depends on the its own value in the previous level, as well as the vertex with index that reverses the i-th bit, where i is the number of level.

an input of the full computation or written by a previous subcomputation. The *space* of a computation or subcomputation is the number of memory locations both read and written. An (l, m)-*partitioning* of a computation is a partitioning of instructions into subcomputations such that each has at most l inputs and at most m outputs. We allow for recomputation—instructions in different subcomputations might compute the same value.

Lemma 3.1.1. *Any computation in the* (M, ω)*-ARAM has an* $((\omega + 1)M, 2M)$*-partitioning such that at most one of the subcomputations has ARAM cost* $Q < \omega M$*.*

Proof. We generate the partitioning constructively. Starting at the beginning, partition the instructions into contiguous blocks such that all but possibly the last block has cost $Q \geq \omega M$, but removing the last instruction from the block would have cost $Q < \omega M$. To remain within the cost bound each such subcomputation can read at most ωM values from large-memory. It can also read the at most M values that are in the small-memory when the subcomputation starts. Therefore it can read at most $(\omega + 1)M$ distinct values from the input or from previous subcomputations. Similarly, each subcomputation can write at most M values to large-memory, and an additional M that remain in small-memory when the subcomputation ends. Therefore it can write at most $2M$ distinct values that are available to later subcomputations or the output. □

3.2 Fast Fourier Transform (FFT)

We now consider lower bounds for the DAG computation problem for the family of FFT DAGs (also called FFT networks, or butterfly networks). The FFT DAG of input size $n = 2^k$ consists of $k + 1$ levels each with n vertices (total of $n \log 2n$ vertices). Each vertex (i, j) at level $i \in 0, \ldots, k - 1$ and row j has two out edges, which go to vertices $(i + 1, j)$ and $(i + 1, j \oplus 2^i)$ ($\oplus$ is the exclusive-or of the bit representation). An example of such DAG with $n = 8$ is shown in Figure 3.1. This is the DAG used by the standard FFT (Fast Fourier Transform) computation. We note that in the FFT DAG there is at most a single path from any vertex to another.

Lemma 3.2.1. *Any (l, m)-partitioning of a computation for simulating an n input FFT DAG has at least $n \log n/(m \log l)$ subcomputations.*

Proof. We refer to all vertices whose values are outputs of any subcomputation, as *partition output vertices*. We assign each such vertex arbitrarily to one of the subcomputations for which it is an output.

Consider the following accounting scheme for fractionally assigning a unit weight for each non-input vertex to some set of partition output vertices. If a vertex is a partition output vertex, then assign the weight to itself. Otherwise take the weight, divide it evenly between its two immediate descendants (out edges) in the FFT DAG, and recursively assign that weight to each. For example, for a vertex x that is not a partition output vertex, if an immediate descendant y is a partition output vertex, then y gets a weight of $1/2$ from x, but if not and one of y's immediate descendants z is, then z gets a weight of $1/4$ from x. Since each non-input vertex is fully assigned across some partition output vertices, the sum of the weights assigned across the partition output vertices exactly equals $|V| - n = n \log n$. We now argue that every partition output vertex can have at most $\log l$ weight assigned to it. Looking back from an output vertex we see a binary tree rooted at the output. If we follow each branch of the tree until we reach an input for the subcomputation, we get a tree with at most l leaves, since there are at most l inputs and at most a single path from every vertex to the output. The contribution of each vertex in the tree to the output is $1/2^i$, where i is its depth (the root is depth 0). The leaves (subcomputation inputs) are not included since they are partition output vertices themselves, or inputs to the whole computation, which we have excluded. By induction on the tree structure, the weight of that tree is maximized when it is perfectly balanced, which gives a total weight of $\log l$.

Therefore since every subcomputation can have at most m outputs, the total weight assigned to each subcomputation is at most $m \log l$. Since the total weight across all subcomputations is $n \log n$, the total number of subcomputations is at least $n \log n/(m \log l)$. □

Theorem 3.2.2 (FFT Lower Bound)**.** *Any solution to the DAG computation problem on the family of FFT DAGs parametrized by input size n has costs $Q(n) = \Omega\left(\frac{\omega n \log n}{\log(\omega M)}\right)$ and $W(n) = \Omega(Q(n) + n \log n)$ on the (M, ω)-ARAM.*

Proof. By Lemma 3.1.1 every computation must have an $((\omega + 1)M, 2M)$-partitioning with subcomputation cost $Q \geq \omega M$ (except perhaps one). Plugging in Lemma 3.2.1 we have $Q(n) \geq \omega M n \log n/(2M \log((\omega + 1)M))$, which gives our bound on $Q(n)$. For $W(n)$ we just add in the cost of the computation of each vertex. □

Note that whichever of ω and M is larger will dominate in the denominator of $Q(n)$. When $\omega \leq M$, these lower bounds match those for the standard external memory model [7,

170] assuming both reads and writes have cost ω. This implies that cheaper reads do not help asymptotically in this case. When $\omega > M$, however, there is a potential asymptotic advantage for the cheaper reads.

3.3 Sorting Networks

A sorting network is a acyclic network of comparators, each of which takes two input keys and returns the minimum of the keys on one output, and the maximum on the other. For a family of sorting networks parametrized by n, each network takes n inputs, has n ordered outputs, and when propagating the inputs to the outputs must place the keys in sorted order on the outputs. A sorting network can be modeled as a DAG in the obvious way. Ajtai, Komlós and Szemerédi [12] described a family of sorting networks that have size $O(n \log n)$ and depth $O(\log n)$. Their algorithm is complicated and the constants are very large. Many simplifications and constant factor improvements have been made, including the well known Patterson variant [227] and a simplification by Seiferas [250]. Recently Goodrich [148] gave a much simpler construction of an $O(n \log n)$ size network, but it requires polynomial depth. Here we show lower bounds of simulating any sorting network on the (M, ω)-ARAM.

Theorem 3.3.1 (Sorting Lower Bound)**.** *Simulating any family of sorting networks parametrized on input size n has $Q(n) = \Omega\left(\frac{\omega n \log n}{\log(\omega M)}\right)$ and $W(n) = \Omega(Q(n) + n \log n)$ on the (M, ω)-ARAM.*

Proof. Consider an (l, m)-partitioning of the computation. Each subcomputation has at most l inputs from the network, and m outputs for the network. The computation is oblivious to the values in the network (it can only place the min and max on the outputs of each comparator). Therefore locations of the inputs and outputs are fixed independent of input values. The total number of choices the subcomputation has is therefore $\binom{l}{m} m! = l!/(l-m)! < l^m$. Since there are $n!$ possible permutations, we have that the number of subcomputations k must satisfy $(l^m)^k \geq n!$. Taking logs of both sides, rearranging, and using Stirling's formula we have $k > \log(n!)/(m \log l) > \frac{1}{2} n \log n/(m \log l)$ (for $n > e^2$). By Lemma 3.1.1 we have $Q(n) > \omega M \frac{1}{2} n \log n/(2M \log((1+\omega)M)) = \frac{1}{4} \omega n \log n/\log((1+\omega)M)$ (for $n > e^2$). □

These bounds are the same as for simulating an FFT DAG, and, as with FFTs, they indicate that faster reads do not asymptotically affect the lower bound unless $\omega > M$. These lower bounds rely on the sort being done on a network, and in particular that the location of all read and writes are oblivious to the data itself. As discussed in the next section, for general comparison sorting algorithms, we can get better upper bounds than indicated by these lower bounds.

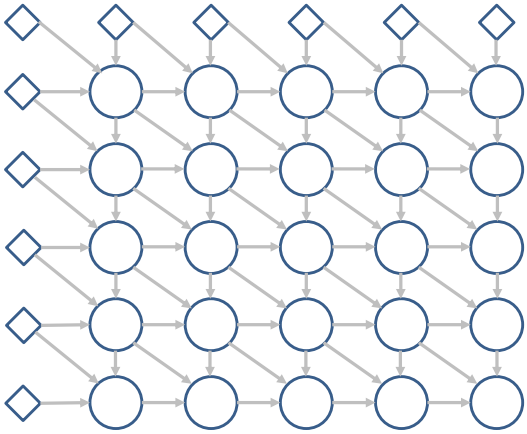


Figure 3.2: An example of the computational DAG of a diamond DAG in a 5×5 grid. The values in the circles need to be computed. The values in the left column and top row can be directly computed from the input, and each other value depends on its top, left, and top-left neighbors. The bottom-right value is the output.

3.4 Diamond DAG

We consider the family of *diamond DAGs* parametrized on size n. Each DAG has n^2 vertices arranged in a $n \times n$ grid such that every vertex $(i, j), 0 \leq i < (n-1), 0 \leq j < (n-1)$ has two out-edges to $(i+1, j)$ and $(i, j+1)$. The DAG has one input at $(0, 0)$ and one output at $(n-1, n-1)$. The family of 3-diamond DAGs additionally have edges from (i, j) to $(i+1, j+1)$. An example of such diamond DAG is given in Figure 3.2 as an example. Diamond DAGs have many applications in dynamic programs, such as for the edit distance (ED), longest common subsequence (LCS), and optimal sequence alignment problems.

Lemma 3.4.1 (Cook and Sethi, 1976)**.** *Solving the DAG computation problem on the family of diamond DAGs of input parameter n (size $n \times n$) requires n space to store vertex values from the DAG.*

Proof. Cook and Sethi [106] show that evaluating the top half of a diamond DAG ($i + j \geq n - 1$) , which they call a pyramid DAG, requires n space to store partial results. Since all paths of the diamond DAG must go through the top half, it follows for the diamond DAG. □

Theorem 3.4.2 (Diamond DAG Lower Bound)**.** *The family of diamond DAGs parametrized on input size n has $Q(n) = \Omega\left(\frac{\omega n^2}{M}\right)$ and $W(n) = \Omega(Q(n) + n^2)$ on the (M, ω)-ARAM.*

Proof. Consider the sub-DAG induced by a $2M \times 2M$ diamond ($a \leq i < a + 2M, b \leq j < b + 2M$) of vertices. By Lemma 3.4.1 any subcomputation that computes the last output vertex of the sub-DAG requires $2M$ memory to store values from the diamond. The extra in-edges along two sides and out-edges along the other two can only make the problem harder. Half of the $2M$ required memory can be from small-memory, so the remaining M must require writing those values to large-memory. Therefore every $2M \times 2M$ diamond requires M writes of values within the diamond. Partitioning the full diamond DAG into

$2M \times 2M$ sub-diamonds, gives us $n^2/(2M)^2$ partitions. Therefore the total number of writes is at least $M \times \frac{n^2}{(2M)^2} = \frac{n^2}{4M}$, each with cost ω. For the work we need to add the n^2 calculations for all vertex values. □

This lower bound is asymptotically tight since a diamond DAG can be evaluated with matching upper bounds by evaluating each $M/2 \times M/2$ diamond sub-DAG as a subcomputation with M inputs, outputs and memory.

These bounds show that for the DAG computation problem on the family of diamond DAGs there is no asymptotic advantage of having cheaper reads. In Section 4.5 we show that for the ED and LCS problems (normally thought of as a diamond DAG computation), it is possible to do better than the lower bounds. This requires breaking the DAG computation rule by partially computing the values of each vertex before all inputs are ready. The lower bounds are interesting since they show that improving asymptotic performance with cheaper reads requires breaking the DAG computation rules.

3.5 Results from Jacob and Sitchinava

Jacob and Sitchinava [182] recently show the lower bounds on permuting, sorting and sparse-matrix vector multiplication, and the results are summarized in Table 3.1. In this series of work, the block size B is considered as a non-constant since the bounds are trivial otherwise. Here we briefly overview the outline of their proofs.

The key challenges here is that we no longer have a fixed computational DAG. Also, the same cost can be incurred by different combinations of reads and writes. To solve this, the proof analyzes batch reads or writes (referred to a "round") such that the total ARAM cost of each round is at least $\omega M/B$, and the small-memory needs to be emptied between rounds.

The proof on permuting is based on a counting argument. This also gives the a lower bound on sorting consequently since sorting is strictly harder.

Assume any algorithm requires R rounds to finish. The algorithm must distinguish all $n!$ possible inputs, leading to

$$\frac{n!}{B!^{n/B}} \leq P(R) \leq \left(\binom{n}{\omega M/B} \binom{\omega M}{M} 2^M \frac{M!}{B!^{M/B}} (3N)^{M/B} \right)^R$$

$P(R)$ represents the number of "block-wise" permutations that can be generated after R rounds. On the right of the inequality, $\binom{n}{\omega M/B}$ is the number of choices of $\omega M/B$ blocks to read out of the inputs. $\binom{\omega M}{M} 2^M$ is the number of ways to choose up to M items to keep in the memory out of the ωM reads from previous. Then block-wisely permuting M items in the small-memory has $M!/(B!^{M/B})$ cases. Finally since the memory are emptied between rounds, we can write back the M/B blocks into the large-memory with cost $\omega M/B$, and

the number of options is no more than $(3N)^{M/B}$. The whole term on the right side upper bounds the number of permutations that can be identified after R rounds.

For the left term, $n!$ is the number of inputs that the algorithm needs to identify, while $B!^{n/B}$ is the number of permutations that one block can add up to. This lower bounds the number of the block-wise permutations that have to be distinguished by a permuting algorithm.

Solving this inequation gives the lower bound of R. Since the ARAM cost of each round is at least $\omega M/B$, the lower bound of permuting and sorting is thus $\Omega(\min\{n, \omega n/B \log_{\omega M/B} n/B\})$.

A lower bound on computing the product of a sparse matrix by a dense vector can be analyzed similarly, using the round-based computation and counting argument. The bound is listed in Table 3.1. The full details of the proof techniques in this section are referred to [182].

Chapter 4

Basic Algorithmic Building Blocks

To start the write-efficient algorithms, we first introduce the basic building blocks that are widely used in designing other algorithms. Therefore, when designing more complicated and advanced algorithms, these building blocks can be directly used, just like in the symmetric setting.

We start by showing that a variety of algorithms that are reasonably easy and optimal in Section 4.1. These algorithms include fast Fourier transform (FFT), search trees and priority queues, and sorting. Some other algorithms like BFS/DFS or planar convex hull and Delaunay triangulation will be described later in the corresponding chapters.

Then later in Section 4.2 and 4.3, we discuss four commonly-used parallel primitives, i.e., reduce, filter, list and tree contractions, and their upper bounds on the Asymmetric NP model. They are intensively used in the rest of the thesis to design more advanced parallel write-efficient algorithms.

Sorting, as one of the most important algorithmic primitives, are discussed separately in Section 4.4. We show various algorithms with optimal bounds on (M, ω)-ARAM, (M, B, ω)-ARAM, and the asymmetric ideal-cache model.

Lastly, we discuss an interesting dynamic programming algorithm to compute edit distance and longest common subsequence of two input strings. In Section 3.4 we show that without any redundant computation, we cannot decrease the number of writes. Here we show that the ARAM cost can be improved by a factor of $\omega^{1/3}$, if we are willing to pay a factor of $O(\omega^{2/3})$ extra work. It is an interesting open problem on whether this ARAM cost is optimal, if work T remains polynomial.

4.1 Primitives on (M, ω)-ARAM Model

Search Trees and Priority Queues. We now consider algorithms for some problems that can be implemented efficiently using balanced binary search trees. In the following discussion we assume $M = O(1)$. Red-black trees with appropriate rebalancing rules

require only $O(1)$ amortized work per update (insertion or deletion) once the location for the key is found [267]. For a tree of size n finding a key's location uses $O(\log n)$ reads but no writes, so the total amortized cost $Q = W = O(\omega + \log n)$ per update in the (M, ω)-ARAM. For arbitrary sequences of searches and updates, $\Omega(\omega + \log n)$ is a matching lower bound on the amortized cost per operation when $M = O(1)$. Since priority queues can be implemented with a binary search tree, insertion and delete-min have the same bounds. It seems more difficult, however, to reduce the number of writes for priority queues that support efficient melding or decrease-key.

Sorting. Sorting can be implemented with $Q = W = O(n(\log n + \omega))$ by inserting all keys into a red-black tree and then reading them off in priority order. We note that this bound on work is better than the sorting network lower bound (Theorem 3.3.1). For example, when $\omega = M = \log n$ it gives a factor of $\log n / \log \log n$ improvement. The additional power is a consequence of being able to randomly write to one of n locations at the leaves of the tree for each insertion. The bound is optimal for W since n writes are required for the output and comparison-based sorting requires $O(n \log n)$ operations.

FFT. For the FFT, the idea is to first split the butterfly DAG into layers of $\log(\omega M)$ levels. Then divide each layer so that the last $\log M$ levels are partitioned into FFT networks of output size M. Attach to each partition all needed inputs from the layer and the vertices needed to reach them (note that these vertices will overlap among partitions). Each extended partition will have ωM inputs and M outputs, and can be computed in M small-memory with $Q = O(\omega M)$, and $W = O((\omega + \log M)M)$. This gives a total upper bound $Q = O(\omega M \cdot n \log n / (M \log(\omega M))) = O(\omega n \log n / \log(\omega M))$, and $W = O(Q(n) + n \log n)$, which matches the lower bound (asymptotically). All computations are done within the DAG model.

4.2 Parallel primitives on Asymmetric NP Model

In this section we consider several basic parallel primitives on the Asymmetric NP model. We note that some primitives inherently require as many writes as reads. For example, all prefix sums needs to write out all the sums. However, when they are used as a step for some other purpose, then the final generation of the values can be folded into whatever needs the values. This idea is used in the output sensitive filter described below.

4.2.1 Reduce

Summing a sequence of values with respect to an associative function $f(x, y)$ is surely the most common parallel function. Of course it only requires a single result so it should be possible to make it write efficient. In the Asymmetric NP there are two methods to do this. The first is simply to use a divide-and-conquer algorithm that recurses on the two halves in parallel, and when they return add the two results. The base case can either be a single element or $O(\log n)$ elements, and then sum those elements sequentially. A

fork can be used to generate two child tasks for the calls. The interesting feature of our model is that this will only require a single write, which is to write the final answer. All other computation can be done in the constant space per task stack space. This may seem impossible since the processors need to communicate. Recall, however, that when we simulate the Asymmetric NP (with binary branching) on a machine, we account for the steals in the cost. These steals are communicating the values among processors. The divide-and-conquer algorithm leads to the following result.

Lemma 4.2.1. *The reduction of n elements can be done in $\Theta(n+\omega)$ work and $\Theta(\log n)$ depth using $\Theta(1)$ writes on the Asymmetric NP model.*

A second way to do a reduce is with bulk-synchronous steps. The first step forks n/ω tasks, each of which sums ω values and writes its sum to large memory. This can be repeated $\log_\omega n$ times to get the sum. The resulting algorithm does $\Theta(n + \omega)$ work and has $\Theta(\omega \log_\omega n)$ depth.

4.2.2 Output-Sensitive Ordered Filter

Given an array A of size n and a predicate ρ on the elements of A, we want to filter out the entries $x \in A$ for which $\rho(x) = 0$, and get a new array A' of size $k \leq n$, where k is the number of elements $x \in A$ for which $\rho(x) = 1$ and A' preserves the relative order of the elements in A. In the classic nested-parallel model, this is easily done in linear work and logarithmic depth by first creating a new array that holds the result of applying ρ to each element, computing a prefix sum on that array to determine the position in A' of each element to be moved, and then moving those elements. However, this requires $\Theta(n)$ writes. In the Asymmetric NP model, this number of writes can be problematic; there exist algorithms where the number of writes depends on k rather than n except for the filtering step.

Our goal is an (output-sensitive) ordered filter algorithm whose writes are proportional to the output size rather than the input size, thereby matching the lower bound on writes. We show the following result:

Lemma 4.2.2. *Ordered filter on an array of size n such that k elements satisfy the given predicate can be done in $\Theta(n + \omega k)$ work and $O(\log n)$ depth using $\Theta(k)$ writes* whp *on the Asymmetric NP model.*

Our algorithm proceeds as follows: we first apply a REDUCE operation with ρ to find k, and allocate an array B of size $O(k)$. Recall that in our model, a reduce operation takes only $O(1)$ writes.

If $k \leq n/\log_2 n$, then we hash the k entries, along with their indices in A into B. This takes only $O(k)$ writes and can be done efficiently in parallel, since we expect few collisions. We then sort the array by the original indices to get an array A' of size k that preserves the elements' relative order in the input array. Because there are less than

$n/\log_2 n$ entries, we can sort them in $O(n + \omega k)$ work and $O(\log n)$ depth *whp*, using a write-efficient parallel sorting algorithm (Section 4.4.1).

If $k > n/\log_2 n$, we divide the array into k equal parts, and then in parallel process each part sequentially. This sequential filtering need only write each non-filtered element once, and it takes $O(\log n)$ depth since each part is small. Finally, we concatenate the results to get our output array. This concatenation can be done in $O(\log n)$ depth by counting the elements in each part and using a prefix sum to find starting indices for each part's output in the final array. Since the prefix sum is only applied to the set of k parts, it uses only $O(k)$ writes.

4.3 List and Tree Contraction on Asymmetric NP Model

In this section we introduce efficient parallel algorithms for list and tree contraction on the Asymmetric NP Model that reduce the number of writes without increasing the reads. In both problems, our goal is to divide up the problem into sub-problems that can be processed in parallel. We say that an ***equal partition*** of a structure of size n is a partition into $O(s)$ contiguous parts, each of which is of size $O(n/s)$ for some $s \leq n$. For example, the partition based on m-critical nodes in [138] is an equal partition.

The list contraction algorithm we present is relatively simple, and uses a common approach to partition the problem size using random samples. Tree contraction however, becomes more challenging when we limit the writes to main memory. We are not aware of any existing tree contraction algorithms (even sequential ones) that solve the problem using bounded local memory. Furthermore, designing a parallel version is hard because we cannot explicitly record information on the tree nodes (this would require too many writes).

Previous parallel tree contraction algorithms use either a top-down approach [183, 215, 217, 254] or a bottom-up approach [138]. All of them require a linear number of writes, and we are unaware of any non-trivial modifications to these algorithms that can reduce the number of writes to $o(n)$. To solve this problem, our algorithm a uses a bottom-up approach. It is based on a new tree-partition algorithm that, instead of partitioning the tree based on sub-tree sizes as done in [138], uses the Euler tour as a tool to refine our partition after randomly selecting sample nodes. Then we prove that the partition we get is an equal partition using a surprisingly simple argument. With this partitioning of the tree, we can combine a sequential space-bounded tree contraction algorithm that we describe with any of the existing tree contraction algorithms to obtain a write-efficient parallel tree contraction algorithm in the Asymmetric NP model.

4.3.1 List Contraction

A linked list is a list of nodes in which each node has a pointer to the next node in the list. A segment on the linked list is defined as the elements between two given nodes. The

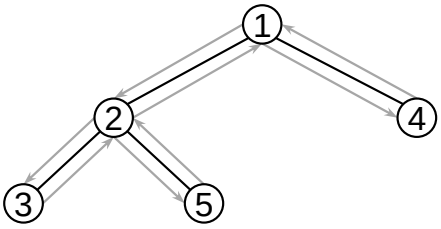


Figure 4.1: The Euler tour of this rooted binary tree is (1, 2, 3, 2, 5, 2, 1, 4, 1), obtained by following the arrows on the edges starting at the root node 1.

list contraction problem is to contract a linked list of length n into a single node (possibly combining values). It has many applications, including list ranking and Euler tours [183]. Sequentially, we can just loop over all nodes by following the pointers which takes a linear number of reads and work, and a constant number of writes to main memory. In the symmetric setting ($\omega = 1$), the standard parallel approach using random mate [21] requires $O(\log n)$ depth and linear reads and writes.

Our algorithm partitions the list in two steps. In the first step, each element in the list is randomly marked with probability s/n for some parameter s. Then in the second step, we start with each marked node, and in parallel, follow the list with the pointers and mark every $\lfloor n/s \rfloor$'th element. The longest chain we have to follow here has length $O((n \log n)/s)$ *whp*, which can be shown using a Chernoff bound. Now clearly all segments between two consecutively marked nodes have size no more than $\lfloor n/s \rfloor$. With the marked nodes, we contract all segments in parallel, with each one done sequentially (terminating when the next marked node is encountered). After that, a standard symmetric version of parallel list contraction is applied on the marked nodes. Each step of the algorithm performs $O(s)$ writes.

The new list contraction algorithm takes linear work, $O((n \log n)/s)$ depth, and $O(s)$ writes in the Asymmetric NP model. This algorithm is efficient when running on a share-memory machine with $p = O(n/(\omega \log n))$ cores. In this case we can just plug in $s = n/\omega$ and get the bounds shown in the theorem. The list partition routine described above is used as a subroutine in our tree contraction algorithm described next.

Theorem 4.3.1. *List contraction of size n can be computed using $O(n)$ work, $O(\omega \log n)$ depth and $O(n/\omega)$ writes with $O(1)$ local memory in the Asymmetric NP model.*

4.3.2 Tree Contraction

The tree contraction problem is to contract a tree with n nodes into a single node (possibly combining node values), and has many applications in parallel computing [183, 215, 217]. We assume that the input is a rooted binary tree and each tree node has pointers to its parent, left child, and right child (if they exist). When the tree is viewed as a directed graph that contains two directed edges for each edge in the tree, the Euler tour [265] of

Algorithm 1: A parallel tree partitioning algorithm

Input: A rooted binary tree T
Output: $O(s)$ partition nodes

1 Apply the parallel list partitioning algorithm on the Euler tour of tree T and mark no more than $O(s)$ tree nodes such that each sublist has length less than n/s
2 **foreach** marked node v **do**
3 Traverse the Euler tour from the position of v's last appearance to the next marked node v', and mark the highest node in this range
4 **return** all marked nodes

the tree is an Eulerian circuit of the directed graph (see Figure 5.2 for an example). It can be constructed implicitly: given the current node and the previous edge, we can check whether the previous edge is from the parent or child of the current node, and according to this information, decide which edge to take next. We define a ***component*** of a tree to be a set of tree nodes that are connected. A subtree is a component, but not vice versa.

In the symmetric setting ($\omega = 1$) doing tree contraction sequentially in linear work is trivial, and classic parallel tree contraction algorithms [138, 183, 215, 217] take $O(\log n)$ depth and $O(n)$ writes. However, many applications, such as arithmetic expression evaluation and subtree size queries on a set of tree nodes, have a sublinear output size. Thus, a natural question to ask is whether we can design a parallel tree contraction algorithm with a sublinear number of writes. We describe such an algorithm in this section.

A new tree partitioning algorithm.

The goal of this algorithm is to find $O(s)$ partition nodes such that each tree component has size at most n/s. The high-level idea of the algorithm is to compute an equal partition of the Euler tour and mark the lowest common ancestor of each component as a partition node. The pseudocode is provided in Algorithm 1, and we explain the details below.

Since we cannot afford to store information per node, our solution is to run the parallel list partitioning algorithm described in Section 4.3.1 on the Euler tour of the tree. However, generating the Euler tour is too expensive, as it would require a linear number of writes. Thus, we need to simulate the parallel list partitioning algorithm. We do so as follows. First, we randomly sample $O(s)$ nodes on the tree. Then, from each sampled tree node, we follow the Euler tour in every direction in parallel. In each direction, we mark every $\lfloor n/(3s) \rfloor$'th element along the path, until we reach the next sampled node on the Euler tour. This step is similar in the list partitioning algorithm, but each node may correspond to multiple (up to three) different locations in the Euler tour, so we traverse the list from all different locations. In expectation, the distance between two samples will be at most $O((n/s) \log n)$ *whp*. We will mark another $O(s)$ nodes during the second step. This process corresponds to Line 1 of the pseudocode.

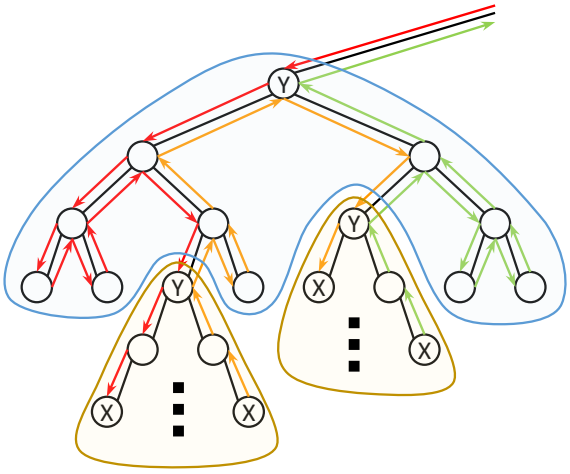


Figure 4.2: An example of how the tree is partitioned into components. A component consists of tree nodes from at most three segments of the Euler tour.

With the $O(s)$ marked nodes, the algorithm now finds and marks $O(s)$ ***partition nodes***, which corresponds to Lines 2–4 of Algorithm 1. A partition node is the highest node (closest to the root) in a tree component consisting of tree nodes in a segment of the Euler tour. For every segment that starts with the first or second appearance of a marked interior node v, the partition node is just the first node in the segment, because the segment cannot go beyond the subtree of v and it terminates no later than the next appearance of v in the Euler tour. Hence, we need to do nothing for these segments since the partition nodes are already marked. We only consider the segments that start with the last appearances of marked nodes. The partition node of each such segment can be computed with constant space by traversing through the segment: we always maintain a pointer p to the highest node so far in the traversal. We start by pointing at the first node in the segment, and whenever we go from our current top node to its parent in the traversal, we update p to point to its parent as well. When the traversal finishes, the node pointed by p is marked. It is easy to see that this marked node is the highest node in the segment.

The partition nodes provide a partitioned tree with $O(s)$ components if we form the tree components by ignoring edges from all partition nodes to their parents. We now show that this partition is an equal partition.

Lemma 4.3.2. *The marked nodes generated in Algorithm 1 partition the tree such that each tree component contains at most n/s tree nodes.*

Proof. We prove this by showing that each component consists of tree nodes from at most three segments of the Euler tour, which is shown in Figure 4.2. Recall that in the

algorithm, we marked every $\lfloor n/3s \rfloor$'th element in the Euler tour, so showing this suffices to prove our claim.

In Figure 4.2, nodes marked by the list partitioning algorithm are shown with the letter "X", and the newly-added partition nodes corresponding to the highest nodes in each segment are marked with "Y". Both types of nodes are considered *marked nodes*. We claim that for each tree component rooted by either an X node or a Y node (e.g., the blue region in Figure 4.2), there exist at most two subtrees (e.g., the two yellow regions in Figure 4.2) that contain marked nodes and are rooted at nodes that are direct children of this component. More specifically, there will be at most one such subtree in each of the left and right subtrees of the root node (we will refer to these two subtrees as *left-side and right-side marked subtrees*). The segment from the last X node in the left subtree (if it exists) on the Euler tour to the first X node in the right subtree (if it exists) marks the root of this component. To see that there is at most one subtree on each side, assume for the sake of contradiction that there are two marked nodes on the same side of the root but not in the same marked subtree (yellow region). Then the lowest common ancestor of these two nodes will be marked as a Y node. This node is assumed to be in this component (the blue region) but actually it will be marked and removed from this component, which leads to a contradiction.

Hence, each component only consists of nodes from at most three segments from the Euler tour. The first segment is the one that enters this component in the Euler tour, and ends at the first X node (based on the Euler tour) in the left-side marked subtree, which is shown as the red arrows in Figure 4.2. The second segment is illustrated as the orange arrows, starting from the last X node in the left-side marked subtree and ending in the first X node in the right-side marked subtree. The last segment, shown as the green arrows, is symmetric to the first segment and leaves this part from the last X node. If there are no marked node in the left side, then the first and second segments are merged into one, and similarly on the right side. Thus each component consists of vertices from at most three segments of the Euler tour.

Since we can guarantee that each segment has size no more than $n/3s$, each tree component contains no more than n/s nodes. □

The overall cost is the sum of the cost of partitioning the Euler tour and the cost of traversing a subset of the segments (starting for the up-edge of each X node). This takes linear work, $O(s)$ writes and $O((n/s)\log n)$ depth *whp*.

Remark: Notice that our tree partition is done by removing (actually ignoring) edges but not vertices (as done in [138]) since this is more efficient on both work and writes in practice. This algorithm also works on all constant-degree trees. However, for an arbitrary tree our algorithm would not work. In fact, if the given tree is a star, then there does not exist any equal partition of non-constant ($\omega(1)$) size, which means that no top-down approaches by tree partitioning will give a write-efficient solution in this case. Therefore,

a preprocessing step that converts the input to a binary tree with linear writes is required before running our algorithm, but if the tree contraction is run multiple times on a given tree, there is still an advantage in using this algorithm.

A sequential algorithm.

We now discuss a sequential and space-bounded tree contraction algorithm that will be used as a subroutine in our parallel algorithm. Previous tree contraction algorithms [138, 215, 217, 254] take either linear space or linear writes to the main memory, which is too costly in the asymmetric setting. Instead we would like to design an algorithm that uses a small amount of local (small) memory and performs no writes to main memory. The tool we use is the tree partitioning algorithm discussed in the previous section, which can be used to partition a tree into components of size no more than n/s using $O(s)$ writes. Our algorithm then contracts each tree component down to a single node, and after that we apply a standard tree contraction algorithm on the marked nodes. To restrict the number of components as well as the size of the components so that all intermediate results fit into a small memory and require no writes, the we recursively apply the tree partitioning algorithm until each component fits in small memory. Suppose that the cutoff size for the base case of the recursion is c. Then the required small memory size is $O(s \log_s(n/c) + c)$, and the work and number of reads is $O(n \log_s(n/c))$. By setting s to either $O(\epsilon n^\epsilon)$ or some constant greater than 1, we obtain the following lemma.

Lemma 4.3.3. *The sequential algorithm presented above contracts a tree of size n requiring $O(1)$ writes, and using a small memory of size $O(n^\epsilon)$ and linear work, where $0 < \epsilon < 1$, or a small memory of size $O(\log n)$ and $O(n \log n)$ work and reads.*

A parallel algorithm.

We now describe a parallel tree-contraction algorithm that uses the tree partitioning algorithm and sequential tree contraction algorithm as subroutines.

The high-level idea for the parallel algorithm is to first partition the tree into small, almost equal-sized components using the tree partitioning algorithm, and then contract each component independently in parallel using the sequential contraction algorithm. Finally, we use a standard parallel tree contraction algorithm to contract the remaining nodes. This step requires a number of writes proportional to the number of remaining nodes, which is much smaller than the original tree size. The algorithm takes s as a parameter and consists of three steps as described below.

Step 1: Tree Partitioning. The first part of the algorithm computes a tree partition such that each component has size $O(n/s)$, where s is a parameter of the algorithm. This step requires $O(n)$ work, $O((n \log n)/s)$ depth *whp* and $O(s)$ writes, if implemented by Algorithm 1.

Step 2: Tree Contraction on Components. With the partition nodes computed in Step 1, we now have a set of components each with size at most $O(n/s)$. Since the

tree components are themselves trees, we can apply our sequential tree contraction algorithm on each of them. The contraction is restricted to be inside each component by not contracting any partition node. Also, we leave the root node of each component uncontracted. After contraction, each component's root has at most one left and one right child, which is either the marked child (the Y nodes in the yellow regions in Figure 4.2) or the a descendant of the root that is in the component (if no marked child on the side). Hence, if the local memory has size $O((n/s)^\epsilon)$, this step takes linear time and $O(n/s)$ depth, and yields an intermediate contracted tree with no more than $O(s)$ nodes.

Step 3: Contraction of Remaining Nodes. In the last step, we apply any existing parallel tree contraction algorithm with linear work and logarithmic depth to contract the tree generated from the last step to a single node. This step costs $O(s)$ reads, writes and work, and $O(\log n)$ depth.

To reduce the number of writes without increasing the asymptotic work complexity, we choose $s = n/\omega$. Hence, the size of the components is at most ω. The following theorem gives the cost of our parallel tree contraction algorithm.

Theorem 4.3.4. *Algorithm 1 can be used to contract a tree with size n using $O(n)$ work, $O(\omega \log n)$ depth and $O(n/\omega)$ writes with $O(\omega^\epsilon)$ local memory in the Asymmetric NP model.*

4.4 Sorting

For sorting algorithm, we first observe that in the RAM model, it is well known that sorting by inserting each key into a balanced search tree requires only $O(n)$ writes with no increase in reads ($O(n \log n)$). However, applying the idea to the Asymmetric NP model is non-trivial, since the well-known parallel sorting algorithms with $O(\log n)$ depth, like Cole's mergesort [102], do not fit into the two framework we discussed in Section 2.3.2. Instead, we extend the parallel sorting algorithm in [68] into a write-efficient manner, and yield an algorithm with $O(n)$ writes, $O(n \log n)$ reads and $O(\log n)$ depth *whp* assuming priority writes (Section 4.4.2).

Since sorting is one of the most fundamental algorithm and widely used as a subroutine in many other algorithms and frameworks, we also consider it on the (M, B, ω)-ARAM (considering the block size B). We show that three asymptotically-optimal sorting algorithms can each be adapted to the (M, B, ω)-ARAM with reduced write costs. First, following [226, 273], we adapt multi-way mergesort by merging $\omega M/B$ sorted runs at a time (instead of M/B as in the original EM version). This change saves writes by reducing the depth of the recursion. Each merge makes ω passes over the runs, using an in-memory heap to extract values for the output run for the pass. Our algorithm and analysis is somewhat simpler than [226, 273]. Second, we present a sample sort algorithm that uses $\omega M/B$ splitters at each level of recursion (instead of M/B in the original EM version). Again, the challenge is to both find the splitters and partition using them while incurring only $O(N/B)$ writes across each level of recursion. Finally, our third sorting algorithm is a

Algorithm 2: Asymmetric-NP Sort

Input: An array of records A of length n
Output: The sorted array

1 Select a sample S from A independently at random with per-record probability $1/\log n$, and sort the sample
2 Use every $(\log n)$-th element in the sorted S as splitters, and for each of the about $n/\log^2 n$ buckets defined by the splitters allocate an array of size $c \log^2 n$
3 In parallel locate each record's bucket using a binary search on the splitters
4 In parallel insert the records into their buckets by repeatedly trying a random position within the associated array and attempting to insert if empty
5 Pack out all empty cells in the arrays and concatenate all arrays
// Step 6 is an optional step used to obtain $O(\omega \log n)$ depth
6 **for** round $r \leftarrow 1$ to 2 **do**
7 | **foreach** array A' generated in previous round (in parallel) **do**
8 | | Deterministically select $|A'|^{1/3} - 1$ samples as splitters and apply integer sort on the bucket number to partition A' into $|A'|^{1/3}$ sub-arrays
9 **foreach** subarray from the previous round **do**
10 | Apply the (M, ω)-ARAM sort
11 **return** the sorted array

heapsort using a buffer-tree-based priority queue. Compared to the original EM algorithm, both our buffer-tree nodes and the number of elements stored outside the buffer tree are larger by a factor of ω, which adds nontrivial changes to the data structure. All three sorting algorithms have the same asymptotic complexity on the (M, B, ω)-ARAM.

4.4.1 Sorting on (M, ω)-ARAM

The number of writes on an (M, ω)-ARAM can be bound for a variety of algorithms and data structures using known techniques. For example, based on the analysis of balanced binary search trees in the previous sections, many trees only require a constant number of rotations per insertion and deletion. Sorting can be done by inserting n records into a balanced search tree data structure, and then reading them off in order. This requires $O(n \log n)$ reads and $O(n)$ writes, for total cost $O(n(\omega + \log n))$. Similarly, we can maintain priority queues (insert and delete-min) and comparison-based dictionaries (insert, delete and search) in $O(1)$ writes per operation.

4.4.2 Sorting on Asymmetric NP Model

We now consider how to sort on the Asymmetric NP model allowing arbitrary write. Algorithm 2 outlines a sample sort (with over-sampling) that does $O(n \log n + \omega n)$ work

($O(n \log n)$ reads and $O(n)$ writes) and has depth $O\left(\log^2 n\right)$. It is similar to other sample sorts [56, 127, 183]. We consider each step in more detail and analyze its cost.

Step 1 can use Cole's parallel mergesort [102] requiring $O(n)$ reads and writes *whp* (because the sample is size $\Theta(n/\log n)$ *whp*), and $O(\omega \log n)$ depth. In step 2 for sufficiently large c, *whp* all arrays will have at least twice as many slots as there are records belonging to the associated bucket [56]. The cost of step 2 is a lower-order term. Step 3 requires $O(n \log n)$ reads, $O(n)$ writes and $O(\log n)$ depth for the binary searches and writing the resulting bucket numbers. Step 4 is an instance of the so-called placement problem (see [236, 239]). This can be implementing by having each record select a random location within the array associated with its bucket and if empty, attempting to insert the record at that location. This is repeated if unsuccessful. Since multiple records might try the same location at the same time, each record needs to check if it was successfully inserted. The expected number of tries per record is constant. Also, if the records are partitioned into groups of size $\log n$ and processed sequentially within the group and in parallel across groups, then *whp* no group will require more than $O(\log n)$ tries across all of its records [236]. Therefore, *whp* the number of reads and writes for this step are $O(n)$ and the depth is $O(\log n)$. Step 5 can be done with a prefix sum, requiring a linear number of reads and writes, and $O(\log n)$ depth. At this point we could apply the asymmetric RAM sort to each bucket giving a total of $O(n \log n)$ reads, $O(n)$ writes and a depth of $O\left(\log^2 n \log\log n\right)$ *whp*.

We can reduce the depth to $O(\log n)$ by further deterministically sampling inside each bucket (step 6) using the following lemma:

Lemma 4.4.1. *We can partition m records into $m^{1/3}$ buckets $M_1, \ldots, M_{m^{1/3}}$ such that for any i and j where $i < j$ all records in M_i are less than all records in M_j, and for all i, $|M_i| < m^{2/3} \log m$. The process requires $O(m \log m)$ reads, $O(m)$ writes, and $O\left(\omega\sqrt{m}\right)$ depth.*

Proof. We first split the m records into groups of size $m^{1/3}$ and sort each group with the RAM sort. This takes $O(m \log m)$ reads, $O(m)$ writes and $O\left(m^{1/3} \log m\right)$ depth. Then for each sorted group, we place every $\log m$'th record into a sample. Now we sort the sample of size $m/\log m$ using Cole's mergesort, and use the result as splitters to partition the remaining records into buckets. Finally, we place the records into their respective buckets by integer sorting the records based on their bucket number. This can be done with a parallel radix sort in a linear number of reads/writes and $O\left(\sqrt{m}\right)$ depth [236].

To show that the largest bucket has size at most $m^{2/3} \log m$, note that in each bucket, we can pick at most $\log m$ consecutive records from each of the $m^{2/3}$ groups without picking a splitter. Otherwise there will be a splitter in the bucket, which is a contradiction. □

Step 6 applies two iterations of Lemma 4.4.1 to each bucket to partition it into sub-buckets. For an initial bucket of size m, this process will create sub-buckets of at most size $O\left(m^{4/9} \log^{5/3} m\right)$. Plugging in $m = O\left(\log^2 n\right)$ gives us that the largest sub-bucket is of

size $O\left(\log^{8/9} n(\log\log n)^{5/3}\right)$. We can now apply the RAM sort to each bucket in $O(\log n)$ depth. This gives us the following theorem.

Theorem 4.4.2. *Sorting n records can be performed using $O(n \log n + \omega n)$ work ($O(n \log n)$ reads and $O(n)$ writes), and in $O(\log^2 n)$ depth* whp *on the Asymmetric NP model allowing concurrent writes.*

Reducing the depth to $O(\log n)$. The depth can be reduced to $O(\log n)$ by combining the result of the sorting algorithm by Jeremy Fineman, which can sort n elements in $O(n \log n)$ reads and writes and $O(\log n)$ depth on the nested-parallel model. The algorithm is recursive. In each level, we assume the problem size is m. We pick $s = \sqrt{m}/\log n$ samples, using a $O(s^2)$ algorithm to sort them in $O(\log s)$ time, and then subselect $p = s/\log n$ as pivots. Then by using the solution for placement problems to put every element into the correct bucket. This algorithm stops when there is less than $O(\log^6 n)$ elements in each bucket. Then elements within each bucket can be sorted using any polylog-depth algorithm. This algorithm is not write-efficient, but can be used to replace Cole's mergesort in Step 1 to achieve $O(\log n)$ depth *whp*.

4.4.3 Sorting on (M, B, ω)-ARAM Model

In this section, we present sorting algorithms for the (M, B, ω)-ARAM model. We show how the three approaches for external-memory sorting—mergesort, sample sort, and heapsort (using buffer trees)—can each be adapted to this asymmetric case.

In each case we trade off a factor of ω additional reads for a larger branching factor ($\omega M/B$ instead of M/B), hence reducing the number of rounds. The ARAM cost of each algorithm matches the non-trivial lower bound given in Section 3.5. It is interesting that the same general approach works for all three types of sorting. The first algorithm, the mergesort, has been described elsewhere [226] although in a different model (their model is specific to NAND flash memory and has different sized blocks for reading and writing, among other differences). Our parameters are therefore different, and our analysis is new. //about Nodari. To the best of our knowledge, our other two algorithms are new.

4.4.3.1 Mergesort

We use an l-way mergesort—i.e., a balanced tree of merges with each merge taking in l sorted arrays and outputting one sorted array consisting of all records from the input. We assume that once the input is small enough a different sort (the *base case*) is applied. For $l = M/B$ and a base case of $n \leq M$ (using any sort since it fits in memory), we have the standard EM mergesort. With these settings there are $\log_{M/B}(n/M)$ levels of recursion, plus the base case, each costing $O(n/B)$ memory operations. This gives the well-known overall bound of $\Theta((n/B)\log_{M/B}(n/B))$ [7].

To modify the algorithm for the asymmetric case, we increase the branching factor and the base case by a factor of ω, i.e., $l = \omega M/B$ and a base case of $n \leq \omega M$. This means that it

Algorithm 3: AEM-Mergesort

Input: An array of records A of length n
Output: The sorted array

1 **if** $|A| \leq \omega M$ **then**
2 Sort A using $\omega|A|/B$ reads and $|A|/B$ writes, and **return**
3 Evenly partition A into $l = \omega M/B$ subarrays $A_1, \ldots, A_l$ (at the granularity of blocks) and recursively apply AEM-Mergesort to each.
4 **Initialize Merge.** Initialize an empty output array O, a load buffer and an empty store buffer each of size B, an empty priority queue Q of size M, an array of pointers $I_1, \ldots, I_l$ that point to the start of each sorted subarray, $c = 0$, and $lastV = -\infty$. Associated with Q is $Q.max$, which holds the maximum element in Q if Q is full, and $+\infty$ otherwise.
5 **while** $c < |A|$ **do**
6 **for** $i \leftarrow 1$ to l **do**
7 Process-Block(i)
8 **while** Q is not empty **do**
9 $e \leftarrow Q.deleteMin$
10 Write e to the store buffer, $c \leftarrow c + 1$
11 If the store buffer is full, flush it to O and update $lastV$
12 **if** e is marked as last record in its subarray block **then**
13 $i = e.subarray$
14 Increment I_i to point to next block in subarray i
15 Process-Block(i)
16 $A \leftarrow O$ // Logically, don't actually copy
17 **Function** *Process-Block(subarray i)*
18 **if** I_i points to the end of the subarray **then return**
19 Read the block I_i into the load buffer
20 **forall** records e in the block **do**
21 **if** $e.key$ is in the range $(lastV, Q.max)$ **then**
22 If Q is full, eject $Q.max$
23 Insert e into Q, and mark if last record in block

is no longer possible to keep the base case in the primary memory, nor one block for each of the input arrays during a merge. The modified algorithm is described in Algorithm 3.

Each merge proceeds in a sequence of rounds, where a round is one iteration of the **while** loop starting on line 5. During each round we maintain a priority queue within the primary memory. Because operations within the primary memory are free in the model, this can just be kept as a sorted array of records, or even unsorted, although a balanced search tree can be a feasible solution in practice. Each round consists of two

phases. The first phase (the **for** loop on line 6) considers each of the l input subarrays in turn, loading the current block for the subarray into the load buffer, and then inserting each record e from the block into the priority queue if not already written to the output (i.e., $e.key > lastV$), and if smaller than the maximum in the queue (i.e., $e.key < Q.max$). This might bump an existing element out of the queue. Also, if a record is the last in its block then it is marked and tagged with its subarray number.

The second phase (the **while** loop starting on line 8) starts writing the priority queue to the output one block at a time. Whenever reaching a record that is marked as the last in its block, the algorithm increments the pointer to the corresponding subarray and processes the next block in the subarray. We repeat the rounds until all records from all subarrays have been processed.

To account for the space for the pointers $I = I_1, \ldots, I_l$, let $\alpha = (\log n)/s$, where s is the size of a record in bits, and n is the total number of records being merged. The cost of the merge is bounded as follows:

Lemma 4.4.3. *$l = \omega M/B$ sorted sequences with total size n (stored in $\lceil n/B \rceil$ blocks, and block aligned) can be merged using at most $(\omega + 1)\lceil n/B \rceil$ reads and $\lceil n/B \rceil$ writes, on the (M, B, ω)-ARAM model with primary memory size $(M + 2B + 2\alpha\omega M/B)$.*

Proof. Each round (except perhaps the last) outputs at least M records, and hence the total number of rounds is at most $\lceil n/M \rceil$. The first phase of each round requires at most $\omega M/B$ reads, so the total number of reads across all the first phases is at most $\omega\lceil n/B \rceil$ (the last round can be included in this since it only loads as many blocks as are output). For the second phase, a block is only read when incrementing its pointer, therefore every block is only read once in the second phase. Also every record is only written once. This gives the stated bounds on the number of reads and writes. The space includes the space for the in-memory heap (M), the load and store buffers, the pointers I ($\alpha\omega M/B$), and pointers to maintain the last-record in block information ($\alpha\omega M/B$). □

We note that it is also possible to keep I in secondary memory. This will double the number of writes because every time the algorithm moves to a new block in an input array i, it would need to write out the updated I_i. The increase in reads is small. Also, if one uses a balanced search tree to implement the priority queue Q then the size increases by $< M(\log M)/s$ in order to store the pointers in the tree.

For the base case when $n \leq \omega M$ we use the following lemma.

Lemma 4.4.4. *$n \leq \omega M$ records stored in $\lceil n/B \rceil$ blocks can be sorted using at most $\omega\lceil n/B \rceil$ reads and $\lceil n/B \rceil$ writes, on the (M, B, ω)-ARAM model with primary memory size $M + B$.*

Proof. We sort the elements using a variant of selection sort, scanning the input list a total of at most ω times. In the first scan, store in memory the M smallest elements seen so far, performing no writes and $\lceil n/B \rceil$ reads. After completing the scan, output all the $\min(M, n)$ elements in sorted order using $\lceil \min(M, n)/B \rceil$ writes. Record the maximum

element written so far. In each subsequent phase (if not finished), store in memory the M smallest records larger than the maximum written so far, then output as before. The cost is $\lceil n/B \rceil$ reads and M/B writes per phase (except perhaps the last phase). We need one extra block to hold the input. The largest output can be stored in the $O(\log M)$ locations we have allowed for in the model. This gives the stated bounds because every element is written out once and the input is scanned at most ω times. □

Together we have:

Theorem 4.4.5. *Algorithm 3 sorts n records using*

$$R(n) \leq (\omega + 1)\left\lceil \frac{n}{B} \right\rceil \left\lceil \log_{\frac{\omega M}{B}} \left(\frac{n}{B} \right) \right\rceil$$

reads, and

$$W(n) \leq \left\lceil \frac{n}{B} \right\rceil \left\lceil \log_{\frac{\omega M}{B}} \left(\frac{n}{B} \right) \right\rceil$$

writes on an (M, B, ω)-ARAM with primary memory size $(M + 2B + 2\alpha\omega M/B)$.

Proof. The number of recursive levels of merging is bounded by $\left\lceil \log_{\frac{\omega M}{B}} \left(\frac{n}{\omega M} \right) \right\rceil$, and when we add the additional base round we have $1 + \left\lceil \log_{\frac{\omega M}{B}} \left(\frac{n}{\omega M} \right) \right\rceil = \left\lceil \log_{\frac{\omega M}{B}} \left(\frac{n}{\omega M} \frac{\omega M}{B} \right) \right\rceil = \left\lceil \log_{\frac{\omega M}{B}} \left(\frac{n}{B} \right) \right\rceil$. The cost for each level is at most $(\omega + 1)\lceil n/B \rceil$ reads and $\lceil n/B \rceil$ writes (only one block on each level might not be full). □

4.4.3.2 Sample Sort

We now describe an l-way randomized sample sort [56, 127] (also called distribution sort), which asymptotically matches the I/O bounds of the mergesort. The idea of sample sort is to partition n records into l approximately equally sized buckets based on a sample of the keys within the records, and then recurse on each bucket until an appropriately-sized base case is reached. As with the mergesort, here we will use a branching factor $l = \omega M/B$. Again this branching factor will reduce the number of levels of recursion relative to the standard EM sample sort which uses $l = M/B$ [7]. We describe how to process each partition and the base case.

The partitioning starts by selecting a set of splitters. This can be done using standard techniques, which we review later. The splitters partition the input into buckets that *whp* are within a constant factor of the average size n/l. The algorithm now needs to bucket the input based on the splitters. The algorithm processes the splitters in ω rounds of size M/B each, starting with the first M/B splitters. For each round the algorithm scans the whole input array, partitioning each value into the one of M/B buckets associated with the splitters, or skipping a record if its key does not belong in the current buckets. One block for each bucket is kept in memory. Whenever a block for one of the buckets is full, it is written out to memory and the next block is started for that bucket. Each ω rounds

reads all of the input and writes out only the elements associated with these buckets (roughly a $1/\omega$ fraction of the input).

The base case occurs when $n \leq \omega M$, at which point we apply the selection sort from Lemma 4.4.4.

Let n_0 be the original input size. The splitters can be chosen by randomly picking a sample of keys of size $m = \Theta(l \log n_0)$, sorting them, and then sub-selecting the keys at positions $m/l, 2m/l, \ldots, (l-1)m/l$. By selecting the constant in the Θ sufficiently large, this process ensures that, *whp* every bucket is within a constant factor of the average size [56]. To sort the samples apply a RAM mergesort, which requires at most $O(((l \log n_0)/B) \log(l \log n_0/M))$ reads and writes. This is a lower-order term when $l = O(n/\log^2 n)$, but unfortunately this bound on l may not hold for small subproblems. There is a simple solution—when $n \leq \omega^2 M^2/B$, instead use $l = n/(\omega M)$. With this modification, we always have $l \leq \sqrt{n/B}$.

It is likely that the splitters could also be selected deterministically using an approach used in the original I/O-efficient distribution sort [7].

Theorem 4.4.6. *The kM/B-way sample sort sorts n records using,* whp*,*

$$R(n) = O\left(\frac{\omega n}{B}\left\lceil \log_{\frac{\omega M}{B}}\left(\frac{n}{B}\right)\right\rceil\right)$$

reads, and

$$W(n) = O\left(\frac{n}{B}\left\lceil \log_{\frac{\omega M}{B}}\left(\frac{n}{B}\right)\right\rceil\right)$$

writes on an (M, B, ω)-ARAM with primary memory size $(M + B + M/B)$.

Proof. (Sketch) The primary-memory size allows one block from each bucket as well as the M/B splitters to remain in memory. Each partitioning step thus requires $\lceil n/B \rceil + \omega M/B$ writes, where the second term arises from the fact that each bucket may use a partial block. Since $n \geq \omega M$ (this is not a base case), the cost of each partitioning step becomes $O(n/B)$ writes and $O(kn/B)$ reads. Because the number of splitters is at most $\sqrt{n} = O(n/\log^2 n)$, choosing and sorting the splitters takes $O(n/B)$ reads and writes. Observe that the recursive structure matches that of a sample sort with an effective memory of size ωM, and that there will be at most two rounds at the end where $l = n/(\omega M)$. As in standard sample sort, the number of writes is linear with the size of the subproblem, but here the number of reads is multiplied by a factor of ω. The standard samplesort analysis thus applies, implying the bound stated.

It remains only to consider the base case. Because all buckets are approximately the same size, the total number of leaves is $O(n/B)$—during the recursion, a size $n > \omega M$ problem is split into subproblems whose sizes are $\Omega(B)$. Applying Lemma 4.4.4 to all leaves, we get a cost of $O(\omega n/B)$ reads and $O(n/B)$ writes for all base cases. □

Extensions for the Private-Cache Model. The above can be readily parallelized. Here we outline the approach. We assume that there are $p = n/M$ processors. We use parallelism

both within each partition, and across the recursive partitions. Within a partition we first find the l splitters in parallel. (As above, $l = \omega M/B$ except for the at most two rounds prior to the base case where $l = n/(\omega M)$.) This can be done on a sample that is a logarithmic factor smaller than the partition size, using a less efficient sorting algorithm such as parallel mergesort, and then sub-selecting l splitters from the sorted order. This requires $O(\omega(M/B + \log^2 n))$ time, where the second term ($O(\omega \log^2 n)$) is the depth of the parallel mergesort, and the first term is the work term $O((\omega/B)((n/\log n)\log n)/P) = O(\omega M/B)$.

The algorithm groups the input into $n/(\omega M)$ chunks of size ωM each. As before we also group the splitters into ω rounds of size M/B each. Now in parallel across all chunks and across all rounds, partition the chunk based on the round. We have $n/(\omega M) \times \omega = n/M$ processors so we can do them all in parallel. Each will require ωM reads and M writes. To ensure that the chunks write their buckets to adjacent locations (so that the output of each bucket is contiguous) we will need to do a pass over the input to count the size of each bucket for each chunk, followed by a prefix sum. This can be done before processing the chunks and is a lower-order term. The time for the computation is $O(\omega M/B)$.

The processors are then divided among the sub-problems proportional to the size of the sub-problem, and we repeat. The work at each level of recursion remains the same, so the time at each level remains the same. For the base case of size $\leq \omega M$, instead of using a selection sort across all keys, which is sequential, we find ω splitters and divide the work among ω processors to sub-select their part of the input, each by reading the whole input, and then sorting their part of size $O(M)$ using a selection sort on those keys. This again takes $O(\omega M/B)$ time. The total time for the algorithm is therefore:

$$O\left(\omega\left(\frac{M}{B} + \log^2 n\right)\left\lceil 1 + \log_{\frac{\omega M}{B}}\left(\frac{n}{\omega M}\right)\right\rceil\right)$$

with high probability. This is linear speedup assuming $\frac{M}{B} \geq \log^2 n$. Otherwise the number of processors can be reduced to maintain linear speedup.

4.4.3.3 I/O Buffer Trees

This section describes how to augment the basic buffer tree [25] to build a priority queue that supports n Insert and Delete-Min operations with an amortized cost of $O((\omega/B)(1 + \log_{\omega M/B} n))$ reads and $O((1/B)(1 + \log_{\omega M/B} n))$ writes per operation. Using the priority queue to implement a sorting algorithm trivially results in a sort costing a total of $O((\omega n/B)(1 + \log_{\omega M/B} n))$ reads and $O((n/B)(1 + \log_{\omega M/B} n))$ writes. These bounds asymptotically match the preceding sorting algorithms, but some additional constant factors are introduced because a buffer tree is a dynamic data structure.

Our buffer tree-based priority queue for the (M, B, ω)-ARAM contains a few differences from the regular EM buffer tree [25]: (1) the buffer tree nodes are larger by a factor ω, (2) consequently, the "buffer-emptying" process uses an efficient sort on ωM elements instead of an in-memory sort on M elements, and (3) to support the priority queue, $O(\omega M)$

elements are stored outside the buffer tree instead of $O(M)$, which adds nontrivial changes to the data structure.

Overview of a buffer tree.

A buffer tree [25] is an augmented version of an (a, b)-tree [175], where $a = l/4$ and $b = l$ for large branching factor l. In the original buffer tree $l = M/B$, but to reduce the number of writes we instead set $l = \omega M/B$. As an (a, b) tree, all leaves are at the same depth in the tree, and all internal nodes have between $l/4$ and l children (except the root, which may have fewer). Thus the height of the tree is $O(1 + \log_l n)$. An internal node with c children contains $c - 1$ keys, stored in sorted order, that partition the elements in the subtrees. The structure of a buffer tree differs from that of an (a, b) tree in two ways. Firstly, each leaf of the buffer tree contains between $lB/4$ and lB elements stored in l blocks.[1] Secondly, each node in the buffer tree also contains a dense unsorted list, called a ***buffer***, of partially inserted elements that belong in that subtree.

We next summarize the basic buffer tree insertion process [25]. Supporting general deletions is not much harder, but to implement a priority queue we only need to support deleting an entire leaf. The insertion algorithm proceeds in two phases: the first phase moves elements down the tree through buffers, and the second phase performs the (a, b)-tree rebalance operations (i.e., splitting nodes that are too big). The first phase begins by appending the new element to the end of the root's buffer. We say that a node is ***full*** if its buffer contains at least lB elements. If the insert causes the root to become full, then a ***buffer-emptying process*** commences, whereby all of the elements in the node's buffer are sorted then distributed to the children (appended to the ends of their buffers). This distribution process may cause children to become full, in which case they must also be emptied. More precisely, the algorithm maintains a list of internal nodes with full buffers (initially the root) and a separate list of leaves with full buffers. The first phase operates by repeatedly extracting a full internal node from the list, emptying its buffer, and adding any full children to the list of full internal or leaf nodes, until there are no full internal nodes.

Note that during the first phase, the buffers of full nodes may far exceed lB, e.g., if all of the ancestors' buffer elements are distributed to a single descendant. Sorting the buffer from scratch would therefore be too expensive. Fortunately, each distribution process writes elements to the child buffers in sorted order, so all elements after the lB'th element (i.e., those written in the most recent emptying of the parent) are sorted. It thus suffices to split the buffer at the lB'th element and sort the first lB elements, resulting in a buffer that consists of two sorted lists. These two lists can trivially be merged as they are being distributed to the sorted list of children in a linear number of I/O's.

[1] Arge [25] defines the "leaves" of a buffer tree to contain $\Theta(B)$ elements instead of $\Theta(lB)$ elements. Since the algorithm only operates on the parents of those "leaves", we find the terminology more convenient when flattening the bottom two levels of the tree. Our leaves thus correspond to what Arge terms "leaf nodes" [25] (not to be confused with leaves) or equivalently what Sitchinava and Zeh call "fringe nodes" [256].

When the first phase completes, there may be full leaves but no full internal nodes. Moreover, all ancestors of each full leaf have empty buffers. The second phase operates on each full leaf one at a time. First, the buffer is sorted as above and then merged with the elements stored in the leaf. If the leaf contains $X > lB$ elements, then a sequence of (a, b)-tree rebalance operations occur whereby the leaf may be split into $\Theta(X/(lB))$ new nodes. These splits cascade up the tree as in a typical (a, b)-tree insert.

Buffer tree with fewer writes.

To reduce the number of writes, we set the branching factor of the buffer tree to $l = \omega M/B$ instead of $l = M/B$. The consequence of this increase is that the buffer emptying process needs to sort $lB = \omega M$ elements, which cannot be done with an in-memory sort. The advantage is that the height of the tree reduces to $O(1 + \log_{\omega M/B} n)$.

Lemma 4.4.7. *It costs $O(\omega X/B)$ reads and $O(X/B)$ writes to empty a full buffer containing X elements using $\Theta(M)$ memory.*

Proof. By Lemma 4.4.4, the cost of sorting the first ωM elements is $O(\omega^2 M/B)$ reads and $O(\omega M/B)$ writes. The distribute step can be performed by simultaneously scanning the sorted list of children along with the two sorted pieces of the buffer, and outputting to the end of the appropriate child buffer. A write occurs only when either finishing with a child or closing out a block. The distribute step thus uses $O(\omega M/B + X/B)$ reads and writes, giving a total of $O(\omega^2 M/B + X/B)$ reads and $O(\omega M/B + X/B)$ writes including the sort step. Observing that full means $X > \omega M$ completes the proof. □

Theorem 4.4.8. *Suppose that the partially empty block belonging to the root's buffer is kept in memory. Then the amortized cost of each insert into an n-element buffer tree is $O((\omega/B)(1 + \log_{\omega M/B} n))$ reads and $O((1/B)(1 + \log_{\omega M/B} n))$ writes.*

Proof. This proof follows from Arge's buffer tree performance proofs [25], augmented with the above lemma. We first consider the cost of reading and writing the buffers. The last block of the root buffer need only be written when it becomes full, at which point the next block must be read, giving $O(1/B)$ reads and writes per insert. Each element moves through buffers on a root-to-leaf path, so it may belong to $O(1 + \log_{\omega M/B} n)$ emptying processes. According to Lemma 4.4.7, emptying a full buffer costs $O(\omega/B)$ reads and $O(1/B)$ writes per element. Multiplying these two gives an amortized cost per element matching the theorem.

We next consider the cost of rebalancing operations. Given the choice of (a, b)-tree parameters, the total number of node splits is $O(n/(lB))$ [25, Theorem 1] which is $O(n/(\omega M))$. Each split is performed by scanning a constant number of nodes, yielding a cost of $O(\omega M/B)$ reads and write per split, or $O(n/(\omega M) \cdot \omega M/B) = O(n/B)$ reads and writes in total or $O(1/B)$ per insert. □

An efficient priority queue with fewer writes.

The main idea of Arge's buffer tree-based priority queue [25] is to store a working set of the $O(lB)$ smallest elements resident in memory. When inserting an element, first add it to the working set, then evict the largest element from the working set (perhaps the one just inserted) and insert it into the buffer tree. To extract the minimum, find it in the working set. If the working set is empty, remove the $\Theta(lB)$ smallest elements from the buffer tree and add them to the working set. In the standard buffer tree, $l = M/B$ and hence operating on the working set is free because it fits entirely in memory. In our case, however, extra care is necessary to maintain a working set that has size roughly ω times larger.

Our (M, B, ω)-ARAM priority queue follows the same idea except the working set is partitioned into two pieces, the alpha working set and beta working set. The ***alpha working set***, which is always resident in memory, contains at most $M/4$ of the smallest elements in the priority queue. The ***beta working set*** contains at most $2\omega M$ of the next smallest elements in the data structure, stored in $O(\omega M/B)$ blocks. The motivation for having a beta working set is that during Delete-Min operations, emptying elements directly from the buffer tree whenever the alpha working set is empty would be too expensive—having a beta working set to stage larger batches of such elements leads to better amortized bounds. Coping with the interaction between the alpha working set, the beta working set, and the buffer tree, is the main complexity of our priority queue. The beta working set does not fit in memory, but we keep a constant number of blocks from the beta working set and the buffer tree (specifically, the last block of the root buffer) in memory.

We begin with a high-level description of the priority-queue operations, with details of the beta working set deferred until later. For now, it suffices to know that we keep the maximum key in the beta working set in memory. To insert a new element, first compare its key against the maximums in the alpha and beta working set. Then insert it into either the alpha working set, the beta working set, or the buffer tree depending on the key comparisons. If the alpha working set exceeds maximum capacity of $M/4$ elements, move the largest element to the beta working set. If the beta working set hits its maximum capacity of $2\omega M$ elements, remove the largest ωM elements and insert them into the buffer tree.

To delete the minimum from the priority queue, remove the smallest element from the alpha working set. If the alpha working set is empty, extract the $M/4$ smallest elements from the beta working set (details to follow) and move them to the alpha working set. If the beta working set is empty, perform a buffer emptying process on the root-to-leftmost-leaf path in the buffer tree. Then delete the leftmost leaf and move its contents to the beta working set.

The beta working set. The main challenge is in implementing the beta working set. An unsorted list or buffer allows for efficient inserts by appending to the last block. The challenge, however, is to extract the $\Theta(M)$ smallest elements with $O(M/B)$ writes—if

$\omega > B$, each element may reside in a separate block, and we thus cannot afford to update those blocks when extracting the elements. Instead, we perform the deletions implicitly.

To facilitate implicit deletions, we maintain a list of ordered pairs $(i_1, x_1), (i_2, x_2), (i_3, x_3), \ldots$, where (i, x) indicates that all elements with index at most i and key at most x are invalid. Our algorithm maintains the invariant that for consecutive list elements (i_j, x_j) and (i_{j+1}, x_{j+1}), we have $i_j < i_{j+1}$ and $x_j > x_{j+1}$ (recall that all keys are distinct).

To insert an element to the beta working set, simply append it to the end. The invariant is maintained because its index is larger than any pair in the list.

To extract the minimum $M/4$ elements, scan from index 0 to i_1 in the beta working set, ignoring any elements with key at most x_1. Then scan from $i_1 + 1$ to i_2, ignoring any element with key at most x_2. And so on. While scanning, record in memory the $M/4$ smallest valid elements seen so far. When finished, let x be the largest key and let i be the length of the beta working set. All elements with key at most x have been removed from the full beta working set, so they should be implicitly marked as invalid. To restore the invariant, truncate the list until the last pair (i_j, x_j) has $x_j > x$, then append (i, x) to the list. Because the size of the beta working set is growing, $i_j < i$. It should be clear that truncation does not discard any information as (i, x) subsumes any of the truncated pairs.

Whenever the beta working set grows too large ($2\omega M$ valid elements) or becomes too sparse (ω extractions of $M/4$ elements each have occurred), we first rebuild it. Rebuilding scans the elements in order, removing the invalid elements by packing the valid ones densely into blocks. Testing for validity is done as above. When done, the list of ordered pairs to test invalidity is cleared.

Finally, when the beta working set grows too large, we extract the largest ωM elements by sorting it (using the selection sort of Lemma 4.4.4).

Analyzing the priority queue.

We begin with some lemmas about the beta working set.

Lemma 4.4.9. *Extracting the $M/4$ smallest valid elements from the beta working set and storing them in memory costs $O(\omega M/B)$ reads and amortized $O(1)$ writes.*

Proof. The extraction involves first performing read-only passes over the beta working set and list of pairs, keeping one block from the working set and one pair in memory at a time. Because the working set is rebuilt after ω extractions, the list of pairs can have at most ω entries. Even if the list is not I/O efficient, the cost of scanning both is $O(\omega M/B + \omega) = O(\omega M/B)$ reads. Next the list of pairs indicating invalid elements is updated. Appending one new entry requires $O(1)$ writes. Truncating and deleting any old entries can be charged against their insertions. □

The proof of the following lemma is similar to the preceding one, with the only difference being that the valid elements must be moved and written as they are read.

Lemma 4.4.10. *Rebuilding the beta working set costs $O(\omega M/B)$ reads and writes.* □

Theorem 4.4.11. *Our priority queue, if initially empty, supports n Insert and Delete-Min operations with an amortized cost of $O((\omega/B)(1 + \log_{\omega M/B} n))$ reads and $O((1/B)(1 + \log_{\omega M/B} n))$ writes per operation.*

Proof. Inserts are the easier case. Inserting into the alpha working set is free. The amortized cost of inserting directly into the beta working set (a simple append) is $O(1/B)$ reads and writes, assuming the last block stays in memory. The cost of inserting directly into the buffer tree matches the theorem. Occasionally, the beta working set overflows, in which case we rebuild it, sort it, and insert elements into the buffer tree. The rebuild costs $O(\omega M/B)$ reads and writes (Lemma 4.4.10), the sort costs $O(\omega^2 M/B)$ reads and $O(\omega M/B)$ writes (by Lemma 4.4.4), and the ωM buffer tree inserts cost $O((\omega^2 M/B)(1 + \log_{\omega M/B} n))$ reads and $O((\omega M/B)(1 + \log_{\omega M/B} n))$ writes (by Theorem 4.4.8). The latter dominates. Amortizing against the ωM inserts that occur between overflows, the amortized cost per insert matches the theorem statement.

Deleting the minimum element from the alpha working set is free. When the alpha working set becomes empty, we extract $M/4$ elements from the beta working set, with a cost of $O(\omega M/B)$ reads and $O(1)$ writes (Lemma 4.4.9). This cost may be amortized against the $M/4$ deletes that occur between extractions, for an amortized cost of $O(\omega/B)$ reads and $O(1/M)$ writes per delete-min. Every ω extractions of $M/4$ elements, the beta working set is rebuilt, with a cost of $O(\omega M/B)$ reads and writes (Lemma 4.4.10) or amortized $O(1/B)$ reads and writes per delete-min. Adding these together, we so far have $O(\omega/B)$ reads and $O(1/B)$ writes per delete-min.

It remains to analyze the cost of refilling the beta working set when it becomes empty. The cost of removing a leaf from the buffer tree is dominated by the cost of emptying buffers on a length-$O(\log_{\omega M/B} n)$ path. Note that the buffers are not full, so we cannot apply Lemma 4.4.7. But a similar analysis applies. The cost per node is $O(\omega^2 M/B + X/B)$ reads and $O(\omega M/B + X/B)$ writes for an X-element buffer. As with Arge's version of the priority queue [25], the $O(X/B)$ terms can be charged to the insertion of the X elements, so we are left with a cost of $O(\omega^2 M/B)$ read and $O(\omega M/B)$ writes per buffer. Multiplying by $O(1 + \log_{\omega M/B} n)$ levels gives a cost of $O((\omega^2 M/B)(1 + \log_{\omega M/B} n))$ reads and $O((\omega M/B)(1 + \log_{\omega M/B} n))$ writes. Because each leaf contains at least $\omega M/4$ elements, we can amortize this cost against at least $\omega M/4$ deletions, giving a cost that matches the theorem. □

With this priority queue, sorting can be trivially implemented in $O((\omega n/B)(1 + \log_{\omega M/B} n))$ reads and $O((n/B)(1 + \log_{\omega M/B} n))$ writes, matching the bounds of the previous sorting algorithms.

4.5 Longest Common Subsequence and Edit Distance

This section describes a more efficient dynamic-programming algorithm for longest common subsequence (LCS) and edit distance (ED) on the (M, ω)-ARAM model. The standard approach for these problems (an $M \times M$ tiling) results in an ARAM cost of $O(mn\omega/M)$ and work of $O(mn + mn\omega/M)$, where m and n are the length of the two input strings. Lemma 3.4.2 states that the standard bound is optimal under the standard DAG computation rule that all inputs must be available before evaluating a node. Perhaps surprisingly, we are able to beat these bounds by leveraging the fact that dynamic programs do not perform arbitrary functions at each node, and hence we do not necessarily need all inputs to begin evaluating a node.

Our main result is captured by the following theorem for large input strings (the general case). For smaller strings, we can do even better and will be discussed later in Section 4.5.2.

4.5.1 The General Case

Theorem 4.5.1. *Let $k_T = \min((\omega/M)^{1/3}, \sqrt{M})$ and suppose $m, n = \Omega(k_T M)$. Then it is possible to compute the ED or length of the LCS with work $W(m, n) = O(mn + mn\omega/(k_T M))$.*

Let $k_Q = \min(\omega^{1/3}, \sqrt{M})$ and suppose $m, n = \Omega(k_Q M)$. Then it is possible to compute the ED or length of the LCS with an ARAM cost of $Q(m, n) = O(mn\omega/(k_Q M))$.

To understand these bounds, our algorithm beats the ARAM cost of the standard tiling algorithm by a k_Q factor. And if $\omega \geq M$, our algorithm (using different tuning parameters) beats the work of the standard tiling algorithm by a k_T factor.

Overview The dynamic programs for LCS and ED correspond to computing the shortest path through an $m \times n$ grid with diagonal edges, where m and n are the string lengths. We focus here on computing the length of the shortest path, but it is possible to output the path as well with the same asymptotic complexity (see Section 4.5.3). Without loss of generality, we assume that $m \leq n$, so the grid is at least as wide as it is tall. For LCS, all horizontal and vertical edges have weight 0; the diagonal edges have weight -1 if the corresponding characters in the strings match, and weight ∞ otherwise. For ED, horizontal and vertical edges have weight 1, and diagonal edges have weights either 0 or 1 depending on whether the characters match. Our algorithm is not sensitive to the particular weights of the edges, and thus it applies to both problems and their generalizations.

Note that the $m \times n$ grid is not built explicitly since building and storing the graph would take $\Theta(mn)$ writes if $mn \gg M$. To get any improvement, it is important that subgrids reuse the same space. The weights of each edge can be inferred by reading the appropriate characters in each input string.

Our algorithm partitions the implicit grid into size-$(hM' \times kM')$, where h and k are parameters of the algorithm to be set later, and $M' = M/c$ for large enough constant $c > 1$ to give sufficient working space in small-memory. When string lengths m and $n \geq m$ are both "large", we use $h = k$ and thus usually work with $kM' \times kM'$ square subgrids. If the smaller string length m is small enough, we instead use parameters $h < k$. To simplify the description of the algorithm, we assume without loss of generality that m and n are divisible by hM' and kM', respectively, and that M is divisible by c.

Our algorithm operates on one $hM' \times kM'$ rectangle at a time, where the edges are directed right and down. The shortest-path distances to all nodes along the bottom and right boundary of each rectangle are explicitly written out, but all other intermediate computations are discarded. We label the vertices $u_{i,j}$ for $1 \leq i \leq hM'$ and $1 \leq j \leq kM'$ according to their row and column in the square, respectively, starting from the top-left corner. We call the vertices immediately above or to the left of the square the *input nodes*. The input nodes are all outputs for some previously computed rectangle. We call the vertices $u_{hM',j}$ along the bottom boundary and $u_{i,kM'}$ along the right boundary the *output nodes*.

The goal is to reduce the number of writes, thereby decreasing the overall cost of computing the output nodes, which we do by sacrificing reads and work. It is not hard to see that recomputing internal nodes enables us to reduce the number of writes. Consider, for example, the following simple approach assuming $M = \Theta(1)$: For each output node of a $k \times k$ square, try all possible paths through the square, keeping track of the best distance seen so far; perform a write at the end to output the best value.[2] Each output node tries $2^{\Theta(k)}$ paths, but only a $\Theta(1/k)$-fraction of nodes are output nodes. Setting $k = \Theta(\lg \omega)$ reduces the number of writes by a $\Theta(\lg \omega)$-factor at the cost of $\omega^{O(1)}$ reads. This same approach can be extended to larger M, giving the same $\lg \omega$ improvement, by computing "bands" of nearby paths simultaneously. But our main algorithm, which we discuss next, is much better as M gets larger (see Theorem 4.5.1).

Path sketch The key feature of the grid leveraged by our algorithm is that shortest paths do not cross, which enables us to avoid the exponential recomputation of the simple approach. The noncrossing property has been exploited previously for building shortest-path data structures on the grid (e.g., [246]) and more generally planar graphs (e.g., [124, 191]). These previous approaches do not consider the cost of writing to large-memory, and they build data structures that would be too large for our use. Our algorithm leverages the available small-memory to compute bands of nearby paths simultaneously. We capture both the noncrossing and band ideas through what we call a path sketch,

[2]This approach requires constant small-memory to keep the best distance, the current distance, and working space for computing the current distance. We also need bits proportional to the path length to enumerate paths.

which we define as follows. The path sketch enables us to cheaply recompute the shortest paths to nodes.

We call every M'-th row in the square a *superrow*, meaning there are h superrows in the square. The algorithm partitions the i-th superrow into *segments* $\langle i, \ell, r\rangle$ of consecutive elements $u_{iM',\ell}, u_{iM',\ell+1}, \ldots, u_{iM',r}$. The main restriction on segments is that $r < \ell + M'$, i.e., each segment consists of at most M' consecutive elements in the superrow. Note that the segment boundaries are determined by the algorithm and are input dependent.

A *path sketch* is a sequence of segments $\langle s, \ell_s, r_s\rangle, \langle s+1, \ell_{s+1}, r_{s+1}\rangle, \langle s+2, \ell_{s+2}, r_{s+2}\rangle, \ldots, \langle i, \ell_i, r_i\rangle$, summarizing the shortest paths to the segment. Specifically, this sketch means that for each vertex in the last segment, there is a shortest path to that vertex that goes through a vertex in each of the segments in the sketch. If the sketch starts at superrow 1, then the path originates from a node above the first superrow (i.e., the top boundary or the topmost M' nodes of the left boundary). If the sketch starts with superrow $s > 1$, then the path originates at one of the M' nodes on the left boundary between superrows $s-1$ and s. Since paths cannot go left, the path sketch also satisfies $\ell_s \le \ell_{s+1} \le \cdots \le \ell_i$.

Evaluating a path sketch Given a path sketch, we refer to the process of determining the shortest-path distances to all nodes in the final segment $\langle i, \ell_i, r_i\rangle$ as *evaluating the path sketch* or *evaluating the segment*, with the distances in small-memory when the process completes. Note that we have not yet described how to build the path sketch, as the building process uses evaluation as a subroutine.

The main idea of evaluating the sketch is captured by Figure 4.3 for the example sketch $\langle 1, 4, 6\rangle, \langle 2, 6, 6\rangle, \langle 3, 8, 9\rangle$. The sketch tells us that shortest paths to $u_{9,8}$ and $u_{9,9}$ pass through one of $u_{3,4}, u_{3,5}, u_{3,6}$ and the node $u_{6,6}$. Thus, to compute the distances to $u_{9,8}$ and $u_{9,9}$, we need only consider paths through the darker nodes and solid edges—the lighter nodes and dashed edges are not recomputed during evaluation.

The algorithm works as follows. First compute the shortest-path distances to the first segment in the sketch. To do so, horizontally sweep a height-$(M'+1)$ column across the $(M'+1)\times kM'$ slab raising above the s-th superrow, keeping two columns in small-memory at a time. Also keep the newly computed distances to the first segment in small-memory, and stop the sweep at the right edge of the segment. More generally, given the distances to a segment in small-memory, we can compute the values for the next segment in the same manner by sweeping a column through the slab. This algorithm yields the following performance.

Lemma 4.5.2. *Given a path sketch $\langle s, \ell_s, r_s\rangle, \ldots, \langle i, \ell_i, r_i\rangle$ in an $hM' \times kM'$ grid with distances to all input nodes computed, our algorithm correctly computes the shortest-path distances to all nodes in the segment $\langle i, \ell_i, r_i\rangle$. Assuming $k \ge h$ and small-memory size $M \ge 5M' + \Theta(1)$, the algorithm requires $O(kM^2)$ operations in small-memory, $O(kM)$ reads, and 0 writes.*

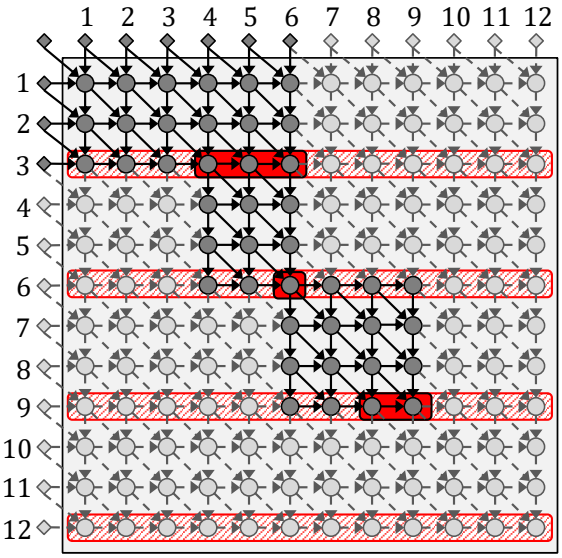


Figure 4.3: Example square grid and path sketch for $M' = 3$ and $h = k = 4$. The circles are nodes in the square. The diamonds are input nodes (outputs of adjacent squares), omitting irrelevant edges. The red slashes are the 4 superrows, and the solid red are the sketch segments.

Proof. Correctness follows from the definition of the path sketch: the sweep performed by the algorithm considers all possible paths that pass through these segments.

The algorithm requires space in small-memory to store two columns in the current slab, the previous segment in the sketch, and the next segment in the sketch, and the two segment boundaries themselves, totaling $4M' + \Theta(1)$ small-memory. Due to the monotonically increasing left endpoints of each segment, the horizontal sweep repeats at most M' columns per supperrow, so the total number of column iterations is $O(kM' + hM') = O(kM')$. Multiplying by M' gives the number of nodes computed.

The main contributor to reads is the input strings themselves to infer the structure/weights of the grid. With M' additional small-memory, we can store the "vertical" portion of the input string used while computing each slab, and thus the vertical string is read only once with $O(hM') = O(kM')$ reads. The "horizontal" input characters can be read with each of the $O(kM')$ column-sweep iterations. An additional k reads suffice to read the sketch itself, which is a lower-order term. □

Building the path sketch The main algorithm on each rectangle involves building the set of sketches to segments in the bottom superrow. At some point during the sketch-building process, the distances to each output node is computed, at which point it can be written out. The main idea of the algorithm is a sketch-extension subroutine: given

segments in the i-th superrow and their sketches, extend the sketches to produce segments in the $(i+1)$-th superrow along with their sketches.

Our algorithm builds up an ordered list of consecutive path sketches, one superrow at a time. The first superrow is partitioned into k segments, each containing exactly M' consecutive nodes. The list of sketches is initialized to these segments.

Given a list of sketches to the i-th superrow, our algorithm extends the list of sketches to the $(i+1)$-th superrow as follows. The algorithm sweeps a height-$(M'+1)$ column across the $(M'+1) \times kM'$ slab between these superrows (inclusive). The sweep begins at the left end of the slab, reading the input values from the left boundary, and continuing across the entire width of the slab. In small-memory, we evaluate the first segment of the i-th superrow (using the algorithm from Lemma 4.5.2). Whenever the sweep crosses a segment boundary in the i-th superrow, again evaluate the next segment in the i-th superrow. For each node in the slab, the sweep calculates both the shortest-path distance and a pointer to the segment in the previous superrow from whence the shortest path originates (or a null pointer if it originates from the left boundary). When the originating segment of the bottom node (the node in the $(i+1)$-th superrow) changes, the algorithm creates a new segment for the $(i+1)$-th superrow and appends it to the sketch of the originating segment. If the segment in the current segment in the $(i+1)$-th superrow grows past M' elements, a new segment is created instead and the current path sketch is copied and spliced into the list of sketches. Any sketch that is not extended through this process is no longer relevant and may be spliced out of the list of sketches. When the sweep reaches a node on the output boundary (right edge or bottom edge of the square), the distance to that node is written out.

Lemma 4.5.3. *The sketching algorithm partitions the i-th superrow into at most ik segments.*

Proof. The proof is by induction over superrows. As a base case, the first superrow consists of exactly k segments. For the inductive step, there are two cases in which a new segment is started in the $(i+1)$-th superrow. The first case is that the originating segment changes, which can occur at most ik times by inductive assumption. The second case is that the current segment grows too large, which can occur at most k times. We thus have at most $(i+1)k$ segments in the $(i+1)$-th superrow. □

Lemma 4.5.4. *Suppose $h \leq k$ and small-memory $M \geq 11M' + \Theta(1)$, and consider an $hM' \times kM'$ grid with distances to input nodes already computed. Then the sketch building algorithm correctly computes the distances to all output boundary nodes using $O((hk)^2M^2)$ operations in small-memory, $O((hk)^2M)$ reads from large-memory, and $O(h^2k + X)$ writes to large-memory, where $X = O(kM)$ is the number of boundary nodes written out.*

Proof. Consider the cost of computing each slab, ignoring the writes to the output nodes. We reserve $5M'+\Theta(1)$ small-memory for the process of evaluating segments in the previous superrow. To perform the sweep in the current slab, we reserve M' small-memory to store one segment in the previous row, M' small-memory to store characters in the "vertical"

input string, $4(M' + 1)$ small-memory to store two columns (each with distances and pointers) for the sweep, and an additional $\Theta(1)$ small-memory to keep, e.g., the current segment boundaries. Since there are at most hk segments in the previous superrow (Lemma 4.5.3), the algorithm evaluates at most hk segments; applying Lemma 4.5.2, the cost is $O(hk^2M^2)$ operations, $O(hk^2M)$ reads, and 0 writes. There are an additional $O(kM^2)$ operations to sweep through the kM^2 nodes in the slab, plus $O(kM)$ reads to scan the "horizontal" input string. Finally, there are $O(hk)$ writes to extend existing sketches and $O(hk)$ writes to copy at most k sketches.

Summing across all h slabs and accounting for the output nodes, we get $O((hk)^2M^2 + hkM^2)$ operations, $O((hk)^2M + hkM)$ reads, and $O(h^2k + X)$ writes. Removing the lower-order terms gives the lemma. □

Combining across all rectangles in the grid, we get the following corollary.

Corollary 4.5.5. *Let $m \leq n$ be the length of the two input strings, with $m \geq M$. Suppose $h = O(m/M)$ and $k = O(n/M)$ with $h \leq k$. Then it is possible to compute the LCS or edit distance of the strings with $O(mnhk)$ operations in small-memory, $O(mnhk/M)$ reads to large-memory, and $O(mnh/M^2 + mn/(hM))$ writes to large-memory.*

Proof. There are $\Theta(mn/(hkM^2))$ size-$(hM/11) \times (kM/11)$ subgrids. Multiplying by the cost of each grid (Lemma 4.5.4) gives the bound. □

Setting $h = k = 1$ gives the standard $M \times M$ tiling with $O(nm)$ work and $O(mn\omega/M)$ ARAM cost. As the size of squares increase, the fraction of output nodes and hence writes decreases, at the cost of more overhead for operations in small-memory and reads from large-memory. Assuming both n and m are large enough to do so, plugging in $h = k = \max\{1, k_T\}$ or $h = k = k_Q$ with a few steps of algebra to eliminate terms yields Theorem 4.5.1.

Proof of Theorem 4.5.1. As long as $h \leq \sqrt{M}$, the number of writes reduces to $O(mn/(hM))$. (Increasing h further causes the number of writes to increase.)

Consider the work bound first. If $k_T \leq 1$, then just use algorithm with $h = k = 1$. Otherwise, let $M' = M/11$ and use the algorithm with $h = k = k_T$. As long as $h = k \leq (\omega/M)^{1/3}$, which is true for k_T, the work of operations is less than the work of writes, giving the bound.

For the ARAM cost, use our algorithm with $h = k = k_Q$. As long as $h = k \leq \omega^{1/3}$, then cost of reads is less than the cost of writes. □

4.5.2 Smaller String Lengths

We now discuss how to improve the ARAM cost for smaller string lengths. If $m \leq M$, then the standard I/O algorithm becomes even better — simply sweep a column through, which remains in small-memory, using $O(m + n)$ reads, no writes, and $O(mn)$ work. Since

there are no writes, we cannot beat that bound. As described already, if $m \geq kM'$ then our algorithm partitions the grid into $kM' \times kM'$ squares, which for larger k saves writes by sacrificing reads and re-computation. The remaining question is what happens when m is larger than small-memory but not too much larger, i.e., $M < m < k_T M'$ or $M < m < k_Q M'$.

When m falls in this range, we apply the algorithm to $m \times kM'$ rectangles, i.e., setting $h = m/M'$. It turns out we can achieve a better bound than Theorem 4.5.1 by increasing k even further. The key observation here is that the bottom of the rectangle no longer needs to be written out because there is no rectangle below it — only the right edge is an output edge. The number of writes per rectangle (Lemma 4.5.4 with $X = hM'$) reduces to $O(h^2k + hM)$. We thus have the following modified version of Corollary 4.5.5

Corollary 4.5.6. *Let $m \leq n$ be the length of the two input strings, with $m \geq M$. Let $h = \Theta(m/M)$, and suppose $k = O(n/M)$ satisfying $h \leq k$. Then it is possible to compute LCS or edit distance of a length m and n input strings with $O(mnhk)$ operations in small-memory, $O(mnhk/M)$ reads to large-memory, and $O(mnh/M^2 + mn/(kM))$ writes to large-memory.*

Proof. There are $\Theta(n/(kM))$ size $m \times (kM/11)$ subgrids. Multiplying by the cost of each grid from Lemma 4.5.4, with $X = hM$, gives $O(nh^2kM)$ operations, $O(nh^2k)$ reads, and $O(nh^2/M + nh/k)$ writes. Substituting one of the h terms with $h = \Theta(m/M)$ gives the theorem. □

The following theorem provides the improved work and ARAM cost in the case that one string is short but the other is long. To understand the bounds here, consider the maximum and minimum values of h for $h = m/M'$ and large ω. If $h = (\omega/M)^{1/3}$, i.e., m is large enough that we can divide into $k_T M' \times k_T M'$ squares, then we get $k'_T = (\omega/M)^{1/3}$ matching the bound in Theorem 4.5.1. As h decreases, the bound improves. In the limit, $h = \Theta(1)$ (or $m = \Theta(M)$), we get $k'_T = \sqrt{\omega/M}$ which is better.

Theorem 4.5.7. *Let $h = \Theta(m/M)$ and suppose that $h \leq k_T$ specified in Theorem 4.5.1. Let $k'_T = \min\{\sqrt{\omega/(hM)}, M/h\}$ and suppose that $n = \Omega(k'_T M)$. Then it is possible to compute the length of the LCS or edit distance with total work of $W(m,n) = O(mn + mn\omega/(k'_T M))$.*

Let $h = \Theta(m/M)$ and suppose that $h \leq k_Q$ specified in Theorem 4.5.1. Let $k'_Q = \min\{\sqrt{\omega/h}, M/h\}$ and suppose $n = \Omega(k'_Q M)$. Then it is possible to compute the length of the LCS or edit distance with an ARAM cost of $Q(m,n) = O(mn\omega/(k'_Q M))$.

Proof. With the restrictions on h, we have $h \leq k$, so Corollary 4.5.6 is applicable. As in proof of Theorem 4.5.1, the second term of the min has the effect that $O(mnh/M^2 + mn/(kM)) = O(mn/(kM))$. The rest of the bound follows by choice of k to makes the cost of writes dominate. □

When n is also small, the bound improves further. In this case, the algorithm consists of building the sketch on a single $m \times n$ grid, so no boundary nodes are output — the only writes that need be performed are the sketch itself.

Theorem 4.5.8. *Let $m \leq n$ be the length of the two input strings, with $m \geq M$. Let $h = \Theta(m/M)$ and let $k = \Theta(n/M)$. Then it is possible to compute the LCS or edit distance of the two strings with work $W(m,n) = O(mnhk + h^2k\omega)$ and ARAM cost $Q(m,n) = O(mnhk/M + h^2k\omega)$.*

Proof. The bound follows directly from Lemma 4.5.4 with $X = 0$ and substituting one hk term in the read/work bounds. □

4.5.3 Recovering the Shortest Path

The standard approach for outputting the shortest path is to trace backwards through the grid from the bottom-rightmost node. This approach assumes that the distances to all internal nodes are known, but unfortunately our algorithm discards distances to interior nodes.

Fortunately, the sketch provides enough information to cheaply traceback a path through each square without any additional writes (except the path itself) and without asymptotically more reads or work. In particular, for any node v in superrow $i+1$, it is not hard to identify a node u in superrow i such that a shortest path to v passes through u. Consider the sketch $\langle s, \ell_s, r_s\rangle, \ldots, \langle i, \ell_i, r_i\rangle, \langle i+1, \ell_{i+1}, r_{i+1}\rangle$ to the segment $\langle i+1, \ell_{i+1}, r_{i+1}\rangle$ that includes v. The vertex u is one of the vertices in the penultimate segment $\langle i, \ell_i, r_i\rangle$ of the sketch, so the goal is to identify which one. To do so, evaluate the sketch to the segment $\langle i, \ell_i, r_i\rangle$. Then perform a horizontal sweep through the final slab, keeping track of the originating vertex from the penultimate segment.

Now suppose we have these vertices u and v that fall along the shortest path and are in consecutive superrows. We also need to identify the path through the slab between u and v. To do that, we apply Hirschberg's [167] recursive low-space algorithm for path recovery in the ED/LCS grid, splitting the horizontal dimension in half on each recursion. Note that the work reduces by a constant fraction in each recursion, but the ARAM cost does not (the only ARAM cost here is from reading the "horizontal" input string), so it may not be immediately obvious that the ARAM cost is cheap enough. Fortunately, after recursing at most $\lg k$ times, the length of the horizontal substring is at most M' and the remaining path-recovery subproblem can be done with no further reads from large-memory.

Putting it all together, tracing a path to the previous superrow requires one sketch evaluation, followed by work that is linear in the area and an ARAM cost that corresponds to reading the horizontal string from large-memory $\lg k$ times. Rounding up loosely, we get work of $O(k(M')^2 + (M)(kM)) = O(kM^2)$ along with $O(kM' + (kM')\lg k) = O(k^2M)$ reads. Multiplying by the h superrows, we have $O(hkM^2)$ work and $O(hk^2M)$ reads from large-memory. Both of these are less than the cost of building the sketch in the first place (Lemma 4.5.4).

Chapter 5

Graph Algorithms

5.1 Overview

In this section we investigate write-efficient graph algorithms. Generally, designing write-efficient algorithms on graph problems is hard because of the difficulty to partition the computation to be small enough to fit into the small-memory on a general graph. In most cases we seek chances in algorithmic designing to trade fewer writes from more reads (and other operations), but in phased Dijkstra introduced in Section 5.4.1 we utilize the small-memory to avoid frequent updates of the priority queue to the large-memory.

In this section we first study undirected graph connectivity and biconnectivity in Section 5.3. We propose sequential and parallel algorithms to solve them using significantly fewer writes than conventional algorithms. Our primary algorithmic tool is the construction of an $o(n)$-sized *implicit decomposition* of a bounded-degree graph G on n nodes, which combined with read-only access to G enables fast answers to connectivity and biconnectivity queries on G. The construction breaks the linear-write "barrier", resulting in costs that are asymptotically lower than conventional algorithms while adding only a modest cost to querying time.

To be more specific, for general graphs with n vertices and m edges, We provide (bi)connectivity oracles that can either be preprocessed in either $O(m + \omega n)$ or $O(\sqrt{\omega}m)$ work with small depth. Then each connectivity query can be answered in $O(1)$ or $O(\sqrt{\omega})$ work, and each biconnectivity query can be answered in $O(1)$ or $O(\sqrt{\omega})$ work. More details of previous and our results can be found in Table 5.1.

Then later in Section 5.4 we investigate write-efficient distance-based graph algorithms. Since most of these distance-based problems are notoriously hard in parallel, we mainly study them in the sequential setting based on (M, ω)-ARAM model. We cover single source shortest paths (SSSP) using Dijkstra in Section 5.4.1 and minimum spanning trees (MST) in Section 5.4.2, and their costs are summarized in Table 5.2. For parallel setting,

	Connectivity		Biconnectivity		Best choice
	Sequential	**Parallel**	**Sequential**	**Parallel**	**when**
Best prior results	$O(m+\omega n)$	$O(\omega m)^\dagger$	$O(\omega m)$	$O(\omega m)^\dagger$	–
This thesis (§5.3.4.2,§5.3.5.2)	$O(m+\omega n)^\dagger$	$O(m+\omega n)^\dagger$	$O(m+\omega n)^\dagger$	$O(m+\omega n)^\dagger$	$m \in \Omega(\sqrt{\omega}n)$
This thesis §5.3.4.3,§5.3.5.3	$O(\sqrt{\omega}m)^\dagger$	$O(\sqrt{\omega}m)^\dagger$	$O(\sqrt{\omega}m)^\dagger$	$O(\sqrt{\omega}m)^\dagger$	$m \in o(\sqrt{\omega}n)$

Table 5.1: Summary of main results for constructing connectivity oracles (n nodes, m edges, †=expected), where $\omega \gg 1$ is the cost of writes to the asymmetric memory. Here "Sequential" indicates the ARAM work, and "Parallel" indicates the work on Asymmetric NP. All parallel algorithms have depth polynomial in $\omega \log n$. For prior results, this table shows the work of the best prior sequential algorithm and parallel algorithm respectively. Compared to prior work, asymmetric memory writes are reduced by up to a factor of ω, yielding improvements in both sequential and parallel settings. Query times are $O(\sqrt{\omega})^\dagger$ (connectivity) and $O(\omega)^\dagger$ (biconnectivity) for the last row and $O(1)$ for the rest. For all algorithms the small symmetric memory is only $O(\omega \log n)$ words.

we also discuss the minimum spanning trees in Section 5.4.2.2, and BFS in Section 5.4.3. The bounds of these algorithms are summarized in Table 5.3.

5.2 Preliminaries and Terminologies

In this section, a graph $G = (V, E)$ has $n = |V|$ vertices and $m = |E|$ edges. Vertices are indexed from 0 to $n-1$. Unless otherwise stated, G can contain self-loops and parallel (duplicate) edges, and is not necessarily connected. We assume a global ordering of the vertices to break ties when necessary. If the degree of every vertex is bounded by a constant, we say the graph is ***bounded-degree***.

We use standard definitions of *spanning tree*, *spanning forest*, *connected component*, *biconnected component*, *articulation points*, *bridge*, and *k-edge-connectivity* on a graph, and *lowest-common-ancestor* (LCA) query on a tree.

A ***spanning tree*** T of an undirected connected graph G is a subgraph that is a tree which includes all of the vertices of G. A ***spanning forest*** of G contains the union of the spanning trees of all connected components in G. The ***lowest-common-ancestor*** (LCA) query for two vertices on a rooted spanning tree requires $O(n)$ work and $O(\log n)$ depth on preprocessing, and $O(1)$ query time [48, 243].

A ***connected component*** of G is a subgraph in which any two vertices are connected to each other by paths via edges in the graph.

Problem	Cost and work on (M, ω)-ARAM
Single-source shortest paths	$Q(n,m) = O\left(\min\left(n\left(\omega + \frac{m}{M}\right), \omega(m + n\log n), m(\omega + \log n)\right)\right)$ $W(n,m) = O(Q(n,m) + n\log n)$
Minimum spanning tree	$Q(n,m) = O\left(m\min\left(\frac{n}{M}, \log n\right) + \omega n\right)$ $W(n,m) = O(Q(n,m) + n\log n)$

Table 5.2: Summary of results on sequential graph algorithms in this thesis on the (M, ω)-ARAM.

Problem	Work on Asymmetric NP	Depth
MST (parallel KKT)	$Q(n,m) = W(n,m) =$ $O(\alpha(n)m + \omega n\log(\min(m/n, \omega)))$	$O(\omega\,\text{polylog}(n))$
Breadth-first search	$Q(n,m) = W(n,m) = O(m + \omega n)$	$O(\omega\Delta\log n)$ *whp*

Table 5.3: Summary of results on parallel graph algorithms in this thesis on the Asymmetric NP. Δ is the diameter of the input graph.

A ***biconnected component*** (also known as a block or 2-connected component) of G is a maximal subgraph such that it is still connected after removing any single vertex in the subgraph. Any connected graph decomposes into a tree of biconnected components called the block-cut tree of the graph. The blocks are attached to each other at shared vertices called ***articulation points***.

A ***bridge*** of G is an edge whose deletion increases the number of connected components of the graph. A connected graph is k***-edge-connected*** if it remains connected whenever fewer than k edges are removed. An unconnected graph is 0-edge connected; a connected graph with bridges is 1-edge-connected; and a bridge-less graph is at least 2-edge-connected.

When $G = (V, E)$ is a weighted graph, the edge lengths are denoted as $l : E \to \mathbb{R}_+$. The ***shortest-path distance*** on the graph G is denoted as $d_G(u, v)$ between nodes u and v in V. Throughout this thesis, we assume that $\min_{x\neq y} d(x, y) = 1$. Let $\Delta = \frac{\max_{x,y} d(x,y)}{\min_{x\neq y} d(x,y)} = \max_{x,y} d(x, y)$, the diameter of the graph G.

5.3 Connectivity / Biconnectivity

5.3.1 Introduction

In this section we look into graph connectivity problems, and in particular whether it is possible to build an "oracle" using a sublinear number of writes that supports fast queries, along with any read-write tradeoffs this entails. We consider undirected connectivity (connected components, spanning forests) and biconnectivity (biconnected components, articulation points, and related 1-edge-connectivity) problems. We do not consider the cost of initially storing the graph in memory, but note that there are many scenarios in which the graph is either represented implicitly, e.g., the Swendsen-Wang algorithm [262], or for which the graph is sampled and used multiple times, e.g., edges selected based on different Boolean hash functions or based on properties (timestamp, weight, relationship, etc.) associated with the edge.

Our results show that if a graph with n vertices and m edges is sufficiently dense, a sublinear number of writes ($o(m)$) can be achieved without asymptotically increasing the number of reads (no tradeoff is required). For bounded-degree graphs, on the other hand, our algorithm achieving a sublinear number of writes ($o(n)$) involves a linear tradeoff between reads and writes. The main technical contribution is a new *implicit* decomposition of a graph that allows writing out information only for a suitably small *sample* of the vertices. We use two models to account for the read-write asymmetry: (i) the (M, ω)-*ARAM model*, in which writes to the asymmetric memory cost $\omega \gg 1$ and all other operations are unit cost; and (ii) its parallel variant, the *Asymmetric NP* model. Both models have a small symmetric memory (a cache) that can be used to help minimize the number of writes to the large asymmetric memory.

Table 5.1 summarizes our main results for these models, showing asymptotic improvements in construction costs over all prior work (sequential or parallel) for these well-studied connectivity problems.

Algorithms with $o(m)$ writes for non-sparse graphs. The first contribution is a group of algorithms that achieve $O(m/\omega + n)$ writes, $O(m)$ other operations, and hence $O(m + \omega n)$ work. While standard sequential BFS- or DFS-based graph connectivity algorithms require only $O(n)$ writes, and hence already achieve this bound, the parallel setting is more challenging. Existing linear-work parallel connectivity algorithms perform $\Theta(m)$ writes [104, 137, 160, 161, 229, 231, 253], and hence are actually $\Theta(\omega m)$ work in the asymmetric memory setting. We show how the algorithm of Shun et al. [253] can be adapted to use only $O(m/\omega + n)$ writes (and $O(m)$ other operations), by avoiding repeated graph contractions and using the write-efficient BFS (discussed later in this section), yielding the first $O(m + \omega n)$ expected work, low-depth parallel algorithm for connectivity in the asymmetric setting. (By ***low depth*** we mean depth polynomial in $\omega \log n$.)

For biconnectivity, the standard output is an array of size m indicating to which biconnected component each edge belongs [108, 183]. Producing this output requires at least m writes, and as a result, the sequential time (and parallel work) ends up being $\Theta(\omega m)$ in the asymmetric memory setting. We present an equally effective representation of the output, which we call the *BC labeling*, which has size only $O(n)$. This leads to a sequential biconnectivity algorithm that constructs the oracle in only $O(m + \omega n)$ time in the asymmetric setting. Moreover, we show how to leverage our new parallel connectivity algorithm to compute the BC labeling in $O(m/\omega + n)$ writes, yielding the first $O(m + \omega n)$ work parallel algorithm for biconnectivity in the asymmetric memory setting. We show:

Theorem 5.3.1. *Graph connectivity and biconnectivity oracles can be constructed in parallel with $O(m + \omega n)$ expected work and $O(\omega^2 \log^2 n)$ depth* whp *on the Asymmetric NP model, and each query can be answered in $O(1)$ work. The symmetric memory is $O(\omega \log n)$ words.*

Algorithms with $o(n)$ writes for sparse graphs. For sparse graphs, the work of our connectivity and biconnectivity algorithms is dominated by the $\Omega(n)$ writes they perform. This led us to explore the following fundamental question: ***Is it possible to construct, using $o(n)$ writes to the asymmetric memory, an oracle for graph connectivity (or biconnectivity) that can answer queries in time independent of n?*** Given that the standard output for these problems (even with BC labeling) is $\Theta(n)$ size even for bounded-degree graphs, one might conjecture that $\Omega(n)$ writes are required. Our main contribution is a (perhaps surprising) affirmative answer to the above question for both the connectivity and biconnectivity problems.

The key technical contribution behind our breaking of the $\Omega(n)$-write "barrier" is the definition and use of an ***implicit k-decomposition*** of a graph. Informally, a *k-decomposition* of a graph G is comprised of a subset S of the vertices, called *centers*, and a mapping $\rho(\cdot)$ that partitions the vertices among the centers, such that (i) $|S| = O(n/k)$, (ii) the number of vertices in each partition is at most k, and (iii) for each center, the induced subgraph on its vertices is connected. However, explicitly storing the center associated with each vertex would require $\Omega(n)$ writes. Instead, an *implicit k-decomposition* defines the mapping implicitly in terms of a procedure that is given only G and S (and a 1-bit labeling on S).

With the new concept of implicit k-decomposition, we present three algorithmic subroutines which together construct connectivity and biconnectivity oracles with $O(m/\sqrt{\omega})$ writes, which is $o(n)$ when $m \in o(\sqrt{\omega}n)$. For clarity of presentation, we begin by assuming the input graph has bounded degree. Section 5.3.6 discusses how to relax this constraint.

We first present an algorithm to compute an implicit k-decomposition that can be constructed in only $O(n/k)$ writes, $O(kn)$ reads, and low depth, and can compute $\rho(v)$ in only $O(k)$ expected reads and no asymmetric memory writes. The intuition behind our construction is first to pick a random subset of the vertices and then map each unpicked vertex to the closest center by performing a BFS on the graph G. Unfortunately, this does

not satisfy the constraint on partition size, so a more sophisticated approach is needed. The unique challenge that arises again and again in the asymmetric context is that the sublinear limitation on the number of writes rules out the approaches used by prior work.

We then show how the implicit k-decomposition can be used to solve graph connectivity and biconnectivity. We define the concept of a *clusters graph*, which contains vertices each representing a cluster and edges between clusters. The key idea is that after precomputing on the clusters graph and storing a constant amount of information about connectivity and biconnectivity on each vertex (corresponding to a cluster in the original graph), a connectivity or biconnectivity query can be answered by only looking at the local structure and preprocessed information on a constant number of clusters. This is straightforward for connectivity queries because we need only compare the labels of the clusters that contains the respective query points. However, this becomes much more challenging in graph biconnectivity since the correspondence between the clusters and biconnected components is non-trivial: a cluster may contain the vertices in many biconnected components while the vertices in a certain biconnected component can belong to different clusters. Therefore, biconnectivity queries require considerable subtleties in the design, to store the appropriate information on the clusters graph that enables each query to access only a constant number of clusters (at most 3). More specifically, we define the concept of the *local graph* of each cluster (it maintains the relationship of biconnectivity of this cluster and its neighbor clusters and can be computed with the cost proportional to the size of this cluster), such that the biconnectivity queries discussed in Section 5.3.5 can be answered by looking up a constant number of local graphs and the information stored in the clusters graph.

Our sequential algorithms have significant algorithmic merits on their own, but we also show that all the algorithms can be made to run in parallel with low depth. We show:

Theorem 5.3.2. *Graph connectivity and biconnectivity oracles can be constructed in $O(m/\sqrt{\omega})$ expected writes and $O(m\sqrt{\omega})$ work on the (M, ω)-ARAM model. The depth on the Asymmetric NP is $O(\omega^{3/2}\log^3 n)$* whp. *Each connectivity query can be answered in $O(\omega^{1/2})$ expected time (work) ($O(\omega^{1/2}\log n)$* whp*) and each biconnectivity query in $O(\omega)$ expected time (work) ($O(\omega \log n)$* whp*). The symmetric memory is $O(\omega \log n)$ words.*

5.3.2 Related Work

Although graph decompositions with various properties have been shown to be quite useful in a large variety of applications (e.g., [1, 33, 34, 35, 63, 201, 216]), to our knowledge none of the prior algorithms provide the necessary conditions for processing graphs with a sublinear number of writes in order to answer connectivity/biconnectivity queries (targeting instead other decomposition properties that are unnecessary in our setting, such as few edges between clusters). For example, Miller et al.'s [216] parallel low-diameter decomposition algorithm requires at least $\Omega(n)$ writes (even if a write-efficient BFS is used), and provides no guarantees on the partition sizes. Similarly, algorithms for size-

balanced graph partitioning (e.g., [22]) require $\Omega(n)$ writes. Our implicit k-decomposition construction is reminiscient of sublinear time algorithms for estimating the number of connected components [46, 88] in its use of BFS from a sample of the vertices. However, their BFS is used for a completely different purpose (approximate counting of $1/n_u$, the inverse of the size of the connected component containing a sampled node u), does not provide a partitioning of the nodes into clusters (two BFS from different nodes can overlap), and cannot be used for connectivity or biconnectivity queries (two BFS from the same connected component may be truncated before intersecting).

5.3.3 Implicit Decomposition

We now introduce the concept of an implicit decomposition. The idea is to partition an undirected graph into connected clusters such that all we need to store to represent the cluster is one representative, which we call the center of the cluster, and some small amount of information on that center (1 bit in our case). The goal is to quickly answer queries on the cluster. The queries we consider are: given a vertex find its center, and given a center find all its vertices. To reduce the amount of symmetric-memory needed, we need all clusters to be roughly the same size. We start with some definitions, which consider an undirected graphs G.

For graph $G = (V, E)$ we refer to the subgraph induced by a subset of vertices as a ***cluster***. A ***decomposition*** of a connected graph $G = (V, E)$ is a vertex subset $S \subset V$, called ***centers***, and a function $\rho(v) : V \to S$, such that the ***cluster*** $C(s) = \{v \in V \mid \rho(v) = s\}$ for each center $s \in S$ is connected. A decomposition is a ***k-decomposition*** if the size of each cluster is upper bounded by k, and $|S| = O(n/k)$. We are often interested in the graph induced by the decomposition, and in particular:

Definition 1 (clusters graph)**.** *Given the decomposition (S, ρ) of a graph $G = (V, E)$, the* **clusters graph** *is the multigraph $G' = (S, \langle\ \{\rho(u), \rho(v)\} : \{u, v\} \in E, \rho(u) \neq \rho(v)\ \rangle)$.*

Definition 2 (implicit decomposition)**.** *An* **implicit decomposition** *of a connected graph $G = (V, E)$ is a decomposition (S, ρ, ℓ) such that $\rho(\cdot)$ is defined implicitly in terms of an algorithm given only G, S, and a (short) labeling $\ell(s)$ on $s \in S$.*

For the algorithms in this thesis, we use implicit k-decompositions. Our goal is to construct and query the decomposition quickly, while using short labels. Our main result is the following.

Theorem 5.3.3. *An implicit k-decomposition (S, ρ, ℓ) can be constructed on an undirected bounded-degree graph $G = (V, E)$ with $|V| = n$ such that:*

- *the construction takes $O(kn)$ operations and $O(n/k)$ writes, both in expectation;*
- *$\rho(v)$ query: finding $\rho(v)$ for any given $v \in V$ takes $O(k)$ operations in expectation and $O(k \log n)$* whp*, and no writes;*
- *$C(s)$ query: finding $C(s)$ for any given $s \in S$ takes $O(k^2)$ operations in expectation, and $O(k^2 \log n)$* whp *and no writes;*

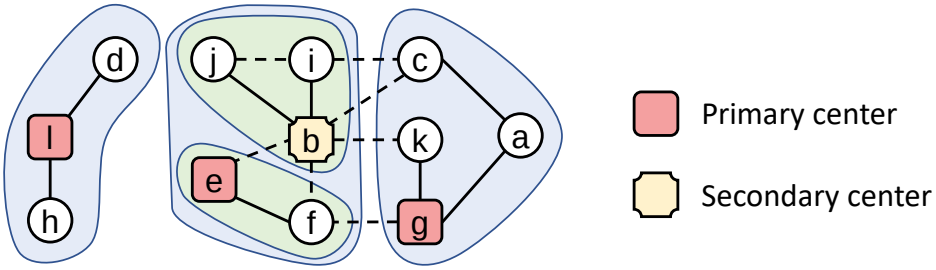


Figure 5.1: An example implicit k-decomposition for $k = 4$ consisting of clusters $\{d, h, l\}$, $\{i, j, b\}$, $\{e, f\}$, and $\{a, c, g, k\}$. In the graph, j's primary center is e (i.e., $\rho_0(j) = e$) and its secondary center is b (i.e., $\rho(j) = b$). Note b is on the shortest path to from j to e. Also note that c is closer to the secondary center b than to g, but picks g as its actual center, because b is not on the shortest path to its primary center. In breaking ties we assume lexicographically smaller letters have higher priorities. The solid lines are on shortest paths from a vertex to its secondary center.

- *the labels $\ell(s), s \in S$ are 1-bit each; and,*
- *construction and queries take $O(k \log n)$ symmetric memory* whp.

Note that this theorem indicates a linear tradeoff between reads (operations) and writes for the construction controlled by k.

At a high level, the construction algorithm works by identifying a subset of centers such that every vertex can quickly find its nearest center without having to keep a pointer to it (which would require too many writes). It first selects a random subset of the vertices where each vertex is selected with probability $1/k$. We call these the ***primary centers*** and denote them as S_0. All other vertices are then assigned to the nearest such center. Unfortunately, a cluster defined in this way can be significantly larger than k (super-polynomial in k). To handle this, the algorithm identifies an additional $O(n/k)$ ***secondary centers***, S_1. Every vertex v is associated with a primary center $\rho_0(v) \in S_0$, and an actual center $\rho(v) \in S = S_0 \cup S_1$. The only values the algorithm stores are the set S and the bits $\ell(s), s \in S$ indicating whether it is primary or secondary. An example is given in Figure 5.1.

In our construction it is important to break ties among equal-length paths in a consistent way, such that subpaths of a shortest path are themselves a unique shortest path. For this purpose we assume the vertices have a total ordering (and comparing two vertices takes constant time). Among two equal hop-length paths from a vertex u, consider the first vertex where the paths diverge. We say that the path with the higher priority vertex at that position is shorter. Let $SP(u, v)$ be the shortest path between u and v under this definition for breaking ties, and $L(SP(u, v))$ be its length such that comparing $L(SP(u, v))$ and $L(SP(u, w))$ breaks ties as defined. By our definition all subpaths of a shortest path

are also unique shortest paths for a fixed vertex ordering. Based on these definitions we specify $\rho_0(v)$ and ρ as follows:

$$\rho_0(v) = \operatorname*{argmin}_{u \in S_0} L(v, u)$$

$$\rho(v) = \operatorname*{argmin}_{u \in S \wedge u \in SP(v, \rho_0(v))} L(v, u)$$

The definitions indicate that a vertex's center is the first center encountered on the shortest path to the nearest primary center. This could either be a primary or secondary center (see Figure 5.1). $\rho(v)$ is defined in this manner to prevent vertices from being reassigned to secondary centers created in other primary clusters, which could result in oversized clusters.

We now describe how to find $\rho(v)$ for any vertex v. First, we find v's closest primary center by running a BFS from v until we hit a vertex in S_0. The BFS orders the vertices by $L(SP(v, \cdot))$. To find $\rho(v)$ we first search for the primary center of v ($\rho_0(v)$) and then identify the first center on the path from v to $\rho_0(v)$, possibly $\rho_0(v)$ itself.

Lemma 5.3.4. *$\rho(v)$ can be found in $O(k)$ operations in expectation, and $O(k \log n)$ operations* whp*, and using $O(k \log n)$ symmetric memory* whp*.*

Proof. Note that the search order from a vertex is deterministic and independent of the sampling used to select S_0. Therefore, the expected number of vertices visited before hitting a vertex in S_0 is k. By tail bounds, the probability of visiting $O(ck \log n)$ vertices before hitting one in S_0 is at most $1/n^c$. The search is a BFS, so it takes time linear in the number of vertices visited. Because the vertices are of bounded degree, placing them in priority order in the queue is easy. Once the primary center is found, a search back on the path gives the actual center. We assume that symmetric memory is used for the search so that no writes to the asymmetric memory are required. The memory used is proportional to the search size, which is proportional to the number of operations; $O(k)$ in expectation and $O(k \log n)$ *whp.* □

The space requirement for the symmetric memory is $O(k \log n)$, which we believe is realistic as we set $k = \sqrt{\omega}$ when using this decomposition later in this chapter.

We use the following lemma to help find $C(s)$ for a center s.

Lemma 5.3.5. *The union of the shortest paths* $\mathrm{SP}(v, \rho(v))$ *for $v \in V$ define a rooted spanning tree on each cluster, with the center as the root (when path edges are viewed as directed towards $\rho(v)$).*

Proof. We first show that this is true for the clusters defined by the primary centers S (i.e., $\rho_0(v)$). We use the notation $SP(v, u) + SP(u, w)$ to indicate joining the two shortest paths at u. Consider a vertex v with $\rho_0(v) = s$, and consider all of the vertices P on

the shortest path from v to s. The claim is that for each $u \in P$, $\rho(u) = s$ and $SP(u, s)$ is a subpath of P. This implies a rooted tree. To see that $\rho(u) = s$ note that the shortest path from u to a primary vertex t has length $L(SP(u, t))$. We can write the length of the shortest path from v to t as $L(SP(v, t)) \leq L(SP(v, u) + SP(u, t))$ and the length of the shortest path from v to s as $L(SP(v, s)) = L(SP(v, u) + SP(u, s))$. Because $\rho_0(v) = s$, we know that $L(SP(v, s)) < L(SP(v, t))\ \forall t \neq s$. Through substitution and subtraction, we see that $L(SP(u, s)) < L(SP(u, t))\ \forall t \neq s$. This means that $\rho_0(u) = s$. We know that $SP(u, s)$ cannot contain the edge b that v takes to reach u in $SP(v, s)$ because $u \in SP(v, s)$. Since the search from u excluding b will have the same priorities as the search from v when it reaches u, $SP(u, s)$ is a subpath of P.

Now consider the clusters defined by $\rho(v)$. The secondary centers associated with a primary center s partition the tree for s into subtrees. Each subtree begins (relative to the root) at a center and ends when encountering another center (this other center is not included). Each vertex in the tree for s will be assigned the correct partition by $\rho(v)$ because each will be assigned to the first secondary center on the way to the primary center. □

The set of solid edges in Figure 5.1 is an example of the spanning forest. This gives the following.

Corollary 5.3.6. *For any vertex v,* $\mathrm{SP}(v, \rho(v)) \subseteq C(\rho(v))$.

Lemma 5.3.7. *For any vertex $s \in S$, its cluster $C(s)$ can be found in $O(k|C(s)|)$ operations in expectation and $O(k|C(s)| \log n)$ operations* whp*, and using $O(|C(v)| + k \log n)$ symmetric memory* whp.

Proof. For any center $s \in S$, identifying all the vertices in its cluster $C(s)$ can be implemented as a BFS starting at s. For each vertex $v \in V$ that the BFS visits, the algorithm checks if $\rho(v) = s$. If so, we add v to $C(s)$ and put its unvisited neighbors in the BFS queue, and otherwise we do neither. By Corollary 5.3.6, any vertex v for which $\rho(v) = s$ must have a path to s only through other vertices whose center is v. Therefore the algorithm will visit all vertices in $C(s)$. Furthermore, because the graph has bounded degree it will only visit $O(C(s))$ vertices not in $C(s)$. Each visit to a vertex u requires finding $\rho(v)$. Our bound on the number of operations therefore follows from Lemma 5.3.4. We use $O(|C(v)|)$ symmetric memory for storing the queue and $C(v)$, and $O(k \log n)$ memory *whp* for calculating $\rho(v)$. □

We now show how to select the secondary centers such that the size of each cluster is at most k. Algorithm 4 describes the process. By Lemma 5.3.5, before the secondary centers are added, each primary vertex in $s \in S_0$ defines a rooted tree of paths from the vertices in its cluster to s. The function SecondaryCenters then recursively cuts up this tree into subtrees rooted at each u that is identified.

Algorithm 4: Constructing Implicit k-Decomposition

Input: Connected bounded-degree graph $G = (V, E)$, parameter k
Output: A set of cluster centers S_0 and S_1 ($S = S_0 \bigcup S_1$)

```
1  Sample each vertex with probability 1/k, and place in S_0
2  S_1 = ∅
3  foreach vertex v ∈ S_0 do
4  |  SecondaryCenters(v, G, S_0)
5  return S_0 and S_1
6  function SecondaryCenters(v, G, S)
7  |  Search from v for the first k vertices that have v as their center. This defines a
   |    tree.
8  |  If the search exhausts all vertices with center v, return.
9  |  Otherwise identify a vertex u that partitions the tree such that its subtree and
   |    the rest of the tree are each at least a constant fraction of k.
10 |  Add u to S_1.
11 |  SecondaryCenters(v, G, S ∪ u)
12 |  SecondaryCenters(u, G, S ∪ u)
```

Lemma 5.3.8. *Algorithm 4 runs in $O(nk)$ operations and $O(n/k)$ writes (both in expectation), and uses $O(k \log n)$ symmetric memory* whp *on the (M, ω)-ARAM Model. It generates a implicit k-decomposition S of G with labels distinguishing S_0 from S_1.*

Proof. The algorithm creates clusters of size at most k by construction (it partitions any cluster bigger than k using the added vertices u). Each call to SecondaryCenters (without recursive calls) will use $O(k^2)$ operations in expectation because we visit k vertices and each one has to search back to v to determine if v is its center. Each call also uses $O(k \log n)$ space for the search *whp* because we need to store the k elements found so far and each $\rho(v)$ uses $O(k \log n)$ space for the search *whp*. Before making the recursive calls, we write out u to S_1, requiring one write per call to SecondaryCenters. The symmetric memory can be reused by the recursive calls.

We are left with showing there are at most $O(n/k)$ calls to SecondaryCenters. There are n/k primary centers in expectation. If there are too many (beyond some constant factor above the expectation), we can try again. Because the graph has bounded degree, we can find a vertex that partitions the tree such that its subtree and the rest of the tree are both at most a constant fraction of k. We can now charge all internal nodes of the recursion against the leaves. There are at most $O(n/k)$ leaves because each defines a cluster of size $\Theta(k)$. Therefore there are $O(n/k)$ calls to SecondaryCenters, giving the stated overall bounds. □

Parallelizing the decomposition. To parallelize the decomposition in Algorithm 4, we make one small change; in addition to adding u to the set of secondary centers at each recursive call to SecondaryCenters, we add all children of v (thus separating v into its own cluster). This guarantees that the height of the tree decreases by at least one on each recursive call, and only increases the number of writes by a constant factor. This gives the following lemma.

Lemma 5.3.9. *On the Asymmetric NP model, Algorithm 4 runs in depth* $O(k^3 \log^2 n)$ whp.

Proof. Certainly selecting the set S_0 can be done in parallel. Furthermore the calls to SecondaryCenters on line 4 can be made recursively in parallel. The depth will be proportional to the depth to each call to SecondaryCenters (not including recursive calls) multiplied by the depth of the recursion. To bound the depth, in the parallel version we also mark the children of the root as secondary centers, which does not increase the number of secondary centers asymptotically (due to the bounded-degree assumption). In this way the height of the tree decreases by one on each recursive call. The depth of the recursion is at most the depth of the tree associated with the primary center $\rho_0(v)$. This is bounded by $O(k \log n)$ *whp* because by Lemma 5.3.4 every vertex finds its primary center within $O(k \log n)$ steps *whp*. The depth of SecondaryCenters (without recursion) is just the number of operations ($O(k^2 \log n)$ *whp*). This gives the bound. □

Extension to unconnected graphs. Note that once a connected component contains at least one primary center, the definition and Theorem 5.3.3 hold. However, it is possible that in a small component, the search of $\rho(\cdot)$ exhausts all connected vertices without finding any primary centers (vertices in the initial sample, S_0). In this case, we check whether the size of the cluster is at least k, and if so, we mark as a primary center the vertex that is the smallest according to the total order on vertices. This marks at most n/k primary centers and the rest of the algorithm remains unchanged. This step is added after line 1 in Algorithm 4, and requires $O(nk)$ work and operations, $O(n/k)$ writes, and $O(k + \log n)$ depth. The cost bound therefore is not changed. If the component is smaller than k, we use the smallest vertex in the component as a center implicitly, but never write it out. The $\rho(\cdot)$ function can easily return this in $O(k)$ operations.

5.3.4 Graph Connectivity and Spanning Forest

This section describes parallel write-efficient algorithms for graph connectivity and spanning forest; that is, identifying which vertices belong to each connected component and producing a spanning forest of the graph. These tasks can be easily accomplished sequentially by performing a breadth-first or depth-first search in the graph with $O(m)$ operations and $O(n)$ writes. While there are several work-efficient parallel algorithms for the problem [104, 137, 160, 161, 229, 231, 253], all of them use $\Omega(n + m)$ writes. This section has two main contributions: (1) Section 5.3.4.2 provides a parallel algorithm using $O(n + m/\omega)$ writes in expectation, $O(n\omega + m)$ expected work, and $O(\omega^2 \log^2 n)$ depth with

high probability; (2) Section 5.3.4.3 gives an algorithm for constructing a connectivity oracle on constant-degree graphs in $O(n/\sqrt{\omega})$ expected writes and $O(n\sqrt{\omega})$ expected total operations. Our oracle-construction algorithm is parallel, having depth $O(\omega^{3/2}\log^3 n)$ *whp*, but it also represents a contribution as a sequential algorithm.

Our parallel algorithm (Section 5.3.4.2) can be viewed as a write-efficient version of the parallel algorithm due to Shun et al. [253]. This algorithm uses a low-diameter decomposition algorithm of Miller et al. [216] as a subroutine, which we review and adapt next in Section 5.3.4.1.

The ***exponential distribution*** with parameter λ is defined by the probability density function:

$$f(x,\lambda) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The mean of the exponential distribution is $1/\lambda$.

5.3.4.1 Low-diameter Decomposition

Here we review the low-diameter decomposition of Miller et al. [216]. The so-called "(β, d)-decomposition" is terminology lifted from their paper, and it should not be confused with our implicit k-decompositions. The details of the decomposition subroutine are important only to extract a bound on the number of writes.

A (β, d)-***decomposition*** of an undirected graph $G = (V, E)$, where $0 < \beta < 1$ and $1 \leq d \leq n$, is defined as a partition of V into subsets $V_1, \ldots, V_\ell$ such that (1) the shortest path between any two vertices in each V_i using only vertices in V_i has length at most d, and (2) the number of edges $(u, v) \in E$ crossing the partition, i.e., such that $u \in V_i$, $v \in V_j$, and $i \neq j$, is at most βm. Miller et al. [216] provide a parallel algorithm for generating a $(\beta, O(\log n/\beta))$-decomposition with $O(m)$ operations and $O((\log^2 n)/\beta)$ depth *whp*. As described by Miller et al., the number of writes performed is also $O(m)$, but this can be improved to $O(n)$. Specifically, the algorithm executes multiple breadth-first searches (BFS's) in parallel, which can be replaced by write-efficient BFS's.

In more detail, the algorithm first assigns each vertex v a value δ_v drawn from an exponential distribution with parameter β (mean $1/\beta$). Then on iteration i of the algorithm, BFS's are started from unexplored vertices v where $\delta_v \in [i, i+1)$ and all BFS's that have already started are advanced one level. At the end of the algorithm, all vertices that were visited by a BFS starting from the same source will belong to the same subset of the decomposition. If a vertex is visited by multiple BFS's in the same iteration, it can be assigned to an arbitrary BFS.[1] The maximum value of δ_v can be shown to be $O(\log n/\beta)$ *whp*, and so the algorithm terminates in $O(\log n/\beta)$ iterations. Each iteration

[1]The original analysis of Miller et al. [216] requires the vertex to be assigned to the BFS with the smaller fractional part of δ_s, where s is the source of the BFS. However, Shun et al. [253] show that an arbitrary assignment gives the same complexity bounds.

requires $O(\log n)$ depth for packing the frontiers of the BFS's, leading to an overall depth of $O(\log^2 n/\beta)$ *whp*. A standard BFS requires operations and writes that are linear in the number of vertices and edges explored, giving a total work of $O(\omega(m+n))$. By using the write-efficient BFS, the expected number of writes for each BFS is proportional to the number of vertices marked (assigned to it), and so the total expected number of writes is $O(n)$. Tasks only need $O(1)$ symmetric memory in the algorithm. This yields the following theorem:

Theorem 5.3.10. *A $(\beta, O(\log n/\beta))$-decomposition can be generated in $O(n)$ expected writes, $O(m+\omega n)$ expected work, and $O(\log^2 n/\beta)$ depth* whp *on the Asymmetric NP model.*

5.3.4.2 Connectivity and Spanning Forest

The parallel connectivity algorithm of [253] applies the low-diameter decomposition recursively with β set to a constant less than 1. Each level of recursion contracts a subset of vertices into a single supervertex for the next level. The algorithm terminates when each connected component is reduced to a single supervertex. The stumbling block for write efficiency is this contraction step, which performs writes proportional to the number of remaining edges.

Instead, our write-efficient algorithm applies the low-diameter decomposition just once, but with a much smaller β, as follows:

1. Perform the low-diameter decomposition with parameter $\beta = 1/\omega + n/m$.
2. Find a spanning tree on each V_i (in parallel) using write-efficient BFS.
3. Create a contracted graph, where each vertex subset in the decomposition is contracted down to a single vertex. To write down the cross-subset edges in a compacted array, employ the write-efficient filter (Section 4).
4. Run any parallel linear-work spanning forest algorithm on the contracted graph, e.g., the algorithm from [104] with $O(\omega \log n)$ depth.

Combining the spanning forest edges across subsets (produced in Step 4) with the spanning tree edges (produced in Step 2) gives a spanning forest on the original graph. Adding the bounds for each step together yields the following theorem. Again only $O(1)$ symmetric memory is required per task.

Theorem 5.3.11. *For any choice of $0 < \beta < 1$, connectivity and spanning forest can be solved in $O(n+\beta m)$ expected writes, $O(\omega n + \beta\omega m + m)$ expected work, and $O(\log^2 n/\beta)$ depth* whp *on the Asymmetric NP model. For $\beta = 1/\omega$, these bounds reduce to $O(n + m/\omega)$ expected writes, $O(m+\omega n)$ expected work and $O(\min\{\omega, m/n\}\log^2 n)$ depth* whp.

Proof. Step 1 has performance bounds given by Theorem 5.3.10, and the expected number of edges remaining in the contracted graph is at most βm. Step 2 performs BFS's on disjoint subgraphs, so summing across subsets yields $O(n)$ expected writes and $O(m+n\omega)$ expected work. Since each tree has low diameter $\mathcal{D} = O(\log n/\beta)$, the BFS's have depth

$O(\mathcal{D} \log n) = O(\log^2 n/\beta)$ *whp* (Section 5.4.3). Step 3 is dominated by the filter, which has a number of writes proportional to the output size of $O(\beta m)$, for $O(m + \beta\omega m)$ work. The depth is $O(\log n)$ (Section 4.2.2). Finally, the algorithm used in Step 4 is not write-efficient, but the size of the graph is $O(n + \beta m)$, giving that many writes and $O(\omega(n + m\beta))$ work. Adding these bounds together yields the theorem. □

5.3.4.3 Connectivity Oracle in Sublinear Writes

A connectivity oracle supports queries that take as input a vertex and return the label (component ID) of the vertex. This allows one to determine whether two vertices belong in the same component. The algorithm is parameterized by a value k, to be chosen later. We assume throughout that the symmetric memory per task is $\Omega(k \log n)$ words and that the undirected graph has bounded degree.

We begin with an outline of the algorithm. The goal is to produce an oracle that can answer for any vertex which component it belongs to in $O(k)$ work. To build the oracle, we would like to run the connectivity algorithm on the clusters graph produced by an implicit k-decomposition. The result would be that all center vertices in the same component be labeled with the same identifier. Answering a query then amounts to outputting the component ID of the center it maps to, which can be queried in $O(k)$ expected work and $O(k \log n)$ work *whp* according to Lemma 5.3.4.

The main challenge in implementing this strategy is that we cannot afford to write out the edges of the clusters graph (as there could be too many edges). Instead, we treat the implicit k-decomposition as an implicit representation of the clusters graph. Given an implicit representation, our connected components algorithm is the following:

1. Find an implicit k-decomposition of the graph.
2. Run the write-efficient connectivity algorithm from Section 5.3.4.2 with $\beta = 1/k$, treating the k-decomposition as an implicit representation of the clusters graph, i.e., querying edges as needed.

As used in the connectivity algorithm, our implicit representation need only be able to list the edges in the clusters graph that are adjacent to a center vertex x. To do so, start at x, and explore outwards (e.g., with BFS), keeping all vertices and edges encountered so far in symmetric memory. For each frontier vertex v, query its center (as in Lemma 5.3.7) — if $\rho(v) = x$, then v's unexplored neighbors are added to the next frontier; otherwise (if $\rho(v) \neq x$) the edge $(x, \rho(v))$ is in the clusters graph, so add it to the output list.

Lemma 5.3.12. *Assuming a symmetric memory of size $\Omega(k \log n)$, the centers neighboring each center in the clusters graph can be listed in no writes, and work, depth, and operations all $O(k^2)$ in expectation or $O(k^2 \log n)$* whp.

Proof. Listing all the vertices in the cluster takes expected work $O(k^2)$ according to Lemma 5.3.7, or $O(k^2 \log n)$ *whp*. The number of vertices in the cluster is $O(k)$, so they can all fit in symmetric memory. Moreover, because each vertex in the cluster has $O(1)$

neighbors, the total number of explored vertices in neighboring clusters is $O(k)$, all of which can fit in symmetric memory. Each of these vertices is queried with a cost of $O(k)$ operations in expectation and $O(k \log n)$ *whp* given the specified symmetric memory size (Lemma 5.3.4). □

Note that a consequence of the implicit representation is that listing neighbors is more expensive, and thus the number of operations performed by a BFS increases, affecting both the work and the depth. The implicit representation is only necessary while operating on the original clusters graph, i.e., while finding the low-diameter decomposition and spanning trees of each of those vertex subsets; the contracted graph can be built explicitly as before. The best choice of k is $k = \sqrt{\omega}$, giving us the following theorem.

Theorem 5.3.13. *A connectivity oracle that answers queries in $O(\sqrt{\omega})$ expected work and $O(\sqrt{\omega} \log n)$ work* whp *can be constructed in $O(n/\sqrt{\omega})$ expected writes, $O(\sqrt{\omega} n)$ expected work, and $O(\omega^{3/2} \log^3 n)$ depth* whp *on the Asymmetric NP model, assuming a symmetric memory of size $\Omega(\sqrt{\omega} \log n)$.*

Proof. The implicit k-decomposition can be found in $O(n/k)$ writes, $O(kn + \omega n/k)$ work, and $O(k^3 \log^2 n)$ depth by Lemmas 5.3.8 and 5.3.9. For $k = \sqrt{\omega}$, these bounds reduce to $O(n/\sqrt{\omega})$ writes, $O(\sqrt{\omega} n)$ work, and $O(\omega^{3/2} \log^3 n)$ depth.

If we had an explicit representation of the clusters graph with $n' = O(n/k)$ vertices and $m' = O(m) = O(n)$ edges, the connectivity algorithm would have $O(n' + m'/k) = O(n/k)$ expected writes, $O(\omega n' + \omega m'/k + m') = O(\omega n/k + n)$ expected work, and $O(k \log^2 n)$ depth *whp* (by Theorem 5.3.11). The fact that the clusters graph is implicit means that the BFS needs to perform $O(k^2)$ additional work (but not writes) per node in the clusters graph, giving expected work $O(\omega n/k + n + k^2 n') = O(\omega n/k + kn)$. To get a high probability bound, the depth is multiplied by $O(k^2 \log n)$, giving us $O(k^3 \log^3 n) = O(\omega^{3/2} \log^3 n)$ for $k = \sqrt{\omega}$. □

We can also output the spanning forest on the contracted graph in the same bounds, which will be used in the biconnectivity algorithms in Sections 5.3.5.3 and 5.3.5.4.

5.3.5 Graph Biconnectivity

In this section we introduce algorithms related to biconnectivity and 1-edge connectivity queries. We first review the classic approach and its output, which requires $\Theta(m)$ writes. Then we propose a new BC (biconnected-component) labeling output, which has size $O(n)$ and can be constructed in $O(n)$ writes. Queries such as determining bridges, articulation points, and biconnected components can be answered in $O(1)$ operations (and no writes) with the BC labeling. Finally we show how an implicit k-decomposition (as generated by Algorithm 4) can be integrated into the algorithm to further reduce the writes to $O(n/\sqrt{\omega})$.

We begin by presenting sequential algorithms that we believe to be new and interesting. Then in Section 5.3.5.4 we show that these algorithms are parallelizable. All of our biconnectivity algorithms use $O(k \log n)$ symmetric memory.

In this section we assume that the graph is connected. If not, we can run the connectivity algorithm and then run the algorithm on each component. The results for a graph are the union of the results of each of its connected components.

5.3.5.1 The Classic Algorithm

The classic Tarjan-Vishkin parallel algorithm [266] to compute biconnected components and bridges of a connected graph is based on the Euler-tour technique. The algorithm starts by building a spanning tree T rooted at some arbitrary vertex. Each vertex is labeled by $\mathit{first}(v)$ and $\mathit{last}(v)$, which are the ranks of v's first and last appearance on the Euler tour of T. The low value $\mathit{low}(v)$ and the high value $\mathit{high}(v)$ of a vertex $v \in V$ are defined as:

$$\mathit{low}(v) = \min\{w(u) \mid u \textit{ is in the subtree rooted at } v\}$$
$$\mathit{high}(v) = \max\{w(u) \mid u \textit{ is in the subtree rooted at } v\}$$

where

$$w(u) = \min\{\mathit{first}(u) \cup \{\mathit{first}(u') \mid (u, u') \textit{ is a nontree edge}\}\}^2$$

Namely, $\mathit{low}(v)$ and $\mathit{high}(v)$ indicate the first and last vertex on the Euler tour that are connected by a nontree edge to the subtree rooted at v. The $\mathit{low}(\cdot)$ and $\mathit{high}(\cdot)$ values can be computed by a reduce on each vertex followed by a leaffix[3] on the subtrees. The computation of low and high takes $O(\log n)$ depth, $O(m+\omega n)$ work, and $O(n)$ writes on the Asymmetric NP model, by using the algorithm and scheduling theorem in Chapter 2 and 4. Then a tree edge is a bridge if and only if the child's *low* and *high* is inclusively within the range of *first* and *last* values of its parent. The cost of this variant of the Tarjan-Vishkin algorithm applied to finding bridges is dominated by the cost of the spanning tree, as given in Theorem 5.3.11.

For biconnected components, the standard output is an array $B[\cdot]$ of size m, where the i-th element in B indicates to which biconnected component the i-th edge belongs [108, 183]. Explicitly writing out B is costly in the asymmetric setting, especially when $m \gg n$. We provide an alternative BC labeling as output that requires only $O(n)$ writes.

5.3.5.2 The BC Labeling

Here we describe the ***BC (biconnected-component) labeling***, which effectively represents biconnectivity output in $O(n)$ space. Instead of storing all edges within each biconnected component, the BC labeling stores a component label for each vertex, and a

[2]If there are multiple edges (u, u') in the graph, none of them are considered here.

[3]Leaffix is similar to prefix but defined on a tree and computed from the leaves to the root.

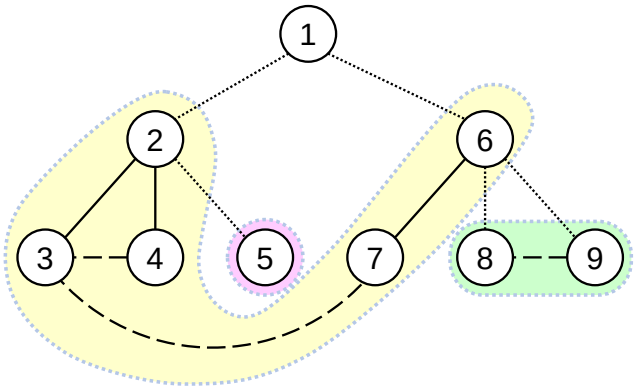


Figure 5.2: An example of the BC labeling of a graph. The spanning tree is rooted at vertex 1. The solid and dot lines indicate tree edges while dot lines are the critical edges. Dash lines are non-tree edges. The vertex labels $l = [1, 1, 1, 2, 1, 1, 3, 3]$ (note that the root does not have a label), and component heads $r = [1, 2, 6]$. Based on the BC labeling the bridges, articulation points, and biconnected components can be easily retrieved as $\{(2, 5)\}$, $\{2, 6\}$, and $\{\{1, 2, 3, 4, 6, 7\}, \{2, 5\}, \{6, 8, 9\}\}$.

vertex for each component. An example of a BC labeling of a graph is shown in Figure 5.2. We will later show how to use this representation along with an implicit decomposition to reduce the writes further.

Definition 3 (BC labeling)**.** *The BC labeling of a connected undirected graph with respect to a rooted spanning tree stores a* **vertex label** $l : V \backslash \{root\} \rightarrow [C]$ *where* C *is the number of biconnected components in the graph, and a* **component head** $r : [C] \rightarrow V$ *of each biconnected component.*

Lemma 5.3.14. *The BC labeling of a connected graph can be computed in* $O(m)$ *operations and* $O(n + m/\omega)$ *writes on the* (M, ω)*-ARAM. Queries to find bridges, articulation points, or biconnected components can be answered in no writes and* $O(1)$ *operations given a BC labeling on a rooted spanning tree.*

The algorithm to compute BC labeling. A vertex $v \in V$ (except for the root) is an articulation point if and only if there exists at least one child u in the spanning tree that has $first(v) \leq low(u)$ and $high(u) \leq last(v)$, and here we call the tree edge between such a pair of vertices a ***critical edge***. The algorithm to compute the BC labeling simply removes all critical edges and runs graph connectivity on all remaining graph edges. Then the algorithm described in Section 5.3.4.2 gives a unique component label that we assign as the vertex label. For each component, its head is the vertex that is its parent in the spanning tree (this parent is unique). Each connected component and its head form a biconnected component.

The correctness of the algorithm can be proven by showing the equivalence of the result of this algorithm and that of the Tarjan-Vishkin algorithm [266].

Because the number of biconnected components is at most n, the spanning tree, vertex labels, and component heads require only linear space. Therefore, the space requirement of the BC labeling is $O(n)$.

Query on BC labeling. We now show that queries are easy with the BC labeling. An edge is a bridge if and only if it is the only edge connecting a single-vertex component and its component head (the biconnected component contains this single edge). The root of the spanning tree is an articulation point if and only if it is the head of at least two biconnected components. Any other vertex is an articulation point if and only if it is the head of at least one biconnected component. A block-cut tree can also be generated from the BC labeling: for each vertex create an edge from itself and its vertex label; and for each component create an edge from the label of this component to the component head. We have a block-cut tree after removing degree-1 nodes corresponding to vertices.

This new representation can be interpreted as an implicit version of the standard output [108, 183] of biconnected components, i.e., the label of the biconnected component of each edge can be reported in $O(1)$ operations. This is simple: we report the label of the endpoint of the edge that is further from the root along the spanning tree. The correctness can be shown in two cases: if the edge is a spanning tree edge, then the component label is stored in the further vertex; otherwise, the two vertices must have the same label and reporting either one gives the label of this biconnected component.

Using BC labeling gives the following theorem (see Section 5.3.5.4 for depth analysis):

Theorem 5.3.15. *Articulation points, bridges, and biconnected components on the Asymmetric NP model take $O(m + n\omega)$ expected work and $O(\min\{\omega, m/n\} \log^2 n)$ depth* whp*, and each query can be answered in $O(1)$ work.*

It is interesting to point out that, the BC labeling can efficiently answer queries that are non-trivial when using the standard output. For example, consider the query: are two vertices in the same biconnected component? With the BC labeling we can answer the query by finding the label of the lower vertex and checking whether the higher one has the same label or is the component head of this component. To the best of our knowledge, answering such queries on the standard representation can be hard, unless other information is also kept (e.g. a block-cut tree).

5.3.5.3 Biconnectivity Oracle in Sublinear Writes

Next we will show how the implicit k-decomposition generated by Algorithm 4 can be integrated into the algorithm to further reduce writes in the case of bounded-degree graphs. Our goal is as follows.

Theorem 5.3.16. *There exists an algorithm that computes articulation points, bridges, and biconnected components of a bounded-degree graph in $O(n\sqrt{\omega})$ expected work, $O(n/\sqrt{\omega})$*

writes and $O(\omega^{3/2}\log^3 n)$ *depth, and each query takes an expected* $O(\omega)$ *work and* $O(\omega \log n)$ *work* whp*, with no writes, on the Asymmetric NP model.*

The overall idea of the new algorithm is to replace the vertices in the original graph with the clusters generated by Algorithm 4. We generate the BC labeling on the clusters graph (so the vertex labels are now the ***cluster labels***), and then show that a biconnectivity query can be answered using only the information on the clusters graph and a constant number of associated clusters. The cost analysis is based on the parameter k, and using $k = \sqrt{\omega}$ gives the result in the theorem.

The BC labeling on the clusters graph.

In the first step of the algorithm we generate the BC labeling on the clusters graph with $k = \sqrt{\omega}$. We root this spanning tree and name it the clusters spanning tree. The head vertex of a cluster is chosen as the ***cluster root*** for that cluster. (The root cluster does not have a cluster root.) For a cluster, we call the endpoint of a cluster tree edge outside of the cluster an ***outside vertex***. The ***outside vertices*** of a cluster is the set of outside vertices of all associated cluster tree edges. Note that all outside vertices except for one are the cluster roots for neighbor clusters.

The local graph of a cluster.

We next define the concept of the local graph of a cluster, so that each query only needs to look up a constant number of associated local graphs. An example of a local graph is shown in Figure 5.3 and a more formal definition is as follows.

Definition 4 (local graph)**.** *The local graph* G' *of a cluster is defined as* $(V_i \cup V_o, E')$*, where* V_i *is the set of vertices in the cluster and* V_o *is the set of outside vertices, and* E' *consists of:*

1. *The edges with both endpoints in this cluster and the associated clusters' tree edges.*
2. *For* c *neighbor clusters sharing the same cluster label, we find the* c *corresponding outside vertices in* V_o*, and connect the vertices with* $c - 1$ *edges.*
3. *For an edge* (v_1, v_2) *with only one endpoint* v_1 *in* V_i*, we find the outside vertex* v_o *that is connected to* v_2 *on the cluster spanning tree, and create an edge from* v_1 *to* v_o*.*

Figure 5.3 shows an example local graph. Solid black lines are edges within the cluster and solid grey lines are cluster tree edges. Neighbor clusters that share a label are shown with dashed outlines and connected via curved dashed lines. e_1 and e_2 are examples of edges with only one endpoint in the cluster, and they are replaced by e_1' and e_2' respectively.

Computing local graphs requires a spanning tree and BC labeling of the clusters graph.

Lemma 5.3.17. *The cost to construct one local graph is* $O(k^2)$ *in expectation and* $O(k^2 \log n)$ whp *on the* (M, ω)*-ARAM.*

Proof. Each cluster in the implicit k-decomposition has at most k vertices, so finding the vertices V_i takes $O(ck)$ cost where c is the cost to compute the mapping $\rho(\cdot)$ of a vertex

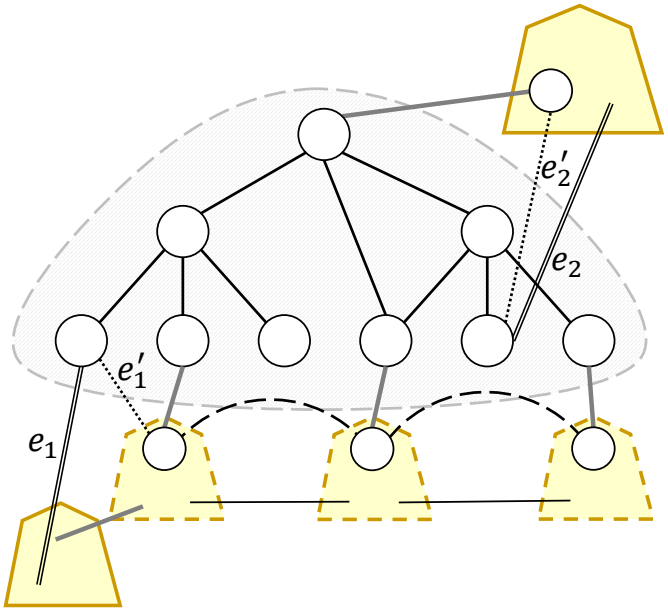


Figure 5.3: An example of a local graph. The vertices in the shaded area is in one cluster. The local graph contains the vertices in the shaded area and the outside vertices shown in smaller circles. Solid lines indicate the edges that are in the clusters and thick grey lines represent cluster tree edges connecting other clusters (which are shown in yellow pentagons). The three neighbor clusters sharing the same cluster label are connected using two edges (dash curves). Edges e_1 and e_2 are the edges that only has one endpoints in the cluster. The other endpoint is set to be the outside vertex connecting the cluster of the other original endpoint of this edge in the cluster spanning tree. Consequently e'_1 and e'_2 are the replaced edges for e_1 and e_2.

Algorithm 5: Sublinear-write algorithm for biconnectivity

Input: Connected bounded-degree graph $G = (V, E)$ and an implicit k-decomposition

1 Apply connectivity algorithm to generate the clusters graph.
2 Compute $low(\cdot)$ and $high(\cdot)$ values of all clusters.
3 Compute the BC labeling of the clusters graph.
`// Bridges and articulation points can be queried`
4 Compute the root biconnectivity of all outside vertices in all local graphs.
5 Apply leafix to identify the articulation point of each cluster root.
`// Biconnectivity and 1-edge connectivity on vertices and edges can be queried`
6 Compute the number of biconnected components in each cluster that are completely within this cluster.
7 Apply prefix sums on the clusters to give an identical label to each biconnected component.
`// The label of biconnected component can be queried`

($O(k)$ in expectation and $O(k \log n)$ *whp*). Since each vertex has a constant degree, there will be at most $O(k)$ neighbor clusters, so $|V_o| = O(k)$. Enumerating and checking the other endpoint of the edges adjacent to V_i takes $O(ck)$ cost. Finding the new endpoint of an edge in category 3 requires constant cost after an $O(n/k)$ preprocessing of the Euler tour of the cluster spanning tree. The number of neighbor clusters is $O(k)$ so checking the cluster labels and adding edges costs no more than $O(k)$. The overall cost to construct one local graph is thus $O(k^2)$ in expectation and $O(k^2 \log n)$ *whp*. After plugging in c is $O(k)$ in expectation and $O(k \log n)$ *whp*, the overall cost matches the bounds in the lemma. □

Queries.

With the local graph and the BC labeling on the clusters graph, many types of biconnectivity queries can be answered. Some of them are easier while other queries require more steps, and the preprocess steps are shown in an overview of Algorithm 5.

Bridges. There are three cases when deciding whether an edge is a bridge: a tree edge in the clusters spanning tree, a cross edge in the clusters spanning tree, or an edge with both endpoints in the same cluster. Deciding which case to use takes constant operations.

A tree edge is a bridge if and only if it is a bridge of the clusters graph, which we can mark with $O(n/k)$ writes while computing the BC labeling. A cross edge cannot be a bridge.

For an edge within a cluster, we use the following lemma:

Lemma 5.3.18. *An edge with both endpoints in one cluster is a bridge if and only if it is a bridge in the local graph of the corresponding cluster.*

Proof. If an edge is a bridge in the original graph it means that there are no edges from the subtree of the lower vertex to the outside except for this edge itself. By applying the modifications of the edges, this property still holds, which means the edge is still a bridge in the local graph and vice versa. □

Checking if an edge in a cluster is a bridge takes $O(k^2)$ work in expectation and $O(k^2 \log n)$ *whp*.

Articulation points. By a similar argument, a vertex is an articulation point of the original graph if and only if it is an articulation point of the associated local graph. Given a query vertex v, we can check whether it is an articulation point the local graph associated to v, which costs $O(k^2)$ work in expectation and $O(k^2 \log n)$ *whp*.

We now discuss how to perform several more complex queries. To start with, we show some definitions and results that are used in the algorithms for queries.

Definition 5 (root biconnectivity)**.** *We say a vertex v in a cluster C's local graph is root-biconnected if v and the cluster root have the same vertex label in C's local graph.*

A root-biconnected vertex v indicates that v can connect to the ancestor clusters without using the cluster root (i.e., the cluster root is not an articulation point to cut v). Another interpretation is that there is no articulation point in cluster C that disconnects v from the outside vertex of the cluster root.

Lemma 5.3.19. *Computing and storing the root biconnectivity of all outside vertices in all local graphs takes $O(nk)$ operations in expectation and $O(n/k)$ writes on the (M, ω)-ARAM.*

The proof is straightforward. The cost to construct the local graphs and compute root biconnectivity is $O(nk)$, and since there are $O(n/k)$ clusters tree edges, storing the results requires $O(n/k)$ writes.

Querying whether two vertices are biconnected. Checking whether two vertices v_1 and v_2 can be disconnected by removing any single vertex in the graph is one of the most commonly-used biconnectivity-type queries. To answer this query, our goal is to find the tree path between this pair of vertices and check whether there is an articulation point on this path that disconnects them.

The simple case is when v_1 and v_2 are within the same cluster. We know that the two vertices are connected by a path via the vertices within the cluster. We can check whether any vertex on the path disconnects these two vertices using their vertex labels.

Otherwise, v_1 and v_2 are in different clusters C_1 and C_2. Assume C_{LCA} is the cluster that contains the LCA of v_1 and v_2 (which can be computed by the LCA of C_1 and C_2 in $O(1)$ operations) and $v_{LCA} \in C_{LCA}$ is the LCA vertex. The tree path between v_1 and v_2 is from v_1 to C_1's cluster root, and then to the cluster root of the outside vertex of C_1's cluster root, and so on, until reaching v_{LCA}, and the other half of the path can be constructed

symmetrically. It takes $O(k^2)$ expected operations to check whether any articulation point disconnects the paths in C_1, C_2 and C_{LCA}. For the rest of the clusters, since we have already precomputed and stored the root biconnectivity of all outside vertices, then applying a leaffix on the clusters spanning tree computes the cluster containing the articulation point of each cluster root. Therefore, checking whether such an articulation point exists on the path between C_1 and C_{LCA} or between C_2 and C_{LCA} that disconnects v_1 and v_2 takes $O(1)$ operations. Thus, checking whether two vertices are biconnected requires $O(k^2)$ operations in expectation and no writes.

Querying whether two vertices are 1-edge connected. This is a similar query comparing to the previous one and the only difference is whether an edge, instead of a vertex, is able to disconnect two vertices. The query can be answered in a similar way by checking whether a bridge disconnects the two vertices on their spanning tree path, which is related to the two clusters containing the two query vertices and the LCA cluster, and the precomputed information for the clusters on the tree path among these three clusters. The cost for a query is also $O(k^2)$ operations in expectation and it requires no writes.

Queries on biconnected-component labels for edges. We now answer the standard queries [108, 183] of biconnected components: given an edge, report a unique label that represents the biconnected component this edge belongs to.

We have already described the algorithm to check whether any two vertices are biconnected, so the next step is to assign a unique label of each biconnected components, which requires the following lemma:

Lemma 5.3.20. *A vertex in one cluster is either in a biconnected component that only contains vertices in this cluster, or biconnected with at least one outside vertex of this cluster.*

Proof. Assume a vertex v_1 in this cluster C is biconnected to another vertex v_2 outside the cluster, then let v_o be the outside vertex of C on the spanning tree path between v_1 and v_2, and v_1 is biconnected with v_o, which proves the lemma. □

With this lemma, we first compute and store the labels of the biconnected components on the cluster roots, which can be finished with $O(nk)$ expected operations and $O(n/k)$ writes with the BC labeling on the clusters graph and the root biconnectivity of outside vertices on all clusters. Then for each cluster we count the number of biconnected components completely within this cluster. Finally we apply a prefix sum on the numbers for the clusters to provide a unique label of each biconnected component in every cluster. Although not explicitly stored, the vertex labels in each cluster can be regenerated with $O(k^2)$ operations in expectation and $O(k^2 \log n)$ operations *whp*, and a vertex label is either the same as that of an outside vertex which is precomputed, or a relative label within the cluster plus the offset of this cluster.

Similar to the algorithm discussed in Section 5.3.5.2, when a query comes, the edge can either be a cluster tree edge, a cross edge, or within a cluster. For the first case the

label biconnected component is the precomputed label for the (lower) cluster root. For the second case we just report the vertex label of an arbitrary endpoint, and similarly for the third case the output is the vertex label of the lower vertex in the cluster. The cost of a query is $O(k^2)$ in expectation and $O(k^2 \log n)$ *whp*.

With the concepts and lemmas in this section, with a precomputation of $O(nk)$ cost and $O(n/k)$ writes, we can also do a normal query with $O(k^2)$ cost in expectation and $O(k^2 \log n)$ *whp* on **bridge-block tree**, **cut-block tree**, and **1-edge-connected components**.

5.3.5.4 Parallelizing Biconnectivity Algorithms

The two biconnectivity algorithms discussed in this section are highly parallelizable. The key algorithmic components include Euler-tour construction, tree contraction, graph connectivity, prefix sum, and preprocessing LCA queries on the spanning tree. Since the algorithms run each of the components a constant number of times, the depth of the algorithm is bounded by the depth of graph connectivity, whose bound is provided in Section 5.3.4 ($O(\omega^2 \log^2 n)$ and $O(\omega^{3/2} \log^3 n)$ *whp* when plugging in β as $1/\omega$ and $1/\sqrt{\omega}$, respectively).[4]

For the sublinear-write algorithm, we assume that computations within a cluster are sequential, and the work is upper bounded by $O(k^2) = O(\omega)$ in expectation and $O(k^2 \log n) = O(\omega \log n)$ *whp* for the computations within a cluster. This term is additive to the overall depth, since after computing the spanning tree (forest) of the clusters, we run all computations within the clusters in parallel and then run tree contraction and prefix sums based on the calculated values. The $O(\omega)$ expected work ($O(\omega \log n)$ *whp*) is also the cost for a single biconnectivity query, and multiple queries can be done in parallel.

5.3.6 Sublinear-Write Algorithms on Unbounded-Degree Graphs

Here we discuss a solution to generate another graph G' which has bounded degree with $O(m)$ vertices and edges, and the connectivity queries on the original graph G can be answered in G' equivalently.

The overall idea is to build a tree structure with ***virtual nodes*** for each vertex that has more than a constant degree. Each virtual node will represent a certain range of the edge list. Considering a star with all other vertices connecting to a specific vertex v_1, we build a binary tree structure with 2 virtual nodes on the first level $v_{1,2\to n/2}$, $v_{1,n/2+1\to n}$, 4 virtual nodes on the second level $v_{1,2\to n/4}, \cdots, v_{1,3n/4+1\to n}$ and so on. We replace the endpoint of an edge from the original graph G with the leaf node in this tree structure

[4]The classic parallel algorithms with polylogarithmic depth solve the Euler-tour construction, tree contraction, and prefix sum, since we here only require linear writes (in terms of number of vertices, $O(n)$ and $O(n/k)$, for the two algorithms) for both algorithms.

that represents the corresponding range with a constant number of edges. Notice that if both endpoints of an edge have large degrees, then they both have to be redirected.

The simple case is for a sparse graph in which most of the vertices are bounded-degree, and the sum of the degrees for vertices with more than a constant number of edges is $O(n/k)$ (or $O(n/\sqrt{\omega})$). In this case we can simply explicitly build a tree structure for the edges of a vertex.

Otherwise, we require the adjacency array of the input graph to have the following property: each edge can query its positions in the edge lists for both endpoints. Namely, an edge (u, v) knows it is the i-th edge in u's edge list and j-th edge in v's edge list. To achieve this, either an extra pointer is stored for each edge, or the edge lists are presorted and the label can be binary searched (this requires $O(\log n)$ work for each edge lookup). With this information, there exists an implicit graph G' with bounded-degree. The binary tree structures can be defined such that given an internal tree node, we can find the three neighbors (two neighbors for the root) without explicitly storing the newly added vertices and edges. Notice that the new graph G' now has $O(m)$ vertices including the virtual ones. The virtual nodes help to generate implicit k-decomposition and require no writes unless they are selected to be either primary or secondary centers.

Graph connectivity is obviously not affected by this transformation. It is easy to check that a bridge in the original graph G is also a bridge in the new graph G' and vice versa. In the biconnectivity algorithm, an edge in G can be split into multiple edges in G', but this will not change the biconnectivity property within a biconnected component, unless the component only contains one bridge edge, which can be checked separately.

This construction, combined with our earlier results, leads to Theorem 5.3.2.

5.3.7 Conclusion

This work provides several algorithms targeted at solving graph connectivity problems considering the read-write asymmetry. Our algorithms make use of an implicit decomposition technique that is applicable beyond the scope of the problems studied in this thesis. By using this decomposition and redundantly performing small computation, we are able to reduce the number of writes in exchange for a small increase in the total number of operations. This allows us to offset the increased cost of writes in anticipated future systems and improve overall performance. Even excluding new memory technology, we believe that research into algorithms with fewer writes provides interesting results from both a theoretical and memory/cache coherence perspective. Our work provides a framework which can be used to develop write-efficient solutions to large graph connectivity problems.

5.4 Distance-based Algorithms

On (M, ω)-ARAM model, breadth-first and depth-first search can be performed with $Q = W = O(\omega n + m)$. In particular each vertex only requires a constant number of writes when it is first added to the frontier (the stack or queue) and a constant number of writes when it is finished (removed from the stack or queue). Searches along an edge to an already visited vertex requires no writes. This implies that several problems based on BFS and DFS also only require $Q = W = O(DFS(n))$. For example, topological sort, biconnected components, and strongly connected components. The analysis is based on the fact that there are only $O(n)$ forward edges in the DFS. However when using priority-first search on a weighed graph (e.g., Dijkstra's or Prim's algorithms) then the problem is more difficult to perform optimally. This is because the priority queue might need to be updated for every edge that is visited. A common theme in many of our algorithms is that they use redundant computations and require a tradeoff between reads and writes. We show how to adapt Dijkstra's single-source shortest-paths algorithm using phases so that the priority queue is kept in small-memory (Section 5.4.1), and briefly sketch how to adapt Borůvka's minimum spanning tree algorithm to reduce the number of shortcuts and hence writes that are needed (Section 5.4.2).

For parallel and write-efficient graph algorithms, we show minimum spanning trees in Section 5.4.2.2, and BFS in Section 5.4.3. The parallelism is from the random exponential growing-with-filtering technique.

5.4.1 Single-Source Shortest Paths

The single-source shortest-paths (SSSP) problem takes a directed weighted graph $G = (V, E)$ and a source vertex $s \in V$, and outputs the shortest distances $d(s, v)$ from s to every other vertex in $v \in V$. For graphs with non-negative edge weights, the most efficient algorithm is Dijkstra's algorithm [114].

In this section we will study (variants of) Dijkstra's algorithm in the asymmetric setting. We describe and analyze three versions (two classical and one new variant) of Dijkstra's algorithm, and the best version can be chosen based on the values of M, ω, the number of vertices $n = |V|$, and the number of edges $m = |E|$.

Theorem 5.4.1. *The SSSP problem on a graph $G = (V, E)$ with non-negative edge weights can be solved with $Q(n, m) = O\left(\min\left(n\left(\omega + \frac{m}{M}\right), \omega(m + n \log n), m(\omega + \log n)\right)\right)$ and $W(n, m) = O(Q(n, m) + n \log n)$, both in expectation, on the (M, ω)-ARAM.*

We start with the classical Dijkstra's algorithm [114], which maintains for each vertex v, $\delta(v)$, a tentative upper bound on the distance, initialized to $+\infty$ (except for $\delta(s)$, which is initialized to 0). The algorithm consists of $n - 1$ iterations, and the final distances from s are stored in $\delta(\cdot)$. In each iteration, the algorithm selects the unvisited vertex u with smallest finite $\delta(u)$, marks it as *visited*, and uses its outgoing edges to relax (update) all of

its neighbors' distances. A priority queue is required to efficiently select the unvisited vertex with minimum distance. Using a Fibonacci heap [128], the work of the algorithm is $O(m + n \log n)$ in the standard (symmetric) RAM model. In the (M, ω)-ARAM, the costs are $Q = W = O(\omega \cdot (m + n \log n))$ since the Fibonacci heap requires asymptotically as many writes as reads. Alternatively, using a binary search tree for the priority queue reduces the number of writes (see Section 4.1) at the cost of increasing the number of reads, giving $Q = W = O(m \log n + \omega m)$. These bounds are better when $m = o(\omega n)$. Both of these variants store the priority queue in large-memory, requiring at least one write to large-memory per edge.

We now describe an algorithm, which we refer to as *phased Dijkstra*, that fully maintains the priority queue in small-memory and only requires $O(n)$ writes to large-memory. The idea is to partition the computation into phases such that for a parameter M' each phase needs a priority queue of size at most $2M'$ and visits at least M' vertices. By selecting $M' = M/c$ for an appropriate constant c, the priority queue fits in small-memory, and the only writes to large-memory are the final distances.

Each phase starts and ends with an empty priority queue P and consists of two parts. A Fibonacci heap is used for P, but is kept small by discarding the M' largest elements (vertex distances) whenever $|P| = 2M'$. To do this P is flattened into an array, the M'-th smallest element d_{max} is found by selection, and the Fibonacci heap is reconstructed from the elements no greater than d_{max}, all taking linear time. All further insertions in a given phase are not added to P if they have a value greater than d_{max}. The first part of each phase loops over all edges in the graph and relaxes any that go from a visited to an unvisited vertex (possibly inserting or decreasing a key in P). The second part then runs the standard Dijkstra's algorithm, repeatedly visiting the vertex with minimum distance and relaxing its neighbors until P is empty. To implement relax, the algorithm needs to know whether a vertex is already in P, and if so its location in P so that it can do a decrease-key on it. It is too costly to store this information with the vertex in large-memory, but it can be stored in small-memory using a hash table.

The correctness of this phased Dijkstra's algorithm follows from the fact that it only ever visits the closest unvisited vertex, as with the standard Dijkstra's algorithm.

Lemma 5.4.2. *Phased Dijkstra's has* $Q(n, m) = O\left(n\left(\omega + \frac{m}{M}\right)\right)$ *and* $W(n, m) = O(Q(n, m) + n \log n)$ *both in expectation (for* $M \leq n$*).*

Proof. During a phase either the size of P will grow to $2M'$ (and hence delete some entries) or it will finish the algorithm. If P grows to $2M'$ then at least M' vertices are visited during the phase since that many need to be deleted with delete-min to empty P. Therefore the number of phases is at most $\lceil n/M' \rceil$. Visiting all edges in the first part of each phase involves at most m insertions and decrease-keys into P, each taking $O(1)$ amortized time in small-memory, and $O(1)$ time to read the edge from large-memory. Since compacting Q when it overflows takes linear time, its cost can be amortized against

the insertions that caused the overflow. The cost across all phases for the first part is therefore $Q = W = O(m\lceil n/M' \rceil)$. For the second part, every vertex is visited once and every edge relaxed at most once across all phases. Visiting a vertex requires a delete-min in small-memory and a write to large-memory, while relaxing an edge requires an insert or decrease-key in small-memory, and $O(1)$ reads from large-memory. We therefore have for this second part (across all phases) that $Q = O(\omega n + m)$ and $W = O(n(\omega + \log n) + m)$. The operations on P each include an expected $O(1)$ cost for the hash table operations. Together this gives our bounds. □

Compared to the first two versions of Dijkstra's algorithm with $Q = W = O(\omega m + \min(\omega n \log n, m \log n))$, the new algorithm is strictly better when $\omega M > n$. More specifically, the new algorithm performs better when $nm/M < \max\{\omega m, \min(\omega n \log n, m \log n)\}$. Combining these three algorithms proves Theorem 5.4.1, when the best one is chosen based on the parameters M, ω, n, and m.

We will discuss the implementation details of Phased Dijkstra later in Section 8.6.2.

5.4.2 Minimum Spanning Tree (MST)

In this section we discuss several commonly-used algorithms for computing a minimum spanning tree (MST) on a weighted graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. Some of them are optimal in terms of the number of writes ($O(n)$). Although loading a graph into large-memory requires $O(m)$ writes, the algorithms are still useful on applications that compute multiple MSTs based on one input graph. For example, it can be useful for computing MSTs on subgraphs, such as road maps, or when edge weights are time-varying functions and hence the graph maintains its structure but the MST varies over time.

Prim's algorithm. All three versions of Dijkstra's algorithm discussed in Section 5.4.1 can be adapted to implement Prim's algorithm [108]. Thus, the upper bounds of ARAM cost and work in Theorem 5.4.1 also hold for minimum spanning trees.

Kruskal's algorithm. The initial sorting phase requires $Q = W = O(m \log n + \omega m)$ (Section 4.4.1). The second phase constructs a MST using union-find without path compression in $O(m \log n)$ work, and performs $O(n)$ writes (the actual edges of the MST). Thus, the complexity is dominated by the first phase. Neither ARAM cost nor work match our variant of Borůvka's algorithm.

5.4.2.1 Borůvka's Algorithm

Borůvka's algorithm consists of at most $\log n$ rounds. Initially all vertices belong in their own component, and in each round, the lightest edges that connect each component to another component are added to the edge set of the MST, and components are merged using these edges. This merging can be done using, for example, depth first search among the components and hence takes time proportional to the number of remaining

components. However, since edges are between original vertices the algorithm is required to maintain a mapping from vertices to the component they belong to. Shortcutting all vertices on each round to point directly to their component requires $O(n)$ writes per round and hence up to $O(n \log n)$ total writes across the rounds. This is not a bottleneck in the standard RAM model but is in the asymmetric case.

We now describe a variant of Borůvka's algorithm which is asymptotically optimal in the number of writes. It requires only $O(1)$ small-memory. The algorithm proceeds in two phases. For the first $\log \log n$ rounds, the algorithm performs no shortcuts (beyond the merging of components). Thus it will leave chains of length up to $\log \log n$ that need to be followed to map each vertex to the component it belongs to. Since there are at most $O(m \log \log n)$ queries during the first $\log \log n$ rounds and each only require reads, the total work for identifying the minimum edges between components in the first phase is $O(m(\log \log n)^2)$. After the first phase all vertices are shortcut to point to their component. We refer to these components as the phase-one components. In the second phase, on every round, we shortcut the phase-one components to point directly to the component they belong to. Since there can only be at most $n/\log n$ phase-one components, and at most $\log n - \log \log n$ rounds in phase-two, the total number of reads and writes for these updates is $O(n)$. During phase two the mapping from a vertex to its component takes two steps: one to find its phase-one component and another to get to the current component. Therefore the total work for identifying the minimum edges between components in the second phase is $O(m \log n)$.

All other work is on the components themselves (i.e., adding the forest of minimum edges and performing DFS to merge components). The number of reads, writes, and other instructions is proportional to the number of components. There are n components on the first round and the number decreases by at least a factor of two on each following round. Therefore the total work on the components is $O(\omega n)$. Summing the costs give the following lemma.

Lemma 5.4.3. *Our variant of Borůvka's algorithm generates a minimum spanning tree on a graph with n vertices and m edges with ARAM cost and work $Q(n, m) = W(n, m) = O(m \log n + \omega n)$ on the (M, ω)-ARAM.*

Theorem 5.4.4. *A minimum spanning tree on a graph $G = (V, E)$ can be computed with ARAM cost $Q(n, m) = O\left(m \min\left(\frac{n}{M}, \log n\right) + \omega n\right)$ and work $W(n, m) = O(Q(n, m) + n \log n)$ on the (M, ω)-ARAM.*

The theorem is a combination of the bounds of Prim's and Borůvka's algorithms (the n/M term is in expectation).

5.4.2.2 KKT Algorithm and the Parallel Version

This section extends Karger, Klein and Tarjan's (KKT) [185] sequential linear-work randomized algorithm for minimum spanning tree/forest. At a high level, their algorithm

proceeds as follows: Randomly sample half of the edges, and calculate a minimum spanning forest on these sampled edges. Use the sampled forest to filter out edges that cannot be part of the overall minimum spanning forest. Specifically, identify all edges $e = (x, y)$ of the graph such that e is the heaviest edge on the cycle it closes in the sampled forest. These edges are discarded. Next, recurse on the remaining graph and perform a constant number "Borůvka steps" to reduce the number of nodes in the graph. The proof that this algorithm runs in linear work hinges on two main facts: first, that the filtering can be done in linear work, and second, that the number of edges filtered out is large in expectation. We will use the following definitions in this section.

Definition 6. *For a given tree T in the graph, and two vertices $x, y \in V$, the path connecting x and y in T (if such a path exists) is denoted by $\tau(x, y)$.*

Definition 7. *An edge $e = (x, y)$ is said to be **heavy** with respect to a tree T on the graph if e closes a cycle in T and it is the heaviest edge in that cycle. That is, $w(e) \geq \max(w(e') \mid e' \in \tau(x, y))$. Any edge that is not heavy with respect to T is said to be **light** with respect to T.*

Recall the well-known *cycle property* of MSTs: a tree T is an MST in a graph G if and only if every non-tree edge closes a cycle in T and is the maximum weight edge on that cycle. Therefore, if all edges of a graph not in T are heavy, then T is an MST of that graph.

The filtering-out of heavy edges is achieved via minimum spanning tree verification algorithms. These algorithms take a tree and a graph as input and determine whether the tree is an MST in the graph. They operate by labeling each edge as light or heavy with respect to the tree. There is extensive work on the verification of minimum spanning trees, and several algorithms are known to operate in linear work [116, 159, 188]. The KKT algorithm then throws out any edges labelled heavy by the verification algorithm, and recurses on the remainder of the graph.

To be efficient, the KKT algorithm requires that when a minimum spanning forest is built on a random subset of the graph, a large fraction of edges in the remaining graph can be filtered out. They show that this holds using the following lemma:[5]

Lemma 5.4.5 (Sampling Lemma)**.** *For a random subset $R \subseteq E$ of size r, and a random subset $S \subseteq E$ of size s, the expected number of edges in S that are light with respect to the MST of R is less than sn/r.*

Using this lemma, Karger et al. prove that their algorithm requires $O(m)$ work with high probability.

Write-efficient MST.

Our goal is to find an MST algorithm that minimizes the number of writes performed without significantly increasing the number of reads. A slight modification of Borůvka's

[5]The statement of this lemma is a slight variation of the version given in [86] and is different from the original paper [185].

Algorithm 6: Write-efficient MST algorithm

Input: A graph $G = (V, E)$
Output: An MST with a set T of edges

```
sampleSize ← max(2n, m/ω)
Edge set T ← {}
while sampleSize < m do
    The set of light edges S_L ← {}
    for s ← 1 to sampleSize do
        Randomly pick an edge e ∈ E
        if e cannot be filtered with respect to T then
            S_L ← S_L + {e}
    T ← KKT(S_L ∪ T)
    sampleSize ← 2 · sampleSize
S_L ← {}
forall e ∈ E do // Take all edges in final round
    if e cannot be filtered with respect to T then
        S_L ← S_L + {e}
T ← KKT(S_L ∪ T)
return T
```

algorithm yields an algorithm which executes in $O(m \log n + \omega n)$ work in the (M, ω)-ARAM model. That is, we can achieve the optimal number of writes ($O(n)$) using $O(m \log n)$ reads. However, we want to find an algorithm that more closely matches the optimal work of the algorithm of Karger et al. [185].

We present an MST algorithm that uses the KKT algorithm as a subroutine, and requires $O(\alpha(n)m)$ reads (where $\alpha(\cdot)$ is the inverse Ackermann function) and $O(n \log(\min(m/n, \omega)))$ writes, resulting in $O(\alpha(n)m + \omega n \log(\min(m/n, \omega)))$ work.

The algorithm is iterative, and proceeds as follows. It begins by taking an $O(n)$ sized random sample of the edges and running KKT on them to find a minimum spanning forest of the sampled graph. Then, in each round, it takes a random sample that is twice as large as the previous one, filters out the edges in the sample that are heavy with respect to the most recently calculated spanning forest, and then runs KKT again on the remaining part of the sample. In each such round, with high probability, we will only be left with $O(n)$ edges from the sample that pass the filtering, and so each round will need $O(n)$ writes. The algorithm proceeds in this way until the final sample includes all edges in the graph. The pseudocode for this algorithm is presented in Algorithm 6.

Analysis.

We start with a sample set S of edges with size $\max(2n, m/\omega)$, and run the KKT algorithm on this sample to calculate a minimum spanning forest on it. We then double the size of the sample set in each round and recalculate, until all edges are processed. However, to save writes, instead of storing all the samples, we use an online filtering algorithm to throw out the edges that are heavy, and only keep the light sample edges. We sample slightly (a constant factor) more than our desired sample size to account for collisions. Note that once the sample size is linear in m, we can sample by flipping a coin with appropriate probabilities for each edge, without increasing the algorithm's total work. We define the edge set S_L to be the subset of S that contains light edges (with respect to the current spanning forest in this round), and the edge set S_H to be the subset consisting of heavy edges. Clearly, $S = S_H \cup S_L$.

Lemma 5.4.6. *At the end of round i of the algorithm, we have a minimum spanning forest on $\Theta(2^{i-1}\max(n, m/\omega))$ edges of the graph in expectation.*

Proof. In round i, we have at least $\Theta(2^{i-1}\max(2n, m/\omega))$ sample edges in expectation, but only build a minimum spanning forest using the edges in S_L, along with the tree we already have. Let T be the set of edges in the current forest, as shown in the pseudocode. Note that by the definition of heavy edges, T is a minimum spanning forest in the graph whose edge set is $S_H \cup T$. Therefore, the MST of $T \cup S_L$ is also a minimum spanning forest on $T \cup S_L \cup S_H$. □

Therefore, at the end of round $\lceil \log_2(\min(m/n, \omega)) \rceil$, we have an MST on the entire graph, and we are done. Note that in the last round, we simply consider all of the edges (without sampling) to ensure that we've seen every edge.

Lemma 5.4.7. *In every round, the expected size of S_L, the number of edges that pass the filtering, is $\Theta(n)$.*

Proof. By Lemma 5.4.6, the MST used to filter edges in round i is a minimum spanning forest on at least $c_1 2^{i-1} n$ edges in expectation for some constant c_1. The sample size in round i is $c_2 2^i n$ in expectation for some constant c_2. By the Sampling Lemma, the expected size of S_L in round i is less than $\dfrac{sn}{r} = \dfrac{c_2(2^i n)n}{c_1 2^{i-1} n} = \Theta(n)$. □

By Lemmas 5.4.6 and 5.4.7, it is easy to see that, excluding any work needed for the filtering, the total number of writes required for this algorithm is $O(n \log(\min(m/n, \omega)))$, and the number of reads is $\sum_{i=1}^{\lceil \log_2(\min(m/n,\omega)) \rceil} \Theta(2^i n) = O(m)$.

Ideally, we would like the filtering step to take no more than $O(n)$ writes per round, and a constant number of reads per edge. There are several MST verification algorithms that take work linear in the number of edges [116, 159, 188]. However, all of these algorithms also take $O(m)$ writes, and are therefore not suitable for us. Alon and Schieber [18] present an online algorithm for minimum spanning tree verification that operates in $O(n)$

preprocessing work, and then $\alpha(n)$ work per queried edge. The queries are done through a look-up in the data structure built in the preprocessing stage, and require no writes. Using their algorithm allows us to execute the entire write-efficient MST algorithm in $O(n \log(\min(m/n, \omega)))$ writes and $O(\alpha(n)m)$ reads.

Alon and Schieber also prove a matching lower bound for the problem of answering online tree product queries, which is a generalization of the MST verification problem. However, they show this lower bound looking at the worst case query. It may be possible to improve upon this result by considering the query time amortized over all of the edges.

Parallel Analysis.

We can parallelize each of the steps of the algorithm. Clearly, the sampling of edges can be done in parallel in constant depth. We then use a parallel version of Alon and Schieber's algorithm [18] to filter out heavy edges. This takes polylogarithmic depth. Cole et al. [104] presented a parallel version of the KKT algorithm that takes linear work and polylogarithmic depth, which we can use instead of the sequential version whenever we call KKT. So simply by using the parallel versions of these algorithms, we achieve a work-efficient polylogarithmic depth algorithm (we only sample $O(\log(\min(m/n, \omega)))$ times, and the depth is only increased by a factor of $O(\log(\min(m/n, \omega)))$).

However, we need to be precise with the number of writes required in the parallel version. After using Alon and Schieber's algorithm to check whether each sampled edge can be filtered out, we need to pack out the edges that passed the filtering to use them in the next round of the algorithm. A standard packing algorithm would execute a number of writes proportional to the total size of the sample, which is too many writes for us. For this, we use the output sensitive filter algorithm presented in Section 4.2.2.

We thus obtain the following theorem:

Theorem 5.4.8. *Given a graph G with m edges and n vertices, an MST of G can be found in $O(\alpha(n)m + \omega n \log(\min(m/n, \omega)))$ work, $O(\text{polylog}(n))$ depth and $O(n \log(\min(m/n, \omega)))$ writes* whp *in the Asymmetric NP model, where $\alpha(\cdot)$ is the inverse Ackerman function.*

5.4.3 Parallel Breadth-First Search

The *breadth-first search* (BFS) problem takes as input an unweighted graph $G = (V, E)$ and a source vertex r, and returns breadth-first search tree rooted at r containing all vertices reachable from r. This section describes a parallel write-efficient BFS algorithm. We will use the notation $n = |V|$ and $m = |E|$. The standard sequential BFS algorithm is write-efficient, but not parallel. It requires $O(m + \omega n)$ work, including $O(n)$ writes (the minimum number of writes required for BFS). On the other hand, the standard parallel level-synchronous BFS algorithm of [53] is not write-efficient, requiring $O(m)$ writes. The algorithm explores the graph in parallel, where round i visits all vertices at a distance i away from r (we call the vertices newly explored in round $i - 1$ the *frontier* for round i). Each frontier vertex visits and writes to all of its unexplored neighbors in parallel,

which causes additional writes when multiple frontier vertices attempt to visit the same vertex simultaneously. This algorithm uses $O(\omega(m+n))$ work, $O(\Delta \log n)$ depth *whp*, and $O(m+n)$ writes, where Δ is the diameter of the input graph.

We now present an algorithm for BFS that runs in $O(m+\omega n)$ work and $O(\Delta \log^2 n)$ depth using only $O(n)$ writes in expectation. The algorithm works like level-synchronous BFS, but makes use of exponential delaying algorithm when visiting the vertices in a round to reduce collisions (and writes) on a shared neighbor. In exponential delaying exploration takes place in iterations, where on each iteration we process a fraction of the vertices on the frontier. We first randomize the order of the frontier vertices, and then on the first iteration we process the first vertex and on iteration $i > 1$, we process the next 2^{i-2} vertices on the frontier. During the exploration process, a vertex checks to see if its neighbor has been visited and only updates that neighbor if it was not visited in a prior round or a prior iteration in the same round.

We now show that the number of writes for this algorithm is $O(n)$ in expectation. Consider a vertex v that is adjacent to the current frontier and has not yet been visited. Let F be the number of vertices on the frontier (without loss of generality, assume it is a power of 2), let $N_F(v)$ be the number of in-neighbors v has on the frontier. We will use $\alpha = 1 - (N_F(v)/F)$ for notational convenience. The probability that v is first visited in iteration 1 is α and in iteration $i > 1$ is:

$$(1-\alpha^{2^{i-2}})\alpha \prod_{j=2}^{i-1} \alpha^{2^{j-2}} = (1-\alpha^{2^{i-2}})\alpha^{2^{i-2}}$$

The expected number of vertices that attempt to visit v in iteration 1 is $N_F(v)/F = 1-\alpha$ and in iteration $i > 1$ is $2^{i-2}(1-\alpha)$. Summing the expectations over all iterations gives:

$$\begin{aligned}
&(1-\alpha)\alpha + \sum_{i=2}^{\log F} 2^{i-2}(1-\alpha)(1-\alpha^{2^{i-2}})\alpha^{2^{i-2}} \\
&< O(1) + \sum_{i=2}^{\log F} 2^{i-2}\alpha^{2^{i-2}} \;=\; O(1)
\end{aligned}$$

where we use $\sum_{x=1}^{\infty} x a^x = O(1)$ for $0 \le a < 1$. This shows that the expected number of writes to a vertex is $O(1)$, and the expected number of writes overall is $O(n)$.

Randomly permuting the vertices sums to linear work overall and $O(\log^2 n)$ depth *whp* per round [217, 254]. The $\log n$ iterations used in the exponential delaying also contributes $O(\Delta \log^2 n)$ to the depth per round. Thus the overall depth is $O(\Delta \log^2 n)$. The number of reads remains $O(m+n)$, which gives the following theorem.

Theorem 5.4.9. *For a graph with n vertices, m edges, and diameter* diam*, our write-efficient breadth-first search algorithm requires $O(m+\omega n)$ work in expectation, $O(\Delta \log^2 n)$ depth* whp*, and $O(n)$ writes in expectation on the Asymmetric NP model.*

Chapter 6

Geometric Algorithms

6.1 Overview

Achieving parallelism (polylogarithmic depth) and optimal write-efficiency simultaneously seems generally hard for many algorithms and data structures in computational geometry. This is because for most geometric algorithms, the computation is not fixed, but decided by the input value. Therefore, simple solutions to partition the computations into smaller chunks and apply the computations within the small-memory are not available. Here, optimal write-efficiency means that the number of writes that the algorithm or data structure construction performs is asymptotically equal to the output size.

This thesis mainly consists of two general frameworks and show how they can be used to design algorithms and data structures from geometry with high parallelism as well as optimal write-efficiency. The first framework is designed for randomized incremental algorithms. Randomized incremental algorithms are relatively easy to implement in practice, and the challenge is in simultaneously achieving high parallelism and write-efficiency. There are several technical parts in this framework. The first part includes several new incremental algorithms, which is the first step of the parallel and write-efficient algorithms. The difficulty of this step is usually in showing the parallelism, and in this approach the new results are based on analyzing the dependence graph of these algorithms. This technique is used in other problems in [61, 164, 225, 254]. The second part is for the write-efficiency (while maintaining parallelism), which further consists of two components: a DAG-tracing algorithm and a prefix doubling technique. The write-efficiency is from the DAG-tracing algorithm, that given a current configuration of a set of objects and a new object, finds the part of the configuration that "conflicts" with the new object. Finding n objects in a configuration of size n requires $O(n \log n)$ reads but only $O(n)$ writes. Once the conflicts have been found, then previous and new parallel incremental algorithms can be used to resolve the conflicts among objects taking linear

reads and writes. This allows for a prefix doubling approach in which the number of objects inserted in each round is doubled until all objects are inserted.

This framework obtains parallel write-efficient algorithms for comparison sort, planar Delaunay triangulation, and k-d trees, all requiring optimal work, linear writes, and polylogarithmic depth. The most interesting result is for Delaunay triangulation (DT). Although DT can be solved in optimal time and linear writes sequentially using the plane sweep method, previous parallel DT algorithms seem hard to make write efficient. Most are based on divide-and-conquer, and seem to inherently require $\Theta(n \log n)$ writes. The DT algorithm in this thesis requires delicate above-mentioned design and analysis in order to achieve parallelism and write-efficiency. For k-d trees, the p-batched incremental construction technique is introduced that maintains the balance of the tree while asymptotically reducing the number of writes.

The second framework is designed for augmented trees, including interval trees, range trees, and priority search trees. The goal is to achieve write-efficiency for both the initial construction as well as future dynamic updates. The framework consists of two techniques. The first technique is to decouple the tree construction from sorting, and introduce parallel algorithms to construct the trees in linear reads and writes after the objects are sorted (the sorting in Chapter 4 can be done with linear writes). Such algorithms provide write-efficient constructions of these data structures, but can also be applied in the rebalancing scheme for dynamic updates—once a subtree is reconstructed once it is unbalanced. The second technique is the α-labeling. Some tree nodes are subselected as critical nodes, and the augmentation is only maintained on these nodes. By doing so the number of tree nodes that need to be written on each update is limited, at the cost of having to read more nodes.

This framework obtains efficient augmented trees in the asymmetric setting. In particular, the trees can be constructed in optimal work and writes, and polylogarithmic depth. For dynamic updates, a trade-off is provided between performing extra reads in queries and updates, while doing fewer writes on updates. A standard algorithm uses $O(\log n)$ reads and writes per update ($O(\log^2 n)$ reads on a 2D range tree). The number of writes can be reduced by a factor of $\Theta(\log \alpha)$ for $\alpha \geq 2$, at a cost of increasing reads by at most a factor of $O(\alpha)$ in the worst case. For example, when the number of queries and updates are about equal, we can improve the work by a factor of $\Theta(\log \omega)$, which is significant given that the update and query costs are only logarithmic.

The previous two frameworks introduce new parallel write-efficient algorithms for comparison sorting, planar Delaunay triangulation, k-d trees, and static and dynamic augmented trees (including interval trees, range trees and priority search trees). We believe the techniques in these frameworks will be useful for designing other algorithms in both the symmetric and asymmetric settings. Also, new parallel write-efficient algorithms for write-sensitive hash tables, and sequential write-efficient algorithms for LP-style algorithms are also discussed in this chapter.

Randomized Incremental Algorithms.

The randomized incremental approach (RIC[1]) has been an extremely useful paradigm for generating simple and efficient algorithms for a variety of problems. There have been dozens of papers on the topic (e.g., see the surveys [218, 248]). Much of the early work was in the context of computational geometry, but the approach has more recently been applied to graph algorithms [101, 107]. The main idea is to insert elements one-by-one in random order while maintaining a desired structure. The random order ensures that the insertions are somehow spread out, and worst-case behaviors are unlikely.

The incremental process is iterative, and hence would appear to be sequential and not write-efficient. In this thesis, we proposed a framework to analyze the computational DAG of the RIC algorithms, to show the parallelism and write-efficiency together in a uniform analysis.

The incremental process would appear sequential since it is iterative, but in practice incremental algorithms are widely used in parallel implementations by allowing some iterations to start in parallel and using some form of locking to avoid conflicts. Many parallel implementations for Delaunay triangulation and convex hull, for example, are based on the randomized incremental approach [49, 82, 98, 99, 113, 144, 202, 230, 252]. In theory, however, after 25 years, there are still no known bounds for parallel Delaunay triangulation using the incremental approach, nor for many other problems.

In this thesis we first show that the incremental approach for Delaunay and many other problem is actually parallel, at least with the right incremental algorithms, and leads to work-efficient polylogarithmic-depth (time) algorithms for the problems. The results are based on analyzing the dependence graph. This technique has recently been used to analyze the parallelism available in a variety of sequential algorithms, including the simple greedy algorithm for maximal independent set [61], the Knuth shuffle for random permutation [254], greedy graph coloring [164], and correlation clustering [225]. The advantage of this method is that one can use standard sequential algorithms with modest change to make them parallel, often leading to very simple parallel solutions. It has also been shown experimentally that this approach leads to quite practical parallel algorithms [60], and to deterministic parallelism [60, 75].

We then show the write-efficient versions of these algorithms. For sorting and Delaunay triangulation, the number of writes is linear, which is optimal since this is the output size. For other linear-work algorithms, the number of writes can be bounded to $O(n^\epsilon)$ for any $\epsilon > 0$. Some analysis here.

The contributions of the section can be summarized as follows.

[1]The letter C is for constrcution. In early works RIC algorithms were used to construct some generalized geometric data structures including convex hull and triangulation. Later work, including this thesis, extended the scope of these algorithms, but we still call them RIC algorithms to follow the convention.

Problem	Work (expected)	Depth (*whp*)	Writes
Comparison sorting (Section 6.4)	$O(n \log^2 n)$	$O(\log n)$	$O(n)$
Planar Delaunay triangulation (Section 6.5)	$O(n \log n)^\dagger$	$O(\log^2 n)$	$O(n)$
k-d tree construction (Section 6.6)	$O(n \log n)^\dagger$	$O(\log^2 n)$	$O(n)$
2D linear programming (Section 6.8.1)	$O(n)$	$O(\log^2 n)$	$O(1)$
Smallest enclosing disk (Section 6.8.3)	$O(n)$	$O(\log^2 n)$	$O(1)$

Table 6.1: Work and depth bounds, and number of writes for the randomized incremental algorithms. The work bounds exclude the extra cost for writes. †: the depth of the linear-write version is $O(\log^2 n \log\log n)$. For the last two algorithms, the write-efficient version does not have the depth guarantees.

1. We describe a framework for analyzing parallelism and write-efficiency in randomized incremental algorithms, and give general bounds on the depth of algorithms with certain dependence probabilities (Section 6.2).
2. We show that randomly ordered insertion into a binary search tree is inherently parallel, leading to an almost trivial comparison sorting algorithm taking $O(\log n)$ depth and $O(n \log n)$ work (i.e., n processors), both with high probability on the priority-write CRCW PRAM (Section 6.4). Surprisingly, we know of no previous description and analysis of this parallel algorithm. The number of writes can remain to be optimal on the Asymmetric NP model. The depth bound is $O(log^2 n)$ since in the Asymmetric NP model we only allow binary instead of arbitrary branching factor.
3. We propose a new randomized incremental algorithm for planar Delaunay triangulation, and then describe a simple way to parallelize it (Section 6.5). The algorithm takes $O(\log^2 n)$ depth with high probability, and $O(n \log n)$ work (i.e., $n/\log n$ processors) in expectation, on the CRCW PRAM. The number of writes can be further reduced to linear on the Asymmetric NP model, with a little sacrifice on the depth of $O(\log\log n)$. It would seem to be by far the simplest work-efficient parallel Delaunay triangulation algorithm.
4. We show that classic sequential randomized incremental algorithms for constant-dimensional linear programming, and smallest enclosing disk can be parallelized (Section 6.8). The number of writes can be reduced to $O(1)$.

6.2 Iteration Dependences for RIC Algorithms

An ***iterative algorithm*** is an algorithm that runs in a sequence of ***steps*** (iterations) in order. When applied to a particular input, we refer to the computation as an ***iterative computation***. Each step i of an iterative computation does some work $W(i)$, and has

some depth $D(i)$ (the steps themselves can be parallel). Step j is said to ***depend*** on step $i < j$ if the computation of step j is affected by the computation of step i. The particular dependences, or even the number of steps, can be a function of the input, and can be modeled as a directed acyclic graph (DAG)—the steps ($I = 1, \ldots, n$) are vertices and dependences between them are arcs (directed edges).

Definition 8 (Iteration Dependence Graph [254])**.** *An **iteration dependence graph** for an iterative computation is a (directed acyclic) graph $G(I, E)$ such that if every step $i \in I$ runs after all predecessor steps in G have completed, then every step will do the same computation as in the sequential order.*

We are interested in the depth (longest directed path) of iteration dependence graphs since shallow dependence graphs imply high parallelism—at least if the dependences can be determined online, and depth of each step $D(i)$ can be appropriately bounded. We refer to the depth of the DAG as the ***iteration depth***, and denote it as $D(G)$. Here we are interested in probabilistic bounds on the iteration depth over random input orders.

An ***incremental algorithm*** is an iterative algorithm that maintains some property over elements while ***inserting*** a new element on each step. We will use $E = \{e_1, \ldots, e_n\}$ to indicate the insertion order of n elements. A ***randomized incremental (RIC) algorithm*** is an incremental algorithm in which the elements are added in a uniformly random order. In randomized incremental algorithms, the presence of a dependence arc between steps i and j will have a probability p_{ij} based on all possible orders (each of the $n!$ orders is a primitive event in the sample space). We are interested in upper bounds on these probabilities, which we will refer to as $\hat{p}_{ij}$. A subtle point is that the exact probabilities p_{ij} are sometimes not independent (e.g., along a path), but the upper bounds $\hat{p}_{ij}$ are, allowing them to be multiplied. We will use backwards analysis [248]—we consider "removing" randomly selected elements one at a time from the end, noting that the analysis of elements $1, \ldots, i$ does not depend on the elements $j > i$.

In this thesis, we consider three types of incremental algorithms, which we refer to as Type 1, 2, and 3, for lack of better names.

Type 1 Algorithms. In these algorithms we analyze the dependence depth by considering all possible paths in the iteration dependence graph and taking a union bound over the probability of each. We describe two algorithms of this type—sorting by insertion into a binary search tree, and incremental planar Delaunay triangulation. In the algorithms (and indeed in just about all incremental algorithms) inserting an element j between two elements $i < j$ and $k > j$ will never add a dependence between i and k (although it might remove one). The property means that we only need to consider the dependence between positions i and $i + 1$ when calculating an upper bound on the probability $\hat{p}_{ij}$ ($j > i$). In particular, for all $j \geq i + 1$ we use $\hat{p}_{i(i+1)} \geq \hat{p}_{ij} \geq p_{ij}$. We use the following lemma.

Lemma 6.2.1. *Consider an iteration dependence graph G of n iterations with $\hat{p}_{ij} = f(i) \geq 1/n$, independent along any path, then*

$$\Pr(D(G) \geq l) < n\left(\frac{e\sum_{i=1}^{n} f(i)}{l}\right)^{l}$$

Proof. Consider a path of length $l-1$, and let $K \subseteq \{1,\ldots,n\}$ be the vertices on the path ($|K| = l$). We have that the probability of the path existing is upper bounded by:

$$P(K) = n\prod_{k\in K} f(k)\ .$$

The multiplicative factor of n is needed to account for the fact that the last element of Kdoes not contribute to the probability of the path and can be as small as $1/n$. We can now take the union bound over all possible paths of length $l-1$, giving:

$$\Pr(D(G) > l-1) \leq X(G) = \sum_{K\subseteq\{1,\ldots,n\},|K|=l} P(K)$$

If $f(i)$ is a constant with value $\hat{p}$ we have:

$$X(G) = \binom{n}{l} n\hat{p}^{l} < n\left(\frac{en\hat{p}}{l}\right)^{l} = n\left(\frac{e\sum_{i=1}^{n} f(i)}{l}\right)^{l}$$

where we use the inequality $\binom{n}{m} < \left(\frac{en}{m}\right)^{m}$.

We now show that unequal (non-constant) probabilities that maintain the same sum $\sum_{i=1}^{n} f(i)$ will only reduce $X(G)$ and hence the upper bound on $\Pr(D(G) \geq l)$. Therefore the probability is maximized by the equation above. Consider two locations i and j such that $f(i) \neq f(j)$. We show that changing these probabilities both to $\hat{p}_m = (f(i) + f(j))/2$ will increase $X(G)$. A path will either go through i but not j, j but not i, neither or both. Clearly the ones through neither will not affect the total sum. For every path through just i there is a path through just j going through the same set of other vertices. If P_r is the product of probabilities of the other vertices in one of these pairs of paths then the contribution to the union bound of both is $f(i)P_r + f(j)P_r = 2\hat{p}_m P_r$. The contribution from these paths is therefore not changed by changing $f(i)$ and $f(j)$ to $\hat{p}_m$. However, the contribution from paths going through both will increase since the old product is $f(i)f(j)P_r$ while the new one is $\hat{p}_m^2 P_r$, which has to be at least as large. □

Corollary 6.2.2. *Consider an iteration dependence graph G of n iterations with $\hat{p}_{ij} \leq c/i$, independent along any path. Then for any $k \geq 2ce^2$ we have*

$$\Pr(D(G) > k\ln n) \in O(1/n^{k-1})\ .$$

Proof. Plugging into Lemma 6.2.1 gives:

$$\begin{aligned}\Pr(D(G) \geq k \ln n) &< n\left(\frac{ec\sum_{i=1}^{n} 1/i}{k \ln n}\right)^{k \ln n} \\ &< n\left(\frac{ec(1+\ln n)}{k \ln n}\right)^{k \ln n} \\ &\leq n(1/e)^{k \ln n} = 1/n^{k-1} \quad \square\end{aligned}$$

To apply the previous lemma or corollary requires showing independence of the upper $\hat{p}_{ij}$ along every path. For sorting this is easy. For Delaunay triangulation the probabilities are not independent among the iterations corresponding to points, but we divide the iterations into sub-iterations, corresponding to the creation of triangles, for which they are independent.

The Type 1 algorithms that we describe can be parallelized by running a sequence of rounds. Each round checks all remaining steps to see if their dependences have been satisfied and runs the steps if so. The algorithms require at most $O(n)$ work per round to check violations. By Theorem 6.2.2, the number of rounds will be $O(\log n)$ *whp*. The total expected work is therefore $O(n \log n)$ for the checks, plus the work for the steps, which is the same as for the sequential variants—$O(n \log n)$ in expectation. The total work is therefore $O(n \log n)$ in expectation.

Type 1 incremental algorithms can be implemented in two ways: one completely online, only seeing a new element at the start of each step, and the other offline, keeping track of all elements from the beginning. In the first case, a structure based on the history of all updates can be built during the algorithm that allows us to efficiently locate the "position" of a new element (e.g., [157]), and in the second case the position of each uninserted element is kept up-to-date on every step (e.g., [100]). The bounds on work are typically the same in either case. Our incremental sort uses an online style algorithm, and the Delaunay triangulation uses an offline one.

Type 2 Algorithms. Here we describe a class of incremental algorithms, called Type 2 algorithms, that have a special structure. The iteration dependence graph for these algorithms is formed as follows: each step j independently has probability at most c/j of being a ***special step*** for some constant c; each special step j has dependence arcs to all steps $i < j$; and all non-special steps have one dependence arc to the closest earlier special step. For Type 2 algorithms, when a special step i is processed, it will check all previous steps, requiring $O(i)$ work and $d(i)$ depth, and when a non-special step is processed it does $O(1)$ work. It can be shown that in expectation each step takes $O(1)$ work, so this means that sequential implementations of Type 2 algorithms take $O(n)$ work in expectation.

Theorem 6.2.3. *A Type 2 incremental algorithm has an iteration dependence depth of* $O(\log n)$ whp*, and can be implemented to run in* $O(n)$ *expected work and* $O(d(n) \log^2 n)$ *depth* whp*, where* $d(n)$ *is the depth of processing a special step.*

Proof. Since the probabilities are independent and the expectation is $\sum_{j=1}^{n} c/j = O(\log n)$, using a Chernoff bound, it is easy to show that the number of special steps is $O(\log n)$ *whp*. With this bound, we can show that the iteration dependence depth, or the length of the longest path in the dependence graph, is $O(\log n)$ *whp* by noticing that there cannot be two consecutive non-special steps in a path (i.e., in the worst case every other vertex in the path is a special step, and there are only $O(\log n)$ of them *whp*).

We now show how parallel linear-work implementations can be obtained. A parallel implementation needs to execute the special steps one-by-one, and for each special step it can do its computation in parallel. For the non-special steps whose closest earlier special step has been executed, their computation can all be done in parallel. To maintain work-efficiency, we cannot afford to keep all unfinished steps active on each round. Instead, we start with a constant number of the earliest steps on the first round and on each round geometrically increase the number of steps processed, similar to the prefix methods described in earlier work on parallelizing iterative algorithms [61].

Without loss of generality, assume $n = 2^k$ for some integer k. We can process batched steps in $k + 1$ rounds—the first round runs the first task and the i-th round ($i > 1$) runs the second half of the first 2^{i-1} tasks (we refer to each batch as a ***prefix***). Each time a prefix is processed it checks all steps, finds the earliest unfinished special step, applies the computation associated with that step, and marks that special step and all earlier steps as finished. Since each step takes $O(1)$ work in expectation, each time a prefix is processed, it applies some computation with work $O(2^{i-2})$ and depth $d(n)$. The maximum/minimum of n elements can be computed in $O(n)$ work and $O(\log n)$ depth, so finding the earliest special step can be done in $O(2^{i-2})$ work and $O(\log n)$ depth *whp*. Marking steps as finished can be done in the same bounds. The number of times a prefix needs to be executed is equal to the number of special steps in the prefix, which for any prefix k is bounded by $\sum_{i=2^{k-2}}^{2^{k-1}-1} c/i = O(1)$ in expectation. Therefore, the work per prefix is $O(2^{i-2})$ in expectation, and summed over all rounds is $O(1) + \sum_{i=2}^{\log n} O(2^{i-2}) = O(n)$ in expectation. Summed across all prefixes, the total number of times they are processed is equal to the iteration dependence depth, which is $O(\log^2 n)$ *whp*, and so we have an overall depth of $O(d(n) \log^2 n)$ *whp*. □

Type 3 Algorithms. In the third type of incremental algorithms, instead of fully abiding by the dependence arcs and bounding the iteration depth, we allow for violations of the dependence arcs, and hence allow computations to differ from the sequential ordering possibly doing some extra work. However, we bound the extra work and show how to resolve the conflicts so the results are the same as in the sequential algorithm.

Consider a set of elements S. We assume that each element $x \in S$ defines a total ordering $<_x$ on all S. This ordering can be the same for each $x \in S$, or different. For example, in sorting the total ordering would be the order of the keys and the same for all

$x \in S$. We say the computation has ***separating dependences*** if the following condition is satisfied.

Definition 9 (separating dependences)**.** *For any three elements $a, b, c \in S$, if $a <_c b <_c c$ or $c <_c b <_c a$, then c can only depend on a if a is inserted first among the three.*

In other words, if b separates a from c in the total ordering for c, and runs first, it will separate the dependence between a and c (also if c runs before a, of course, there is no dependence from a to c). Again using sorting as an example, if we insert b into a BST first (or use it as a pivot in quicksort), it will separate a from c and they will never be compared (each comparison corresponds to a dependence).

Lemma 6.2.4. *In a randomized incremental algorithm that has separating dependences, $\hat{p}_{ij} = 2/i$ is an upper bound on p_{ij}.*

Proof. Consider the total ordering $<_j$. The elements are inserted in random order, and so the probability that i is the nearest element before j among the first i is at most $1/i$. If it is not the nearest, it has already been separated from j by an earlier insertion. Similarly for the nearest element after j, giving a total probability of $2/i$. □

Corollary 6.2.5. *The number of dependences in a randomized incremental algorithm with separating dependences is $O(n \log n)$ in expectation.*[2]

This comes simply from the sum $\sum_{j=2}^{n} \sum_{i=1}^{j-1} p_{ij}$ which is bounded by $2n \ln n$. This leads to yet another proof that quicksort, or randomized insertion into a binary search tree, does $O(n \log n)$ comparisons in expectation. This is not the standard proof based on $p_{ij} = 2/(j - i + 1)$ being the probability that the i'th and j'th smallest elements are compared. Here the p_{ij} represent the probability that the i'th and j'th elements in the random order are compared.

In this thesis we introduce graph algorithms that have separating dependences with respect to insertion of the vertices, and there is a dependence from vertex i to vertex j if a search from i (e.g., shortest path or reachability) visits j.

To allow for parallelism we permit iterations to run concurrently in rounds. This means that we might not separate elements that were separated in the sequential order (for example, if i separated j from k in the sequential order, but we run i and j concurrently, then they might both have a dependence to k). Also, for each algorithm we describe a way to combine results from steps that run in parallel so they give an identical result as the sequential order. If all elements are randomly permuted and the rounds are of geometrically increasing size starting with constant size, as with Type 2 algorithms, the approach wastes at most a constant factor in extra dependences (extra visited vertices in our graph algorithms).

[2] Also true *whp*.

Theorem 6.2.6. *A randomized incremental algorithm with separating dependences can run in $O(\log n)$ parallel rounds over the iterations and every element will have $O(\log n)$ incoming dependence arcs in expectation (for a total of $O(n \log n)$).*

Proof. As in the proof of Theorem 6.2.3, assume without loss of generality $n = 2^k - 1$ for some integer k, and we process batched steps in k rounds where the batch size increases by powers of 2 $(1, 2, 4, 8, \ldots)$. Clearly the number of rounds is $O(\log n)$. Consider the number of incoming arcs to the last element x to be inserted since it is the worst case in expectation. We can consider all elements at each round happening at the very beginning of the round, since in a parallel execution no other elements in the round will separate any elements in the round from x (although all previous rounds can). For round i, the beginning of the round is at position 2^{i-1}. Therefore by Lemma 6.2.4, the probability from any element in round i is $2/2^{i-1}$, and there are 2^{i-1} such elements giving 2 as the expected number of incoming arcs to x from elements in round i. When summed across the $\log n$ rounds we get $2 \log n$, which gives us the $O(\log n)$ bound on incoming arcs as claimed. □

As a side remark we note that by batching we have increased the number of incoming arcs over the sequential algorithm by a factor of about $(2 \log_2 n)/(2 \ln n) = \log_2 e \approx 1.44$.

6.3 General Techniques for Incremental Algorithms

In this section, we first introduce our framework for randomized incremental algorithms. Our goal is to have a systematic approach for designing geometric algorithms that are highly parallel and write-efficient.

Our observation is that it should be possible to make randomized incremental algorithms write-efficient since each newly added object in expectation only conflicts with a small region of the current configuration. For instance, in planar Delaunay triangulation, when a randomly chosen point is inserted, the expected number of encroached triangles is 6. Therefore, resolving such conflicts only makes minor modifications to the configuration during the randomized incremental constructions, leading to algorithms using fewer writes. The challenges are in finding the conflicted region of each newly added object write-efficiently and work-efficiently, and in adding multiple objects into the configuration in parallel without affecting write-efficiency. We will discuss the general techniques to tackle these challenges based on the *history* graph [76, 157], and then discuss how to apply them to develop parallel write-efficient algorithms for comparison sorting in Section 6.4, planar Delaunay triangulation in Section 6.5, and k-d tree construction in Section 6.6.

6.3.1 DAG Tracing

We now discuss how to find the conflict set of each newly added object (i.e., only output the conflict primitives) based on a history (directed acyclic) graph [76, 157] in a parallel and write-efficient fashion. Since the history graphs for different randomized

incremental algorithms can vary, we abstract the process as a DAG tracing problem that finds the conflict primitives in each step by following the history graph.

Definition 10 (DAG tracing problem)**.** *The DAG tracing problem takes an element x, a DAG $G = (V, E)$, a root vertex $r \in V$ with zero in-degree, and a boolean predicate function $f(x, v)$. It computes the vertex set $S(G, x) = \{v \in V \mid f(x, v)$ and out-degree$(v) = 0\}$.*

We call a vertex v ***visible*** if $f(x, v)$ is true.

Definition 11 (tracable property)**.** *We say that the DAG tracing problem has the tracable property when $v \in V$ is visible only if there exists at least one direct predecessor vertex u of v that is visible.*

Variable	Description
$D(G)$	the length of the longest path in G
$R(G, x)$	the set of all visible vertices in G
$S(G, x)$	the output set of vertices

Theorem 6.3.1. *The DAG tracing problem can be solved in $O(|R(G, x)|)$ work, $O(D(G))$ depth and $O(|S(G, x)|)$ writes when the problem has the tracable property, each vertex $v \in V$ has a constant degree, $f(x, v)$ can be evaluated in constant time, and the small-memory has size $O(D(G))$. Here $R(G, x)$, $D(G)$, and $S(G, x)$ are defined in the previous table.*

Proof. We first discuss a sequential algorithm using $O(|R(G, x)|)$ work and $O(|S(G, x)|)$ writes. Because of the tracable property, we can use an arbitrary search algorithm to visit the visible nodes, which requires $O(R(G, x))$ writes since we need to mark whether a vertex is visited or not. However, this approach is not write-efficient when $|S| = o(|R(G, x)|)$, and we now propose a better solution.

Assume that we give a global ordering $\prec_v$ of the vertices in G (e.g., using the vertex labels) and use the following rule to traverse the visible nodes based on this ordering: a visible node $v \in V$ is visited during the search of its direct visible predecessor u that has the highest priority among all visible direct predecessors of v. Based on this rule, we do not need to store all visited vertices. Instead, when we visit a vertex v via a directed edge (u, v) from u, we can check if u has the highest priority among all visible predecessors of v. This checking has constant cost since v has a constant degree and we assume the visibility of a vertex can be verified in constant time. As long as we have a small-memory of size $O(D(G))$ that keeps the recursion stack and each vertex in V has a constant in-degree, we can find the output set $S(G, x)$ using $O(|R(G, x)|)$ work and $O(|S(G, x)|)$ writes.

We note that the search tree generated under this rule is unique and deterministic. Therefore, this observation allows us to traverse the tree in parallel and in a fork-join manner: we can simultaneously fork off an independent task for each outgoing edges of the current vertex, and all these tasks can be run independently and in parallel. The parallel depth, in this case, is upper bounded by $O(D(G))$, the depth of the longest path in the graph. □

Here we assume the graph is explicitly stored and accessible, so we slightly modify the algorithms to generate the history graph, which is straightforward in all cases in this section.

6.3.2 The Prefix-Doubling Approach

The sequential version of randomized incremental algorithms process one object (e.g., a point or vertex) in one iteration. The prefix-doubling approach splits an algorithm into multiple rounds, with the first round processing one iteration and each subsequent round doubling the number of iterations processed. This high-level idea is widely used in parallel algorithm design. We show that the prefix-doubling approach combined with the DAG tracing algorithm can reduce the number of writes by a factor of $\Theta(\log n)$ in a number of algorithms. In particular, our variant of prefix doubling first processes $n/\log n$ iterations using a standard write-inefficient approach (called as the ***initial round***). Then the algorithm runs $O(\log \log n)$ ***incremental rounds***, where the i'th round processes the next $2^{i-1}n/\log n$ iterations.

6.4 Comparison Sorting

The first algorithm that we consider is sorting by incrementally inserting into a binary search tree (BST) with no rebalancing (w.l.o.g. we assume that no two keys are equal)[3]. It is well-known that for a random insertion order, this takes $O(n \log n)$ expected time. We apply our approach to show that the sequential incremental algorithm is also efficient in parallel. Algorithm 7 gives pseudocode that works either sequentially or in parallel. A step is one iteration of the **for** loop on Line 2. For the parallel version, the **for** loop should be interpreted as a **parallel for**, and the assignment on Line 7 should be considered a priority-write—i.e., all writes happen synchronously across the n iterations, and when there are writes to the same location, the smallest value gets written. The sequential version does not need the check on Line 8 since it is always true.

6.4.1 Analysis on the Iterative Dependences

The dependence between iterations in the algorithm is in the check if ${}^*P$ is empty in Line 6. This means that iteration j depends on $i < j$ if and only if the node for i is on the path to j. This leads to the following lemma.

Lemma 6.4.1. *For keys in random order,* INCREMENTALSORT *iteration j depends on iteration $i < j$ with probability at most $2/i$, and this upper bound is independent of all choices for $k > i$.*

[3]Sorting is not a geometric algorithm but is a good example to illustrate the key concepts and our framework.

Algorithm 7: INCREMENTALSORT

Input: A sequence $K = \{k_1, \ldots, k_n\}$ of keys.
Output: A binary search tree over the keys in K.
// $^*P$ reads indirectly through the pointer P.
// The check on Line 8 is only needed for the parallel version.

```
1  Root ← a pointer to a new empty location
2  for i ← 1 to n do
3  |  N ← newNode(k_i)
4  |  P ← Root
5  |  while true do
6  |  |  if *P = null then
7  |  |  |  write N into the location pointed to by P
8  |  |  |  if *P = N then
9  |  |  |  |  break
10 |  |  if N.key < *P.key then
11 |  |  |  P ← pointer to *P.left
12 |  |  else
13 |  |  |  P ← pointer to *P.right
14 return Root
```

Proof. The proof follows the standard analysis (e.g. [248]). We consider the probability for steps i and $i+1$ (and hence an upper bound for all $j > i$). Since it was inserted last, node i is a leaf in the BST when $i+1$ is inserted. Node i will therefore only be on the path to $i+1$ if they are neighbors in **sorted**$(1, \ldots, i+1)$. Node $i+1$ has at most two neighbors, each which is added on step i with probability $1/i$ (independent of all choices of $k > i$), giving $\hat{p}_i = 2/i$. □

Along with Corollary 6.2.2 this implies the following.

Corollary 6.4.2. *Insertion of n keys into a binary search tree in random order has iteration dependence depth $O(\log n)$* whp.

We note that since iterations only depend on the path to the key, the transitive reduction of the iteration dependence graph is simply the BST itself. In general, e.g. Delaunay triangulation in the next section, the dependence structure is not a tree.

Lemma 6.4.3. *The parallel version of* INCREMENTALSORT *generates the same tree as the sequential version, and for a random order of n keys runs in $O(n \log n)$ work and $O(\log n)$ depth* whp *on a priority-write CRCW PRAM.*

Proof. They generate the same tree since whenever there is a dependence, the earliest step wins. The number of rounds of the while loop is bounded by the iteration depth

($O(\log n)$ *whp*) since for each iteration, each round checks a new dependence (i.e., each round traverses one level of the iteration dependence graph). Since each round takes constant depth on the priority-write CRCW PRAM with n processors, this gives the required bounds. □

6.4.2 A Linear-Write Version

We discuss how the DAG-tracing algorithm and prefix doubling in Section 6.3 reduce the number of writes in this algorithm.

Linear-write and $O(\log^2 n \log\log n)$-depth incremental sort. We discuss a linear-write parallel sorting algorithm based on the prefix-doubling approach. The initial round constructs the search tree for the first $n/\log_2 n$ elements using Algorithm 7. For the i'th incremental round where $1 \leq i \leq \lceil \log_2 \log_2 n \rceil$, we add the next $2^{i-1}n/\log_2 n$ elements into the search tree. In an incremental round, instead of directly running Algorithm 7, we first find the correct position of each element to be inserted (i.e., to reach line 7 in Algorithm 7). This step can be implemented using the DAG tracing algorithm, and in this case the DAG is just the search tree constructed in the previous round. The root vertex r is the tree root, and $f(x, v)$ returns true iff the the search of element x visit the node v. Note that the DAG is actually a rooted tree, and also each element only visits one tree node in each level and ends up in one leaf node (stored in P), which means this step requires $O(2^{i-1}n)$ work, $O(\log n)$ depth *whp*, and $O(2^{i-1}n/\log n)$ writes in the i'th round.

After each element finds the empty leaf node that it belongs to, in the second step in this round we then run Algorithm 7, but using the pointer P that was computed in the first step. We refer to the elements in the same empty leaf node as belonging to the same *bucket*. Notice that the depth of this step in one incremental round is upper bounded by the depth of a random binary search tree which is $O(\log n)$ *whp*, so this algorithm has $O(\log^2 n \log\log n)$ depth *whp*: $O(\log\log n)$ rounds, and in each round there are $O(\log n)$ levels.

We now analyze the expected number of writes of in the second step. In each incremental round the number of elements inserted is the same as the number of elements already in the tree. Hence it is equivalent to randomly throwing k balls into k bins, where k is the number of elements to be inserted in this incremental round. Assume that the adversary picks the relative priorities of the elements within each bin, so that it takes $O(b^2)$ work and to sort b elements within each bucket in the worst case. We can show that the probability that there are b elements in a bucket is $\Pr(b) = \binom{k}{b} \cdot 1/k^b(1-1/k)^{k-b}$, and $\Pr(b+1) < \Pr(b) \cdot c_1$ when $b > c_2$, for some constant $c_1 < 1$ and $c_2 > 1$. The expected number of writes within each bucket in this incremental round is therefore:

$$k \cdot \sum_{i=0}^{k} i^2 \Pr(i) < k\left(O(1) + \sum_{i=c_2}^{k} i^2 \cdot \Pr(c_2) c_1^{i-c_2}\right) = O(k)$$

Hence the overall number of writes is also linear. Algorithm 7 sorts b elements in a bucket with $O(b)$ depth, and *whp* the number of balls in each bin is $O(\log k)$, so the depth in this step is included in the depth analysis in the previous paragraph. Combining the work and depth gives the following lemma.

Lemma 6.4.4. INCREMENTALSORT *for a random order of n keys runs in $O(n \log n + \omega n)$ expected work and $O(\log^2 n \log\log n)$ depth* whp *on Asymmetric NP model with priority-write.*

Improving the depth to $O(\log^2 n)$. We can improve the depth to $O(\log^2 n)$ as follows. Notice that for $b = c_3 \log\log n$ and $c_4 > 1$,

$$\sum_{i=b}^{k} \Pr(i) < \sum_{i=b}^{k} \Pr(c_2) c_1^{i-c_2} = \log^{-c_4} k$$

This indicates that only a small fraction of the buckets in each incremental round are not finished after $b = c_3 \log\log n$ iterations of the while-loop on Line 5 of Algorithm 7.

In the depth-improved version of the algorithm, the while-loop terminates after $b = c_3 \log\log n$ iterations, and postpones these insertions (and all further insertions into this subtree in future rounds) to a final round. The final round simply runs another round of Algorithm 9 and inserts all uninserted elements (not write-efficiently). Clearly the depth of the last round is $O(\log^2 n)$, since it is upper bounded by the depth of running Algorithm 7 for all n elements. The depth of the whole algorithm is therefore $O(\log^2 n) + O(\log n \log\log n) \cdot O(\log\log n) + O(\log^2 n) = O(\log^2 n)$ *whp*.

We now analyze the number of writes in the final round. The probability that a bucket in any round does not finish is $\log^{-c_4} k$, and pessimistically there are in total $O(n \cdot \log^{-c_4} k)$ of such buckets. We also know that using Chernoff bound the maximum size of a bucket after the first round is $O(\log^2 n)$ *whp*. The number of writes in the last round is upper bounded by the overall number of uninserted elements times the tree depth, which is $O(n) \cdot \log^{-c_4} n \cdot O(\log^2 n) \cdot O(\log n) = o(n)$ by setting c_3 and c_4 appropriately large. This leads to the main theorem.

Theorem 6.4.5. INCREMENTALSORT *for a random order of n keys runs in $O(n \log n + \omega n)$ expected work and $O(\log^2 n)$ depth* whp *on Asymmetric NP model with priority-write.*

Note that this gives a much simpler work/write-optimal logarithmic-depth algorithm for comparison sorting than the write-optimal parallel sorting algorithm in Chapter 4 that is based on Cole's mergesort [102], although this algorithm is randomized and requires priority-writes.

6.5 Planar Delaunay Triangulation

A Delaunay triangulation (DT) in the plane is a triangulation of a set of points P such that no point in P is inside the circumcircle of any triangle (the circle defined by

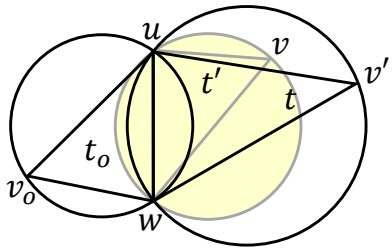


Figure 6.1: An illustration of the procedure of ReplaceTriangle. For each edge (u, w) that is a boundary of v's encroached triangle t, we find the triangle t_o on the other side of (u, w), generate the new triangle t', and recompute the encroaching set $E(t')$. Notice that the new (colored) circumcircle for t' (the encroaching region for t') can only contain points that are in the circumcircles of t and t_o.

the triangle's three corner points). We say a point ***encroaches*** on a triangle if it is in the triangle's circumcircle, and will assume for simplicity that the points are in general position (no three points on a line or four points on a circle). Delaunay triangulation can be solved sequentially in optimal $O(n \log n)$ work. There are also several work-efficient parallel algorithms that run in polylogarithmic depth [31, 51, 238], but they are all quite complicated.

The widely-used incremental Delaunay algorithms, due to their simplicity, date back to the 1970s [150]. They are based on the rip-and-tent idea: for each point p in order, rip out the triangles p encroaches on and tent over the resulting cavity with a ring of triangles centered at p. The algorithms differ in how the encroached triangles are found, and how they are ripped and tented. Clarkson and Shor [100] first showed that randomized incremental 3D convex hull is efficient, running in $O(n \log n)$ time in expectation, which by reduction implies the same results for DT. Guibas et al. (GKS) showed a simpler direct incremental algorithm for DT [157] with the same bounds, and this has become the standard version described in textbooks [109, 121, 218] and often used in practice. The GKS algorithm uses a history of triangle updates to locate the triangle t that p is in. It then searches out for all other encroached triangles. The algorithm, however, is inherently sequential since for certain inputs and certain points in the input, the search from t will likely have depth $\Theta(n)$, and hence a single step can take linear depth.

Our goal is to use an incremental DT algorithm for which the steps themselves can be parallelized. For this purpose we use an offline variant of an algorithm by Boissonnat and Teillaud [76]. We show that the iteration depth is $O(\log n)$ *whp* although this requires analyzing substeps. We further show that each step can be parallelized leading to a simple parallel algorithm with $O(n \log n)$ work in expectation and $O(\log^2 n)$ depth *whp*.

Our variant is described in Algorithm 8. For each triangle $t \in M$ it maintains the set of uninserted points that encroach on t, denoted as $E(t)$. On each step i the algorithm selects the triangles that point i encroach on (all already known), removes these triangles and

Algorithm 8: IncrementalDT

Input: A sequence $V = \{v_1, \ldots, v_n\}$ of points in the plane.
Output: DT(V).
Maintains: A set of triangles M, and for each $t \in M$, the points that encroach on it, $E(t)$.

```
1  t_b ← a sufficiently large bounding triangle
2  E(t_b) ← V
3  M ← {t_b}
4  for i ← 1 to n do
5  |  foreach triangle t ∈ M with v_i ∈ E(t) do
6  |  |  ReplaceTriangle(M,t,v_i)
7  return M

8  function ReplaceTriangle(M,t,v)
9  |  foreach edge (u, w) ∈ t (three of them) do
10 |  |  if (u, w) is a boundary of v's encroached region then
11 |  |  |  t_o ← the other triangle sharing (u, w)
12 |  |  |  t' ← (u, w, v)
13 |  |  |  E(t') ← {v' ∈ E(t) ∪ E(t_o) | inCircle(v', t')}
14 |  |  |  M ← M ∪ {t'}
15 |  M ← M \ {t}
```

replaces them with new ones (see Figure 6.1). All work on uninserted points is done in determining $E(t')$ for each new triangle t', and for each new triangle only requires going through two existing sets, $E(t)$ and $E(t')$. This justified by Fact 6.5.1 [76]. Determining which triangles encroach on a point can be implemented by keeping a mapping of points to encroached triangles.

Fact 6.5.1. *When adding a triangle $t' = (u, w, v)$ for a new point v, and for the two old triangles t and t_o that shared the edge (u, w), we have $E(t) \cap E(t_o) \subseteq E(t') \subseteq E(t) \cup E(t_o)$.*

Proof. Let $t = (u, w, v')$ be the triangle being removed and $t_o = (w, u, v_o)$ be the other triangle sharing the edge (u, w). The new point v must be in the circumcircle of t since it is removing it, but cannot be in the circumcircle of t_o since then it would be removing t_o as well and (u, w) would not be a boundary. The circumcircle of t' therefore must be contained in the union of the circumcircles of t and t_o, and must contain the intersection (see Figure 6.1). □

6.5.1 The Work Bound

A time bound for IncrementalDT of $O(n \log n)$ follows from the analysis of Boissonnat and Teillaud [76], and more indirectly from Clarkson and Shor [100]. However for

completeness and to show precise (within constant factor) bounds we include a bound on the number of InCircle tests here. We note that due to Fact 6.5.1, the InCircle test is not required for points that appear in both $E(t_o)$ and $E(t)$.

Theorem 6.5.2. *IncrementalDT on n points in random order does at most $24n \ln n + O(n)$ InCircle tests in expectation.*

Proof. On step i, for each point at $j > i$ we consider the boundary of the region j encroaches on. We define each of the boundary edges by its two endpoints (u, w) along with the (up to) two points sharing a triangle with (u, w). Note that in ReplaceTriangle a point is only tested for encroachment on the triangle (u, w, v) if its boundary (u, w, v_o, v') is being deleted and replaced with (u, w, v_o, v). We can therefore charge every comparison to the creation of a boundary of a point, and spend it when deleted.

Consider steps i and $i + 1$ (recall we can use $i + 1$ as a surrogate for any $j > i$). By Euler's formula, the average degree of a node in a planar graph is at most 6. Therefore, since $i + 1$ is selected uniformly at random (among $1, \ldots, i + 1$), its expected boundary size will be at most 6. Each boundary involves up to 4 points from $1, \ldots, i$, so the probability that the random point removed on step i is one of them is at most $4/i$. Therefore, the total expected number of boundaries of $i + 1$ (and hence any $j > i$) added on step i is at most $6 \times 4/i = 24/i$. If C is the number of in-circle tests, this gives:

$$\mathbb{E}[C] \leq 3n + \sum_{j=2}^{n} \sum_{i=1}^{j} 24/i \leq 24n \ln n + O(n)$$

where the $3n$ term comes from having to charge for the creation of the initial bounding triangle. □

6.5.2 Analysis on the Iterative Dependences

We now consider the dependence depth of our algorithm. One approach to parallelizing the algorithm is to on each parallel round have every uninserted point check if its dependences are satisfied, and insert itself if so. It turns out that two points i and $i + 1$ are dependent if and only if immediately before either is added, their encroached regions overlap by at least an edge. Unfortunately this means that the probabilities of the dependence arcs (i, j) and (j, k) are not independent. In particular if j has a large encroached region, this increases both p_{ij} and p_{jk}. For example, consider a wagon wheel—$(n - 1)$ points nearly on a circle, and a single point at the hub. When the hub point is inserted at j, it will have dependence arcs from all previous points, and to all future points.

We therefore consider a more fine-grained dependence structure that relaxes the dependences. The observation is that not all triangles added by a point need to be added on the same round. In particular, ReplaceTriangle only depends on the triangle it is replacing and the three neighbors. We therefore can run ReplaceTriangle(M, t, v) as

Algorithm 9: ParIncrementalDT

Input: A sequence $V = \{v_1, \ldots, v_n\}$ of points in the plane.
Output: DT(V).
Maintains: $E(t)$, the points that encroach on each triangle t.

```
1 t_b ← a sufficiently large bounding triangle
2 E(t_b) ← V
3 M ← {t_b}
4 while E(t) ≠ ∅ for any t ∈ M do
5 |   parallel foreach triangle t ∈ M do
6 |   |   Let t_1, t_2, t_3 be the three neighboring triangles
7 |   |   if min(E(t)) ≤ min(E(t_1) ∪ E(t_2) ∪ E(t_3)) then
8 |   |   |   ReplaceTriangle(M, t, min(E(t)))
9 return M
```

long as among the points encroaching on t and the three neighbors of t, there is no earlier point than v. An equal point is fine, since that would be the same point.

Algorithm 9 describes such a parallel variant. Since the triangles for a given point can be added on different rounds, the mesh is not necessarily self consistent after each round. We therefore assume that if a neighboring triangle (i.e., t_1, t_2 or t_3) is already deleted it can be ignored, and if not yet added, then ReplaceTriangle cannot proceed until added. A hash table mapping pairs of vertices representing edges to their up to two adjacent triangles can be used to find neighboring triangles. We assume that there is a synchronization point before Line 8.

Note that there is a one-to-one correspondence between the calls to ReplaceTriangle in the sequential and parallel algorithm—i.e., they are the "same" algorithm but just with a different ordering. We believe that this parallel version is even simpler than the sequential version since it does not require a mapping from points to encroached triangles.

For a sequence of points V, let $G_T(V) = (T, E)$ be the dependence graph defined by ParIncrementalDT(V) in the following way. The vertices T corresponds to triangles created by the algorithm, and for each call to ReplaceTriangle(M, t, v_i) we place an arc from triangle t and its three neighbors (t_1, t_2 and t_3) to each of the one, two, or three triangles created by ReplaceTriangle. Note that we can associate each triangle with the point v_i that created it. This is an iteration dependence graph over all iterations, including subiterations that create triangles.

Theorem 6.5.3. *For points V in random order, $D(G_T(V)) = O(\log n)$ whp.*

Proof. For a sequence of points V let $T(V, i)$ be the set of triangles created by point v_i, and let $E_{t,t'}(V)$ be the indicator variable for a dependence arc from triangle t to t' given V. Let $\hat{p}_{ij}$ be an upper bound for the total probability that triangles created by v_i have an arc

to any single triangle created by v_j (uniformly random over all permutations of the input). More precisely:

$$\hat{p}_{ij} \geq \frac{1}{|V|!}\left(\sum_{V' \in \text{perms}(V)}\left(\max_{t \in T(V',j)}\left(\sum_{t' \in T(V',i)} E_{t,t'}(V')\right)\right)\right).$$

Consider a path going through a triangle created by each point $K \subseteq \{1,\ldots,n\}$. If the $\hat{p}_{ij}$ are independent along any path, then the probability of such a path is bounded by the product of the $\hat{p}_{ij}$ along the path and the expectation on the number of triangles on the last point (which is 6 and independent of the $\hat{p}_{ij}$). For $\hat{p}_{ij} = f(i) \geq 1/n$ this gives a total bounded by $6n\prod_{k\in K} f(k)$. We can now apply the proof of Lemma 6.2.1, where the $\hat{p}_{ij}$ are interpreted in this new way, and the probability along a path $K \subseteq \{1,\ldots,n\}$ is interpreted as the probability that any path exists involving triangles created by those points. As in the proof of Lemma 6.2.1, the union bound over all possible subsets K of length l gives an overall upper bound on the probability of any path of length $l-1$:

$$\Pr(D(G_T(V)) > l-1) \leq \sum_{K\subseteq\{1,\ldots,n\},|K|=l} 6n\prod_{k\in K} f(k)$$

and assuming that $\hat{p}_{ij} = f(i) = O(1/i)$ gives $O(\log n)$ depth *whp* (the argument about equal probabilities being the worst case still holds).

We are therefore left with showing that the $\hat{p}_{ij} = O(1/i)$ is a valid upper bound, and that this bound is independent along any path (allowing us to multiply them). As usual we consider steps (points) i and $j = i+1$, and hence an upper bound for any $j > i$. We consider one triangle t' created at $i+1$. Every triangle t' depends on 4 triangles—the t that was sacrificed for it in ReplaceTriangle, and its three neighbors (see Figure 6.2). These 4 triangles have six corners in total, any one of which could be the point i. Three of those points (a, b and c in the figure) would create three triangles that t' depends on. The other three (d, e and f in the figure) only create one triangle t' depends on. Therefore given that the point i is selected uniformly at random from i points the total probability that triangles at i have an arc to t' (a triangle at $i+1$) is bounded by $\hat{p}_{ij} = (3\cdot 3 + 3\cdot 1)/i = 12/i$.

The probabilities are independent since $\hat{p}_{ij} = 12/i$ does not depend on the point j, or indeed any of the points selected in positions $(i+1),\ldots,n$. For example, conditioned on the center of the wagon wheel being at j, $\hat{p}_{ij} = 12/i$ is still an upper bound. With $\hat{p}_{ij} = O(1/i)$, and independence of the $\hat{p}_{ij}$ along paths, we can apply our variant of Lemma 6.2.1 (described above), and Corollary 6.2.2 for the result. □

Theorem 6.5.4. *ParIncrementalDT (Algorithm 9) runs in $O(n\log n)$ work in expectation and $O(\log^2 n)$ depth* whp *on the CRCW PRAM.*

Proof. The number of rounds of ParIncrementalDT is $D(G_T(V))$ since the iteration dependence graph is defined by the algorithm. Each round has depth $O(\log n)$ for merging

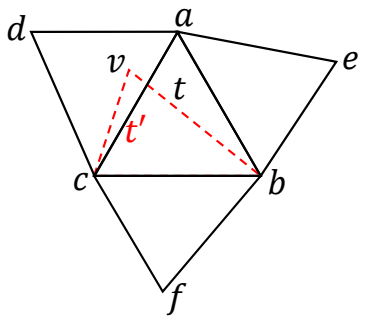


Figure 6.2: The dependence of t' on four previously created triangles.

the encroached sets and load balancing, for an overall depth of $O(\log^2 n)$ *whp*. Assuming each triangle maintains its minimum index, checking if a triangle is safe to process takes constant work. Since there are be at most $O(n)$ triangles on any round (true even though the mesh is not necessarily consistent), each round does at most $O(n)$ work to check all the triangles, for a total of $O(n \log n)$ work across rounds *whp*. The rest of the work is no more than the sequential version, which is $O(n \log n)$ in expectation. □

6.5.3 A Linear-Write Version

We now discuss a write-efficient version of the incremental Delaunay algorithm. We use the DAG tracing and prefix-doubling techniques introduced in Section 6.3. The algorithm first computes the DT of the $n/\log_2 n$ earliest points in the randomized order, using the non-write-efficient version. This step requires linear writes. It then runs $O(\log \log n)$ incremental rounds and in each round adds a number of points equal to the number of points already inserted.

To insert points, we need to construct a search structure in the DAG tracing problem. We can modify the BGSS algorithm to build such a structure. In fact, the structure is effectively a subset of the edges of the dependence graph $G_T(V)$. In particular, in the algorithm the only inCircle test is on Line 13. In this test, to determine if a point encroaches t', we need only check its two ancestors t and t_o (we need not also check the two other triangles neighboring t, as needed in $G_T(V)$). This leads to a DAG with depth at most as large as $G_T(V)$, and for which every vertex has in-degree 2. The out-degree is not necessarily constant. However, by noting that there can be at most a constant number of outgoing edges to each level of the DAG, we can easily transform it to a DAG with constant out-degree by creating a copy of a triangle at each level after it has out-neighbors. This does not increase the depth, and the number of copies is at most proportional to the number of initial triangles ($O(n)$ in expectation) since the in-degrees are constant. We refer to this as the *tracing structure*. An example of this structure is shown in Figure 6.3.

The tracing structure can be used in the DAG tracing problem (Definition 10) using the predicate $f(v, t) = \text{inCircle}(v, t)$. This predicate has the traceable property since a

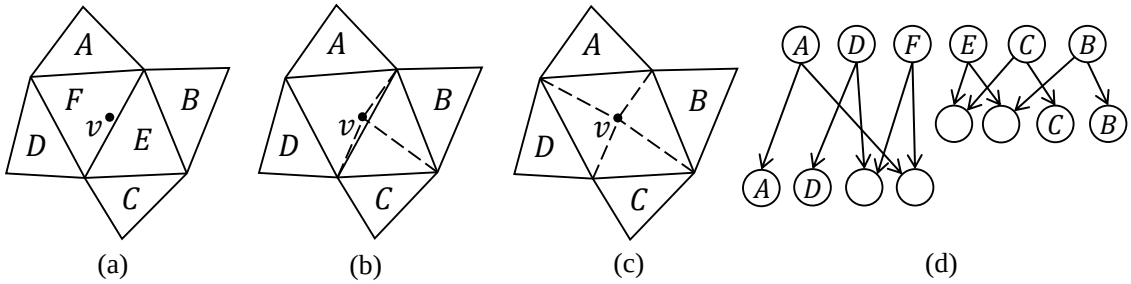


Figure 6.3: An example of the tracing structure. Here a point v is added and the encroaching region contains triangles E and F (subfigure (a)). Four new triangles will be generated and replace the two previous triangles. They may or may not be created in the same round, and in this example this is done in two substeps (subfigures (b) and (c)). Part of the tracing structure is shown in subfigure (d). Four neighbor triangles A, B, C, and D are copied, and four new triangles are created. An arrow indicates that a point is encroached by the head triangle only if it is encroached by the tail triangle.

point can only be added to a triangle t' (i.e., encroaches on the triangle) if it encroached one of the two input edges from t and t_o. We can therefore use the DAG tracing algorithm to find all of the triangles encroached on by a given point v starting at the initial root triangle t_b.

We first construct the DT of the first $n/\log_2 n$ points in the initial round using Algorithm 9 while building the tracing structure. Then at the beginning of each incremental round, each point traces down the structure to find its encroached triangles, and leaves itself in the encroached set of that triangle. Note that the encroached set for a given point might be large, but the average size across points is constant in expectation.

We now analyze the cost of finding all the encroached triangles when adding a set of new points. As discussed, the depth of G is upper bounded by $O(\log n)$ *whp*. The number of encroached triangles of a point x can be analyzed by considering the degree of the point (number of incident triangles) if added to the DT. By Euler's formula, the average degree of a node in a planar graph is at most 6. Since we add the points in a random order, the expected value of $|S(G, x)|$ in Theorem 6.3.1 is constant. Finally, the number of all encroached (including non-leaf) triangles of this point is upper bounded by the number of inCircle tests. Then $|R(G, x)|$, the expected number of visible vertices of x, is $O(\log n)$ (Theorem 4.2 in [68]).

After finding the encroached triangles for each point being added, we need to collect them together to add them to the triangle. This step can be done in parallel with a semisort, which takes linear expected work (writes) and $O(\log^2 m)$ depth *whp* [154], where m is the number of inserted points in this round. Combining these results leads to the following lemma.

Lemma 6.5.5. *Given $2m$ points in the plane and a tracing structure T generated by Algorithm 9 on a randomly selected subset of m points, computing for each triangle in T the points that encroach it among the remaining m points takes $O(m \log m + \omega m)$ work ($O(m)$ writes) and $O(\log^2 n)$ depth* whp *in the Asymmetric NP model.*

The idea of the algorithm is to keep doubling the size of the set that we add (i.e., prefix doubling). Each round applies Algorithm 9 to insert the points and build a tracing structure, and then the DAG tracing algorithm to locate the points for the next round. The depth of each round is upper bounded by the overall depth of the DAG on all points, which is $O(\log n)$ *whp*, where n is the original size. We obtain the following theorem.

Theorem 6.5.6. *Planar Delaunay triangulation can be computed using $O(n \log n + \omega n)$ work (i.e., $O(n)$ writes) in expectation and $O(\log^2 n \log\log n)$ depth* whp *on the Asymmetric NP model with priority-writes.*

Proof. The original Algorithm 9 in [68] has $O(\log^2 n)$ depth *whp*. In the prefix-doubling approach, the depth of each round is no more than $O(\log^2 n)$, and the algorithm has $O(\log\log n)$ rounds. The overall depth is hence $O(\log^2 n \log\log n)$ depth *whp*.

The work bound consists of the costs from the initial round, and the incremental rounds. The initial round computes the triangulation of the first $n/\log_2 n$ points, using at most $O(n)$ inCircle tests, $O(n)$ writes and $O(\omega n)$ work. For the incremental rounds, we have two components, one for locating encroached triangles in the tracing structure, and one for applying Algorithm 9 on those points to build the next tracing structure. The first part is handled by Lemma 6.5.5. For the second part we we can apply a similar analysis to Theorem 4.2 of [68]. In particular, the probability that there is a dependence from a triangle in the i'th point (in the random order) to a triangle added by a later point at location j in the ordering is upper bounded by $24/i$. Summing across all points in the second half (we have already resolved the first half) gives:

$$\mathbb{E}[C] \leq \sum_{i=m+1}^{2m} \sum_{j=i+1}^{2m} 24/i = O(m) \ .$$

This is a bound on both the number of reads and the number of writes. Since the points added in each round doubles, the cost is dominated by the last round, which is $O(n \log n)$ reads and $O(n)$ writes, both in expectation. Combined with the cost of the initial round gives the stated bounds. □

6.6 Space-Partitioning Data Structures

Space partitioning divides a space into non-overlapping regions.[4] This process is usually applied repeatedly until the number of objects in a region is small enough, so that

[4]The other type of partitioning is object partitioning that subdivides the set of objects directly (e.g., R-tree [158, 207], bounding volume hierarchies [152, 274]).

we can afford to answer a query in linear work within the region. We refer to the tree structure used to represent the partitioning as the space-partitioning tree. Commonly-used space-partitioning trees include binary space partitioning trees, quad/oct-trees, k-d trees, and their variants, and are widely used in computational geometry [109, 162], computer graphics [15], integrated circuit design, learning theory, etc.

In this section, we propose write-efficient construction and update algorithms for k-d trees [44]. We discuss how to support dynamic updates write-efficiently in Section 6.6.2, and we discuss how to apply our technique to other space-partitioning trees in Section 6.6.3.

6.6.1 k-d Tree Construction and Queries

k-d trees have many variants that facilitate different queries. We start with the most standard applications on range queries and nearest neighbor queries, and discussions for other queries are in Section 6.6.3. A range query can be answered in $O(n^{(k-1)/k})$ worst-case work, and an approximate $(1+\epsilon)$-nearest neighbor (ANN) query requires $\log n \cdot O(1/\epsilon)^k$ work assuming bounded aspect ratio,[5] both in k-dimensional space. The tree to achieve these bounds can be constructed by always partitioning by the median of all of the objects in the current region either on the longest dimension of the region or cycling among the k dimensions. The tree has linear size and $\log_2 n$ depth [109], and can be constructed using $O(n \log n)$ reads and writes. We now discuss how to reduce the number of writes to $O(n)$.

One solution is to apply the incremental construction by inserting the objects into a k-d tree one by one. This approach requires linear writes, $O(n \log n)$ reads and polylogarithmic depth. However, the splitting hyperplane is no longer based on the median, but the object with the highest priority pre-determined by a random permutation. The expected tree depth can be $c \log_2 n$ for $c > 1$, but to preserve the range query cost we need the tree depth to be $\log_2 n + O(1)$ (see details in Lemma 6.6.1). Motivated by the incremental construction, we propose the following variant, called p-batched incremental construction, which guarantees both write-efficiency and low tree depth.

The p-batched incremental construction.

The p-batched incremental construction is a variant of the classic incremental construction where the dependence graph is a tree. Unlike the classic version, where the splitting hyperplane (splitter) of a tree node is immediately set when inserting the object with the highest priority, in the p-batched version, each leaf node will buffer at most p objects before it determines the splitter. We say that a leaf node *overflows* if it holds more than p objects in its buffer. We say that a node is *generated* when created by its parent, and *settled* after finding the splitters, creating leaves and pushing the objects to the leaves' buffers.

[5]The largest aspect ratio of a tree node on any two dimensions is bounded by a constant, which is satisfied by the input instances in most real-world applications.

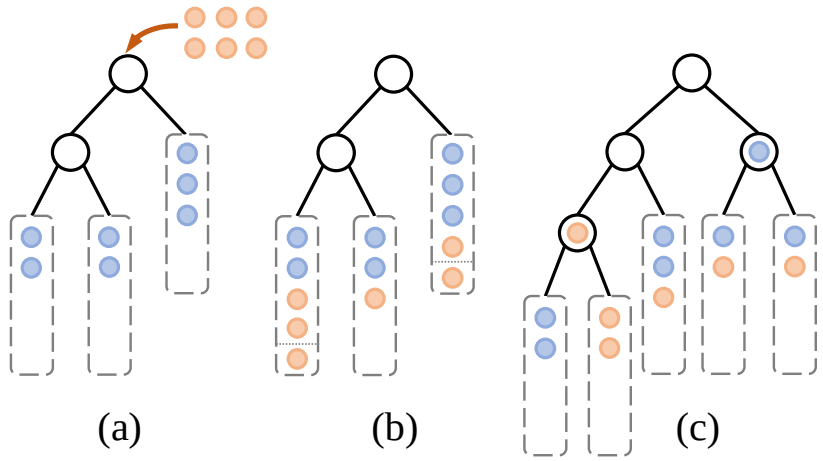


Figure 6.4: An illustration of one round in the p-batched incremental construction for $p = 4$. Subfigure (a) shows the initial state of this round. Then the new objects (shown in orange) are added to the buffers in the leaves, as shown in subfigure (b). Two of the buffers overflow, and so we settle these two leaves as shown in subfigure (c).

The algorithm proceeds in rounds, where in each round it first finds the corresponding leaf nodes that the inserted objects belong to, and adds them into the buffers of the leaves. Then it settles all of the overflowed leaves, and starts a new round. An illustration of this algorithm is shown in Figure 6.4. After all objects are inserted, the algorithm finishes building the subtree of the tree nodes with non-empty buffers recursively. For write-efficiency, we require the small-memory size to be $\Omega(p)$, and the reason will be shown in the cost analysis.

We make a partition once we have gathered p objects in the corresponding subregion based on the median of these p objects. When $p = 1$, the algorithm is the incremental algorithm mentioned above, but the range query cost cannot be preserved. When $p = n$, the algorithm constructs the same tree as the classic k-d tree construction algorithm, but requires more than linear writes unless the small-memory size is $O(n)$, which is impractical when n is large. We now try to find the smallest value of p that preserves the query cost, and we analyze the cost bounds accordingly.

6.6.1.1 Range Query

We use the following lemma to analyze the cost of a standard k-d range query (on an axis-aligned hypercube for $k \geq 2$).

Lemma 6.6.1. *A k-d range query costs $O(2^{((k-1)/k)h})$ using our k-d tree of height h.*

Proof Sketch. A k-d range query has at most $2k$ faces that generate $2k$ half-spaces, and we analyze the query cost of each half-space. Since each axis is partitioned once in every k consecutive levels, one side of the partition hyperplane perpendicular to the query face

will be either entirely in or out of the associated half-space. We do not need to traverse that subtree (we can either directly report the answer or ignore it). Therefore every k levels will expand the search tree by a factor of at most 2^{k-1}. Thus the query cost is $O(2^{((k-1)/k)h})$. □

Lemma 6.6.2. *For our p-batched k-d tree, $p = \Omega(\log^3 n)$ guarantees the tree height to be no more than $\log_2 n + O(1)$ whp.*

Proof. We now consider the p-batched incremental construction. Since we are partitioning based on the median of p random objects, the hyperplane can be different from the actual median. To get the same cost bound, we want the actual number of objects on the two sides to differ by no more than a factor of ϵ *whp*. Since we pick p random samples, by a Chernoff bound the probability that more than $1/2p$ samples are within the first $(1/2 - \epsilon/4)n$ objects is upper bounded by $e^{-p\epsilon^2/24}$. Hence, the probability that the two subtree weights of a tree node differ by more than a factor of ϵ is no more than $2e^{-p\epsilon^2/24}$. This ϵ controls the tree depth, and based on the previous analysis we want to have $n(\frac{1}{2} + \frac{\epsilon}{4})^{\log_2 n/p + O(1)} < p$. Namely, we want the tree to have no more than $\log_2 n/p + O(1)$ levels *whp* to reach the subtrees with less than p elements, so the overall tree depth is bounded by $\log_2 n/p + O(1) + \log_2 p = \log_2 n + O(1)$. Combining these constraints leads to $\epsilon = O(1)/\log_2 n$ and $p = \Omega(\log^3 n)$. □

Lemma 6.6.2 indicates that setting $p = \Omega(\log^3 n)$ gives a tree height of $\log_2 n + O(1)$ *whp*, and Lemma 6.6.1 shows that the corresponding range query cost is $O(2^{((k-1)/k)(O(1)+\log_2 n)}) = O(n^{(k-1)/k})$, matching the standard range query cost.

6.6.1.2 ANN Query

If we assume that the input objects are well-distributed and the k-d tree satisfies the bounded aspect ratio, then the cost of a $(1 + \epsilon)$-ANN query is proportional to the tree height. As a result, $p = \Omega(\log n)$ leads to a query cost of $\log n \cdot O(1/\epsilon)^k$ *whp*.[6]

6.6.1.3 Parallel Construction and Cost Analysis

To get parallelism, we use the prefix-doubling approach, starting with $n/\log n$ objects in the first round. The number of reads of the algorithm is still $\Theta(n \log n)$, since it is lower bounded by the cost of sorting when $k = 1$, and upper bounded by $O(n \log n)$ since the modified algorithm makes asymptotically no more comparisons than the classic implementation. We first present the following lemma.

Lemma 6.6.3. *When a leaf overflows at the end of a round, the number of objects in its buffer is $O(p)$* whp *when $p = \Omega(\log n)$.*

[6] Actually the tree depth is $O(\log n)$ even when $p = 1$. However, for write-efficiency, we need $p = \Omega(\log n)$ to support efficient updates as discussed in Section 6.6.2 that requires the two subtree sizes to be balanced at every node.

Proof Sketch. In the previous round, assume n' objects were in the tree. At that time no more than $p-1$ objects are buffered in this leaf node. Then in the current round another n' objects are inserted, and by a Chernoff bound, the probability that the number of objects falling into this leaf node is more than $(c+1)p$ is at most $e^{-c^2 p/2}$. Plugging in $p = \Omega(\log n)$ proves the lemma. □

We now bound the parallel depth of this construction. The initial round runs the standard construction algorithm on the first $n/\log_2 n$ objects, which requires $O((\log^2 p + \log n)\log n) = O(\log^2 n)$ depth. Then in each of the next $O(\log\log n)$ incremental rounds, we need to locate leaf nodes and a parallel semisort to put the objects into their buffers. Both steps can be done in $O(\log^2 n)$ depth *whp* [154]. Then we also need to account for the depth of settling the leaves after the incremental rounds. When a leaf overflows, by Lemma 6.6.3 we need to split a set of $O(p)$ objects for each leaf, which has a depth of $O(\log^2 p) = O(\log\log n)$ using the classic approach, and is applied for no more than a constant number of times *whp* by Lemma 6.6.3.

We now analyze the number of writes this algorithm requires. The initial round requires $O(n)$ writes as it uses a standard construction algorithm on $n/\log_2 n$ objects. In the incremental rounds, $O(1)$ writes *whp* are required for each object to find the leaf node it belongs to and add itself to the buffer using semisorting [154]. From Lemma 6.6.3, when finding the splitting hyperplane and splitting the object for a tree node, the number of writes required is $O(p)$ *whp*. Note that after a new leaf node is generated from a split, it contains at least $p/2$ objects. Therefore, after all incremental rounds, the tree contains at most $O(n/p)$ tree nodes, and the overall writes to generate them is $O((n/p)\cdot p) = O(n)$. After the incremental rounds finish, we need $O(n)$ writes to settle the leaves with non-empty buffers, assuming $O(p)$ cache size. In total, the algorithm uses $O(n)$ writes *whp*.

Theorem 6.6.4. *A k-d tree that supports range and ANN queries efficiently can be computed using $O(n\log n + \omega n)$ expected work (i.e., $O(n)$ writes) and $O(\log^2 n)$ depth* whp *in the Asymmetric NP model. For range query the small-memory size required is $\Omega(\log^3 n)$.*

6.6.2 k-d Tree Dynamic Updates

Unlike many other tree structures, we cannot rotate the tree nodes in k-d trees since each tree node represents a subspace instead of just a set of objects. Deletion is simple for k-d trees, since we can afford to reconstruct the whole structure from scratch when a constant fraction of the objects in the k-d tree have been removed, and before the reconstruction we just mark the deleted node (constant reads and writes per deletion via an unordered map). In total, the amortized cost of each deletion is $O(\omega + \log n)$. For insertions, we discuss two techniques that optimize either the update cost or the query cost.

6.6.2.1 Logarithmic Reconstruction [224]

We maintain at most $\log_2 n$ k-d trees of sizes that are increasing powers of 2. When an object is inserted, we create a k-d tree of size 1 containing the object. While there are trees of equal size, we flatten them and replace the two trees with a tree of twice the size. This process keeps repeating until there are no trees with the same size. When querying, we search in all (at most $\log_2 n$) trees. Using this approach, the number of reads and writes on an insertion is $O(\log^2 n)$, and on a deletion is $O(\log n)$. The costs for range queries and ANN queries are $O(n^{(k-1)/k})$ and $\log^2 n \cdot O(1/\epsilon)^k$ respectively, plus the cost for writing the output.

If we apply our write-efficient p-batched version when reconstructing the k-d trees, we can reduce the writes (but not reads) by a factor of $O(\log n)$ (i.e., $O(\log n)$ and $O(1)$ writes per update).

When using logarithmic reconstruction, querying up to $O(\log n)$ trees can be inefficient in some cases, so here we show an alternative solution that only maintains a single tree.

6.6.2.2 Single-Tree Version

As discussed in Section 6.6.1, only the tree height affects the costs for range queries and ANN queries. For range queries, Lemma 6.6.2 indicates that the tree height should be $\log_2 n + O(1)$ to guarantee the optimal query cost. To maintain this, we can tolerate an imbalance between the weights of two subtrees by a factor of $O(1/\log n)$, and reconstruct the subtree when the imbalance is beyond the constraint. In the worst case, a subtree of size n' is rebuilt once after $O(n'/\log n)$ insertions into the subtree. Since the reconstructing a subtree of size n' requires $O(n' \log n' + \omega n')$ work, each inserted object contributes $O(\log n \log n' + \omega \log n)$ work to every node on its tree path, and there are $O(\log n)$ such nodes. Hence, the amortized work for an insertion is $O(\log^3 n + \omega \log^2 n)$. For efficient ANN queries, we only need the tree height to be $O(\log n)$, which can be guaranteed if the imbalance between two subtree sizes is at most a constant multiplicative factor. Using a similar analysis, in this case the amortized work for an insertion is $O(\log^2 n + \omega \log n)$.

6.6.3 Extension to Other Trees and Queries

In Section 6.6.1 we discussed the write-efficient algorithm to construct a k-d tree that supports range and ANN queries. k-d trees are also used in many other queries in real-world applications, such as ray tracing, collision detection for non-deformable objects, n-body simulation, and geometric culling (using BSP trees). The partition criteria in these applications are based on some empirical heuristics (e.g., the surface-area heuristic [142]), which generally work well on real-world instances, but usually with no theoretical guarantees.

The p-batched incremental construction can be applied to these heuristics, as long as each object contributes linearly to the heuristic. Let us consider the surface-area

heuristic [142] as an example, which does an axis-aligned split and minimizes the sum of the two products of the subtree's surface area and the number of objects. Instead of sorting the coordinates of all objects in this subtree and finding the optimal split point, we can do approximately by splitting when at least p of the objects are inserted into a region. When picking a reasonable value of p (like $O(\log^2 n)$ or $O(\log^3 n)$), we believe the tree quality should be similar to the exact approach (which is a heuristic after all). However, the p-batched approach does not apply to heuristics that are not linear in the size of the object set. Such cases happen in the Callahan-Kosaraju algorithm [81] when the region of each k-d tree node shrinks to the minimum bounding box, or the object-partitioning data structures (like R-trees or bounding volume hierarchies) where each object can contribute arbitrarily to the heuristic.

6.7 Augmented Trees

An ***augmented tree*** is a tree that keeps extra data on each tree node other than what is used to maintain the balance of this tree. We refer to the extra data on each tree node as the ***augmentation***. In this section, we introduce a framework that gives new algorithms for constructing both static and dynamic augmented trees including interval trees, 2D range trees, and priority search trees that are parallel and write-efficient. Using these data structures we can answer 1D stabbing queries, 2D range queries, and 3-sided queries (defined in Section 6.7.1). For all three problems, we assume that the query results need to be written to the large-memory. Our results are summarized in Table 6.2. We improve upon the traditional algorithms in two ways. First, we show how to construct interval trees and priority search trees using $O(n)$ instead of $O(n \log n)$ writes (since the 2D range tree requires $O(n \log n)$ storage we cannot asymptotically reduce the number of writes). Second, we provide a tradeoff between update costs and query costs in the dynamic versions of the data structures. The cost bounds are parameterized by α. By setting $\alpha = O(1)$ we achieve the same cost bounds as the traditional algorithms for queries and updates. α can be chosen optimally if we know the update-to-query ratio r. For interval and priority trees, the optimal value of α is $\min(2 + \omega/r, \omega)$. The overall work without considering writing the output can be improved by a factor of $\Theta(\log \alpha)$. For 2D range trees, the optimal value of α is $2 + \min(\omega/r, \omega)/\log_2 n$.

We discuss two techniques in this section that we use to achieve write-efficiency. The first technique is to decouple the tree construction from sorting, and we introduce efficient algorithms to construct interval and priority search trees in linear reads and writes after the input is sorted. Sorting can be done in parallel and write-efficiently (linear writes).Using this approach, the tree structure that we obtain is perfectly balanced.

The second technique that we introduce is the α-labeling technique. We mark a subset of tree nodes as *critical* nodes by a predicate function parameterized by α, and only maintain augmentations on these critical nodes. We can then guarantee that every update only modifies $O(\log_\alpha n)$ nodes, instead of $O(\log n)$ nodes as in the classic algorithms. At a

	Construction	Query	Update
Classic interval tree	$O(\omega n \log n)$	$O(\omega k + \log n)$	$O(\omega \log n)$
WE interval tree	$O(\omega n + n \log n)$	$O(\omega k + \alpha \log_\alpha n)$	$O((\omega + \alpha) \log_\alpha n)$
Classic priority search tree	$O(\omega n \log n)$	$O(\omega k + \log n)$	$O(\omega \log n)$
WE priority search tree	$O(\omega n + n \log n)$	$O(\omega k + \alpha \log_\alpha n)$	$O((\omega + \alpha) \log_\alpha n)$
Classic range Tree	$O(\omega n \log n)$	$O(\omega k + \log^2 n)$	$O((\log n + \omega) \log n)$
WE range tree	$O((\alpha + \omega)\, n \log_\alpha n)$	$O(\omega k + \alpha \log_\alpha n \log n)$	$O((\alpha \log n + \omega) \log_\alpha n)$

Table 6.2: A summary of the work cost of the data structures discussed in Section 6.7. In all cases, it is assumed that the tree contains n objects (intervals or points). For interval trees and priority search trees, the number of writes in the construction can be reduced from $O(\log n)$ per element to $O(1)$. For dynamic updates, the number of writes per update can be reduced by a factor of $\Theta(\log \alpha)$ at the cost of increasing the number of reads in update and queries by a factor of α for any $\alpha \geq 2$.

high level, the α-labeling is similar to the weight-balanced B-tree (WBB tree) proposed by Arge et al. [26, 27] for the external-memory (EM) model [7]. However, as we discuss in Section 6.7.3, directly applying the EM algorithms [13, 26, 27, 255, 256] does not give us the desired bounds in our model. Secondly, our underlying tree is still binary. Hence, we mostly need no changes to the algorithmic part that dynamically maintains the augmentation in this trees, but just relax the balancing criteria so the underlying search trees can be less balanced. An extra benefit of our framework is that bulk updates can be supported in a straightforward manner. Such bulk updates seem complicated and less obvious in previous approaches. We propose algorithms on our trees that can support bulk updates write-efficiently and in polylogarithmic depth.

The rest of this section is organized as follows. We first provide the problem definitions and review previous results in Section 6.7.1. Then in Section 6.7.2, we introduce our post-sorted construction technique for constructing interval and priority search trees using a linear number of writes. Finally, we introduce the α-labeling technique to support a tradeoff in query and update cost for interval trees, priority search trees, and range trees in Section 6.7.3.

6.7.1 Preliminaries and Previous Work

We define the *weight* or *size* of tree node or a subtree as the number of nodes in this subtree plus one. The "plus one" guarantees that the size of a tree node is always the sum of the sizes of its two children, which simplifies our discussion. This is also the standard balancing criteria used for weight-balanced trees [222].

6.7.1.1 Interval Trees and the 1D Stabbing Queries

An *interval tree*[7] [109, 120, 208] organizes a set of n intervals $S = \{s_i = (l_i, r_i)\}$ defined by their left and right endpoints. The key on the root of the interval tree is the median of the $2n$ endpoints. This median divides all intervals into two categories: those completely on its left/right, which then form the left/right subtrees recursively, and those covering the median, which are stored in the root. The intervals in the root are stored in two lists sorted by the left and right endpoints respectively. In this thesis, we use red-black trees to maintain such ordered lists to support dynamic updates and refer to them as the *inner trees*. In the worst case, the previous construction algorithms scan and copy $O(n)$ intervals in $O(\log n)$ levels, leading to $O(n \log n)$ reads and writes.

The interval tree can be used to answer a 1D stabbing query: given a set of intervals, report a list of intervals covering the specific query point p_q. This can be done by searching p_q in the tree. Whenever p_q is smaller (larger) than the key of the current node, all intervals in the current tree node with left (right) endpoints smaller than p_q should be reported. This can be done efficiently by scanning the list sorted by left (right) endpoints. The overall query cost is $O(\omega k + \log n)$ (where k is the output size).

6.7.1.2 2D Range Trees and the 2D Range Queries

The *2D range tree* [45] organizes a set of n points $p = \{p_i = (x_i, y_i)\}$ on the 2D plane. It is a tree structure augmented with an inner tree, or equivalently, a two-level tree structure. The outer tree stores every point sorted by their x-coordinate. Each node in the outer tree is augmented with an inner tree structure, which contains all the points in its subtree, sorted by their y-coordinate.

The 2D range tree can be used to answer the 2D range query: given n points in the 2D plane, report the list of points with x-coordinate between x_L and x_R, and y-coordinate between y_B and y_T. Such range queries using range trees can be done by two nested searches on (x_L, x_R) in the outer tree and (y_B, y_T) in at most $O(\log n)$ associated inner trees. Using balanced BSTs for both the inner and outer trees, a range tree can be constructed with $O(n \log n)$ reads and writes, and each query takes $O(\log^2 n + k)$ reads and $O(k)$ writes (where k is the output size). A range tree requires $O(n \log n)$ storage so the number of writes for construction is already optimal.

6.7.1.3 Priority Search Trees and 3-sided Range Queries

The *priority search tree* [109, 209] (priority tree for short) contains a set of n points $p = \{p_i = (x_i, y_i)\}$ each with a *coordinate* (x_i) and a *priority* (y_i). There are two variants of priority trees, one is a search tree on coordinates that keeps a heap order of the priorities as the augmented values [27, 209]. The other one is a heap of the priorities, where each node is augmented with a splitter between the left and right subtrees on the coordinate dimension [109, 209]. The construction of both variants uses $O(n \log n)$ reads and writes

[7]There exist multiple versions of interval trees. In this thesis, we use the version described in [109].

as shown in the original papers [109, 209]. For example, consider the second variant. The root of a priority tree stores the point with the highest priority in p. All the other points are then evenly split into two sets by the median of their coordinates which recursively form the left and right subtrees. The construction scans and copies $O(n)$ points in $O(\log n)$ levels, leading to $O(n \log n)$ reads and writes for the construction.

Many previous results on dynamic priority search trees use the first variant because it allows for rotation-based updates. In this thesis, we discuss how to construct the second variant allowing reconstruction-based updates, since it is a natural fit for our framework. We also show that bulk updates can be done write-efficiently in this variant. For the rest of this section, we discuss the second variant of the priority tree.

The priority tree can be used to answer the 3-sided queries: given a set of n points, report all points with coordinates in the range $[x_L, x_R]$, and priority higher than y_B. This can be done by traversing the tree, skipping the subtrees whose coordinate range do not overlap $[x_L, x_R]$, or where the priority in the root is lower than y_B. The cost of each query is $O(\omega k + \log n)$ for an output of size k [109].

6.7.2 The Post-Sorted Construction

For interval trees and priority search trees, the standard construction algorithms [108, 109, 120, 208, 209, 260] require $O(n \log n)$ reads and writes, even though the output is only of linear size. This section describes algorithms for constructing them in an optimal linear number of writes. Both algorithms first sort the input elements by their x-coordinate in $O(\omega n + n \log n)$ work and $O(\log^2 n)$ depth using the write-efficient comparison sort described in Section 6.4. We now describe how to build the trees in $O(n)$ reads and writes given the sorted input. For a range tree, since the standard tree has $O(n \log n)$ size, the classic construction algorithm is already optimal.

6.7.2.1 Interval Tree

After we sort all $2n$ coordinates of the endpoints, we can first build a perfectly-balanced binary search tree on the endpoints using $O(n)$ reads and writes and $O(\log n)$ depth. We now consider how to construct the inner tree of each tree node.

We create a lowest common ancestor (LCA) data structure on the keys of the tree nodes that allows for constant time queries. This can be constructed in $O(n)$ reads/writes and $O(\log^2 n)$ depth [47, 183]. Each interval can then find the tree node that it belongs to using an LCA query on its two endpoints. We then use a radix sort on the n intervals. The key of an interval is a pair with the first value being the index of the tree node that the interval belongs to, and the second value being the index of the left endpoint in the pre-sorted order. The sorted result gives the interval list for each tree node sorted by left endpoints. We do the same for the right endpoints. This step takes $O(n)$ reads/writes overall. Finally, we can construct the inner trees from the sorted intervals in $O(n)$ reads/writes across all tree nodes.

Parallelism is straightforward for all steps except for the radix sort. The number of possible keys can be $O(n^2)$, and it is not known how to radix sort keys from such a range work-efficiently and in polylogarithmic depth. However, we can sort a range of $O(n \log n)$ in $O(\omega n)$ expected work and $O(\log^2 n)$ depth *whp* [236]. Hence our goal is to limit the first value into a $O(\log n)$ range. We note that given the left endpoint of an interval, there are only $\log_2(2n)$ possible locations for the tree node (on the tree path) of this interval. Therefore instead of using the tree node as the first value of the key, we use the level of the tree node, which is in the range $[1, \ldots, O(\log n)]$. By radix sorting these pairs, so we have the sorted intervals (based on left or right endpoint) for each level. We observe that the intervals of each tree node are consecutive in the sorted interval list per level. This is because for tree nodes u_1 and u_2 on the same level where u_1 is to the left of u_2, the endpoints of u_1's intervals must all be to the left of u_2's intervals. Therefore, in parallel we can find the first and the last intervals of each node in the sorted list, and construct the inner tree of each node. Since the intervals are already sorted based on the endpoints, we can build inner trees in $O(n)$ reads and writes and $O(\log^2 n)$ depth [65].

6.7.2.2 Priority Tree

In the original priority tree construction algorithm, points are recursively split into sub-problems based on the median at each node of the tree. This requires $O(n)$ writes at each level of the tree if we explicitly copy the nodes and pack out the root node that is removed. To avoid explicit copying, since the points are already pre-sorted, our write-efficient construction algorithm passes indices determining the range of points belonging to a sub-problem instead of actually passing the points themselves. To avoid packing, we simply mark the position of the removed point in the list as invalid, leaving a hole, and keep track of the number of valid points in each sub-problem.

Our recursive construction algorithm works as follows. For a tree node, we know the range of the points it represents, as well as the number of valid points n_v. We then pick the valid point with the highest priority as the root, mark the point as invalid, find the median among the valid points, and pass the ranges based on the median and number of valid points (either $\lfloor (n_v - 1)/2 \rfloor$ or $\lceil (n_v - 1)/2 \rceil$) to the left and right sub-trees, which are recursively constructed. The base case is when there is only one valid point remaining, or when the number of holes is more than the valid points. Since each node in the tree can only cause one hole, for every range corresponding to a node, there are at most $O(\log n)$ holes. Since the size of the small-memory is $\Omega(\log n)$, when the number of valid points is fewer than the number of holes, we can simply load all of the valid points into the small-memory and construct the sub-tree.

To efficiently implement this algorithm, we need to support three queries on the input list: finding the root, finding the k-th element in a range (e.g., the median), and deleting an element. All queries and updates can be supported using a standard tournament tree where each interior node maintains the minimum element and the number of valid

nodes within the subtree. With a careful analysis, all queries and updates throughout the construction require linear reads/writes overall. The details are provided later in this section.

The parallel depth is $O(\log^2 n)$—the bottleneck lies in removing the points. There are $O(\log n)$ levels in the priority tree and it costs $O(\log n)$ writes for removing elements from the tournament tree on each level. For the base cases, it takes linear writes overall to load the points into the small-memory and linear writes to generate all tree nodes. The depth is $O(\log n)$.

We summarize our result in this section in Theorem 6.7.1.

Theorem 6.7.1. *An interval tree or a priority search tree can be constructed with pre-sorted input in $O(\omega n)$ expected work and $O(\log^2 n)$ depth* whp *on the Asymmetric NP model.*

The tournament tree for constructing priority trees. We now discuss the tournament tree on a list that can support the RangeMin and the k-th element in a range, and at the meantime an element can be removed. Each interior tree node maintains the minimum element and the number of valid nodes within the subtree. We now show that given the tree of size n, answering all queries in construction uses linear reads and writes.

For a RangeMin query on (x, y), we start from the left corresponding to x, keep going up on the tree until the node that its subtree contains y, and traverse the tree to find y. In this process, we use the maintained values in the tree nodes to update the RangeMin when the corresponding ranges of the subtrees are within (x, y). The reads of such a query is $O(\log(y - x + 1))$. We can query the k-th element in a range similarly.

Since the tree is fully balanced, the tree height is $\log_2 n$. In the i-th level (the root is the first level), there are $O(2^i)$ queries and the sum of the query ranges is $O(n)$. The overall query cost on one level is maximized when all the query ranges are the same, which is $O(2^i \cdot \log(n/2^i))$. The overall cost across all levels is $\sum_{i=1}^{\log_2 n} O(2^i \cdot \log(n/2^i)) = O(n)$.

Deleting an element naïvely in a tournament tree costs $O(\log n)$ writes in the worst case. Our observation is that, once we delete an element of a tree node corresponding to a range (x, y), we know that all further queries are either entirely within (x, y) or disjoint (x, y). We therefore only update the ancestors of the deleted nodes whose range is within (x, y), and there are at most $O(\log(y-x+1))$ of such ancestors. The overall writes required in all deletions has the same form as the overall reads in the queries, which is $O(n)$.

6.7.3 Dynamic Updates using Reconstruction-Based Rebalancing

Dynamic updates (insertions and deletions) are often supported on augmented trees [108, 109, 120, 208, 209] and the goal of this section is to support updates write-efficiently, at the cost of performing extra reads to reduce the overall work. Traditionally, an insertion or deletion costs $O(\log n)$ for interval trees and priority search trees, and $O(\log^2 n)$ for range trees. In the asymmetric setting, the work is multiplied by ω. To reduce the overall work,

we introduce an approach to select a subset of tree nodes as *critical* nodes, and only update the balance information of those nodes (the augmentations are mostly unaffected). The selection of these critical nodes are done by the α-labeling introduced in Section 6.7.3.1. Roughly speaking, for each tree path from the root to a leaf node, we have $O(\log_\alpha n)$ critical nodes marked such that the subtree weights of two consecutive marked nodes differ by about a factor of $\alpha \geq 2$. By doing so, we only need to update the balancing information in the critical nodes, leading to fewer tree nodes modified in an update.

Arge et al. [26, 27] use a similar strategy to support dynamic updates on augmented trees in the external-memory (EM) model, in which a block of data can be transferred in unit cost [7]. They use a B-tree instead of a binary tree, which leads to a shallower tree structure and fewer memory accesses in the EM model. However, in the Asymmetric NP model, modifying a block of data requires work proportional to the block size, and directly using their approach cannot reduce the overall work. Inspired by their approach, we propose a simple approach to reduce the work of updates for the Asymmetric NP model.

The main component of our approach is *reconstruction-based rebalancing* using the *α-labeling* technique. We can always obtain the sorted order via the tree structure, so when imbalance occurs, we can afford to reconstruct the whole subtree in reads and writes proportional to the subtree size and polylogarithmic depth. This gives a unified approach for different augmented trees: interval trees, priority search trees, and range trees.

We introduce the α-labeling idea in Section 6.7.3.1, the rebalancing algorithm in Section 6.7.3.2, and its work analysis in Section 6.7.3.3. We then discuss the maintenance of augmented values for different applications in Section 6.7.3.4. We mention how to parallelize bulk updates in Section 6.7.3.5.

6.7.3.1 α-Labeling

The goal of the α-labeling is to maintain the balancing information at only a subset of tree nodes, the critical nodes, such that the number of writes per update is reduced. Once the augmented tree is constructed, we label the node as a critical node if for some integer $i \geq 0$, (1) its subtree weight is between $2\alpha^i$ and $4\alpha^i - 2$ (inclusive); or (2) its subtree weight is $2\alpha^i - 1$ and its sibling's subtree weight is $2\alpha^i$. All other nodes are *secondary* nodes. As a special case, we always treat the root as a virtual critical node, but it does not necessary satisfy the invariants of critical nodes. Note that all leaf nodes are critical nodes in α-labeling since they always have subtrees of weight 2. When we label a critical node, we refer to its current subtree weight (which may change after insertions/deletions) as its *initial weight*. Note that after the augmented tree is constructed, we can find and mark the critical nodes in $O(n)$ reads/writes and $O(\log n)$ depth. After that, we only maintain the subtree weights for these critical nodes, and use their weights to balance the tree.

Fact 6.7.2. *For a critical node A, $2\alpha^i - 1 \leq |A| \leq 4\alpha^i - 2$ holds for some integer i.*

This fact directly follows the definition of the critical node.

For two critical nodes A and B, if A is B's ancestor and there is no other critical node on the tree path between them, we refer to B as A's *critical child*, and A as B's *critical parent*. We define a *critical sibling* accordingly.

We show the following lemma on the initial weights.

Lemma 6.7.3. *For any two critical nodes A and B where A is B's critical parent, their initial weights satisfy* $\max\{(\alpha/2)|B|, 2|B|-1\} \leq |A| \leq (2\alpha+1)|B|$.

Proof. Based on Fact 6.7.2, we assume $2\alpha^i - 1 \leq |A| \leq 4\alpha^i - 2$ and $2\alpha^j - 1 \leq |B| \leq 4\alpha^j - 2$ for some integers i and j. We first show that $i = j+1$. It is easy to check that j cannot be larger than or equal to i. Assume by contradiction that $j < i-1$. With this assumption, we will show that there exists an ancestor of B, which we refer to it as y, which is a critical node. The existence of y contradicts the fact that A is B's critical parent. We will use the property that for any tree node x the weight of its parent $p(x)$ is $2|x|-1 \leq |p(x)| \leq 2|x|+1$.

Assume that B does not have such an ancestor y. Let z be the ancestor of B with weight closest to but no more than $2\alpha^{i-1}$. We consider two cases: (a) $|z| \leq 2\alpha^{i-1} - 2$ and (b) $|z| = 2\alpha^{i-1} - 1$. In case (a) z's parent $p(z)$ has weight at most $2|z|+1 = 4\alpha^{i-1} - 3$. $|p(z)|$ cannot be less than $2\alpha^{i-1}$ by definition of z, and so $y = p(z)$, leading to a contradiction. In case (b), z's sibling does not have weight $2\alpha^{i-1}$, otherwise $y = z$. However, then $|p(z)| \leq 2|z| = 4\alpha^{i-1} - 2$, and either z is not the ancestor with weight closest to $2\alpha^{i-1}$ or $y = p(z)$.

Given $i = j+1$, we have $(\alpha/2)|B| \leq |A| \leq (2\alpha+1)|B|$ (by plugging in $2\alpha^i - 1 \leq |A| \leq 4\alpha^i - 2$ and $2\alpha^{i-1} - 1 \leq |B| \leq 4\alpha^{i-1} - 2$). Furthermore, since A is B's ancestor, we have $2|B| - 1 \leq |A|$. Combining the results proves the lemma. □

6.7.3.2 Rebalancing Algorithm based on α-Labeling

We now consider insertions and deletions on an augmented tree. Maintaining the augmented values on the tree are independent of our α-labeling technique, and differs slightly for each of the three tree structures. We will further discuss how to maintain augmented values in Section 6.7.3.4.

We note that deletions can be handled by marking the deleted objects without actually applying the deletion, and reconstructing the whole subtree once a constant fraction of the objects is deleted. Therefore in this section, we first focus on the insertions only. We analyze single insertions here, and discuss bulk insertions later in Section 6.7.3.5. Once the subtree weight of a critical node A reaches twice the initial weight s, we reconstruct the whole subtree, label the critical nodes within the subtree, and recalculate the initial weights of the new critical nodes. An exception here is that, if $s \leq 4\alpha^i - 2$ and $2\alpha^{i+1} - 1 \leq 2s$ for a certain i, we do not mark the new root since otherwise it violates the bound stated in Lemma 6.7.4 (see more details in Section 6.7.3.3) with A's critical parent. After this reconstruction, A's original critical parent gets one extra critical child, and the two affected

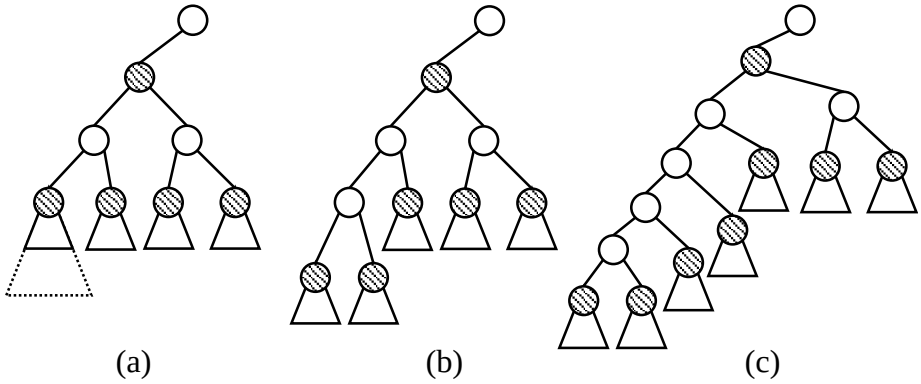


Figure 6.5: An illustration of rebalancing based on α-labeling. The critical nodes are shaded. The case after construction is shown in (a) with solid borders. After some insertions, the size of one of the subtrees grows to twice its initial weight (dashed lines in (a)), so the algorithm reconstructs the subtree, as shown in (b). As we keep inserting new nodes along the left spine, the tree will look like what is shown in (c), but Lemma 6.7.4 guarantees that the subtree of the topmost critical node will be reconstructed before it gets more than $4\alpha + 2$ critical children. The lemma also guarantees that on the path from a critical node to any of its critical children, there can be at most $4\alpha - 1$ secondary nodes.

children now have initial weights the same as A's initial weight. If imbalance occurs at multiple levels, we reconstruct the topmost tree node. An illustration of this process is shown in Figure 6.5.

We can directly apply the algorithms in Section 6.7.2 to reconstruct a subtree as long as we have the sorted order of the (end)points in this subtree. For interval and range trees, we can acquire the sorted order by traversing the subtree. using linear work and $O(\log n)$ depth [254]. For priority trees, since the tree nodes are not stored in-order, we need to insert all interior nodes into the tree in a bottom-up order based on their coordinates (without applying rebalancing) to get the total order on coordinates of all points (the details and cost analysis can be found later in this section). After we have the sorted order, a subtree of weight n can be constructed in $O(\omega n)$ work and $O(\log^2 n)$ depth.

As mentioned, we always treat the root as a virtual critical node, but it does not necessary satisfy the invariants of critical nodes. By doing so, once the weight of the whole tree doubles, we reconstruct the entire tree. We need $\Omega(n)$ insertions for one reconstruction on the root (there can be deletions). The cost for reconstruction is $O(\omega n)$ for interval trees and priority trees, and $O(\omega n \log_\alpha n)$ for range trees (shown in Section 6.7.3.4). The amortized cost is of a lower order compared to the update cost shown in Theorem 6.7.8.

Ordering the nodes within the subtree of a priority tree. Since our priority tree is not a search tree by default, we need an extra step to obtain the ordering of the points in a subtree. This can be trivially achieved by inserting the points in the critical nodes into their subtrees in a bottom-up manner (without balancing the tree). Inserting an object into

a tree requires $O(1)$ writes, so the overall writes are linear. By Corollary 6.7.6, the subtree depth is $O(\alpha \log_\alpha m)$ for a subtree of size m. After the all insertions, the tree depth can be increased by at most $O(\log_\alpha m)$. For an critical node A such that $2\alpha^i - 1 \leq |A| \leq 4\alpha^i - 2$ for some integer i, the number of reads required to find the leaf node is proportional to the tree depth, $O(\alpha i)$. By Lemma 6.7.4, the number critical nodes with the same i decreases geometrically with the increasing of i, the overall reads is asymptotically bounded by the level where $i = 1$. The overall number of reads is therefore $O(\alpha m)$. The total cost to get the ordering is $O((\omega + \alpha)m)$ for a subtree of size m.

6.7.3.3 Cost Analysis of the Rebalancing

To show the rebalancing cost, we first prove some properties about our dynamic augmented trees.

Lemma 6.7.4. *In a dynamic augmented tree with α-labeling, we have* $\max\{(\alpha/4)|B|, (3/2)|B| - 1\} \leq |A| \leq (4\alpha + 2)|B|$ *for any two critical nodes A and B where A is B's critical parent.*

Proof. For any critical node A in the tree, the subtree weight of its critical child B can grow up to a factor of 2 of B's initial weight, after which the subtree is reconstructed to two new critical nodes with the same initial weight of B. A's weight can grow up to a factor of 2 of A's initial weight, without affecting B's weight (i.e., all insertions occur in A's other critical children besides B). Combining these observations with the result in Lemma 6.7.3 shows this lemma except for the $(3/2)|B| - 1 \leq |A|$ part. Originally we have $2|B| - 1 \leq |A|$ after the previous reconstruction. $|A|$ grows together when $|B|$ grows, and right before the reconstruction of B we have $(3/2)|B| - 1 \leq |A|$. □

Lemma 6.7.4 shows that each critical node has at most $4\alpha + 2$ critical children, and so that there are at most $4n + 1$ secondary nodes to connect them. This leads to the following corollary.

Corollary 6.7.5. *The length of the path from a critical node to its critical parent is at most* $4\alpha + 1$.

Combining Lemma 6.7.4 and Corollary 6.7.5 gives the following result.

Corollary 6.7.6. *For a leaf node in a tree with α-labeling, the tree path to the root contains* $O(\log_\alpha n)$ *critical nodes and* $O(\alpha \log_\alpha n)$ *nodes.*

Corollary 6.7.6 shows the number of reads during locating a node in an augmented tree, and the number of critical nodes on that path.

With these results, we now analyze the cost of rebalancing for each insertion. For a critical node with initial weight W, we need to insert at least another W new nodes into this subtree before the next reconstruction of this critical node. Theorem 6.7.1 shows that the amortized cost for each insertion in this subtree is therefore $O(\omega)$ on this node. Based on Corollary 6.7.6, the amortized cost for each insertion contains $O(\log_\alpha n)$ writes

and $O(\alpha \log_\alpha n)$ reads. In total, the work per insertion is $O((\alpha + \omega) \log_\alpha n)$, since we need to traverse $O(\alpha \log_\alpha n)$ tree nodes, update $O(\log_\alpha n)$ subtree weights, and amortize $O(\omega \log_\alpha n)$ work for reconstructions.

We note that any interleaving insertions can only reduce the amortized cost for deletions. Therefore, both the algorithm and the bound can be extended to any interleaving sequence of insertions and deletions. Altogether, we have the following result, which may be of independent interest.

Theorem 6.7.7. *Using reconstruction-based rebalancing based on the α-labeling technique, the amortized cost of each update (insertion or deletion) to maintain the balancing information on a tree of size n is $O((\omega + \alpha) \log_\alpha n)$.*

6.7.3.4 Handling Augmented Values

Since the underlying tree structure is still binary, minor changes to the trees are required for different augmentations.

Interval trees. We do not need any changes for the interval tree. Since we never apply rotations, we directly insert/delete the interval in the associated inner tree with a cost of $O(\log n + \omega)$.

Range trees. For the range tree, we only keep the inner trees for the critical nodes. As such, the overall augmentation weight (i.e., overall weights of all inner trees) is $O(n \log_\alpha n)$. For each update, we insert/delete this element in $O(\log_\alpha n)$ inner trees (Corollary 6.7.6), and the overall cost is $O((\log n + \omega) \log_\alpha n)$. Then each query may look into no more than $O(\alpha \log_\alpha n)$ inner trees each requiring $O(\log n)$ work for a 1D range query. The overall cost for a query is therefore $O(\omega k + \alpha \log_\alpha n \log n)$.

Priority trees. For insertions on priority trees, we search its coordinate in the tree and put it where the current tree node is of lower priority than the new point. The old subtree root is then recursively inserted to one of its subtrees. The cost can be as expensive as $O(\omega\alpha \log_\alpha n)$ when a point with higher priority than all tree nodes is inserted. To address this, points are only stored in the critical nodes, and the secondary nodes only partition the range, without holding points as augmented values. This can be done by slightly modifying the construction algorithm in Section 6.7.2. During the construction, once the current node is a secondary node, we only partition the range, but do not find the node with the highest priority. Since all leaf nodes are critical, the tree size is affected by at most a factor of 2. With this approach, each insertion modifies at most $O(\log_\alpha n)$ nodes, and so the extra work per insertion for maintaining augmented data is $O((\alpha + \omega) \log_\alpha n)$. A deletion on priority trees can be implemented symmetrically, and can lead to cascading promotions of the points. Once the promotions occur, we leave a dummy node in the

original place of the last promoted point, so that all of the subtree sizes remain unchanged (and the tree is reconstructed once half one the nodes are dummy). The cost of a deletion is also $O((\alpha + \omega)\log_\alpha n)$.

Combining the results above gives the following theorem.

Theorem 6.7.8. *Given any integer $\alpha \geq 2$, an update on an interval or priority search tree requires $O((\omega + \alpha)\log_\alpha n)$ amortized work and a query costs $O(\omega k + \alpha \log_\alpha n)$; for a 2D range tree, the query and amortized update cost is $O((\alpha \log n + \omega)\log_\alpha n)$ and $O(\omega k + \alpha \log_\alpha n \log n)$.*

6.7.3.5 Bulk Updates

One of the benefits of our reconstruction-based approach is that, bulk updates on these augmented trees can be supported directly. In the case we change the inner trees of interval and range trees as treaps. For a treap of size n and a bulk update of size m, the expected cost of inserting or deleting this bulk is $O(\omega m + m\log(n/m))$ using treaps (Chapter 8), and the depth is $O(\log m \log n)$ *whp* [65, 260, 261]. We note that there are data structures supporting bulk updates in logarithmic expected depth in the PRAM model [16, 55], but not in the binary forking model discussed in the content of this thesis.

Again deletions are trivial. For interval and range trees, we can just mark all the objects in parallel but apply deletions to inner trees, which requires constant writes per deletion. For priority tree, we delete can delete points in a top-down manner, using $O((\alpha + \omega)\log_\alpha n)$ work per point and $O(\alpha \log_\alpha n \log n)$ depth. We now sketch an outline on the bulk insertion.

Assume the bulk size is m and less than n since otherwise we can afford to reconstruct the whole augmented tree. We first sort the bulk using $O(m \log m + \omega m)$ work and $O(\log^2 n)$ depth. Then we merge the sorted list into the augmented tree recursively, and check each critical node in the tree in a top-down manner.

At any time and for a critical node, we use binary search to decide the new objects inserted in this subtree. If the overall size of the subtree and the newly added objects overflows $4\alpha^i - 2$, we reconstruct the subtree by first flattening the tree nodes, merging with new nodes, rebuilding the subtree using the algorithm in Section 6.7.2, and marking the critical nodes in this subtree. To guarantee Lemma 6.7.4, we do not mark the critical nodes with subtree size greater than or equal to $2\alpha^{i+1} - 1$. By doing so, the proof of Lemma 6.7.4 still holds. The whole process takes $O(n')$ operations in $O(\log n')$ depth for a subtree with n' nodes. Note that at least $O(n')$ nodes are inserted between two consecutive reconstructions so that the cost can be amortized. Otherwise, we just recursively check all the critical children of this node. Once there are no objects within one subtree, we stop the recursion in this subtree.

Note that the range of a binary search can be limited by the range of the critical parent. The overall cost for all binary searches is $O(\alpha m \log(n/m))$ [65] (no writes), and the depth

is $O(\alpha \log m \log_\alpha n)$. In summary, the amortize work for merging m new objects in to the tree is $O(\alpha m \log(n/m) + \omega m \log_\alpha n)$, and the depth is $O(\alpha \log m \log_\alpha n)$.

We now discuss the bulk updates for the augmented values. Again for interval trees and range trees, we can just merge all inserted objects into the corresponding inner trees. Using treaps as the inner trees, merging m' objects to a search tree with n' takes $O(m' \log(n'/m') + \omega m')$ work and $O(\omega \log m' \log n')$ depth. As a result, for interval and range trees, the work per object in the bulk updates is always no more than the single insertion, and the depth is polylogarithmic for any bulk size.

The bulk update for priority trees is similar to the constructions. Once there exists an inserted point with higher priority than the root node, we replace root node with this inserted node, and insert the original root node into the corresponding subtree. Then we leave a hole in the inserted list and recursively apply this process. This process terminates at the time either the subtree root overflows, we reach a leaf node, or there are more holes than new objects. The maintenance of augmentations of priority tree takes $O((\alpha + \omega)m \log_\alpha n)$ amortized work and $O(\alpha \log_\alpha^2 n)$ depth.

6.8 Linear-Work Algorithms

In this section, we study several problems from low-dimensional computational geometry that have linear-work randomized incremental algorithms. These algorithms fall into the Type 2 category of algorithms defined in Section 6.2, and their iteration depth is polylogarithmic *whp*. To obtain linear-work parallel algorithms, we process the steps in prefixes, as described in Section 6.2. For simplicity, we describe the algorithms for these problems in two dimensions, and and briefly note how they can be extended to any fixed number of dimensions.

6.8.1 Linear Programming

Constant-dimensional linear programming (LP) has received significant attention in the computational geometry literature, and several parallel algorithms for the problem have been developed [11, 17, 89, 111, 119, 145, 147, 251]. We consider linear programming in two dimensions. We assume that the constraints are given in general position and the solution is either infeasible or bounded. We note that these assumptions can be removed without affecting the asymptotic cost of the algorithm [247]. The standard randomized incremental algorithm [247] adds the constraints one-by-one in a random order, while maintaining the optimum point at any time. If a newly added constraint causes the optimum to no longer be feasible (a tight constraint), we find a new feasible optimum point on the line corresponding to the newly added constraint by solving a one-dimension linear program, i.e., taking the minimum or maximum of the set of intersection points of other earlier constraints with the line. If no feasible point is found, then the algorithm reports the problem as infeasible.

The iteration dependence graph is defined with the constraints as steps, and fits in the framework of Type 2 algorithms from Section 6.2. The steps corresponding to inserting a tight constraint are the special steps. Special steps depend on all earlier steps because when a tight constraint executes, it needs to look at all earlier constraints. Non-special steps depend on the closest earlier special step i because it must wait for step i to execute before executing itself to retain the sequential execution (we can ignore all of the earlier constraints since i will depend on them). Using backwards analysis, a step j has a probability of at most $2/j$ of being a special step because the optimum is defined by at most two constraints and the constraints are in a randomized order. Furthermore, the probabilities are independent among different steps. This gives us a dependence depth of $O(\log n)$ *whp* as discussed in Section 6.2.

As described in the proof of Theorem 6.2.3, our parallel algorithm executes the steps in prefixes. Each time a prefix is processed, it checks all of the constraints and finds the earliest one that causes the current optimum to be infeasible using line-side tests. The check per step takes $O(1)$ work and processing a violating constraint at step i takes $O(i)$ work and $O(1)$ depth *whp* to solve the one-dimensional linear program which involves minimum/maximum operations. Applying Theorem 6.2.3 with $d(n) = O(1)$ gives the following theorem.

Theorem 6.8.1. *The randomized incremental algorithm for 2D linear programming can be parallelized to run in $O(n)$ work in expectation and $O(\log n)$ depth* whp *on an arbitrary-CRCW PRAM.*

We note that the algorithm can be extended to the case where the dimension d is greater than two by having a randomized incremental d-dimensional LP algorithm recursively call a randomized incremental algorithm for solving $(d-1)$-dimensional LPs. This increases the iteration dependence depth (and hence the depth of the algorithm) to $O(d! \log^{d-1} n)$ *whp.* and increases the expected work to $O(d!n)$. We note that we can generate the random permutation only once and reuse it for the sub-problems. Although we lose independence, the expectation is not affected, and since there are only a constant (a function of d) number of sub-problems with high probability, the high probability bound for the depth is not affected.

6.8.2 Closest Pair

The ***closest pair*** problem takes as input a set of points in the plane and returns the pair of points with the smallest distance between each other. We assume that no pair of points have the same distance. A well-known expected linear-work algorithm [143, 162, 186, 235] works by maintaining a grid and inserting the points into the grid in a random order. The grid partitions the plane into regions of size $r \times r$ where each non-empty region stores the points inside the region and r is the distance of the closest pair so far (initialized to the distance between the first two points). It is maintained using a hash table. Whenever a new

point is inserted, one can check the region the point belongs in and the eight adjacency regions to see whether the new value of r has decreased, and if so, the grid is rebuilt with the new value of r. The check takes $O(1)$ work as each region can contain at most nine points, otherwise the grid would have been rebuilt earlier. Therefore insertion takes $O(1)$ work, and rebuilding the grid takes $O(i)$ work where i is the number of points inserted so far. Using backwards analysis, one can show that point i has probability at most $2/i$ of causing the value of r to decrease, so the expected work is $\sum_{i=1}^{n} O(i) \cdot (2/i) = O(n)$.

This is a Type 2 algorithm, and the iteration dependence graph is similar to that of linear programming. The special steps are the ones that cause the grid to be rebuilt, and the dependence depth is $O(\log n)$ *whp*. Rebuilding the grid involves hashing, and can be done in parallel in $O(i)$ work and $O(\log^* i)$ depth *whp* for a set of i points [141]. We also assume that the points in each region are stored in a hash table, to enable efficient parallel insertion and lookup in linear work and $O(\log^* i)$ depth. To obtain a linear-work parallel algorithm, we again execute the algorithm in prefixes. Applying Theorem 6.2.3 with $d(n) = O(\log^* n)$ gives the following theorem.

Theorem 6.8.2. *The randomized incremental algorithm for closest pair can be parallelized to run in $O(n)$ work in expectation and $O(\log n \log^* n)$ depth* whp *on an arbitrary-CRCW PRAM.*

We note that the algorithm can be extended to d dimensions where the depth is $O(\log d \log n \log^* n)$ *whp* and expected work is $O(c_d n)$ where c_d is some constant that depends on d.

6.8.3 Smallest Enclosing Disk

The ***smallest enclosing disk*** problem takes as input a set of points in two dimensions and returns the smallest disk that contains all of the points. We assume that no four points lie on a circle. Linear-work algorithms for this problem have been described [211, 278], and in this section we will study Welzl's randomized incremental algorithm [278]. The algorithm inserts the points one-by-one in a random order, and maintains the smallest enclosing disk so far (initialized to the smallest disk defined by the first two points). Let v_i be the point inserted on the i'th iteration. If an inserted point v_i lies outside the current disk, then a new smallest enclosing disk is computed. We know that v_i must be on the smallest enclosing disk. We first define the smallest disk containing v_1 and v_i, and scan through v_2 to v_{i-1}, checking if any are outside the disk (call this procedure **Update1**). Whenever v_j ($j < i$) is outside the disk, we update the disk by defining the disk containing v_i and v_j and scanning through v_1 to v_{j-1} to find the third point on the boundary of the disk (call this procedure **Update2**). **Update2** takes $O(j)$ work, and **Update1** takes $O(i)$ work plus the work for calling **Update2**. With the points given in a random order, the probability that the j'th iteration of **Update1** calls **Update2** is at most $2/j$ by a backwards analysis argument, so the expected work of **Update1** is

$O(i) + \sum_{j=1}^{i}(2/j) \cdot O(j) = O(i)$. The probability that **Update1** is called when the i'th point is inserted is at most $3/i$ using a backwards analysis argument, so the expected work of this algorithm is $\sum_{i=1}^{n}(3/i) \cdot O(i) = O(n)$.

This is another Type 2 algorithm whose iteration dependence graph is similar to that of linear programming and closest pair. The points are the steps, and the special steps are the ones that cause **Update1** to be called, which for step i has at most $3/i$ probability of happening. The dependence depth is again $O(\log n)$ *whp* as discussed in Section 6.2.

Our work-efficient parallel algorithm again uses prefixes, both when inserting the points, and on every call to **Update1**. We repeatedly find the earliest point that is outside the current disk by checking all points in the prefix with an in-circle test and taking the minimum among the ones that are outside. **Update1** is work-efficient and makes $O(\log n)$ calls to **Update2** *whp*, where each call takes $O(1)$ depth *whp* as it does in-circle tests and takes a maximum. As in the sequential algorithm, each step takes $O(1)$ work in expectation. Applying Theorem 6.2.3 with $d(n) = O(\log n)$ *whp* (the depth of a executing a step and calling **Update1**) gives the following theorem.

Theorem 6.8.3. *The randomized incremental algorithm for smallest enclosing disk can be parallelized to run in $O(n)$ work in expectation and $O(\log^2 n)$ depth* whp *on an arbitrary-CRCW PRAM.*

The algorithm can be extended to d dimension, with $O(d! \log^d n)$ depth *whp*, and $O(c_d n)$ expected work for some constant c_d that depends on d. Again, we can use the same randomized order for all sub-problems.

6.8.4 Constant-Write Versions

We now introduce a write-efficient version of LP-type algorithms that reduces the number of writes to $O(1)$ while maintaining $O(n)$ expected reads. Here we use the recent work by Har-Peled [163] which only requires constant extra space but is still work-efficient. Here we assume δ is the dimension of the problem.

The key idea in Har-Peled's approach is to simply replace the random permutation by an arbitrary random sequence that has a uniform distribution over the constraints (e.g., a hash function). We can in parallel generate such a sequence by a standard pseudo-random generator by Mulmuley [218]. The output is uniformly random on $[n] = \{1, \ldots, n\}$ and ϕ-wise independent, each element can be computed in $O(\phi)$ time, and the whole algorithm uses $O(c'\phi)$ space for an integer constant $c' \geq 12$. Har-Peled shows that plugging this sequence into the original algorithms gives the same asymptotic work bounds. This is because with probability less than $1/(20\delta)$ each integer in $[n]$ does not appears in the first $\bar{c}n$ elements in this sequence for any constant $\bar{c} \geq 8(5 + \lceil \ln \delta \rceil)^2$. Taking the union bound shows that the probability that any of the constraints in the final configuration does not appear is small: $1/20$. Notice that repeated numbers are not a problem in these algorithms. At the end of the algorithm, we can verify the solution by testing all of the constraints, and

if a violation is found (with probability no more than $1/20$), we just restart the algorithms. Finally, we need different generators for different recursive levels in the algorithm. Other details can be found in [163].

Har-Peled also shows that, as long as $\phi \geq 6\delta + 9$, the probability that the i'th constraint violates the current configuration is $O(1/i)$. This gives the expected linear work bound.

Note that in most of these LP-type problems, storing the current configuration only uses constant space ($O(\delta)$ to be more accurate). Given a small-memory of constant size that can hold the configuration and a constant number of other variables and registers to run such an algorithm, there are no writes required in the whole algorithm (except to write the output). This includes problems like low-dimension linear programming and smallest enclosing disk. However, there exist special cases like the incremental closest-pair algorithm, which requires linear space to store the configuration. Here we denote *constant-size LP-type problems* as those requiring constant space ($O(\delta)$ words) to store the configuration.

Theorem 6.8.4. *Given $O(\delta^2 \log n)$ random bits and a small-memory of constant size, a δ-dimensional constant-size LP-type problem on n objects can be solved in $O(\delta!\, n)$ work and $O(1)$ writes for output on the (M, ω)-ARAM model.*

To get the guarantee for parallelism, we can plug in the generators for $(\log n)$-wise hash functions.

6.9 Write-Efficient Convex Hull

The ***planar convex hull*** problem takes as input a set of points in 2D and generates the smallest polygon (represented as a list of segments) that contains all of the points.

If the algorithm is insensitive to the output size, the fastest algorithms for computing a convex hull require $O(n \log n)$ work. This can be achieved by several algorithms that first sort the points by increasing x-coordinate, and then apply a $O(n)$ work step on the sorted points to compute the hull (see, e.g., [23, 149]). We note that we can trivially obtain an algorithm with $O(n \log n)$ reads and $O(n)$ writes, by first using a write-efficient sort introduced in Section 4.4, followed by the same post-processing step that takes $O(n)$ reads/writes. The sort can be done in $O(\log n)$ depth *whp*, and the post-processing step can be done in parallel in $O(n)$ work and $O(\log n)$ depth [146].

If the algorithm is sensitive to the output size, the symmetric work can be decreased to $O(n \log h)$, where h is the number of points on the hull. Naïve implementations of these algorithms would require $O(n \log h)$ reads/writes, respectively, while the minimum number of writes required is much lower since only the h points and segments on the hull need to be written. Our goal in this section is to develop work-efficient parallel algorithms where the number writes is asymptotically lower than the number of reads.

Algorithm 10: Output-sensitive (Upper) Convex Hull

Input: A set of n two-dimensional points in general position.

1: Place the points into h buckets each of size $O(n/h)$, where the x-coordinates of all points in bucket $i \in [0, \min(h-1)]$ are less than the x-coordinates of all points in buckets $j > i$.
   (a) Pick $\Theta(h \log n)$ random samples, sort them, and use every $(\log n)$'th sample as a splitter.
   (b) Have the remaining points each do a binary search on the splitters to determine which bucket they belong to.
   (c) Use prefix sums to determine and appropriate offsets into buckets for each point.
   (d) Have all points write to the appropriate offset into their bucket.
2: If there is only one bucket B, search up the tree of bridges and perform one of the following:
   (a) If all points in the buckets are on or below a bridge, then do nothing.
   (b) If there are points in B not covered by a bridge, then apply a standard output-sensitive convex hull algorithm on an input containing points in the bucket and the points forming the bridges $Br_{B,L}$ and $Br_{B,R}$.
3: Split points into two sets, L and R, where L contains points in the first $\lceil h/2 \rceil$ buckets, and R contains the remaining points.
4: Find the bridge between L and R.
5: Recursively apply Steps 2–5 on each of L and R, storing the bridge computed in each sub-problem as the left and right child, respectively, of the bridge in Step 4.
6: Obtain final solution by traversing down the tree of bridges.

6.9.1 An Output-Sensitive Algorithm

Obtaining a write-efficient output-sensitive convex hull algorithm with $O(n \log h)$ work requires more effort because we can no longer directly apply a comparison sort. We now describe how to obtain an algorithm with $O(n \log h)$ reads and $O(n \log \log h)$ writes.

Our algorithm uses divide-and-conquer and borrows ideas from [85, 189]. We first assume that we know the value of h and that $h = O(n/\log n)$ (to make oversampling work); we will remove these assumptions later. We also assume without loss of generality that no points have the same x-coordinate. We describe how to compute the upper hull (the hull above the line from the leftmost point to the rightmost point) and the lower hull can be computed analogously. The main steps of the algorithm are shown in Algorithm 10.

The algorithm is a divide-and-conquer algorithm but to avoid data movement we approximately pre-sort the points. In particular, we split the points into h buckets each of size $O(n/h)$, where the x-coordinates of all points in bucket $i \in [0, h-1]$ are less than the

x-coordinates of all points in buckets $j > i$. By picking $\Theta(h \log n)$ samples, sorting them, and using every $(\log n)$'th element as a splitter, the buckets can be shown to have size $O(n/h)$ *whp* using Chernoff bounds. This step is described as Step 1 of Algorithm 10.

If the input contains a single bucket, then we have reached the base case (Step 2), which we will describe how to handle shortly. Otherwise, the algorithm splits the points into approximately two halves, L and R (Step 3). This step requires no data movement, since the points have already been placed into their respective buckets. We then find the ***bridge*** of the upper hull between L and R, which is a line passing through a point in each set such that all points lie below the line (Step 4). The bridge can be found by solving the following two-dimensional linear program, where the bridge is the line $y = \alpha x + \beta$ and x_{mid} is the x-coordinate of a vertical line between L and R (which can be computed as a value arbitrarily close to the x-coordinate of the splitter element [212]):

$$\begin{aligned} &\text{minimize} && \alpha x_{mid} + \beta \\ &\text{subject to:} && \alpha x_{p_i} + \beta \geq y_{p_i} \quad \forall i \in L \cup R \end{aligned}$$

We describe how to solve 2D linear programs write-efficiently in Section 6.8.1.

The bridge found in Step 4 is stored as the root of a tree of bridges, and we recursively compute the tree of bridges on each of L and R in Step 5.

We now describe the base case, when there is only a single bucket B. The bucket searches up the tree of bridges, and considers any bridge whose x-range intersects with the bucket's x-range. A bridge can either lie on or above all points in B, have only a left endpoint in B, or have only a right endpoint in B. If we find any bridge that covers all of B, then no new convex hull edges will be generated from B and we are done. Otherwise, we find the one or two bridges that cover the most points in the bucket, and solve a subproblem with points in B and the up-to-two bridges using a standard $O(n \log h)$-work output-sensitive convex hull algorithm.

Call the set of the bridges with a left endpoint in the bucket $Br_{B,L}$, and the set of bridges with a right endpoint in the bucket $Br_{B,R}$. We wish to include the bridge in each set that covers the most points in the bucket. For a bridge $b \in Br_{B,L}$, let p_b be the endpoint of b in bucket B. The bridge in $Br_{B,L}$ that satisfies this criteria is a bridge b with the minimum value of x_{p_b}, since all points in B are below the bridge, and bridges $b \in Br_{B,L}$ cover all points in B with x-coordinate greater than x_{p_b}. If there are ties, then any bridge with x_{p_b} equal to the minimum value suffices. Similarly, the bridge in $Br_{B,R}$ that satisfies this criteria is a bridge b with the maximum value of x_{p_b}. Both of these bridges can be found during the search up the tree of bridges.

To obtain the final solution (Step 6), we start at the root of the tree of bridges, include the bridge in the solution, and recursively include into the solution the bridges in the descendants of the root that have not been already covered by a previously included bridge. To determine whether to include a bridge, we can search up the tree to see whether it has

been covered. If the base case is reached, then all bridges formed from the set of points in the bucket are included.

Cost analysis We now analyze the cost of Algorithm 10. Step 1a can be done using write-efficient sorting in $O(\log n)$ depth *whp* using $O(n \log h)$ reads and $O(n)$ writes. Step 1b takes $O(n \log h)$ reads, $O(n)$ writes, and $O(\log h)$ depth. Steps 1c takes $O(n)$ reads and writes, and $O(\log n)$ depth. Finally, Step 1d, takes $O(n)$ reads and writes, and $O(1)$ depth. So the cost for this pre-processing is $O(n \log h)$ reads, $O(n)$ writes and $O(\log n)$ depth *whp*.

The number of levels of recursion of Steps 2–5 is $O(\log h)$ *whp* since there are h buckets at the beginning and each sub-problem contains half as many buckets.

Step 2 takes $O(\log h)$ reads, $O(1)$ writes, and $O(\log h)$ depth to find $Br_{B,L}$ and $Br_{B,R}$ per bucket. Summed over all buckets, this takes $O(h \log h)$ reads, $O(h)$ writes and $O(\log h)$ depth. Each sub-problem solved using a standard output-sensitive algorithm contains $O(n/h)$ points, and only generates segments on the upper convex hull of the original point set, since we included the bridges coming in from both sides of the bucket. The total number of operations is $\sum_{i=1}^{h} O((n/h)\log(1 + h_i))$, where h_i is the number of points on the hull in bucket i and $h = \sum_{i=1}^{h} h_i$. The sum is maximized when all h_i's are equal, giving a total of $O(n)$ reads and writes. The algorithm of Kirkpatrick and Seidel [189] can be parallelized to take $O(n \log h)$ work and $O(\log^2 n)$ depth [139]. Note that the algorithm that we apply on the buckets on must take $O(n \log h)$ work overall for the value of h that we guessed, not the number of points on the actual convex hull. Once the amount of work done on the buckets exceeds cn for some constant c, we can assume that our guess of h is wrong, and terminate. To keep track of the work, we can modify the Kirkpatrick and Seidel algorithm that we use on the buckets to increment a shared counter in global memory whenever an operation is performed, and also check the counter before performing an operation and if it is above cn then terminate. The total number of reads/writes for maintaining the counter is cn, which is within our bounds.

Step 3 requires constant work/depth since the points are pre-sorted, and Step 4 requires $O(n)$ reads, $O(\log n)$ writes, and $O(\log n \log n)$ depth *whp* as shown in Section 6.8.1. There are $O(\log h)$ levels of recursion so the overall number of reads in this step is $O(n \log h)$ in expectation. The number of writes from this step satisfies the recurrence $W(n) = 2W(n/2) + O(\log n)$ which solves to $O(n)$. The total depth is $O(\log^2 n \log h)$ as each of the two recursive calls in Step 5 can be executed in parallel.

In Step 6, the searches up the tree take $O(\log h)$ reads and $O(1)$ writes for a total of $O(h \log h)$ reads and $O(h)$ writes. The depth is $O((\omega + \log h) \log h)$.

Overall, the algorithm requires $O(n \log h)$ reads, $O(n)$ writes, and $O(\log h \log^2 n)$ depth *whp*.

As done in [85], to remove the assumption that we know h, we will repeatedly guess h and apply the above algorithm until our guess is above the true value of h. On the i'th

application of the algorithm, our guess will be $h^* = 3^{2^i}$, so in total we require $O(\log\log h)$ iterations until $h^* \geq h$. The number of reads per iteration is $\sum_{i=0}^{O(\log\log h)} O(n \log 3^{2^i}) = O(n \log h)$. The number of writes is $O(n)$ per iteration, for a total of $O(n \log\log h)$. Finally, the depth is $O(\log^2 n \log h \log\log h)$ *whp* To remove the assumption that $h = O(n/\log n)$, once our guess of h exceeds $cn/\log n$ for some constant c we call the output-insensitive algorithm described earlier, which takes $O(n \log h)$ reads, $O(n)$ writes, and $O(\log n)$ depth *whp*. We obtain the following theorem.

Theorem 6.9.1. *A planar convex hull can be computed with $O(n(\log h + \omega \log\log h))$ expected work, $O(\log^2 n \log h \log\log h)$ depth* whp*, and $O(n \log\log h)$ writes* whp *under the Asymmetric NP model.*

6.9.2 Another Output-Sensitive Algorithm

Here we describe an algorithm with $O(nh)$ reads, $O(n)$ writes, and $O(\log^2 n)$ depth *whp*, obtained by modifying the algorithm of Kirkpatrick and Seidel [189]. Their original algorithm finds a bridge on the two halves of the points, uses the bridge to filter out a constant fraction of the points, and recursively finds the hull of the remaining points on each half. The number of levels of recursion is $O(\log h)$ and the number of sub-problems solved is h. Their algorithm as described takes $O(n \log h)$ reads and writes, and $O(\log n \log h)$ depth.

Our goal is to reduce the number of writes. Instead of moving the points such that points for a sub-problem are contiguous, we just inspect all of the points each time we need to find a bridge in a sub-problem. Furthermore, we do not filter out any points. Therefore, each time we need to find the bridge, we use the 2D linear programming algorithm in Section 6.8.1, taking $O(n)$ reads and $O(\log n)$ writes. The number of times we solve the 2D linear program is h, giving a total of $O(nh)$ reads and $O(h \log n)$ writes. We can find the splitters that divide the points approximately evenly, which is needed for the linear program, by taking a random sample of $\min(n, h \log n)$ elements at the beginning, sorting them, and using every $(\log n)$'th element as a splitter, as done in our first algorithm. This takes $O(n \log h)$ reads, $O(n)$ writes, and $O(\log n)$ depth *whp* in total. Since the 2D linear programming algorithm requires a randomized order, we generate a random permutation of the constraints at the beginning, and use it throughout the algorithm, taking $O(n)$ reads and writes and $O(\log n)$ depth. The two recursive calls can happen in parallel, so the depth is $O((\omega + \log n) \log n \log h)$ *whp*. To reduce the overall number of writes to $O(n)$, if the algorithm has not terminated after $O(\log\log n)$ levels of recursion (after $O(n \log n)$ reads and $O(\log^2 n)$ writes have been done in solving the LPs), we can switch to the output-insensitive algorithm that takes $O(n \log n)$ reads and $O(n)$ writes. This gives the following theorem.

Theorem 6.9.2. *A planar convex hull can be computed with $O(n(\min(h, \log n)+\omega))$ expected work, $O(\log^2 n \log h)$ depth* whp*, and $O(n)$ writes under the Asymmetric NP model.*

Chapter 7

Cache-Oblivious Algorithms for Dynamic Programming and Linear Algebra

7.1 Overview

Recall from Section 2.1.3 and 2.3.4, the *ideal-cache model* [131] is widely used in designing algorithms that optimize the communication between CPU and memory. The model is comprised of an unbounded memory and a cache of size M. Data are transferred between the two levels using cache lines of size B, and all computation occurs on data in the cache. An algorithm is *cache-oblivious* if it is unaware of both M and B. The goal of designing such algorithms is to reduce the *cache complexity*[1] (or the *I/O cost* indistinguishably) of an algorithm, which is the number of cache lines transferred between the cache and the main memory assuming an optimal (offline) cache replacement policy. Sequential cache-oblivious algorithms are flexible and portable, and adapt to all levels of a multi-level memory hierarchy. Such algorithms are well studied [28, 79, 110]. Regarding parallelism, as discussed in Section 2.3.4, Blelloch et al. [59] suggest that analyzing the depth and sequential cache complexity of an algorithm is sufficient for deriving upper bounds on parallel cache complexity.

In this chapter, we focus on a class of cache-oblivious algorithms that have a similar computation structure as matrix multiplication and can be coded up in nested for-loops. Their implementations are based on a divide-and-conquer approach that partitions the ranges of the loops and recurses on the subproblems until the base case is reached. Such algorithms provide efficient solutions to many problems solved using dynamic programming (e.g., the LWS/GAP/RNA/Parenthsis problems) and in linear algebra (e.g.,

[1] In this chapter, we refer to it as *symmetric cache complexity* to distinguish from the case when reads and writes have different costs.

matrix multiplication, Gaussian elimination, LU decomposition) [59, 92, 94, 96, 131, 181, 263, 270, 271, 279].

With the arrival of new non-volatile memory (NVM) technologies [171, 177], We have the desire for write-efficient algorithms. However, these classic cache-oblivious algorithms based on divide-and-conquer use asymptotically the same numbers of reads and writes to the main memory, which is inefficient on asymmetric memories. Meanwhile, given the various combinations of computation structures and data dependencies, proving the lower bound and designing the individual algorithm for each specific problem can take significant effort.

An additional motivation to study these algorithms is that, the computation of these algorithms usually involves complicated data dependencies that are entangled with the divide-and-conquer approach, making the algorithm design and analysis less obvious. As a result, although these algorithms have been studied for 20 years, some of them are suboptimal regarding the sequential *symmetric* cache complexity and/or parallel depth, or are complicated and can be greatly simplified. For example, previous algorithms to solve the GAP problem (i.e., the GAP recurrence) and protein accordion folding shown in [92, 94, 96, 181, 263, 270] have cache complexity $\Theta(n^3/B\sqrt{M})$. However, because of the complication of data dependencies, as we show in this thesis, this bound is not-optimal. Some other problems in higher (greater than three) dimensions (e.g., the RNA problem) are even harder, and directly applying the automatic systems in [92, 181] cannot yield algorithms with optimal cache complexity. In short, we lack a general method to analyze the lower and upper bounds of the cache complexity on these problems. Similar situations show up in terms of parallelism as well, and there exist spaces for improvements.

To tackle the challenges, in this chapter we propose a level of abstraction to analyze these cache-oblivious algorithms. Previous algorithms are usually designed and analyzed based on the number of nested loops, or the number of the dimensions of which the input and output are stored and organized. However, we observe that the key underlying factor in determining the cache complexity of these computations is the **number of input entries** involved in each basic computation cell (e.g., two input values for the multiplication in matrix product). This observation and the associated solution is one of the reasons we show improvements of the bounds. We abstract this relationship as k-d grid computation structures (henceforth k-d grids). A more formal definition is given Section 7.3. We then discuss the lower bounds to compute such k-d grids with and without considering the asymmetric cost between writes and reads. We also provide highly-parallelized algorithms with optimal work and cache complexity, that compute a k-d grid assuming no data dependency within it.

We claim that the extra level of abstraction helps us to better understand the difficulties (lower bounds) of some problems. This is because the structures of the algorithms and recurrences in this chapter (without considering data dependence) are always equivalent to one or more k-d grids. Hence the lower bounds on k-d grids apply to these algorithms

and recurrences, indicating whether the previous algorithms are optimal, and if not, what improvement can be achieved. For write-efficient algorithms, we can also show the minimum weighted reads and writes required. We note that the lower bounds shown in this setting are non-trivial, and raised as an unsolved problem at the beginning of the research of asymmetric algorithms [64, 84] for many years.

For upper bounds, the data dependencies between the computations come in. However, we observe that the computations of all cache-oblivious algorithms in this chapter can all be abstracted as or decomposed into multiple k-d grids without any local dependencies within each of them. Then by applying the efficient algorithms on k-d grids to these problems, we show better upper bounds as well as parallel depths. For problems we provide lower bounds, all but one algorithms can be shown optimal. We note that the decomposition is independent of whether the read and write costs are symmetric. Therefore, we will show that once we propose an optimal algorithm on k-d grids in the asymmetry setting, we automatically improve the cache complexities of all the problems from the classic algorithms that do not consider such asymmetry.

We believe that the framework for analyzing cache-oblivious algorithms based on k-d grids provides a better understanding of these algorithms. In particular, the contributions include:

- Algorithms with improved symmetric cache complexity on many problems, including the GAP recurrence, protein accordion folding, and RNA recurrence. We show that the previous cache bound $O(n^3/B\sqrt{M})$ for the GAP recurrence and protein accordion folding is not optimal, and we improve the bound to $O(n^2/B \cdot (n/M + \log \min\{n/\sqrt{M}, \sqrt{M}\}))$ and $\Theta(n^2/B \cdot (1 + n/M))$ respectively[2]. For RNA recurrence, we show an optimal cache complexity of $\Theta(n^4/BM)$, which improves the best existing result by $\Theta(M^{3/4})$.
- We show improved, linear parallel depth for cache-oblivious algorithms solving all-pair shortest-paths, Gaussian elimination, triangular system solver, LWS recurrences and protein accordion folding. For some of these problems, the depth bounds are achieved by previous algorithms [115, 264], but they assumes a much stronger model and the algorithms are also much more complicated (more discussion in Section 7.2). Our approaches are under the standard nested-parallel model and are much simpler. Our algorithms are not in-place, but in Section 7.6.1 (before Theorem 7.6.6) we show that the extra storage can bounded to arbitrarily small (any value no less than the cache size).
- We provide write-efficient cache-oblivious algorithms for all problems we discussed in this chapter, including matrix multiplication, many linear algebra algorithms,

[2]The improvement is $O(\sqrt{M})$ from an asymptotic perspective (i.e., n approaching infinity). For smaller range of n that $O(\sqrt{M}) \leq n \leq O(M)$, the improvement is $O(n/\sqrt{M}/\log(n/\sqrt{M}))$ and $O(n/\sqrt{M})$ respectively for the two cases. (The computation fully fit into the cache when $n < O(\sqrt{M})$.)

all-pair shortest-paths, and a number of dynamic programming recurrences. If a write costs ω times more than a read (the formal computational model shown in Section 7.2), the asymmetric cache complexity is improved by a factor of $\Theta(\omega^{1/2})$ or $\Theta(\omega^{2/3})$ on each problem compared to the previous results [66]. We also show that this improvement is optimal under certain assumptions (the CBCO paradigm, defined in Section 7.4.2).

- The analysis framework is concise. In this chapter, we discuss the lower bounds and parallel algorithms on a dozen or so computations and DP recurrences, which can be further applied to dozens of real-world problems[3]. The results are shown in both settings with or without considering the asymmetric costs between reads and writes.

7.2 Preliminaries and Related Work

Computational models. In this chapter we use the computation discussed in Chapter 2 to measure the cost of an algorithm. Unlike previous chapters, because we study and improve the cache-oblivious algorithms in both the symmetric and asymmetric settings, we make a few changes of the terminologies in this chapter for the ease of algorithm description. We refer to the cache complexity in the symmetric setting (Section 2.1.3) as the ***symmetric cache complexity***. Then, we refer to the cache complexity in the (M, ω)-ARAM (Section 2.3.4) as the ***asymmetric cache complexity***. Since we focus on reducing the I/Os and throughout this chapter all algorithms require optimal asymptotic arithmetic operations, the work W is the total number of arithmetic operations. The work in (M, ω)-ARAM is just W plus the asymmetric cache complexity.

In the analysis, we always assume that the input size is much larger than the cache size (which is usually the case in practice). Otherwise, both the upper and the lower bounds on cache complexity also include the term for output in either the symmetric or the asymmetric settings. For simplicity, this term is ignored in the asymptotic analysis.

Parallel and cache-oblivious algorithms for dynamic programming and linear algebra. Dynamic Programming (DP) is an optimization strategy that decomposes a problem into subproblems with optimal substructure. It has been studied for over sixty years [9, 40, 108]. For the problems that we consider in this chapter, the parallel DP algorithms were already discussed by a rich literature in the eighties and nighties (e.g., [122, 134, 136, 173, 174, 242]). Later work not only considers parallelism, but also optimizes symmetric cache complexity (e.g., [59, 92, 93, 94, 96, 115, 131, 181, 259, 263, 264, 270]). The algorithms in linear algebra that share the similar computation structures (but with different orders in the computation) are also discussed (e.g., [96, 115, 271, 279]).

[3]Like in this thesis we abstract the "2-knapsack recurrence", which fits into our k-d grid computation structure and applies to many algorithms.

Discussions about write-avoiding algorithms by Carson et al. [84].

Carson et al. also discussed algorithms using less writes in the paper [84]. They concluded that many cache-oblivious algorithms like matrix multiplication could not be write-avoiding. Their definition of write-avoiding is different from our write-efficiency, and it requires the algorithm to reduce writes without asymptotically increasing reads. Hence, their negative conclusion does not contradict the result in this thesis.

We now show an example that optimal number of reads leads to worse overall asymmetric cache complexity. Let's say a write is n times more expansive than a read. A smart cache-oblivious matrix multiplication algorithm should apply n^2 inner products, and the asymmetric cache complexity is $O(n^3/B)$—$O(n/B)$ reads and $O(1/B)$ writes per inner product. However, the algorithms using optimal number of reads requires $O(n^3/B\sqrt{M})$ reads and writes, so the overall cache complexity is $O(n \cdot n^3/B\sqrt{M})$. The algorithm requiring more reads is a factor of $O(n/\sqrt{M})$ better on the asymmetric cache complexity. Notice that we always assume $n = \omega(\sqrt{M})$ since otherwise the whole computation is trivially in the cache and has no cost. As a result, an algorithm with a good asymmetric cache complexity does not need to be write-avoiding. Since their assumption is strong, although the claims in their paper are true, we believe that there are no direct causal relationships between their results and write-efficient algorithms (their results can be viewed as side-results of the analysis in Section 7.4).

The algorithms on asymmetric memory in this chapter all require extra reads, but can greatly reduce the overall asymmetric cache complexity compared to the previous cache-oblivious algorithms. The goal of this chapter is to find the optimal cache-oblivious algorithms for any given write-read asymmetry ω.

Discussions on previous work.

We now discuss several possible confusions of this chapter.

We define and analyze the k-d grid computation structure. Similar grid structures and the bounds on computing them have been discussed in many previous work (e.g., [8, 36, 37, 92, 94, 96, 180, 181, 270]). However, we note that the definition in this thesis is different from those papers. In the k-d grid, the dimension of the grid is related to the number of entries per basic computation unit (formal definition in Section 7.3), not the dimension of the input/output arrays. As a result, the readers should not confuse the k-d grid in this thesis with the previous grid structures.

Since we believe that the abstraction and definition are new, the proof of lower bounds in Section 7.4 focuses on the simplest sequential setting with a limited-size cache, which is different from the parallel or distributed settings discussed with infinite-size local memory in previous work (e.g., [8, 36, 37, 180]). It is an interesting future work to extend the results in this thesis to the more complicated settings, especially for the asymmetric setting.

Although the definition of k-d grid is different, in the special case when the number of input entries per basic block is the same as the dimension of input/output arrays, the

analysis based on k-d grid provides the same sequential symmetric cache complexity as the previous work [92, 94, 96, 131, 181, 270]. The sequential symmetric versions of these algorithms based on the k-d grid are the same as or similar to the previous algorithms. One of such examples is the matrix multiplication [131], and we recommend the reader to use it as an instantiation to understand our improved parallel and asymmetric algorithms in Section 7.5 and 7.6. This is also the case for other linear algebra algorithms in Section 7.7.2, and the LWS and Parenthesis recurrences in Section 7.8.1 and 7.8.4. For the algorithms with the same bounds as previous, we give another way to describe and analyze them – this is a minor result, which we do not claim it as a contribution.

Some previous work [115, 264] achieves the same linear depth bounds in several problems discussed in this thesis. We note that they assume a much stronger model (i.e., enabling the DAG to be non-nested and dynamically unfolded), so their algorithms either need a specially designed scheduler [115], or cannot benefit from the scheduling results discussed in this preliminary section. Our algorithms are much simpler and under the nested-parallel model.

The dynamic programming recurrences discussed in this chapter have non-local dependencies (definition given in [136]), and we point out that they are pretty different from the problems like edit distance or stencil computations (e.g., [97, 129, 168, 195]) that only have local dependencies. We did not consider other types of dynamic programming approaches like rank convergence or hybrid r-way DAC algorithms [95, 203, 204] that cannot guarantee processor- and cache-obliviousness simultaneously.

7.3 The k-d Grid Computation Structure

The k-d grid computation structure (short for the k-d grid) is defined as a k-dimensional grid C of size $n_1 \times n_2 \times \cdots \times n_k$. Here we consider k to be a small constant greater than 1. This computation requires $k-1$ input arrays $I_1, \cdots, I_{k-1}$ and generate one output array O. Each array has $k-1$ dimensions and is the projection of the grid out of one different dimension. Each ***cell*** in the grid represents some certain computation that requires $k-1$ inputs and generates a temporary value. This temporary value is "added" to the corresponding location in the output array, and we assume this addition is associative. The $k-1$ inputs of this cell are the projections of this cell removing each (but not the last) dimensions, and the output is the projection removing the last dimension. They are referred to as the input and output ***entries*** of this cell. When computing each cell, the input and output entries must be in the cache. Figure 7.1 illustrates such a computation in 2 and 3 dimensions.

We refer to a k-d grid computation structure as a ***square grid computation structure*** (short for a square grid) of size n if it has size $n_1 = \cdots = n_k = n$. More concisely, we say a k-d grid has size n if it is square and of size n.

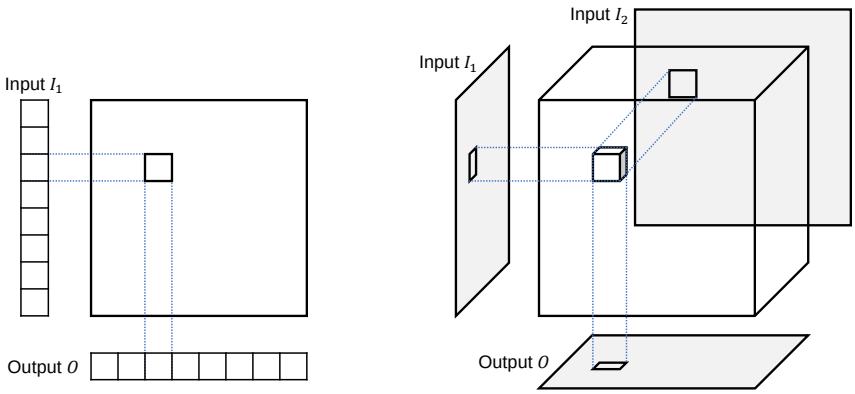


Figure 7.1: An illustration of a 2d and a 3d grid. The left figure shows the 2d case where the input I_1 and output O are 1d arrays, and each computation cell requires exactly one entry in I_1 as input, and update one entry in O. For the 3d case on the right, the inputs and output are 2d arrays, and each computation cell requires one entry from input I_1 and one from input I_2. The input/output entries of each cell are the projections of this cell on different 2d arrays.

For instance, when multiplying two matrices of size n-by-n on a semiring $(\times, +)$, the computation of the classic cubic algorithm exactly matches the 3d square grid. A corresponding 2d case is when computing a matrix-vector multiplication where the matrix is implicit (i.e., we can query the value of each cell, but the matrix is not stored explicitly). Such applications are commonly seen in dynamic programming algorithms.

The key aspect to decide the dimension of a computation is the number of inputs that each basic cell element requires. For example, when multiplying two dense tensors, although each tensor may have multiple dimensions, each multiplication operation is only based on two entries, so it is a 3d grid. The cache-oblivious algorithms discussed in this thesis are based on k-d grids with $k = 2$ or 3, but we can also find applications with larger k (e.g., a Nim game with some certain rules on multiple piles [77]).

For some dynamic programming and linear algebra algorithms in this chapter, the computations can be abstracted as k-d grids, but a constant fraction of the grid cells are empty. We call such a grid as an α-***full*** grid for some constant $0 < \alpha < 1$ if at least an $\alpha \pm o(1)$ fraction of the cells are non-empty. We will show that all properties we show for a k-d grid also work for the α-full case, since the constant α affects the analysis of neither the lower bounds nor the algorithms.

7.4 The Lower Bounds

We first discuss the lower bounds of the cache complexities for a k-d grid computation structure, which sets the target to design the algorithms in the following sections. In Section 7.4.1 we show the symmetric cache complexity. This is a direct extension of the

classic result by Hong and Kong [170] to an arbitrary dimension. Then in Section 7.4.2 we discuss the asymmetric cache complexity when writes are more expensive than reads, which is more interesting and has involved analyses.

7.4.1 Symmetric Cache Complexity

The symmetric cache complexity of a k-d grid is simple to analyze, yielding to the following result:

Theorem 7.4.1. *The symmetric cache complexity of a k-d grid computation structure with size n is $\Omega\left(\frac{n^k}{M^{1/(k-1)}B}\right)$.*

Proof. Let's say the computations of n^k cells are sequentialized in an array. For blocks of size $S = 2^k M^{k/(k-1)}$, the minimum number of inputs and outputs of each block is at least $2^{k-1}kM \geq 2M$. Since only a total amount of M entries can be held in the cache at the beginning of the computation, the cache complexity to load the input/output and finish the computation for such a block is $\Omega(M/B)$. Since there are $n^k/S = \Theta(n^k M^{-k/(k-1)})$ of such blocks, the cache complexity of the entire computation is $\Omega(M/B) \cdot n^k/S = \Omega(n^k/(M^{1/(k-1)}B))$. □

Notice that the proof does not assume cache-obliviousness, but the lower bound is asymptotically tight by applying a sequential cache-oblivious algorithm that is based on 2^k-way divide-and-conquer [131].

7.4.2 Asymmetric Cache Complexity

We now consider the asymmetric cache complexity of a k-d grid computation structure. Unfortunately, this case is significantly harder than the symmetric setting. Again for simplicity we first analyze the square grid of size n, which can be extended to the more general cases similar to [131].

Interestingly, there is no specific pattern that a cache-oblivious algorithm has to follow. Some existing algorithms use "buffers" to support cache-obliviousness (e.g., [25]), and many others use a recursive divide-and-conquer framework. For the recursive approaches, when the cache complexity of the computation is not leaf-dominated (like various sorting algorithms [131]), a larger fan-out in the recursion is more preferable (usually set to $O(\sqrt{n})$). Otherwise, when it is leaf-dominated, existing efficient algorithms all pick a constant fan-out in the recursion in order to reach the base case as soon as possible. All problems we discuss in this chapter are in this category, so we make our analysis under the following constraints. More discussion about this constraint in given in Section 7.9.

Definition 12 (CBCO paradigm)**.** *We say a divide-and-conquer algorithm is under the* constant-branching cache-oblivious (CBCO) paradigm *if it has an input-value independent*

computational DAG, such that each task has constant[4] *fan-outs of its successive subtasks until the base cases, and the partition of each task is decided by the (sub)problem size and independent with the cache parameters (M and B).*

Notice that ω is a parameter of the main memory, instead of a cache parameter. One can define resource-obliviousness [103] so that the value of ω is not exposed to the algorithms, but this is out of the scope of this chapter.

We now prove the (sequential) lower bound on the asymmetric cache complexity of a k-d grid under the CBCO paradigm. The two places that we use this assumption in the proof are the constant branching of the recursion, and "scale-free" of the cache-oblivious algorithms: the structure in the recursive levels around the boundary of cache size is identical.

Theorem 7.4.2. *The asymmetric cache complexity of k-d grid is $\Omega\left(\frac{n^3\omega^{1/k}}{M^{1/(k-1)}B}\right)$ under the CBCO paradigm.*

Proof. We prove the lower bound using the same approach in Section 7.4.1—analyzing blocks of size S based on the sequence of the operations executed by the algorithm. The cache can hold M entries as temporary space for the computation. For the lower bound, we only consider the computation in each cell without considering the step adding the calculated value back to the output array. Again when computing each cell, the k input and output entries have to be in the cache.

For a block of operations of size S, the cache needs to hold the entries in $I_1, \cdots, I_{k-1}$ and O corresponding to the cells in this sequence at least once during the computation. The number of entries is minimized when the sequence of operations are within a k-d cuboid of size $S = a_1 \times a_2 \times \cdots \times a_k$ where the projections on I_i and O are $(k-1)$-d arrays with sizes $a_1 \times \cdots \times a_{i-1} \times a_{i+1} \times \cdots \times a_k$ and $a_1 \times \cdots \times a_{k-1}$. Namely, the number of entries is at least $S/B \cdot 1/a_i$ for the corresponding input or output array.

Note that the input arrays are symmetric to each other regarding the access cost, but in the asymmetric setting storing the output is more expensive. As a result, the cache complexity is minimized when $a_1 = \cdots = a_{k-1} = a$, and let's denote $a_k = ar$ where r is the ratio between a_k and other a_i. Here we assume $r \geq 1$ since reads are cheaper. Due to the scale-free property that M and n are arbitrary, r should be fixed (within a small constant range) for the entire recursion.

Similar to the analysis in the proof of Theorem 7.4.1, for a block of size S, the read transfers required by the cache is $\Omega\left(\frac{n^k}{SB} \cdot \max\{a^{k-1}r - M, 0\}\right)$, where n^k/S is the number of such blocks, and $\max\{a^{k-1}r - M, 0\}/B$ lower bounds the number of reads per block because at most M entries can be stored in the cache from the previous block. Similarly,

[4]It can exponentially depend on k where we assume k is a constant.

the write cost is $\Omega\left(\frac{\omega n^k}{SB} \cdot \max\{a^{k-1} - M, 0\}\right)$. In total, the cost is:

$$\begin{aligned} Q &= \Omega\left(\frac{n^k}{SB} \cdot \left(\max\{a^{k-1}r - M, 0\} + \omega \max\{a^{k-1} - M, 0\}\right)\right) \\ &= \Omega\left(\frac{n^k}{SB}\left(\max\{S^{(k-1)/k}r^{1/k} - M, 0\} + \omega \max\left\{\frac{S^{(k-1)/k}}{r^{(k-1)/k}} - M, 0\right\}\right)\right) \end{aligned}$$

The second step is due to $S = O(a^k r)$.

The cost decreases as the increase of S, but it has two discontinuous points $S_1 = M^{k/(k-1)}/r^{1/(k-1)}$ and $S_2 = M^{k/(k-1)}r$. Therefore,

$$\begin{aligned} Q &= \Omega\left(\frac{n^k}{S_1 B} S_1^{(k-1)/k} r^{1/k} + \frac{n^k}{S_2 B}\left(S_2^{(k-1)/k} r^{1/k} + \frac{\omega S_2^{(k-1)/k}}{r^{(k-1)/k}}\right)\right) \\ &= \Omega\left(\frac{n^k}{S_1^{1/k} B} r^{1/k} + \frac{n^k}{S_2^{1/k} B}\left(r^{1/k} + \frac{\omega}{r^{(k-1)/k}}\right)\right) = \frac{n^3}{M^{1/k} B}\left(r^{1/k} + \frac{\omega}{r}\right) \end{aligned}$$

When $r = \omega^{(k-1)/k}$, Q is minimized to be $\Omega\left(\frac{n^3 \omega^{1/k}}{M^{1/(k-1)} B}\right)$ and this leads to the theorem. □

7.5 A Matching Upper Bound on Asymmetric Memory

In the sequential and symmetric setting, the classic cache-oblivious divide-and-conquer algorithms to compute the k-d grid (e.g., 3D case in [131]) is optimal. In the asymmetric setting, the proof of Theorem 7.4.2 indicates that the classic algorithm is not optimal. The key to improve the cost is the balancing factor $r = \omega^{(k-1)/k}$ that leads to more cheap reads and less expensive writes in each sub-computation.

We now show that the lower bound in Theorem 7.4.2 is tight by a (sequential) cache-oblivious algorithm with such asymmetric cache complexity. The algorithm is given in Algorithm 11, which can be viewed as a variant of the classic approach with minor modifications on how to partition the computation. Notice that in line 6 and 10, "conceptually" means the partitions are used for the ease of algorithm description. In practice, we can just pass the ranges of indices of the subtask in the recursion, instead of actually partitioning the arrays.

Compared to the the classic approaches (e.g., [131]) that partition the largest input dimension among n_i, the only underlying difference in the new algorithm is in line 4: when partitioning the dimension not related to the output array O, n_k has to be $\omega^{(k-1)/k}$ times larger than $n_1, \cdots, n_{k-1}$. This modification introduces an asymmetry between the input size and output size of each subtask, which leads to an improvement in the cache efficiency.

Algorithm 11: ASYM-ALG$(I_1, \cdots, I_{k-1}, O)$

Input: $k-1$ input arrays $I_1, \cdots, I_{k-1}$, read/write asymmetry ω
Output: Output array O

1 The computation has size $n_1 \times n_2 \times \cdots \times n_k$
2 **if** $I_1, \cdots, I_{k-1}, O$ are small enough **then**
3 | Solve the base case and **return**
4 $i \leftarrow \arg\max_{1 \le i \le k}\{n_i x_i\}$ where $x_k = \omega^{-(k-1)/k}$ and $x_j = 1$ otherwise for $1 \le j < k$
5 **if** $i = k$ **then**
6 | (Conceptually) equally partition $I_1, \cdots, I_{k-1}$ into $\{I_{1,a}, I_{1,b}\}, \cdots, \{I_{k-1,a}, I_{k-1,b}\}$ on k-th dimension
7 | ASYM-ALG$(I_{1,a}, \cdots, I_{k-1,a}, O)$
8 | ASYM-ALG$(I_{1,b}, \cdots, I_{k-1,b}, O)$
9 **else**
10 | (Conceptually) equally partition $I_1, \cdots, I_{i-1}, I_{i+1}, \cdots, I_{k-1}, O$ into $\{I_{1,a}, I_{1,b}\}, \cdots, \{I_{k-1,a}, I_{k-1,b}\}, \{O_a, O_b\}$ on i-th dimension
11 | ASYM-ALG$(I_{1,a}, \cdots, I_{i-1,a}, I_i, I_{i+1,a}, \cdots, I_{k-1,a}, O_a)$
12 | ASYM-ALG$(I_{1,b}, \cdots, I_{i-1,b}, I_i, I_{i+1,b}, \cdots, I_{k-1,b}, O_b)$

For simplicity, we show the asymmetric cache complexity for square arrays (i.e., $n_1 = \cdots = n_k$) and $n = \Omega(\omega^{(k-1)/k}M)$, and the general case can be analyzed similar to [131].

Theorem 7.5.1. *Algorithm 11 computes the k-d grid of size n with asymmetric cache complexity* $\Theta\left(\frac{n^k \omega^{1/k}}{M^{1/(k-1)}B}\right)$.

Proof. We separately analyze the numbers of reads and writes required by Algorithm 11. In the sequential execution of Algorithm 11, each subproblem only requires $O(1)$ extra temporary space. Also, our analysis ignores rounding issues since they will not affect the asymptotic bounds.

When starting from the square grid at the beginning, the algorithm first partitions in the first $k-1$ dimensions (via line 10 to 12) into $\omega^{(k-1)/k}$-by-$\omega^{(k-1)/k}$ subproblems (refer to as *second-phase* subproblems) each with size $(n/\omega^{(k-1)/k}) \times \cdots \times (n/\omega^{(k-1)/k}) \times n$, and then partition k dimensions in turn.

The number of writes of the algorithm $W(n)$ (to array O) follows the recurrences that:

$$W'(n) = 2^k W'(n/2) + O(1)$$

and

$$W(n) = (\omega^{(k-1)/k})^{k-1} \cdot \left(W'(n/\omega^{(k-1)/k}) + O(1)\right)$$

where $W'(n)$ is the number of writes required by the second-phase subproblems with the size of O being $n \times \cdots \times n$. The base case is when $W'(M^{1/(k-1)}) = O(M/B)$. Solving the recurrences gives $W'(n/\omega^{(k-1)/k}) = O\left(\frac{n^k \omega^{1-k}}{M^{1/(k-1)}B}\right)$, and $W(n) = O\left(\frac{n^k \omega^{(1-k)/k}}{M^{1/(k-1)}B}\right)$.

We can analyze the reads similarly by defining $R(n)$ and $R'(n)$. The recurrences on reads are:

$$R'(n) = 2^k R'(n/2) + O(1)$$

and

$$R(n) = (\omega^{(k-1)/k})^{k-1} \cdot \left(R'(n/\omega^{(k-1)/k}) + O(1)\right)$$

The difference occurs in the base case since the input fits into the cache when $n = M^{1/(k-1)}/\omega^{1/k}$. Namely, $R'(M^{1/(k-1)}/\omega^{1/k}) = O(M/B)$. Therefore, by solving the recurrences, we have $R'(n/\omega^{(k-1)/k}) = O\left(\frac{n^k \omega^{2-k}}{M^{1/(k-1)}B}\right)$ and $R(n) = O\left(\frac{n^k \omega^{1/k}}{M^{1/(k-1)}B}\right)$.

The overall (sequential) asymmetric cache complexity for Algorithm 11 is:

$$Q(n) = R(n) + \omega W(n) = O\left(\frac{n^k \omega^{1/k}}{M^{1/(k-1)}B}\right)$$

and combining with the lower bound of Theorem 7.4.2 proves the theorem. □

Comparing to the classic approach, the new algorithm improves the asymmetric cache complexity by a factor of $O(\omega^{(k-1)/k})$, since the classic algorithm requires $\Theta(n^k/(M^{1/(k-1)}B))$ reads and writes. Again here we assume n^{k-1} is much larger than M. Otherwise, the lower and upper bounds should include $\Theta(\omega n^{k-1}/B)$ for just writing down the output O.

7.6 Parallelism

We now show the parallelism in computing the k-d grids. We show that the parallel versions of the cache-oblivious algorithms only has polylogarithmic depth, indicating that they are highly parallelized.

7.6.1 The Symmetric Case

We first discuss the classic algorithm on symmetric memory. For a square grid, the algorithm partition the k-dimensions in turn until the base case is reached.

Notice that in every k consecutive partitions, there are no dependencies in $k-1$ of them, which we can take full parallelism from them. The only exception is during the partition in the k-th dimension, whereas both subtasks share the same output array O and cause write concurrence. If such two subtasks are sequentialized (like in [131]), the depth is $D(n) = 2D(n/2) + O(1) = O(n)$.

We now introduce the algorithm with logarithmic depth. As just explained, to avoid the two subtasks from editing the same elements in the output array O, our algorithm works in the following way when partitioning the k-th dimension:

1. Allocating two stack-allocated temporary arrays with the same size of the output array O before the two recursive function calls.
2. Applying computation for the k-d grid in two subtasks with different output arrays that are just allocated (with no concurrency to the other subtask).
3. When both subtasks finish, the computed values are merged back in parallel, with work proportional to the output size and $O(\log n)$ depth.
4. Deallocating the temporary arrays.

Notice that the algorithm also works if we only allocate temporary space for one of the subtasks, while the other subtask still works on the original space for the output array. This can be a possible improvement in practice, but in high dimensional case ($k > 2$) it requires more complicated details to pass the pointers of the output arrays to descendant nodes, aligning arrays to cache lines, etc. Theoretically, this version does not change the bounds except for the stack space in Lemma 7.6.1 when $k = 2$.

We first analyze the cost of square grids of size n in the **symmetric** setting, and will discuss the asymmetric setting later.

Lemma 7.6.1. *The overall stack space for a subtask of size n is $O(n^{k-1})$.*

Proof. When partitioning the output (k-th) dimension, the algorithm allocates and computes two subtasks of size $n/2$. This leads to the following recurrence:

$$S(n) = 2S(n/2) + O(n^{k-1})$$

which solves to $S(n) = O(n^{k-1})$ when $k > 2$. When $k = 2$, we can apply the version that only allocates temporary space for one subtask, which changes the constant before $S(n/2)$ to 1, and yields to the same bound as $S(n) = O(n)$. Note that we only need to analyze one of the branches, since the temporary spaces that are not allocated in the direct ancestor of this subtask have already been deallocated, and will be reused for later computations for the current branch. □

With the lemma, we have the following corollary:

Corollary 7.6.2. *A subtask of size $n \leq M^{1/(k-1)}$ can be computed within a cache of size $O(M)$.*

This corollary indicates that this modified parallel algorithm has the same sequential cache complexity since it fits into the cache in the same level as the classic algorithm (the only minor difference is the required cache size increases by a small constant factor). Therefore we can apply the a similar analysis in [131] ($k = 3$ in the chapter) to show the following lemma:

Lemma 7.6.3. *The sequential symmetric cache complexity of the parallel cache-oblivious algorithm to compute a k-d grid of size n is* $O(n^3/M^{1/(k-1)}B)$.

Assuming that we can allocate a chunk of memory in constant time, the depth of this approach is simply $O(\log^2 n)$—$O(\log n)$ levels of recursion, each with $O(\log n)$ depth for the additions [59].

We have shown the parallel depth and symmetric cache complexity. Therefore, we have the following result for parallel symmetric cache complexity by applying the scheduling theorem in the preliminary section.

Corollary 7.6.4. *The k-d grid of size n can be computed with the parallel symmetric cache complexity of* $O(n^3/M^{1/(k-1)}B + pM\log^2 n)$ *with private caches, or* $O(n^3/M^{1/(k-1)}B)$ *with share caches of size* $M + pB\log^2 n$.

Lemma 7.6.1 shows that the extra space required is $S_1 = O(n^{k-1})$ for sequentially running the parallel algorithm. Naïvely the parallel space requirement is pS_1, which can be very large. We now show a better upper bound for the extra space.

Lemma 7.6.5. *The overall space requirement of the k-d grid is* $O(p^{1/k}n^{k-1})$.

Proof. We analyze the amount of space allocated in total for all processors. Lemma 7.6.1 indicates that if the root of the computation on one processor has the output array of size $(n')^{k-1}$, then the space requirement for this task is $O((n')^{k-1})$. There can be at most 2^k processors starting with their computations of size $n^{k-1}/2^{k-1}$, $(2^k)^2$ of size $n^{k-1}/(2^{k-1})^2$, until $(2^k)^q$ processors of size $n^{k-1}/(2^{k-1})^q$ where $q = \log_{2^k} p$, when each of the p processors all get a task. This is the most pessimistic case that maximizes the overall space requirement, which is:

$$\sum_{h=1}^{\log_{2^k} p} O\left(\frac{n^{k-1}}{(2^{k-1})^h}\right) \cdot (2^k)^h = p \cdot O\left(\frac{n^{k-1}}{(2^{k-1})^{\log_{2^k} p}}\right) = O(p^{1/k}n^{k-1})$$

which gives the stated bound. □

We believe the space requirement for the parallel cache-oblivious algorithm is acceptable since it is asymptotically the same as the most intuitively (non-cache-oblivious) parallel algorithm that partitions the computation into p square subtasks each with size $n/p^{1/k}$. In practice nowadays it is easy to fit several terabyte main memory onto a single powerful machine such that the space requirement can usually be satisfied. For example, when $k = 2$, $p = 100$ and the main memory can hold 10^{12} entries, the grid needs to contain $\approx 10^{22}$ cells to exceed the memory size: such computation can be too big to run on a single shared-memory machine. For $k \geq 3$, the extra term $p^{1/k}$ only affects a small range of input sizes that cannot fit. Even if it falls into this small region, we can always slightly change the algorithm to bound the extra space. We first partition the input dimensions for

$\log_2 p$ rounds to bound the largest possible output size to be $O(n^{k-1}/p)$ that one processor can steal (similar to the case discussed in Section 7.6.2). Then the overall extra space is limited to $O(n^{k-1})$, the same as the input/output size. If needed, the constant in the big-O can also be bounded. Such change will not affect the cache complexity and the depth as long as the main memory size is larger than pM. In practice, it is always several orders of magnitude larger than pM. As a conclusion, we believe the extra space requirement is acceptable in most cases in practice. Throughout the rest of the chapter, we focus on the cache complexity and the depth, similar to the previous work [59].

Combining all results gives the following theorem:

Theorem 7.6.6. *There exists a cache-oblivious algorithm for k-d grid computation structure of size n that requires $\Theta(n^k)$ work, $\Theta\left(\frac{n^k}{M^{1/(k-1)}B}\right)$ symmetric cache complexity, $O(\log^2 n)$ depth, and $O(p^{1/k}n^{k-1})$ main memory size.*

7.6.2 The Asymmetric Case

Algorithm 11 considers the write-read asymmetry, which involves some minor changes to the classic cache-oblivious algorithm. In terms of parallelism, the changes only affect the order of the partitioning of the k-d grid in the recurrence, but not the parallel version and the analysis in Section 7.6.1. As a result, the depth of the parallel variant of Algorithm 11 is also $O(\log^2 n)$. The extra space requirement is actually decreased, because the asymmetric algorithm has a higher priority in partitioning the input dimensions that does not requires allocation temporary space.

Lemma 7.6.7. *The space requirement of Algorithm 11 on p processors is $O(n^{k-1}(1+p^{1/k}/\omega^{(k-1)/k}))$.*

Proof. Algorithm 11 first partition the input dimensions until $q = O(\omega^{(k-1)^2/k})$ subtasks are generated. Then the algorithm will partition k dimensions in turn. If $p < q$, then each processor requires no more than $O(n^{k-1}/q)$ extra space at any time, so the overall extra space is $O(p \cdot n^{k-1}/q) = O(n)$. Otherwise, the worst case appears when $O(p/q)$ processors work on each of the subtasks. Based on Lemma 7.6.5, the extra space is bounded by $O((p/q)^{1/k} \cdot q \cdot n^{k-1}/q) = O(p^{1/k}n^{k-1}/\omega^{(k-1)/k})$. Combining the two cases gives the stated bounds. □

Lemma 7.6.7 indicates that Algorithm 11 requires extra space no more than the input/output size asymptotically when $p = O(\omega^{k-1})$, which should always be true in practice.

The challenge arises in scheduling this computation. The scheduling theorem for the asymmetric case constraints on the non-leaf stack memory to be a constant size. This contradicts the parallel version in Section 7.6.1. This problem can be fixed based on Lemma 7.6.1 that upper bounds the overall extra memory on one task. Therefore the stack-allocated array can be moved to the heap space. Once a task is stolen, the first allocation will annotate a chunk of memory with size order of $|O|$ where O is the current

output. Then all successive heap-based memory allocation can be simulated on this chunk of memory. In this manner, the stack memory of each node corresponding to a function call is constant, which allows us to apply the scheduling theorem in Chapter 2.

Theorem 7.6.8. *Algorithm 11 with input size n requires* $\Theta(n^k)$ *work,* $\Theta\left(\frac{n^k \omega^{1/k}}{M^{1/(k-1)}B}\right)$ *asymmetric cache complexity, and* $O(\log^2 n)$ *depth for a k-d grid of size n.*

7.7 Numerical Algorithms and All-Pair Shortest Paths

In this section we discuss matrix multiplication, Kleene's algorithm on all pair shortest-paths, and some linear algebra algorithms including Strassen algorithm, Gaussian elimination (LU decomposition), and triangular system solver. The common theme in these algorithms is that their computation structures are very similar to that of matrix multiplication, which is a 3d grid. Strassen algorithm is slightly different and introduced separately in Appendix 7.7.6. Other algorithms are summarized in Section 7.7.2 and the details are given in Appendix 7.7.3–7.7.5.

We show improved asymmetric cache complexity for all problems. For Gaussian elimination and triangular system solver, we show linear depth algorithms in both symmetric and asymmetric settings which are based on the parallel algorithm discussed in Section 7.6. There exist work-optimal and sublinear depth algorithm for APSP [269], but we are unaware of how to make it I/O-efficient. Compared to previous linear depth algorithms in [115], our algorithm is in the classic nested-parallel model and can be scheduled using the work-stealing scheduler. Also, we believe our algorithms are significantly simpler.

7.7.1 Matrix Multiplication

The combinatorial matrix multiplication (definition in Section 7.2) is one of the simplest cases of the 3d grid. Given a semiring $(\times, +)$, in matrix multiplication each cell corresponds to a "$\times$" operation of the two corresponding input values and the "$+$" operation is associative. Since there are no dependencies between the operations, we can simply apply Theorem 7.6.6 and 7.6.8 to get the following result.

Corollary 7.7.1. *Combinatorial matrix multiplication of size n can be solved in* $\Theta(n^3)$ *work, optimal symmetric and asymmetric cache complexity of* $\Theta\left(\frac{n^3}{B\sqrt{M}}\right)$ *and* $\Theta\left(\frac{\omega^{1/3}n^3}{B\sqrt{M}}\right)$ *respectively, and* $O(\log^2 n)$ *depth.*

7.7.2 Result Overview on All-Pair Shortest Paths and Linear Algebra Algorithms

We now discuss the well-known cache-oblivious algorithms to solve all-pair shortest paths (APSP) on a graph, Gaussian elimination (LU decomposition), and triangular system

solver. These algorithms share similar computation structures and can usually be discussed together. Chowdhury and Ramachandran [94, 96] categorized matrix multiplication, APSP, and Gaussian Elimination into the Gaussian Elimination Paradigm (GEP) and discussed a unified framework to analyze complexity, parallelism and actual performance. We show how the parallel depth and the asymmetric cache complexity can be improved using the algorithms we just introduced in Section 7.5 and 7.6.

We discuss the details of these cache-oblivious algorithms later in this section. The common theme in these algorithms is that, the computation takes one or two square matrix(ces) of size $n \times n$ as input, applies n^3 operations, and generates output as a square matrix of size $n \times n$. Each output entry is computed by an inner product of one column and one row of either the input matrices or the output matrix in some intermediate state. Namely, the output $A_{i,j}$ requires input $B_{i,k}$ and $C_{k,j}$ for $1 \leq k \leq n$ (A, B and C may or may not be the same matrix). Therefore, we can apply the results of 3d grids on these problems.[5] Note that some of the grids are full (e.g., Kleene's algorithm) while others are not, but they are all α-full and contain $O(n^3)$ operations.

The data dependencies in these algorithms are quite different from each other, but the recursions for cache complexity $Q(n)$ and depth $D(n)$ for APSP, Gaussian elimination and triangular system solver are all in the following form:

$$Q(n) = \beta\, Q(n/2) + \gamma\, Q_{3C}(n/2)$$
$$D(n) = 2\, D(n/2) + \delta\, D_{3C}(n/2)$$

where $Q_{3C}(n)$ and $D_{3C}(n)$ are the cache complexity and depth of a 3d grid of size n. Here, as long as the the recursive subtask fits into the cache together with the 3d grid computation and the β, γ and δ are constants and satisfy $\beta < 8$, we can show the following bounds.

Theorem 7.7.2. *Kleene's algorithm for APSP, Gaussian elimination and triangular system solver of size n can be computed in $\Theta(n^3)$ work, symmetric and asymmetric cache complexity of $O\left(\frac{n^3}{B\sqrt{M}}\right)$ and $O\left(\frac{\omega^{1/3}n^3}{B\sqrt{M}}\right)$ respectively, and $O(n)$ depth.*

7.7.3 All-Pair Shortest-Paths (APSP)

An all-pair shortest-paths (APSP) problem takes a (usually directed) graph $G = (V, E)$ (with no negative cycles) as input. Here we discuss the Kleene's algorithm (first mentioned in [125, 132, 190, 219], discussed in full details in [10]). Kleene's algorithm has the same computational DAG as Floyd-Washall algorithm [126, 275], but it is described in a divide-and-conquer approach, which is already I/O-efficient, cache-oblivious and highly parallelized.

[5]For Gaussian Elimination $A_{k,k}$ is also required, but $A_{k,k}$ is only on the diagonal, which requires a lower-order of cache complexity to load when computing a sub-cubic of a 3d grid.

Algorithm 12: KLEENE(A)

Input: Distance matrix A initialized based on the input graph $G = (V, E)$
Output: Computed Distance matrix A

1 $A_{00} \leftarrow$ KLEENE(A_{00})
2 $A_{01} \leftarrow A_{01} + A_{00}A_{01}$
3 $A_{10} \leftarrow A_{10} + A_{10}A_{00}$
4 $A_{11} \leftarrow A_{11} + A_{10}A_{01}$
5 $A_{11} \leftarrow$ KLEENE(A_{11})
6 $A_{01} \leftarrow A_{01} + A_{01}A_{11}$
7 $A_{10} \leftarrow A_{10} + A_{11}A_{10}$
8 $A_{00} \leftarrow A_{00} + A_{10}A_{01}$
9 **return** A

The pseudocode of Kleene's algorithm is provided in Algorithm 12. The matrix A is partitioned into 4 submatrices indexed as $\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$. The matrix multiplication is defined in a closed semi-ring with $(+, \min)$.

Kleene's algorithm is a divide-and-conquer algorithm to compute APSP. Its high-level idea is to first compute the APSP between the first half of the vertices only using the paths between these vertices. Then by applying some matrix multiplication we update the shortest-paths between the second half of the vertices using the computed distances from the first half. We then apply another recursive subtask on the second half vertices. The computed distances are finalized, and we use them to again update the shortest-paths from the first-half vertices.

The asymmetric cache complexity $Q(n)$ of this algorithm follows the recursion of:

$$Q(n) = 2Q(n/2) + 6Q_{3C}(n/2)$$
$$D(n) = 2D(n/2) + 2D_{3C}(n/2)$$

Considering this cost, the recursion is root-dominated, which indicates that computing all-pair shortest paths of a graph has the same upper bound on cache complexity as matrix multiplication.

7.7.4 Gaussian Elimination

Gaussian elimination (without pivoting) is used in solving of systems of linear equations and computing LU decomposition of symmetric positive-definite or diagonally dominant real matrices. Given a linear system $AX = b$, the algorithm proceeds in two phases. The first phase modifies A into an upper triangular matrix (updates B accordingly),

which is discussed in this section. The second phase solves the values of the variables using back substitution, which is shown in Section 7.7.5.

The process of Gaussian elimination can be viewed as a three nested-loops and computing the value of $A_{i,j}$ requires $A_{i,k}$, $A_{k,j}$ and $A_{k,k}$ for all $1 \leq k < i$. If required, the corresponding value of the LU decomposition matrix can be computed simultaneously. The underlying idea of divide-and-conquer approach is almost identical to Kleene's algorithm, which partitions A into four quadrants $\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$. The algorithm: (1) recursively computes A_{00}; (2) updates A_{10} and A_{01} using A_{00}; (3) updates A_{11} using A_{10} and A_{01}; and (4) recursively computes A_{11}.

Note that each inter-quadrant update in step (2) and (3) is a 3d grid, which gives the following recurrence:

$$Q(n) = 2Q(n/2) + 4Q_{3C}(n/2)$$
$$D(n) = 2D(n/2) + 3D_{3C}(n/2)$$

7.7.5 Triangular System Solver

A Triangular System Solver computes the back substitution step in solving the linear system. Here we assume that it takes as input a lower triangular $n \times n$ matrix T (can be computed using the algorithm discussed in Section 7.7.4) and a square matrix B and outputs a square matrix X such that $TX = B$. A triangular system can be recursively decomposed as:

$$\begin{aligned} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} &= \begin{bmatrix} T_{00} & 0 \\ T_{10} & T_{11} \end{bmatrix} \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} \\ &= \begin{bmatrix} T_{00}X_{00} & T_{00}X_{01} \\ T_{10}X_{00} + T_{11}X_{10} & T_{10}X_{01} + T_{11}X_{11} \end{bmatrix} \end{aligned}$$

such that four equally sized subquadrants X_{00}, X_{01}, X_{10}, and X_{11} can be solved recursively. In terms of parallelism, the two subtasks of X_{00} and X_{01} are independent, and need to be solved prior to the other independent subtasks X_{10}, and X_{11}.

The asymmetric cache complexity $Q(n)$ of this algorithm follows the recursion of:

$$Q(n) = 4Q(n/2) + 2Q_{3C}(n/2)$$
$$D(n) = 2D(n/2) + D_{3C}(n/2)$$

7.7.6 Strassen Algorithm

Strassen algorithm computes matrix multiplication on a ring. Given two input matrices A and B and the output matrix $C = AB$, the algorithm partitions A, B and C into quadrants,

applies seven recursive matrix multiplications on the sums or the differences of the quadrants, and each quadrant of C can be calculated by summing a subset of the seven intermediate matrices. This can be done in $O(n^{\log_2 7})$ work, $O(n^{\log_2 7}/M^{\log_4 7-1}B)$ cache complexity and $O(\log^2 n)$ depth.

Technically the computation structure of Strassen is not a k-d grid, but we can apply a similar idea in Section 7.5 to reduce asymmetric cache complexity. We still use r as the balancing factor between reads and writes (set to be $\omega^{2/3}$ in classic matrix multiplication). Given square input matrices, the algorithm also partition the output into r-by-r submatrices, and then run the 8-way divide-and-conquer approach to compute the matrix multiplication. This gives the following recurrences on work (T), reads, writes and depth based on the output size n:

$$\begin{aligned} T'(n) &= 7T'(n/2) + O(n^2) \\ R'(n) &= 7R'(n/2) + O(n^2/B) \\ W'(n) &= 7W'(n/2) + O(\omega n^2/B) \\ D'(n) &= D(n/2) + O(\log n) \end{aligned}$$

with the base cases $T'(1) = r$, $R'(\sqrt{rM}) = M/B$, $W'(\sqrt{M}) = \omega M/B$, and $D'(1) = 1$. They solve to

$$\begin{aligned} R(n) = r^2 R'(n/r) &= O(n^{\log_2 7} r^{2-\log_4 7} M^{1-\log_4 7}/B) \\ W(n) = r^2 W'(n/r) &= O(\omega n^{\log_2 7} r^{2-\log_2 7} M^{1-\log_4 7}/B) \\ D(n) &= O(\log^2 n) \end{aligned}$$

The case when $r = \omega^{\log_7 4}$ gives the minimized cache complexity of

$$Q(n) = O(n^{\log_2 7}\omega^{\log_7 16-1}M^{1-\log_4 7}/B) \approx O(n^{2.8}\omega^{0.42}/BM^{0.4})$$

an $O(\omega^{0.58})$ improvement over the non-write-efficient version. In this setting the work is $O(n^{\log_2 7}\omega^{\log_7 64-2})$, a factor of $O(\omega^{0.14})$ or $O(\omega^{1/7})$ extra work.

7.8 Dynamic Programming Recurrences

In this section we discuss a number of new results for dynamic programming (DP). To show interesting lower and upper bounds on parallelism and cache efficiency in either symmetric and asymmetric setting, we focus on the specific DP recurrences instead of the problems. We assume the operations in the recurrences are atomic, and not decomposable or batchable.

The goal of this section is to show how the DP recurrences can be viewed as and decomposed into the k-d grids. Then the lower and upper bounds discussed in Section 7.4 and 7.5, as well as the analysis of parallelism in Section 7.6, can be easily applied to

the computation of these DP recurrences. When the dimension of the input/output is the same as the number of entries in each grid cell, then the sequential and symmetric versions of the algorithms in this section are the same as the existing ones discussed in [92, 94, 96, 131, 270], but the others are new. Also, the asymmetric versions and most parallel versions are new. We improve the existing results on symmetric/asymmetric cache complexity, as well as parallel depth.

Symmetric cache complexity. We show improved algorithms on a number of problems when the number of entries per cell differs from the dimension of input/output arrays, including the GAP recurrence, protein accordion folding, and RNA recurrence. We show that the previous cache bound $O(n^3/B\sqrt{M})$ for the GAP recurrence and protein accordion folding is not optimal, and we improve the bounds in Theorem 7.8.2 and 7.8.3. For RNA recurrence, we show an optimal cache complexity of $\Theta(n^4/BM)$ in Theorem 7.8.2, which improves the best existing result by $O(M^{3/4})$.

Asymmetric cache complexity. By replacing the computation of the k-d grid using the asymmetric version discussed in Section 7.5, we show a uniform approach to provide write-efficient algorithms for all problems in this section. We also prove the optimality of all these algorithms expect for the GAP recurrence.

Parallelism. The parallelism of these algorithms is provided by using the parallel algorithms discussed in Section 7.6 as the underlying building blocks for computing the DP recurrences. We can achieve polylogarithmic depth in computing the 2-knapsack recurrence, and linear depth in LWS recurrence and protein accordion folding. The linear depth for LWS can be achieved by previous work [115, 264], but they are not in the nested parallel model and does not have the guarantee by the randomized work-stealing scheduler. Meanwhile, our algorithms are arguably simpler.

7.8.1 LWS Recurrence

The LWS (least-weighted subsequence) recurrence [169] is one of the most commonly-used DP recurrences in practice. Given a real-valued function $w(i, j)$ for integers $0 \leq i < j \leq n$ and D_0, for $1 \leq j \leq n$,

$$D_j = \min_{0 \leq i < j} \{D_i + w(i, j)\}$$

This recurrence is widely used as a textbook algorithm to compute optimal 1D clustering [192], line breaking [193], longest increasing sequence, minimum height B-tree, and many other practical algorithms in molecular biology and geology [135, 136], computational geometry problems [6], and more applications in [194]. Here we assume that $w(i, j)$ can be computed in constant work based on a constant size of input associated to i and j, which is true for all these applications. Although different special properties of the weight function w can lead to specific optimizations, the study of recurrence itself is interesting, especially regarding cache efficiency and parallelism.

We note that the computation structure (without considering dependencies) of this recurrence is a standard 2d grid. Each cell requires the input entry of D_i, computes $D_i + w(i, j)$ and updates D_j as the output entry, so Theorem 7.4.1 and 7.4.2 show lower bounds on cache complexity on this recurrence (the grid is (1/2)-full).

We now introduce cache-oblivious implementation considering the data dependencies. Recent work by Chowdhury and Ramachandran [94] solves the recurrence with $O(n^2)$ work and $O(n^2/BM)$ symmetric cache complexity. The algorithm is simply a divide-and-conquer approach and we describe and extend it based on k-d grids. A task of range (p, q) computes the cells (i, j) such that $p \leq i < j \leq q$. To compute it, the algorithm generates two equal-size subtasks (p, r) and $(r + 1, q)$ where $r = (p + q)/2$, solves the first subtask (p, r) recursively, then computes the cells corresponding to $w(i, j)$ for $p \leq i \leq r < j \leq q$, and lastly solves the subtask $(r + 1, q)$ recursively. Note that the middle step exactly matches a 2d grid with no dependencies between the cells, which can be directly solved using the algorithms in Section 7.5. This leads to the cache complexity and depth to be:

$$Q(n) = 2Q(n/2) + Q_{2C}(n/2)$$

$$D(n) = 2D(n/2) + D_{2C}(n/2)$$

Here $2C$ denotes the computation of a 2d grid. The recurrence is root-dominated and $D(1) = 1$. This solves to the following theorem.

Theorem 7.8.1. *The LWS recurrence can be computed in $\Theta(n^2)$ work, $\Theta\left(\frac{n^2}{BM}\right)$ and $\Theta\left(\frac{\omega^{1/2}n^2}{BM}\right)$ optimal symmetric and asymmetric cache complexity respectively, and $O(n)$ depth.*

7.8.2 GAP Recurrence

The GAP problem [134, 136] is a generalization of the edit distance problem that has many applications in molecular biology, geology, and speech recognition. Given a source string X and a target string Y, we can apply a sequence of consecutive deletes corresponds to a gap in X, and a sequence of consecutive inserts corresponds to a gap in Y. For simplicity here we assume both strings with length n, but the algorithms and analyses can easily adapt to the more general case. Since the cost of such a gap is not necessarily equal to the sum of the costs of each individual deletion (or insertion) in that gap, we define $w(p, q)$ $(0 \leq p < q \leq n)$ to be the cost of deleting the substring of X from $(p + 1)$-th to q-th character, and $w'(p, q)$ for inserting the substring of Y accordingly.

Let $D_{i,j}$ be the minimum cost for such transformation from the prefix of X with i characters to the prefix of Y with j characters, the recurrence for $i, j > 0$ is:

$$D_{i,j} = \begin{cases} \min_{0 \leq q < j}\{D_{i,q} + w(q, j)\} \\ \min_{0 \leq p < i}\{D_{p,j} + w(p, i)\} \end{cases}$$

corresponding to either inserting or deleting a substring. The boundary is set to be $D_{0,0} = 0$, $D_{0,j} = w(0, j)$ and $D_{i,0} = w'(0, i)$. The diagonal dependency from $D_{i-1,j-1}$ can be added if required, without affecting the asymptotic analysis.

The best existing algorithms on GAP Recurrence [94, 263] have symmetric cache complexity of $O(n^3/B\sqrt{M})$. This bound seems to be reasonable, since in order to compute $D_{i,j}$, we need the input of two vectors $D_{i,q}$ and $D_{p,j}$, which is similar to matrix multiplication and other algorithms in Section 7.7. However, as indicated in this thesis, each update in GAP only requires one entry, while matrix multiplication has two. Therefore, if we ignore the data dependencies, the first line of the GAP recurrence can be viewed as n LWS recurrences, independent of the dimension of i (similarly for the second line). This derives a lower bound on cache complexity to be that of an LWS recurrence multiplied by $2n$, which is $\Omega(n^3/BM)$ (assuming $n > M$). Hence, the gap is $\Theta(\sqrt{M})$ between the lower and upper bounds.

We now discuss an I/O-efficient algorithm to reduce this gap. Unfortunately, the algorithm is not optimal, and we leave it as an open problem. Chowdhury and Ramachandran's approach [94] is based on divide-and-conquer to compute output D. The algorithm recursively partitions D into four equal-size quadrants D_{00}, D_{01}, D_{10} and D_{11}, and starts to compute D_{00} recursively. After this is done, it uses the computed value in D_{00} to update D_{01} and D_{10} using the recurrence. This step can be considered to compute $2 \times n'/2$ LWS recurrences (with no data dependencies) each with size $n'/2$ (assuming D has size $n' \times n'$). Then the algorithm computes D_{01} and D_{10} within their own ranges. After that, it updates D_{11} using the results from D_{01} and D_{10}, and solves D_{11} recursively at the end.

Our modified version reorganizes the data layout and the order of computation to take advantage of our I/O-efficient and parallel algorithm on 2d grids. Since the GAP recurrence has two independent sections in different directions, we keep two copies of D, one organized in column major and the other in row major. Then when computing on the inter-quadrant updates (e.g., from D_{00} to D_{01}) we start $n'/2$ parallel tasks, each to compute a 2d grid on the corresponding row or column, taking the input and output with the correct representation. This update takes the work and cache complexity shown in Theorem 7.8.1. We also need to keep the consistency of the two copies. After the update of a quadrant D_{01} or D_{10} is finished, we apply a matrix transpose [59] to update the other copy of this quadrant, and the cost of the transpose is a lower-order term. For the quadrant D_{11}, we wait until the two updates from D_{01} and D_{10} finish, and then apply the matrix transpose to update the values in each other. It is easy to check that by induction, the values in both copies in a quadrant are update-to-date after each recursion.

The updated algorithm still requires $\Theta(n^3)$ work since it does not require extra asymptotic work. The cache complexity and depth satisfy:

$$Q(n) = 4Q(n/2) + 4n \cdot Q_{2C}(n/2)$$

$$D(n) = 3D(n/2) + 2D_{2C}(n/2)$$

The base cases are $Q(\sqrt{M}) = O(M/B)$ and $Q_{2C}(m) = O(m/B)$ for $m \leq M$ for the symmetric setting. Hence, we have $Q(M) = O((M \log_2 \sqrt{M})/B)$. The top-level computation is root dominated. Therefore, if $n > M$, $Q(n) = O(n^2 Q(M)/M) + O(1) \cdot Q_{2C}(n) = O(n^2/B \cdot (n/M + \log_2 \sqrt{M}))$. Otherwise, $Q(n) = O(n^2 \log_2(n/\sqrt{M})/B)$. Similarly we can compute the asymmetric results.

Theorem 7.8.2. *The GAP recurrence can be computed in $\Theta(n^3)$ work, $O(n^{\log_2 3})$ depth, symmetric cache complexity of*

$$O\left(\frac{n^2}{B} \cdot \left(\frac{n}{M} + \log_2 \min\left\{\frac{n}{\sqrt{M}}, \sqrt{M}\right\}\right)\right)$$

and asymmetric cache complexity of

$$O\left(\frac{n^2}{B} \cdot \left(\frac{\omega^{1/2} n}{M} + \omega \log_2 \min\left\{\frac{n}{\sqrt{M}}, \sqrt{M}\right\}\right)\right)$$

Compared to the previous result [92, 94, 96, 181, 263, 270], the improvement on the symmetric cache complexity is asymptotically $O(\sqrt{M})$ (i.e., n approaching infinity). For smaller range of n that $O(\sqrt{M}) \leq n \leq O(M)$, the improvement is $O(n/\sqrt{M}/\log(n/\sqrt{M}))$. (The computation fully fit into the cache when $n < O(\sqrt{M})$.)

Protein accordion folding. The recurrence for protein accordion folding [270] is $D_{i,j} = \max_{1 \leq k < j-1}\{D_{j-1,k} + w(i,j,k)\}$ for $1 \leq j < i \leq n$, with $O(n^2/B)$ cost to precompute $w(i,j,k)$. Although there are some minor differences, from the perspective of the computation structure, the recurrence can basically be viewed as only containing the first section of the GAP recurrence. As a result, the same lower bounds of GAP can also apply to this recurrence.

In terms of the algorithm, we can compute n 2d grids with the increasing order of j from 1 to n, such that the input are $D_{j-1,k}$ for $1 \leq k < j-1$ and the output are $D_{i,j}$ for $j < i \leq n$. For short, we refer to a 2d grid as a task. However, the input and output arrays are in different dimensions. To handle it, we use the similar method as the GAP algorithm that keeps two separate matrices, one in column-major and one in row-major. They are used separately to provide the input and output for the 2d grid. We apply the transpose in a divide-and-conquer manner: once the first half of the tasks finish, we transpose all computed values from the output matrix to the input matrix (which is a square matrix), and then compute the second half of the task. Both matrix transposes in the first and second halves are applied recursively with geometrically decreasing sizes. The correctness of this algorithm can be verified by checking the data dependencies so that all required values are computed and moved to the correct positions before they are used for further computations.

The cache complexity is from two subroutines: the computations of 2d grids and matrix transpose. The cost of 2d grids is simply upper bounded by n times the cost of

each task, which is $O(n^2/B \cdot (1 + n/M))$ and $O(n^2/B \cdot (\omega + \omega^{1/2}n/M))$ for symmetric and asymmetric cache complexity, and $O(n \log^2 n)$ depth. For matrix transpose, the cost can be verified in the following recursions.

$$Q(n) = 2Q(n/2) + Q_{Tr}(n/2)$$

$$D(n) = 2D(n/2) + D_{Tr}(n/2)$$

where Tr indicates the matrix transpose. The base case is $Q(\sqrt{M}) = O(M/B)$ and $D(1) = 1$. Applying the bound for matrix transpose [59] provides the following theorem.

Theorem 7.8.3. *Protein accordion folding can be computed in $O(n^3)$ work, symmetric and asymmetric cache complexity of $\Theta\left(\frac{n^2}{B}\left(1 + \frac{n}{M}\right)\right)$ and $\Theta\left(\frac{n^2}{B}\left(\omega + \frac{\omega^{1/2}n}{M}\right)\right)$ respectively, and $O(n \log^2 n)$ depth.*

The cache bounds in both symmetric and asymmetric cases are optimal with respect to the recurrence.

7.8.3 RNA Recurrence

The RNA problem [136] is a generalization of the GAP problem. In this problem a weight function $w(p, q, i, j)$ is given, which is the cost to delete the substring of X from $(p + 1)$-th to i-th character and insert the substring of Y from $(q + 1)$-th to j-th character. Similar to GAP, let $D_{i,j}$ be the minimum cost for such transformation from the prefix of X with i characters to the prefix of Y with j characters, the recurrence for $i, j > 0$ is:

$$D_{i,j} = \min_{\substack{0 \le p < i \\ 0 \le q < j}} \{D_{p,q} + w(p, q, i, j)\}$$

with the boundary values $D_{0,0}$, $D_{0,j}$ and $D_{i,0}$. This recurrence is widely used in computational biology, like to compute the secondary structure of RNA [276].

While the cache complexity of this recurrence seems to be hard to analysis in previous papers, it fits into the framework of k-d grids straightforwardly. Since each computation in the recurrence only requires one input value, the whole recurrence can be viewed as a 2d grid, with both the input and output as D. The 2d grid contains a constant fraction of the cells, so we can apply the lower bounds in Section 7.5 here.

Again for a matching upper bound, we need to consider the data dependency. We can apply the similar technique in GAP algorithm to partition the output D into four quadrants, compute D_{00}, then D_{01} and D_{10}, and finally D_{11}. Each inter-quadrant update corresponds to a 1/2-full 2d grid. Here maintaining two copies of the array is not necessary with the tall-cache assumption $M = \Omega(B^2)$. Applying the similar analysis in GAP gives the following result:

Theorem 7.8.4. *RNA recurrence can be computed in $\Theta(n^4)$ work, optimal symmetric and asymmetric cache complexity of $\Theta\left(\frac{n^4}{BM}\right)$ and $\Theta\left(\frac{\omega^{1/2}n^4}{BM}\right)$ respectively, and $O(n^{\log_2 3})$ depth.*

7.8.4 Parenthesis Recurrence

The Parenthesis recurrence solves the following problem: given a linear sequence of objects, an associative binary operation on those objects, and the cost of performing that operation on any two given (consecutive) objects (as well as all partial results), the goal is to compute the min-cost way to group the objects by applying the operations over the sequence. Let $D_{i,j}$ be the minimum cost to merge the objects indexed from $i+1$ to j (1-based), the recurrence for $0 \le i < j \le n$ is:

$$D_{i,j} = \min_{i<k<j} \{D_{i,k} + D_{k,j} + w(i,k,j)\}$$

where $w(i,k,j)$ is the cost to merge the two partial results of objects indexed from $i+1$ to k and those from $k+1$ to j. Here the cost function is only decided by a constant-size input associated to indices i, j and k. $D_{i,i+1}$ is initialized, usually as 0. The applications of this recurrence include the matrix chain product, construction of optimal binary search trees, triangulation of polygons, and many others shown in [108, 135, 136, 282].

The computation of this recurrence (without considering dependencies) is a (1/3)-full 3d grid, which has the same lower bound shown in Corollary 7.7.1.

The divide-and-conquer algorithm that computes this recurrence is usually hard to describe (e.g., it takes several pages in [92, 181] although they also describe their systems simultaneously). We claim that under the view of our k-d grids, this algorithm is conceptually as simple as the other algorithms. Again this divide-and-conquer algorithm partitions the state D into quadrants, but at this time one of them (D_{10}) is empty since $D_{i,j}$ does not make sense when $i > j$. The quadrant D_{01} depends on the other two. The algorithm first recursively computes D_{00} and D_{11}, then updates D_{01} using the computed values in D_{00} and D_{11}, and finally recursively computes D_{01}. Here D_{01} is square, so the recursive computation of D_{01} is almost identical to that in RNA or GAP recurrence (although the labeling of the quadrants is slightly changed): breaking a subtask into four quadrants, recursively solving each of them in the correct order while applying inter-quadrant updates in the middle. The only difference is when the inter-quadrant updates are processed, each update requires two values, one in D_{01} and another in D_{00} or D_{11}. This is the reason that Parenthesis is 3d while RNA and GAP are 2d. The correctness of this algorithm can be shown inductively.

Theorem 7.8.5. *The Parenthesis recurrence can be computed in $\Theta(n^3)$ work, optimal symmetric and asymmetric cache complexity of $\Theta\left(\frac{n^3}{B\sqrt{M}}\right)$ and $\Theta\left(\frac{\omega^{1/3}n^3}{B\sqrt{M}}\right)$ respectively, and $O(n^{\log_2 3})$ depth.*

7.8.5 2-Knapsack Recurrence

Given A_i and B_i for $0 \le i \le n$, the 2-knapsack recurrence computes:

$$D_i = \min_{0 \le j \le i} \{A_j + B_{i-j} + w(j, i-j, i)\}$$

for $0 \le i \le n$. The cost function $w(j, i-j, i)$ relies on constant input values related on indices i, $i-j$ and j. To the best of our knowledge, this recurrence is first discussed in this thesis. We name is the "2-knapsack recurrence" since it can be interpreted as the process of finding the optimal strategy in merging two knapsacks, given the optimal local arrangement of each knapsack stored in A and B. Although this recurrence seems trivial, the computation structure of this recurrence actually forms some more complicated DP recurrence. For example, many problems on trees[6] can be solved using dynamic programming, such that the computation essentially applies the 2-knapsack recurrence a hierarchical (bottom-up) manner.

We start by analyzing the lower bound on cache complexity of the 2-knapsack recurrence. The computational grid has two dimensions, corresponding to i and j in the recurrence. If we ignore B in the recurrence, then the recurrence is identical to LWS (with no data dependencies), so we can apply the lower bounds in Section 7.8.1 here.

Note that each update requires two input values A_j and B_{i-j}, but they are not independent. When computing a subtask that corresponding to $(i, j) \in [i_0, i_0 + n_i] \times [j_0, j_0 + n_j]$, the projection sizes on input and output arrays A, B and D are no more than n_j, $n_i + n_j$ and n_i. This indicates that the computation of this recurrence is a variant of 2d grid, so we can use the same algorithm discussed in Section 7.5.

Corollary 7.8.6. *2-knapsack recurrence can be computed using $O(n^2)$ work, optimal symmetric and asymmetric cache complexity of $\Theta\left(\frac{n^2}{BM}\right)$ and $\Theta\left(\frac{\omega^{1/2}n^2}{BM}\right)$, and $O(\log^2 n)$ depth.*

7.9 Conclusion and Future Work

In this thesis, we abstract the k-d grid as a basic block to analyze the computation structure of many classic cache-oblivious algorithms. This abstraction provides a more simple and intuitive framework on better understanding the actual computation of these algorithms, proving lower bounds, and designing algorithms that are both I/O-efficient and highly parallelized. It also provides a unified framework to bound the asymmetric cache complexity of these algorithms.

[6]Such problems can be: (1) computing a size-k independent vertex set on a tree that maximizes overall neighbor size, total vertex weights, etc.; (2) tree properties such that the number of subtrees of certain size, tree edit-distance, etc.; (3) many approximation algorithms on tree embeddings of an arbitrary metric [62, 70]; and many more.

Because of the new perspective to review these algorithms, we already provide many new results, but also observe many new open problems. Among them are:

1. The only non-optimal algorithm regarding cache complexity in this chapter is for the GAP recurrence. The I/O cost has an additional term of $O((n^2 \log M)/B)$. Although in practice this term will not dominate the running time (the computation has $O(n^3)$ arithmetic operations), it is theoretically interesting to know if we can get rid of this term (even without the constraint of cache-obliviousness or based on divide-and-conquer).
2. We show our algorithms in the asymmetric setting are optimal under the assumption of constant-branching (the CBCO paradigm). Since the cache-oblivious algorithms discussed in this chapter are leaf-dominate, we believe this assumption is always true. We wonder if this assumption is necessary (i.e., if there exists a proof without using it, or if there are cache-oblivious algorithms on these problems with non-constant branching but still I/O-optimal).
3. The parallel symmetric cache complexity Q_p on p processors is $Q_1 + O(pDM/B)$, which is a loose upper bound when D is large. Although it might be hard to improve this bound on any general computation under randomized work-stealing, it can be a good direction to show tighter bounds on more regular computation structures like the k-d grids or other divide-and-conquer algorithms. We conjecture that the additive term can be shown as optimal (i.e., $O(pM/B)$) for the k-d grid computation structures.
4. In this chapter we mainly discussed the lower bounds and algorithms for square grid computation structures, which is the setting of the problems in this chapter (e.g., APSP, dynamic programming recurrences). It is interesting to see a more general analysis on k-d grids with arbitrary shape, and such results may apply to other applications like the computation of tensor algebra.

Chapter 8

Experimental Verifications of Write-Efficient Algorithms

8.1 Overview

This section discusses how the write-efficient algorithms work in practice, more than just the theoretical analysis asymptotically.

Early works include the study of this read-write asymmetry on NAND Flash chips [41, 123, 133, 226] and algorithms targeting database operators [90, 272, 273]. However, most of the papers either treat NVMs as external memories, or are based on hardware simulators for existing architecture, which may have many concerns that we will further discuss in Section 8.2.

This thesis, in Chapter 2, formally defined and analyzed several sequential and parallel computational models with good caching and scheduling guarantees. Recall that the basic model, which is the Asymmetric RAM (ARAM), extends the well-known external-memory model [7] and parameterizes the asymmetry using ω, which corresponds to the cost of a write relative to a read to the non-volatile main memory. The cost of an algorithm on the ARAM, the **asymmetric I/O cost**, is the number of write transfers to the main memory multiplied by ω, plus the number of read transfers. This model captures different system consideration (latency, bandwidth, or energy) by simply plugging in different values of ω, and also allows algorithms to be analyzed theoretically. Based on this idea, many interesting algorithms (and lower bounds) are designed and analyzed in this thesis.

In this chapter, we will further show the exact numbers of reads and writes of the algorithms instead of just the asymptotic bounds, and our goal is to bridge the gap between theory and practice. We also try to study and understand which algorithmic techniques are useful in designing practical write-efficient algorithms. As the first trial in this direction, it seems impossible to implement and test all algorithms in the previous chapters. As a result, we focus on several of the most commonly-seen algorithmic building blocks in modern

programming: unordered set/map implemented using **hash tables**, set/map implemented using **balanced binary search trees**, **comparison sort**, and graph traversal algorithms: **breadth-first search** for unweighted graphs and **Dijkstra's algorithm** for weighted graphs.

Unfortunately, no non-volatile main memory is currently available, making it impossible to get real timings. Furthermore, details about latency and other parameters of the memory and how they will be incorporated into the architecture are also not available. This makes detailed cycle-level simulation (e.g., PTLsim [232], MARSSx86 [214] or ZSim [244]) of questionable utility. However, it is quite feasible to count the number of reads and write to main memory while simulating a variety of cache configurations. For I/O-bounded algorithms, these numbers can be used as reasonable proxies for both running time (especially when implemented in parallel) and energy consumption.[1] Moreover, conclusions drawn from these numbers can likely give insights into tradeoffs between reads and writes among different algorithms.

For these reasons, we propose a framework based on a software simulator that can efficiently and precisely measure the number of read and write transfers of an algorithm using different caching policies. We also consider variants in caching policies that might lead to improvements when read and write are not the same.

We also note that designing write-efficient algorithms falls in a high dimensional parameter space since the asymmetries on latency, bandwidth, and energy consumption between reads and writes are different. Here we abstract this as a single value ω. This value together with the cache size M and cache-line size B (set to be 64 bytes in this chapter) form the parameter space of an algorithm.

Our framework provides a simple, clean and hardware-independent method to analyze and experiment the performance on the asymmetric memory. We investigate the algorithmic techniques and learn lessons from the experiments that generally apply for a reasonably large parameter space of ω, M and B. This framework also allows monitoring, reasoning and debugging the code easily, so it can remain useful even after the new hardware is available.

With the framework, we design, implement and discuss many different algorithms and data structures and their write-efficient implementations. Although some of the implementations are standard, like quicksort and the classic hash tables, many others, including the k-level hash tables, sample sort and phased Dijkstra, require careful algorithmic design, analysis, and coding. Under our cost measure, which is the asymmetric I/O cost, we show better approaches on all problems we study in this chapter, compared to the most basic and commonly-used ones on symmetric memories. We understand that there are more advanced versions of the algorithms and data structures discussed in this

[1]The energy consumption of main memory is a key concern since it costs 25-50% energy on data centers and servers [198, 200, 206].

chapter on some specific applications, and how to implement them write-efficiently is an interesting topic for future work.

With the algorithms and their experimental results, we draw many interesting algorithmic strategies and guidance in designing write-efficient algorithms. A common theme is to trade (more) reads for (fewer) writes (apparently it is hard to directly decrease the writes since this can improve the performance on symmetric memory as well and should have been investigated already). Some interesting lessons we learned and can be valuable to share are listed as follows, which can suggest some potential directions to design and engineer write-efficient algorithms in the future.

1. Indirect addressing is less problematic. In the classic setting, indirect addressing should be avoided if possible, since each addressing can be a random access to the memory. However, when writes are expensive, moving the entire data is costly, while indirect addressing only modifies the pointers (at the cost of a possible random access per lookup).
2. Multiple candidate positions for a single entry in a data structure can help. It can be a good option to use more reads per lookup but apply less frequent data movements, when the size of a data structure changes significantly. This is a common strategy we have applied in this experiment chapter to provide an algorithmic tradeoff between reads and writes.
3. It is usually worth to investigate existing algorithms that move or modify the data less. These algorithms can be less efficient in the symmetric setting due to various reasons (e.g., more random accesses, less balanced), but the property that they use fewer writes can be useful in the asymmetric setting (like samplesort vs. quicksort, treap vs. AVL or red-black tree).
4. In-cache data structures should draw more attention[2]. Since the data structures are kept in the cache (or small symmetric memory), the algorithm requires significantly less writes to the large asymmetric memory, although may require extra reads to compensate for less information we can keep within the data structure. In this experiment chapter, we discuss Dijkstra's algorithm on shortest-paths as an example, and such idea can also be applied to computing minimum spanning tree, sorting, and many other problems.

8.2 Discussions on Previous Experiments

[2]Similar ideas appear in many existing models already, like the external-memory model or the streaming model. However, the motivations are different: these models restrict the amount of space that can be used in the computation, while in our case the data structures are used to reduce the writes to the asymmetric main memory without including too many extra reads.

There exist a rich literature to show the read-write asymmetry on the new memories [14, 32, 84, 91, 117, 118, 172, 176, 187, 196, 205, 234, 280, 281, 284, 285], and all other papers in this thesis. Regarding adapting softwares for such read-write asymmetry, some work has studied the system aspect. For example, there exist many papers on how to balance the writes across the chip to avoid uneven wear-out of locations in the context of NAND Flash chips [41, 123, 133, 226].

More closely-related papers targeting database operators: Chen et al. [90] and Viglas [272, 273] presented several interesting and write-efficient sequential algorithms for searching, hash joins and sorting. These are the early and inspirational attempts to design algorithms with fewer writes. Instead of formally proposing new computational models and analyzing the asymptotic cost, they mainly show the performance by the experiment results assuming external memories rather than main memories, or on the cycle-based simulators for existing architecture. For the latter case however, the prototypes of the new memories are still under development, and yet nobody actually knows the exact parameters of the new memories, or how they are incorporated into the actual architecture which is required for the setup of the cycle-based simulator. As far as we know, there is no available cycle-based simulator at the present time for the new memories. In the meantime, the asymmetries on latency, bandwidth, and energy consumption between reads and writes are different, and any of these constraints can be the bottleneck of an algorithm. Hence, designing algorithms on asymmetric memory are in a multiple-dimension parameter space, rather than just recording the running time from a simulator. Therefore, it is essential to develop theoretical models and tools that accounts for, and abstract this asymmetry and use them to analyze algorithms on future memory.

This thesis formally defined several sequential and parallel computational models that take asymmetric read-write costs into account. Based on the computational models, many interesting algorithms (and lower bounds) are designed and analyzed in Chapter 3, 4, 5, 6, and 7, in both sequential and parallel settings.

8.3 Our Model and Simulator

To start with, we discuss how to measure the performance of algorithms on asymmetric memories. We begin with the computational model that estimates the cost of an algorithm. This model requires the numbers of read and write transfers between the non-volatile memory and the cache, so later we introduce how the numbers of an algorithm can be simulated. Unlike the existing symmetric memories, a simple cache policy like LRU does not work on some asymmetric settings. Thus in Section 8.3.2 we briefly summarize the solutions to fix it, and then the cache simulator given in Section 8.3.3 captures this number with different cache policies.

8.3.1 The Cost Model for Asymmetric Memory

In this subsection we review the models defined in Chapter 2 that are used in this chapter.

The most commonly-used cost measure of an algorithm is the time complexity based on the RAM model (Section 2.1.1), which is the overall number of instructions and memory accesses executed in this algorithm. Nowadays, since the actual latency of an access to the main memory is at least two orders of magnitudes more expensive than a CPU instruction, the *I/O cost* based on the external-memory model [7] (Section 2.1.2) is widely used to analyze the cost of an I/O-bounded algorithm. This model assumes a *small-memory* (cache) of size $M \geq 1$, and a *large-memory* of unbounded size. Both memories are organized in blocks (cache-lines) of B words. The CPU can only access the small-memory (with no cost), and it takes unit cost to transfer a single block between the small-memory and the large-memory. This cost measure estimates the running time reasonably well for I/O-bounded algorithms, especially in multi-core parallelism. An efficient algorithm in practice should achieve optimality in both the time complexity and the I/O cost.

To account for more expensive writes on future memories, here we adopt the idea of an (M, ω)-Asymmetric RAM: similar to the external-memory model, transferring a block from large-memory to small-memory takes unit cost; on the other direction, the cost is either 0 if this block is clean and never modified, or $\omega \gg 1$ otherwise. The **asymmetric I/O cost** Q[3] of an algorithm is the overall costs for all memory transfers.

8.3.2 Cache Policies

The cache policy is defined and introduced in Section 2.1.3 and 2.3.4. Either the classic external-memory model or the new ARAM assumes that we can explicitly manipulate the cache in the algorithm. This largely simplifies the analysis, and in many cases is provably within a constant factor of a more realistic cache's performance. For example, the standard least-recent used (LRU) policy is 2-competitive against the optimal offline cache-replacement sequence.

However, the competitive ratio does not hold in the asymmetric setting. Consider a cache with $k = M/B$ cache-lines and a memory access pattern that repeatedly writes to $k - 1$ cache-lines and read from other $k - 1$ cache-lines. An ideal cache policy will keep all $k - 1$ cache-lines associated to writes, so the I/O cost of each round is $k - 1$ for $k - 1$ read misses. An LRU policy however causes a cache miss for every single memory access, leading the I/O cost of each round to $\omega(k - 1) + k - 1$. This overhead is proportional to ω, which can be significant and problematic.

The solution is affected by the architecture, depending on whether software explicitly controls a DRAM buffer or not [90, 105, 197, 233]. If so, then the cost measures on the these models are just the costs in practice, but programmers are responsible for managing

[3]Throughout the experiment chapter, we abbreviate it as the *I/O cost*, unless stated otherwise explicitly.

what to put on the small-memory and guaranteeing correctness. The other option is to leave the hardware to control the small-memory. In this case, Section 2.3.4 shows that if the small-memory is partitioned into two equal-size pools and each of them is maintained using LRU policy, the performance is 3-competitive against the optimal offline cache-replacement sequence (e.g. using 3× space and incurring no more than 3× cost).

We consider three different cache policies in the experiment. The **Classic** policy maintains the small-memory as one memory pool and uses the LRU policy for replacement. The **SplitPool** policy keeps two separate memory pools and each runs the LRU policy. The **Static** policy allows static allocation in one memory pool, and the unallocated memory space is maintained using the LRU policy.

8.3.3 The Cache Simulator

The goal of this chapter is to discuss new algorithmic approaches that minimize the asymmetric I/O cost to the main memory on a variety of fundamental data structures and algorithms. To capture the number of reads and writes to the main memory, we developed a software simulator that can adapt to different cache policies introduced in Section 8.3.2. The cache simulator is composed of an ordered map that keeps tracks of the time stamp of the last visit to each cache-line in the current cache, and an unordered map that stores the mapping from each cache-line to the corresponding location in the ordered map if this cache-line is currently in the cache. Interestingly, the implementation of this cache simulator is a natural application of the techniques discussed in this experiment chapter.

The cache simulator encapsulates a new structure Array that is used in coding algorithms in this experiment chapter. It is like a regular array that can be dynamically allocated and freed, and supports two functions: Read and Write to a specific location in this array. The Arrays are responsible for reporting the memory accesses of the algorithm to the cache simulator, and the cache simulator will update the state of the cache accordingly. Therefore, coding using the Arrays is not different from regular programming much.

The memory accesses to loop variables and temporary variables are ignored, as well as the call stack. This is because the number of such variables is small in all of the algorithms in the experiment (usually no more than 10). Meanwhile, the call stack of all algorithms in this paper has size $O(\log n)$. The overall amount of uncaptured space is orders of magnitudes smaller than the amount of fast memory in our experiments.

The cache consists of one or two memory pools, depending on different cache policies discussed in Section 8.3.2. We will explicitly indicate the cache policy used in each of our experiments. The cache simulator maintains two counters in each memory pool: the number of **read transfers**, and the number of **write transfers**. When testing each algorithm on a specific input instance, the cache is emptied at the beginning and flushed at the end. A read or write is free if the location is already in the cache; otherwise the corresponding cache-line is loaded, the counter of read transfer increments by 1, and

the least-recently-used cache-line in this pool is evicted. Also, a write will mark the dirty-bit of the cache-line to be `true`. When evicting a dirty cache-line, the counter of write transfer increments by 1. Notice that memory reads can cause write transfers, and memory writes can lead to read transfers.

When simulating the **Classic** policy (i.e., the standard one), we also verified our simulated results to ZSim (cycle-level simulator for current architecture), and the numbers always differ by no more than 10% when the parameters are set correctly.

8.4 Sets and Maps

Sets and maps are two of the most commonly-used data types in modern programming. Most programming languages either have them built in as basic types (e.g. python) or supply them as standard libraries (C++, C#, Java, Scala, Haskell, ML).

These data types and the data structures to implement them are used extensively. Not only many other data types can be built based on them, just in the content of the experiment of thesis, unordered sets and maps are used in BFS in Section 8.6.1, Dijkstra's algorithm in Section 8.6.2, and the cache simulator, while ordered sets and maps are used in Dijkstra's algorithm in Section 8.6.2 and the cache simulator.

8.4.1 Unordered Sets and Maps

Our implementation of unordered sets and maps is based on hash tables that support **lookup**, **insertion**, and **deletion**. The hash tables discussed in this section use open addressing and linear probing, since the goal of the data structure is to try to minimize the I/O cost focusing on smaller entries (accessing and reading larger entries are costly anyway so different hash-table implementations make minor differences). For simplicity, we assume no duplicate keys, and it is straightforward to handle the duplicates with minor modifications. In this setting, each operation of the hash table reads a small number of cache-lines, and an insertion or deletion will modify exactly one cache-line that contains the location of the key and will be eventually written back to the large-memory.

The challenge emerges when the set size changes dynamically. For an efficient implementation, we hope the overall size of the hash table to be neither too large nor too small. If the load factor passes 80%, linear probing's performance drastically degrades. On the other hand, we want the hash table size to be reasonably small to better utilize the small-memory (cache), since each cache-line holds more entries in this case. In practice, some implementations keep the load factor up- and lower-bounded by some constant. For example, a typical implementation keeps the occupancy of the hash table between 1/8 and 1/2, and the size doubles or shrinks by half if the number of entries exceeds this range. Such resizing reinserts p entries before at least $p/2$ insertions and deletions (where p is the set/map size). When reads and writes have approximately the same cost, the extra cost for such resizing is small compared to the query and update costs (e.g., the queries

read from lots of memory locations). In the asymmetric setting however, the reads cost much less, but the extra writes in resizing can be significant: the resizing can incur at most twice ($p/(p/2) = 2$) the writes compared to the initial insertions (3× writes in total). Hence, our goal is to discuss an alternative approach that optimizes such extra writes.

8.4.1.1 The k-level Hash Table

Instead of keeping one hash table, our main idea is to maintain a small number k of hash tables simultaneously, where k is a pre-determined parameter. In particular, the k-level hash table *HashTable* is initialized with k arrays $HashTable_{1,\cdots,k}$ with size $2^{c'+i}$ for $1 \leq i \leq k$ (or smaller in specific applications) and a constant c'. In practice we set c' to be 5.

For insertions, when the overall load factor exceeds some threshold r, we allocate a new chunk of memory with the double size of the largest current array, and the smallest hash table is discarded after all elements in it have been reinserted back. Similarly for deletions, if the occupancy of the hash tables drops below a threshold l, a small array with half of the size of the current smallest hash table is allocated, and the largest table is freed after the entries in it being reinserted. For instance, a valid k-level hash table may contain two arrays of size $2^{15} = 32768$ and $2^{16} = 65536$, when $k = 2$ and 30000 entries in the current configuration. The pseudocode of the k-level hash table is given in Algorithm 13. The occupancy range $0 < l < r < 1$ indicates when the resizing happens (a valid set of parameters can be $1/8$ and $1/2$). A classic implementation can be viewed as the special case of the k-level hash table when $k = 1$.

We now analyze the I/O cost Q of the k-level hash table. Here we assume that the size of the k-level hash table is larger than the small-memory and $1 - r < 1/B$, so that one single lookup, insertion or deletion in a single level in the hash table on average requires no more than $c < 2$ cache-line loads to find the location.

Lookup. In a k-level hash table, a lookup requires ck instead of c read transfers (c is the constant just defined) in the worst case (can quit earlier once the entry is found). The cost increases by a factor of k at most.

Insert. There are two definitions of insertions: an insertion that the key is known to be not in the set/map, or an insertion that it is unknown whether the key is in this set/map. Both cases are commonly-used. In this paper, we take the first definition and analyze the cost of this type of insertions. The second type of insertion can be viewed as a lookup first, then an insert if the lookup fails.

When inserting an element in a k-level hash table, we always try the smaller tables first. Once all tables are full, we resize it. More details can be found in Algorithm 13.

The I/O cost Q of an insertion comes in two parts: the cost of the initial insertion to the hash table, and the cost of this entry in future hash-table resizings. The cost of the initial insertion is no more than $c + \omega$, where c is the number of cache-line reads to find

Algorithm 13: The k-level hash table

Input: Parameter k, occupancy range l and r

```
1  Function LOOKUP(x)
2      for i ← 1 to k do
3          p ← HashTable_i.LOOKUP(x)
4          if p ≠ null then return (i, p)
5      return null
6  Function INSERT(x) // x is not in HashTable
7      for i ← 1 to k do
8          if HashTable_i.occupancy < r then
9              HashTable_i.INSERT(x)
10             return
11     Allocate HashTable_{k+1} of size 2 · HashTable_k.size
12     Relabel the hash tables with indices from 0 to k
13     foreach y ∈ HashTable_0 do
14         INSERT(y)
15     Free HashTable_0
16 Function DELETE(x; i, p) // x is located p-th in HashTable_i
17     HashTable_i.DELETE(x, p)
18     if Overall occupancy is less than l (and HashTable_1.size > 1) then
19         Allocate HashTable_0 of size HashTable_1.size/2
20         Relabel the hash tables with indices between 1 to k + 1
21         foreach y ∈ HashTable_{k+1} do
22             INSERT(y)
23         Free HashTable_{k+1}
```

the position to insert, plus ω, one cache-line write for the actual insertion. The cost of resizing is more complicated to analyze.

We note that although a specific entry can be reinserted multiple times during different resizing processes, the overall number of element reinsertion is bounded, and thus we can a amortize the work. A resizing occurs when an insertion comes in and the hash table contains exactly $r \cdot 2^p(2^k - 1)$ elements for some positive integer p. In this case, at most $r \cdot 2^p$ entries (the size of the smallest hash table), are reinserted during the resizing. The total number of insertions from the last resizing is at least $r \cdot 2^{p-1}(2^k - 1)$ (assuming $4l \le r$), so the amortized I/O cost Q of reinsertion for each insertion is upper bounded by $\dfrac{(c+\omega)r \cdot 2^p}{r \cdot 2^{p-1}(2^k-1)} = (c+\omega) \cdot 2/(2^k - 1)$.

In the asymmetric setting when $\omega \gg 1$, the I/O cost of each insertion is approximately $\omega \cdot (1 + 2/(2^k - 1))$, indicating that compared to the classic implementation where $k = 1$, in the worst-case the improvement when $k = 2, 3, 4$ is about 44%, 57% and 62% respectively. The asymptotic improvement when $k \to +\infty$ is 67%.

Delete. A deletion in the k-level hash table is similar to an insertion except that a lookup for the location is required (details in Algorithm 13). The cost of the initial deletion is $ck + \omega$. A resizing of the hash table can occur after at least $l \cdot 2^p(2^k - 1)$ deletions for some positive integer p, and the current hash table keeps $l \cdot 2^p(2^k - 1)$ entries. However, it is

possible that all of these entries are in the last hash table so they are all reinserted. We note that when reinserting the elements from the discarded array, we always try smaller arrays first. This means that a reinserted entry, if not being deleted in the future, will not be reinserted again in the next $\min(k-1, \log_2 r/2l)$ shrinking resizings. Namely, the amortized extra cost of a deletion in future resizings is about ω/k if l is set to be about $2^{-k}r$. The overall I/O cost for a deletion is $Q = ck + \omega(1 + 1/k)$.

We have bounded of the I/O cost of each lookup, insertion or deletion, and the overall cost Q can be estimated by summing the amount of each operation multiplied by the cost of this operation. In practice, insertions and deletions can interleave. For example, when a deletion comes after an insertion, the number of entries remains the same, which leads to no further cost for these two updates afterward. The exact cost is also affected by the pattern of the sequence of the operations, and we will show by experiments.

8.4.1.2 Experiments

In the experiment we test the performance of our k-level hash table, including the numbers of read transfers and write transfers, I/O costs, and wall-clock running time, on different patterns of queries. In all experiments, we insert 1 million elements to an empty hash table, each of which is a 4-byte integer, into the hash table, and we vary the number of queries. The simulated cache contains 10,000 cache-lines and uses the *classic* policy, and for wall-clock running time we run the code on a PC with Intel i7-2600 CPU and 8GB RAM. The occupancy rate is set to be $l = 0.2$ and $r = 0.8$. We have tried other parameters (r between 0.6 and 0.8 and $l = r/4$). The results slightly vary, but all general conclusions in this section still hold.

Non-deletion cases.

Many applications, like webpage caching or the breadth-first searches, only insert but never delete elements in a hash table. Our experiment starts with this simpler case. We first show the relationship between k (the number of hash tables) and the numbers of read transfers and write transfers for a variety of insertion/query ratios, and the results are shown in Table 8.1. We fix the number of insertions to be one million, and query α times after each insertion. We vary α from 0, 1/8, to 8 ($\alpha < 1$ indicates one query per $1/\alpha$ insertions). About 50% query keys are in the hash table (this ratio affects the I/O cost since a successful query can terminate earlier). The number of levels k varies from 1 to 4. In Table 8.2, we show the overall I/O costs, which are the weighted sums assuming two typical values of the write-read ratio ω, 10 and 100.

We first look at the number of write transfers. When there is no query (i.e., the first column, just inserting 1 million entries), the numbers of writes are consistent with our analysis for insertions in Section 8.4.1.1. The only exception here is that cache can hold a constant fraction of the elements, which batches the writes and reduces the number of memory transfers. However, the relative trend in each column remains unchanged. Namely, the number of writes always decreases as the increase of k regardless of the

10^6 insertions, $\alpha \times 10^6$ queries where α is from 0 to 8, cache size is 10,000 cache-lines.

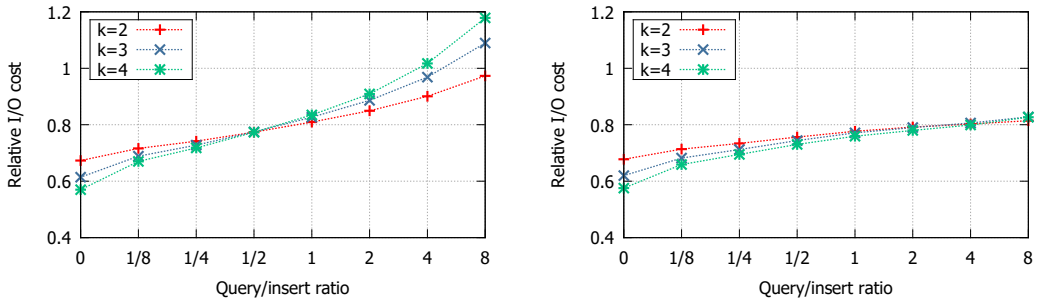
α	0		1/8		1/4		1/2		1		2		4		8	
	RT	**WT**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**
k=1	1.35	1.17	1.44	1.18	1.52	1.19	1.69	1.21	2.02	1.24	2.68	1.27	4.00	1.31	6.64	1.34
k=2	0.85	0.79	1.06	0.84	1.23	0.87	1.54	0.91	2.09	0.96	3.11	1.00	5.07	1.03	8.94	1.05
k=3	0.76	0.72	1.08	0.80	1.32	0.85	1.73	0.90	2.44	0.95	3.76	0.99	6.31	1.02	11.32	1.05
k=4	0.70	0.67	1.11	0.78	1.40	0.82	1.89	0.88	2.74	0.93	4.30	0.97	7.33	1.00	13.31	1.03

Table 8.1: Numbers of read and write transfers of k-level hash tables with different query/insert ratios. Numbers of read and write transfers are divided by 1M.

The I/O costs of k-level hash tables with the same settings in Table 8.1.

	ω = **10**								ω = **100**							
α	**0**	**1/8**	**1/4**	**1/2**	**1**	**2**	**4**	**8**	**0**	**1/8**	**1/4**	**1/2**	**1**	**2**	**4**	**8**
k=1	13.0	13.2	13.4	13.8	14.4	15.4	17.1	20.0	117.9	119.3	120.5	122.7	125.8	129.9	134.8	140.5
k=2	8.8	9.5	10.0	10.7	11.7	13.1	15.4	19.5	79.9	85.1	88.4	92.8	97.7	102.9	108.2	114.4
k=3	8.0	9.1	9.8	10.7	11.9	13.7	16.5	21.8	73.1	81.4	85.8	91.3	97.0	102.7	108.7	116.3
k=4	7.4	8.9	9.6	10.7	12.0	14.0	17.4	23.6	67.9	78.6	83.8	89.6	95.5	101.3	107.7	116.1

Table 8.2: The I/O costs of k-level hash tables with different query/insert ratios. The write-read ratio ω are selected to be typical projected values 10 (latency, bandwidth) and 100 (energy). Results are based on the numbers in Table 8.1. Numbers in red with underlines indicate the best choice of k that minimizes the I/O cost in this setting, and numbers in blue indicate better I/O costs comparing to the classic hash table implementation (i.e., $k = 1$).



(a) $\omega = 10$ (b) $\omega = 100$

Figure 8.1: Relative I/O cost of k-level hash table with different k. The I/O cost is divided by the $k = 1$ case, so every data point below 1 indicates an improvement in such case. Numbers are from Table 8.2.

ratio between queries and updates. The number of writes is reduced by 33%, 40% and 43% when $k = 2, 3, 4$ respectively. Such improvement also shows up in the overall I/O cost in Table 8.2.

We note that more queries cause more reads, and larger k also leads to more reads. Since these reads flush the cache-lines, the numbers of writes in these cases also marginally increase. The optimal choice of k is decided by the update/query distribution as well as the write-read ratio ω. In general, more queries lead to worse performance with larger k, and larger ω prefers larger k. In Table 8.2, we underline the numbers indicating the best choice of k in that specific setting. The experiment results indicate that picking k to be 2 or 3 is always a good choice when $\omega = 10$, and 3 or 4 when $\omega = 100$.

Wall-clock running time.

We also measure the actual running time of the previously stated operations on a real machine. In the current platform with symmetric memory-access costs, the write-read ratio is close to 1. Here we show that, the weighted sums of the I/O measures in Table 8.1 almost match the actual running times shown in Table 8.3, when plugging $\omega = 1$. Therefore, it is reasonable to believe that, the I/O cost shown in Table 8.2 can reasonably well project the performance for the future memory, when the bandwidths for reads and writes become asymmetric. (The argument for energy consumptions holds independently with this running time and other architectural issues.) Meanwhile, it is interesting to point out that, when insertions are more than queries, using k-level hash table with $k = 2$ is actually faster even in the current platform.

Our implementation has a much lower I/O cost compared to separate chaining. We run the same experiment using the STL unordered set (with the same hash function and other setups), and our hash table is at least 3-4 times faster in all cases.

Insertion and deletion cases.

We also tested the performance of k-level hash table on deletions. We first insert 1 million elements and then remove them all. After each insertion or deletion, we query α times. The other settings are the same as the non-deletion case. The results on the numbers of read transfer and write transfer are shown in Table 8.4, and the overall I/O cost with write-read ratio ω to be 10 and 100 are shown in Table 8.5.

From the results, we get almost the same trend as the non-deletion cases. Compared to the classic implementation (i.e., $k = 1$), the overall number of write transfer is reduced by 25%, 32%, and 36% when no queries are involved, and the improvement is slightly decreased when more queries come in, since more read accesses flush out the cache-lines. We note that the number of read transfers required by a deletion is more than that for an insertion, since in each deletion we need to locate the element in the hash table, which requires to look up in most k hash table levels. Hence, compared to the non-deletion cases, slightly smaller values for k are more preferable. The best choice of k in each case is underlined and shown in Table 8.5. For smaller $\omega = 10$ (indicating bandwidth and latency), the best choice of k varies based on different query/update ratios, but $k = 2$ is always an acceptable choice. When considering the energy consumption ($\omega \approx 100$), a larger k, like 3 or 4, is more desirable in all cases.

Wall-clock running time (milliseconds), 10^6 insertions, $\alpha \times 10^6$ queries where α is from 0 to 8.

k	0	1/8	1/4	1/2	1	2	4	8
1	61	65	69	77	95	129	191	321
2	36	45	53	69	96	163	277	512
3	31	45	57	80	124	213	392	739
4	30	51	66	95	156	278	519	999

Table 8.3: Wall-clock running time (milliseconds) of k-level hash table with different combinations of k and query/insert ratios. Numbers in blue indicate a better performance comparing to the classic implementation (i.e., $k = 1$).

1 million insertions then 1 million deletions, α times 2 million queries where α is from 0 to 4, 10,000 cache-lines.

k	0		1/4		1		4	
	RT	WT	RT	WT	RT	WT	RT	WT
1	2.55	2.32	2.95	2.36	4.14	2.44	8.91	2.52
2	2.28	1.75	3.15	1.88	5.45	2.04	14.14	2.18
3	2.47	1.59	3.79	1.80	7.11	1.98	19.61	2.12
4	2.72	1.50	4.44	1.75	8.68	1.95	24.67	2.08

Table 8.4: Numbers of read and write transfers of k-level hash tables with different query/(insert+delete) ratios. Numbers of read and write transfers are devided by 1M.

The I/O costs of k-level hash tables with the same settings in Table 8.4.

k	$\omega = 10$				$\omega = 100$			
	0	1/4	1	4	0	1/4	1	4
1	25.8	26.6	28.5	34.2	235.0	239.4	247.6	261.4
2	19.7	22.0	25.9	35.9	176.8	191.6	209.6	232.0
3	18.4	21.8	26.9	40.8	161.4	183.6	205.1	231.4
4	17.7	22.0	28.2	45.5	152.3	179.6	203.5	233.1

Table 8.5: The I/O costs of k-level hash tables with different query vs. insert/delete ratios. The write-read ratio ω are 10 and 100. Results are based on the numbers in Table 8.4. Numbers in red with underlines indicate the best choice of k that minimizes the I/O cost, and numbers in blue indicate better I/O costs comparing to the classic hash table implementation (i.e., $k = 1$).

8.4.1.3 Conclusions

We proposed a new data structure, the k-level hash table, to implement unordered set and map, that has the same space utilization compared to the classic open-addressing hash tables. The key idea in the k-level hash table is to keep multiple instead of one level

of hash tables. As a result, the algorithm uses fewer writes during resizings, at the cost of more reads in other operations.

The best choice of k is decided by the ratio of updates and queries. Our experiment shows that $k = 2$ always leads to a lower or similar I/O cost when the query/insert ratio is no more than 8, compared to the classic $k = 1$ setting. For the ratio of write/read cost is larger (like 100), larger values of k, like 3 or 4, are even more preferable than the $k = 2$ case.

8.4.2 Ordered Sets and Maps

The implementations of ordered sets and maps are based on some form of balanced tree (or tree-like) data structure and, at minimum, support **lookup**, **insertion**, and **deletion** in logarithmic time. Some set-set functions such as **union**, **intersection**, and **difference** are also required in many scenarios.

One commonly-used data structure to maintain ordered sets and maps is the self-balancing binary search tree (BST). Most languages and libraries use either AVL tree [5] or red-black tree [39, 156] to implement them, while some other implementations of weight-balanced trees [222] and treaps [24, 249] also exist. Here we first analyze the I/O cost on some existing solutions.

8.4.2.1 I/O cost on BSTs

For simplicity, here we assume that the small-memory size is $M = O(1)$. Locating the key for a lookup, insertion or deletion requires to load and compare to $\Theta(\log n)$ tree nodes on the balanced binary search trees (BSTs), The I/O cost is $\Theta(\log n)$, which is also the lower bound of such operations.

For an insertion or deletion, we also need to modify the tree accordingly. In the asymmetric setting, weight-balanced trees [222] are not a good option since we have to update the subtree sizes all the way to the root. This update leads to $\Theta(\log n)$ writes to the large-memory per update. For the other types of BSTs, we show their I/O costs on insertions and deletions individually.

Red-black trees. Red-black trees [39, 156] have the simplest update rules among these balanced BSTs. With the classic rebalancing rules and careful implementation, it requires only $O(1)$ amortized time per update (insertion or deletion) after locating the key [267]. Also, red-black trees require no extra cost to update balancing information except for the tree nodes involved in rotations (unlike the case in AVL trees). As a result, red-black trees have an optimal amortized I/O cost $Q = \Theta(\log n)$ per a lookup and $Q = \Theta(\omega + \log n)$ per insertion or deletion on the (M, ω)-ARAM.

AVL trees. An insertion in AVL trees requires at most two rotations (a double rotation) [5]. Unlike the red-black trees however, we need to track and update the balance factors along the path from the root to the modified tree node. We now bound the number of updated

balance factors to be a constant. If we store the height of the subtree in each tree node, the difference of the two subtree heights can be checked in constant time. Since the number of subtrees of height d in an AVL tree is no more than $n/c^{\lfloor d/2 \rfloor}$ for a tree with n nodes and some constant $c > 1$ [65], the number of increments of the counts for n nodes is $\sum_{d \geq 1} d \cdot (n/c^{\lfloor d/2 \rfloor}) = O(n)$. On average, an insertion needs $O(n)/n = O(1)$ writes.

The deletions of AVL trees is more complicated and $\Theta(\log n)$ rotations can be applied on every deletion. Amani et al. [20] recently showed that there exists such a sequence of $3n$ intermixed insertions and deletions on an initially empty AVL tree that takes $\Theta(n \log n)$ rotations. This instance indicates that the classic implementation of AVL trees has an I/O cost $Q = \Theta(\omega \log n)$ per deletion in the worst case.

Treaps. A treap, also called as a randomized search tree [24, 249], is a Cartesian tree in which each key is given a randomly chosen numeric priority, and the inorder traversal order of the nodes is the same as the sorted order of the keys. The priority for any non-leaf node must be greater than or equal to the priority of its children.

When inserting an element into a treap with $n - 1$ elements or removing an element from n elements, The updated element only compares to the elements that each has a higher priority than the other elements between this element and the updated element. The number of such comparisons is $\sum_{j \in [n] \setminus \{i\}} 1/|i - j| = O(\log n)$ in expectation[4], where i is the position of the updated element in the total order of n elements [249].

The number of rotations can be computed similarly. For an insertion, a rotation happens once the inserted element has a higher priority than all elements in the entire subtree. Again if we assume the inserted element ranked i-th in the total order, the probability that it rotates up for the j-th ranked tree node is $1/|i - j|^2$ (i.e., the j-th element has higher priority than all elements between, and lower than the priority of the inserted node). The overall expected number of rotations per insertion is $\sum_{j \in [n] \setminus \{i\}} 1/|i - j|^2 = O(1)$ in expectation. We can show the constant writes per deletion accordingly.

We note that unlike an AVL tree or a red-black tree, a treap does not require updates to the balancing criteria, which means that we never need to modify the information in each tree node after it is inserted. As a result, an insertion, deletion or query on a treap requires $O(\log n)$ reads *whp* and an insertion or deletion requires $O(1)$ writes in expectation.

8.4.2.2 Join-based Implementation

Blelloch et al. [65, 261] recently proposed a framework that supports fast bulk operations. This framework supports efficient single or multiple insertions or deletions on AVL trees, red-black trees, treaps and weight-balanced trees, as well as union, intersection and set difference on two BSTs (of the same kind). This framework is very concise: each function can be implemented within a dozen lines of code and are independent with the specific balancing criteria in different types of BSTs. The functions rely on only one helper

[4]Also with high probability.

operation Join [3, 4, 65, 258, 267], which deals with the tree balancing and can be treated as a black box. Under this framework, the implementation of these functions is highly efficient and parallelized.

Join(T_L, k, T_R): The Join function takes two balanced BSTs (of the same balancing schemes) T_L and T_R and a key k, and returns a new balanced BST for which the in-order values are a concatenation of the in-order values of T_L, then k, and then the in-order values of T_R. In order to keep the ordering in the tree nodes, the Join function assumes k to be greater than all keys in T_L and smaller than all keys in T_R. Join handles all issues related to rebalancing, and is the only function that knows about and maintains the balance invariants. The Join algorithms for each of the balancing schemes are given in [65]. Meanwhile, Join is the only function that *writes* to the memory. More specifically, all operations that change the attributes of a tree node (e.g., linking to new children, height-maintaining for AVL or red-black trees, color-changing in red-black trees, etc.) are restricted in Join. This property also greatly simplifies the counting of writes in our simulator and makes the optimizations to reduce writes easier.

Split$(T, k) \rightarrow (T_L, T_R)$: Split can be viewed as the inverse of Join that takes a balanced BST and a key k, and splits the tree into two smaller trees T_L that contains keys smaller than key k, and T_R for keys larger than key k. Split can be trivially implemented using Join by a recursive approach.

Bulk Operations. Using Join as a black box, either single or bulk updates can be implemented simply and generally across different balancing schemes. As an example, we give the algorithm of taking the union of two BSTs (or sets/maps) in Algorithm 14. The algorithm is based on divide-and-conquer: T_1's root is used to partition all elements in two trees into two disjoint set, one with T_1's left subtree and T_l, and the other with T_1's right subtree and T_r. Then we union the two subproblems recursively and independently, and join the two output trees using T_1's root. The correctness and running time are shown in [65].

There are several benefits of using this framework for our implementation and experiment. First, as we just explained, different updates have the uniform code on different types of BSTs (except for the Join), which justifies the performance by the different balancing criteria for the BSTs, instead of the different implementations for different trees. Second, although the joined-based implementation operates on two trees, one can check that when one tree contains only a singleton element, the algorithm runs the same as the algorithm of a single insertion on each type of the BSTs. The deletion can also be implemented by taking the difference by the original tree and a tree with a single element. As a result, the joined-based implementation is strictly more powerful. Lastly, we can also run interesting experiments on more operations like bulk updates, and compare the results on different BSTs.

Algorithm 14: The union function

```
1 Function UNION(T1, T2)
2     if T1 = NIL then return T2
3     if T2 = NIL then return T1
4     ⟨Tl, Tr⟩ ← SPLIT(T2, T1's root)
5     return JOIN(T1's root, UNION(T1's left subtree, Tl), UNION(T1's right subtree, Tr))
```

8.4.2.3 Experiments

In this section, we show our experimental results on the counts of read/write transfers for different settings. Due to the page limit, our experiment mainly focuses on the performance of various binary search trees (AVL trees, red-black trees, and treaps) with different batch sizes.

In the experiment, we first insert $m = 10^6$ million integers as keys to a tree T (empty at the beginning), drawn from a uniform distribution from 32-bit unsigned integers, and then delete them in a uniformly random order. The insertion and deletion are grouped in batches of size s, indicating that the insertions are $\lceil m/s \rceil$ unions on the main tree T with trees of size s. The deletions are also batched in $\lceil m/s \rceil$ bulks of size s. In our experiment, we construct a smaller tree for each batch and then call the union or difference function we just mentioned. We note that if all update elements are given in advance, we can also sort them and put them in a list. Since this is a more specific case, our experiment is based on the tree-tree updates.

The node size in all different types of tress is 16 bytes. Each tree node stores four 4-byte data blocks to hold the key, the left and right pointers, and the balancing information. The cache contains 10,000 cache-lines, similar to the setting for unordered sets.

Table 8.6 shows the experimental results on BSTs with different balancing schemes with various batch sizes. The numbers are read and write transfers per update.

Batch size 1.

We first look at the case corresponding the single insertions and deletions, and the results are shown in Figure 8.2 and 8.3. Regarding the number of writes, treaps show the best performance. This is easy to understand since treaps do not modify any information for rebalancing during insertions/deletions. The structure of treaps is deterministic once the priorities are decided. The priorities are set before the merging (deleting), and never changed later. For such reasons, treaps require much fewer writes per update compared to AVL and red-black trees.

The AVL and red-black trees maintain the balancing information on each tree node that needs to be updated when the subtree height changes. Then more writes are used due to these updates. Between these two types of BSTs, red-black trees require more writes, since there are also some color flips on the siblings of the updated tree path. Such flips are

(a) Insertion / Union

Batch	AVL		RB-Tree		Treap		$\omega = 10$			$\omega = 100$		
Size	RT	WT	RT	WT	RT	WT	AVL	RB-T	Treap	AVL	RB-T	Treap
1	11.28	2.83	12.00	3.48	16.61	1.79	39.6	46.8	34.5	295	360	196
1k	12.67	2.89	13.29	3.66	17.18	1.89	41.6	49.9	36.1	302	379	206
10k	6.18	2.68	6.40	3.01	7.33	1.85	33.0	36.5	25.8	274	308	192
100k	2.54	1.84	2.54	1.86	2.56	1.66	20.9	21.2	19.2	186	189	169

(b) Deletion / Difference

Batch	AVL		RB-Tree		Treap		$\omega = 10$			$\omega = 100$		
Size	RT	WT	RT	WT	RT	WT	AVL	RB-T	Treap	AVL	RB-T	Treap
1	13.83	2.72	16.17	5.17	17.85	1.98	41.1	67.8	37.7	286	533	216
1k	15.11	2.79	17.65	5.21	18.52	2.08	43.0	69.8	39.3	294	539	227
10k	8.17	2.69	10.43	3.44	9.09	2.06	35.1	44.9	29.7	278	355	215
100k	3.22	1.99	3.54	2.43	3.23	1.78	23.2	27.8	21.1	203	246	182

(c) Average Tree Depth

AVL	RB-Tree	Treap
19.39	19.62	26.48

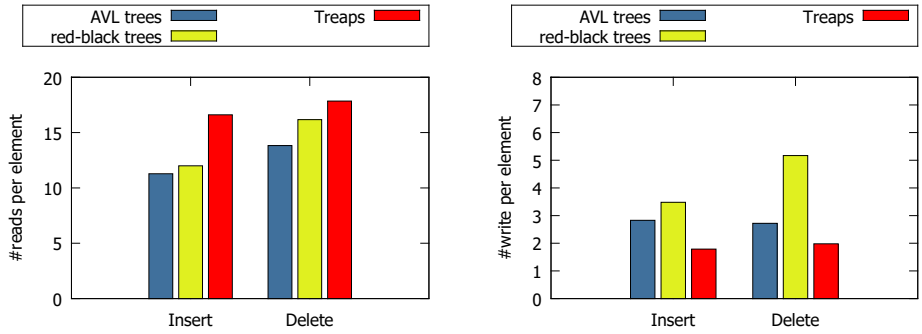
Table 8.6: Numbers of read and write transfers and asymmetric I/O costs of different BSTs with various batch sizes. The numbers are divided by 10^6 (i.e., per inserted/deleted elements). The write-read ratio ω are selected to be typical projected values 10 (latency, bandwidth) and 100 (energy).

extremely cheap in the classic symmetric setting but will cost much in the asymmetric setting when writes become expensive.

The fewer writes for treaps come together with the extra cost on more reads. Treaps are less strictly balanced compared to the other two trees, which saves the writes to maintaining such balancing, but leads to larger average tree depth (shown in Table 8.6(c), about 30% deeper than the other trees). The average depths for AVL and red-black trees are close to optimal, which is 18.96 for a perfectly balanced tree with 10^6 nodes. Because of the shallower average depth, the number of reads required in either updates or queries is much small on these two trees.

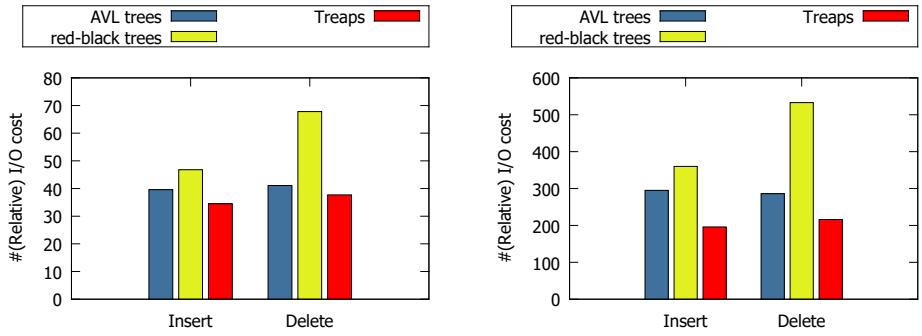
Larger batch sizes.

We now discuss how does the batch size affect the numbers of read and write transfers. The numbers are given in Table 8.6.



(a) Number of reads (b) Number of writes

Figure 8.2: The number of read and write transfers of different BSTs on single insertion/deletion. Data are from the first rows in Table 8.6(a) and 8.6(b).



(a) $\omega = 10$ (b) $\omega = 100$

Figure 8.3: The I/O costs of different BSTs on single insertion/deletion. Data are from the first rows in Table 8.6(a) and 8.6(b).

.

When the batch size increases but remains small, the reads and writes almost remain the same and slightly increase. When the batch size is smaller, the elements in a batch and the paths to visit them are always loaded into the cache once and always stay there. Therefore, the different batch sizes less than 100 do not affect the performance much.

The peak I/O cost per update is around batch size 1000, as we show here. In such case, the overall footprint of a batch update no longer fit into the cache, which may lead to two loads per tree node (once in the split phase and the other in the join phase).

However, the cost turns down when the batch size grows over 1000. The reason is that, the number of tree nodes visited during each bulk update is $\Theta(s \log(m/s))$ (recall that s is the bulk size), which is also the time complexity [65] of this process. Namely, we need to visit $O(\log(m/s))$ tree nodes per inserted node, so we touch fewer nodes as s goes up. Compared to the single updates, each node on multiple tree paths[5] is only looked at

[5]Usually on the top part of the tree. For example, every single insertion visits the root node.

and compared with once in the joined-based bulk updates, and this is from which the improvement comes. Since the top levels in the trees are visited more frequently, they usually stay in the cache. As a result, the saved memory accesses for small batch size are hidden by the function of the cache. However, when the batch size keeps growing and exceed the cache size, then the saved memory accesses lead to lower reads, as shown in Table 8.6.

The number of writes also decreases for larger batch sizes in AVL and red-black trees, because of the previously stated reason. When the elements are inserted or deleted one by one, the balancing factor of a node on by multiple tree paths can be updated multiple times. On the other hand, when the tree is updated in a bulk, such information will be updated by at most once at the join point, which saves the number of writes to the asymmetric memory. However, since treaps do not need to update such information, we cannot observe a significant drop-off on writes for larger batch sizes.

Queries.

We have not explicitly tested the I/O costs for queries, since most queries (like finding or checking a key, locating the k-th element) have the same memory access pattern as the insertions. The only difference is that they do not modify the tree, so there will be no writes. These updates may flush the dirty cache lines, and thus slightly increase the writes. Our experiment shows that if we have the same number of queries and insertions, the number of writes increases by no more than 5%, which is insignificant. Therefore, we believe that we can ignore such changes in the most cases.

8.4.2.4 Conclusions

In this section, we theoretically analyze the asymmetric I/O costs of different types of binary search trees. We show that red-black trees, the insertions for AVL trees, and treaps in expectation have an optimal asymptotic cost ($\Theta(\omega + \log n)$ per update).

We then test the actual performance by conducting experiments based on the join-based implementation, and show that treaps have the best update cost in most cases. The advantage comes from a looser balancing constraint, which also leads to a larger tree depth and query costs. As a result, AVL tree will be a better option if the queries are much more than the updates.

8.5 Sorting

Sorting is one of the most fundamental algorithms and building blocks in algorithm design and programming. In this section, we analyze, performance engineer and experiment the performance of the existing sorting algorithms in the asymmetric setting, which include quicksort, mergesort, BST sort and samplesort.

8.5.1 Algorithms and Implementations

We discuss four sorting algorithms, which are either used as the state-of-the-art implementations or have efficient theoretical guarantees on asymmetric I/O cost. All four algorithms requires $O(n \log n)$ (optimal) comparisons.

Quicksort. Quicksort is one of the commonly-used sorting algorithms in both sequential and parallel setting. It picks a random pivot (or the median among several random samples) that partitions the array into three contiguous subsets containing the input values that are less than, equal to, and larger than the pivot respectively, based on a scan-based approach. Then two recursive calls are made to the less-than and larger-than subsets, until the base case (we set it to be 16 elements) is reach and the algorithm switch to an insertion sort. In our implementation, the pivot is selected to be the median of three random positions.

We now analyze the I/O cost of quicksort. Once the subproblem size is no more than M, the small-memory size, the subproblem will fit into the cache and no more read and write transfers are required for this subtask thereafter. The I/O cost Q is thus $O(\omega n/B \cdot \log(n/M))$, where the number of memory transfers to partition the array in the same recursive level add up to $O(n/B)$ and *whp* the algorithm requires $O(\log(n/M))$ levels to reach the subtask with size no more than M.

Mergesort. Mergesort is another textbook sorting algorithm that is stable and easy to parallel. The implement of mergesort has different versions, and here we discuss an I/O-efficient version. The algorithm partitions the array into two equal-length parts and recursively sort every subproblem individually. After the computation of both subtasks is finished, a scan-based process merges the two sorted subsequences into the final output. To implement the algorithm efficiently, we use the rotating arrays so that every element is moved only once in one merge process. An extra round of data copy is applied if the latest merge leaves the sorted result in the temporal array.

Similarly to quicksort, no further memory transfers are used in subproblems with size no more than M. Therefore the I/O cost Q of mergesort is also $O(\omega n/B \cdot \log(n/M))$. More specifically, $Q = (1+\omega)n/B \cdot \lceil \log_2(n/M) \rceil$ when $\lceil \log_2(n/M) \rceil$ is even, and otherwise there is an extra $(1+\omega)n/B$.

BST sort. BST sort treat the input as an ordered set and all elements are maintained in a binary search tree. Then the in-order traversal of the tree is the sorted output. The BST can either be balanced, or does not apply any rotation but the elements are inserted in a random order. Both cases give I/O cost of $O(n \log n + n\omega)$ on n input elements. It is typically used when the input is adaptive: we can insert and delete elements dynamically, while the sorted result are maintained at any time. In this case, some balancing schemes are required if the insertions and deletions are biased.

In our implementation elements are inserted in a random order, which is done in two phases: the first phase generates n uniformly random integers $a_1, \cdots, a_n$ between 1 to n using some hash functions, and inserting the a_i-th elements into the tree with *no* rotations

(check whether this element is already inserted before the actual insertion); then in the second phase we insert every uninserted element into the tree in the input order. This guarantees that *whp* the gap between the elements inserted in the first phase is $O(\log n)$ in the output order. Hence the tree depth is $O(\log n)$ disregarding any input order, and thus inserting one element requires $O(\log n)$ reads and two random writes (the boolean flag that this node is inserted, and to change the parent's pointer).

Samplesort. Samplesort is a class of sorting algorithms that are based on divide-and-conquer paradigm with multiple pivots and especially commonly-used in the multicore setting. To minimize the I/O cost and number of memory transfers, in this paper, we discuss a cache-aware implementation of samplesort.

Given the cache size M and block size B, if the input size is smaller than M then we simply call quicksort. Otherwise, we pick M random samples from the input, sort the samples, and pick the B-th, $2B$-th, ... , $(M-B)$-th samples as the **pivots**[6] and they form M/B buckets. Then for each input element, we binary search (an implicit search tree) its associated bucket. Finally, we again samplesort the elements in each bucket individually and recursively.

The algorithm partition the input into M/B almost equal-size subproblems with $O(n/B)$ read and write transfers. The I/O cost of this algorithm on average is $O\left(\frac{\omega n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right)$.

The number of write transfer of samplesort in our implementation is highly optimized so that one round of samplesort only requires three sequential reads and one sequential write per input element. After picking the pivots, we first loop over the input to binary search the associate bucket of each element. However, we do not store this value; instead, we only modify the counters of each bucket. After that, we have known the number of elements in each bucket, and we apply a prefix sum compute the offset of each element. Finally, all elements are distributed to their associated buckets based on the offsets. The algorithm only requires three reads and one write for each element, and all other operations are all within the cache. Notice that after one round, the data are stored in another array, so the final results will be moved back to the original array if they happen to be in the other one.

There do exist other sorting algorithms like heapsort, shellsort, bitonic sort, etc. Their performances regarding I/O cost however, are less competitive against the previous sorting algorithms due to either more work or inefficient memory-access pattern. We did not compare with previous work in [273] in this paper, since they are not optimal in terms of comparisons and have tunable parameters in the algorithm, that makes the comparison among them inconclusive. Instead as a very first paper on this topic, we focus on simple

[6]If $M < n/B$ then we pick n/B samples, and this will happen only once *whp* in the algorithm. In practice we replace M to be M/c' for some small constant $c' > 1$ to ensure everything that should be in the cache is actually in the cache.

Algorithm	I/O cost
Quicksort	$O(\omega n/B \cdot \log(n/M))^*$
Mergesort	$O(\omega n/B \cdot \log(n/M))$
BST sort	$O(n \log n + n\omega))$
Samplesort	$O(\omega n/B \cdot \log_{M/B}(n/B))^*$

Table 8.7: List of I/O costs on sorting algorithms. (*) indicates the bounds hold with high probability ($1 - n^{-c}$ for arbitrary $c > 0$).

algorithms and draw interesting conclusions that can also be useful in implementing these more complicated approaches in the future.

8.5.2 Experiments

In the experiment we test our implementations of the four sorting algorithms on the numbers of read and write transfers with various cache sizes. We set the input into two categories: one that we indeed move the data, and the other that the algorithm sorts the indirect pointers pointing to the data fields. The first case is when we are sorting integers, real numbers or any structures with small and fixed size like graph edges, key-value pairs of integers and/or pointers, etc. The second case is when the input is irregular, like strings, texts, images, or any structure that is either large or the sizes vary among different elements. In this case, moving the data is expensive, so we will sort and output a list of indices pointing to the actual data fields. Both cases are widely used in practice.

Performance on sorting float-point numbers. We now show the experiments that perform data movement. We sort 10 million random i.i.d. double-precision float-point numbers and in this case the block size $B = 8$. The numbers of comparisons and read and write transfers of four algorithms are summarized in Table 8.8. The I/O costs of the algorithms for two typical values of ω (10 and 100) are visualized in Figure 8.5.

Both quicksort and mergesort require about $\log_2(n/M)$ rounds, which is 8 to 16, to fully fit the subproblems into the small-memory. Both algorithms require approximately the same number of writes, since quicksort is in-place while mergesort is not and has double memory footprint, but the partition of quicksort is not exactly even. These factors offset each other. However, the read transfer are doubled in mergesort, and in practice causing its less efficient practical performance. Mergesort requires fewer comparisons, but this does not lead to a significant difference in modern architecture, compared to the I/O cost to the main memory.

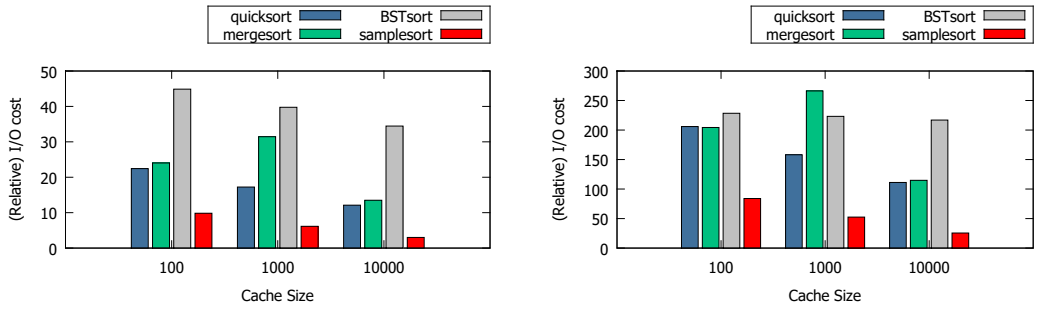
BST sort, as we analyzed theoretically, requires about $\log_2(nB/M)$ reads and two random writes plus some small cost on initializations. BST sort uses less writes compared to quicksort or mergesort when $2B < \log_2(n/M)$, which is hard to be satisfied given the

Cache	Quicksort			Mergesort			BST sort			Samplesort		
Size	RT	WT	#comp	RT	WT	#comp	RT	WT	#comp	RT	WT	#comp
100	2.06	2.04	39.49	4.01	2.00	24.11	24.49	2.04	43.76	1.60	0.82	63.05
1000	1.58	1.57	39.49	3.11	1.56	24.11	19.39	2.04	43.76	1.01	0.51	56.63
10000	1.11	1.10	39.49	2.25	1.13	24.11	14.17	2.03	43.76	0.50	0.25	52.47

Table 8.8: Number of read and write transfers of different sorting algorithms on 10^7 random i.i.d. double-precision float-point numbers. Values in the table are numbers of read and write transfers and comparisons per input element. Cache size is measured by the number of 64-byte cache-lines.

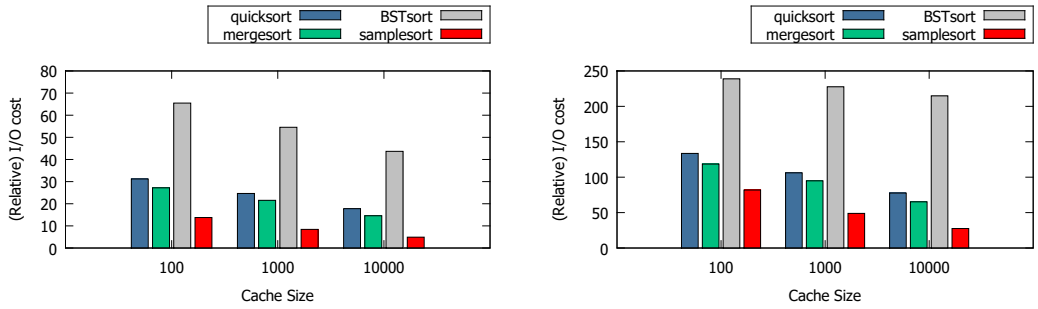
Cache	Quicksort			Mergesort			BST sort			Samplesort		
Size	RT	WT	#comp	RT	WT	#comp	RT	WT	#comp	RT	WT	#comp
100	19.83	1.14	37.68	17.04	1.02	23.30	46.24	1.93	40.93	6.19	0.79	35.97
1000	15.57	0.91	37.68	13.42	0.81	23.30	35.30	1.92	40.93	3.93	0.45	34.34
10000	11.12	0.67	37.68	8.96	0.56	23.30	24.65	1.90	40.93	2.38	0.38	34.43

Table 8.9: Number of read and write transfers of different sorting algorithms on 2×10^6 indices pointing to structures with average size of 64 bytes. Other setup is the same as that in Table 8.8.



(a) $\omega = 10$ (b) $\omega = 100$

Figure 8.4: I/O costs of different sorting algorithms on 10^7 **doubles** with various read-write asymmetry ω. Numbers are from Table 8.8.



(a) $\omega = 10$ (b) $\omega = 100$

Figure 8.5: I/O costs of different sorting algorithms on 2×10^6 **pointers** with various read-write asymmetry ω. Numbers are from Table 8.9.

parameters of the hardware. BST sort thus will still be less efficient to sort small and fixed-structure entries in future memories.

The experiment result of samplesort follows our analysis as well: each round of sample sort approximately requires three sequential reads and one sequential write, and the algorithm uses roughly $\lceil \log_{M/B}(n/B) \rceil$ rounds to reach the base case. Other costs (sampling and sorting pivots, transposing the offsets, etc.) are negligible.

Based on our previous experience, after highly optimizing the performance in the sequential setting, samplesort is slightly slower than quicksort (but much faster in multicore parallelism) even though the I/O cost of samplesort is lower. We believe the reason to be that, samplesort does not explicitly utilize L1 and L2 caches (the binary search, offset transpose, etc.) so most small-memory accesses are toward L3 cache. The memory access pattern for quicksort, however, is mostly sequential scanning, which is highly optimized (like by prefetching) in the current architecture. Since the write costs will significantly increase for future non-volatile main memory, we project that samplesort will be more efficient in this setting, since the I/O cost will very likely be the bottleneck of all these sorting algorithms (similar to the existing multicore parallel setting).

Sorting larger entries. As shown previously, although BST sort requires only at most random two writes per insertion, this cost is still significant compared to other sorting algorithms that have better spatial locality in memory access. However, when the size of each input entry becomes larger, each cache-line holds fewer entries and the number of write transfers per entry increases.

We run the experiment with a fixed number of inputs (10^7) and cache-lines (1000), and varies the size of each input entry from 8 bytes to 64 bytes. The results are shown in Table 8.10. Based on the analysis in Section 8.5.1, for quicksort, mergesort, and samplesort, the I/O cost is proportional to the entry size. However, BST sort is mostly not affected. The number of reads just marginally increases since the cache can hold less top-tree levels, and slightly more writes are required since the overall footprint of the search tree increases as the growth of the entry size. From the results, we can see that when the entry size becomes at least 64 bytes, BST sort outperforms all other algorithms in the asymmetric setting.

Performance on sorting via pointers. In this experiment, we sort 2 million random strings that the characters are stored contiguously in the memory with an average size of 64 bytes. The input also contains 2 million 8-byte pointers to the strings, and we only sort the pointers.

The implementations of quicksort, mergesort, and BST sort are identical except that each comparison requires two indirect addresses to locate the data field. Samplesort, since it is cache-aware, requires two modifications: first, no oversampling for the pivots since now each pivot requires one or two cache-lines for its data; second, instead binary-search

Entry	Quicksort		Mergesort		BSTsort		Samplesort	
Size	RT	WT	RT	WT	RT	WT	RT	WT
8	1.58	1.57	3.25	1.63	19.39	2.04	1.01	0.51
16	3.26	3.17	7.00	3.50	21.11	2.29	2.09	1.08
32	6.58	6.13	15.00	7.50	23.04	2.79	4.25	2.24
64	13.18	11.58	32.00	16.00	25.35	3.79	8.91	4.87

Table 8.10: Numbers of read and write transfers of different sorting algorithms with various data sizes (bytes). The input size is 10^7 and the cache contains 1000 cache-lines.

the bucket of each element twice, we now store the bucket label after the first search to avoid the second search (roughly double the writes while half the reads).

The numbers of read and write transfers are reported in Table 8.9 and visualized for two typical values of ω (10 and 100) in Figure 8.5. Writes of quicksort, mergesort, and BST sort stay approximately the same as the previous experiment (the decrease in values is due to fewer input elements). Reads are now increased by about $\log_2(nB/M)$, the cost of locating the data fields.

For each round in sample sort, each element now requires one random read, three sequential reads and two sequential writes (i.e., 1.375 read transfers and 0.25 write transfer). Also, one extra random read per element is in the base case to load the data into the cache. The algorithm requires $O(\log_{M/B}(n))$ rounds to reach the base case, and in the experiment, this round number is approximately 3, 2 and 1 for three cache sizes. Finally, output needs to be moved back when reaching the base case in an odd number of rounds.

As what we understand, the wall-clock performance of samplesort when sorting pointers on current architecture is already faster than the other sorts. This is because samplesort requires fewer I/Os and comparisons, and all algorithms randomly access the memory so techniques like prefetching cannot help. This gap will be even enlarged in future NVMs when the write costs being exaggerated.

8.5.3 Conclusions

Based on our implementations and experiments, the following conclusions and the projection of the techniques for future non-volatile main-memories can be drawn.

Samplesort. Samplesort generally requires fewer I/Os than other sorting algorithms. On existing hardware, since samplesort does not explicitly optimize for L1/L2 cache, its sequential performance is slightly slower than quicksort. However, in the multicore parallel setting, samplesort is always the fastest due to its efficiency on I/O cost. We predict that samplesort will play a more significant role with the future hardware even in

the sequential setting, because of the lower number of accessing to the large asymmetric memory and the asymmetry of bandwidth and energy on the new memories.

Sorting indirect pointers. The advantage of samplesort will be enlarged when sorting via pointers. This is because samplesort requires fewer rounds to finish, and therefore it uses fewer reads, fewer writes, and fewer comparisons compared to other algorithms. This also matches the observation on the existing symmetric setting that samplesort is more efficient on wall-clock performance.

BST sort. Although BST sort only requiring constant writes on large-memory per update, it is still inefficient compared to other approaches when sorting integers or float-point numbers because of its lacking on utilizing the cache-lines. However, when sorting entries with at least 64 bytes (i.e., a cache-line size), the I/O cost of BST sort outperforms the rest three algorithms.

8.6 Graph Traversal Algorithms

We now provide the full version of the graph-traversal algorithms in this paper. We discuss two of the most commonly-used graph-traversing algorithms: Breadth-first search (BFS), and Dijkstra's Algorithm. We show that with appropriate implementations, these algorithms can write much fewer to the main memory, comparing to the classic implementations.

Throughout this section we assume the input graph $G = (V, E)$ contains $n = |V|$ vertices and $m = |E|$ edges.

8.6.1 Breadth-First Search

Breadth-first search (BFS) is an algorithm for graph traversing or searching. It starts at the source node, which is an arbitrary node of a graph, and explores the vertices with respect to the distances (the number of hops) from the source node. BFS is commonly-used in computing single-source shortest paths on unweighted graphs, as a subroutine for graph radii estimation, eccentricity estimation and betweenness centrality, and as a basic building block for other graph algorithms like graph connectivity, reachability, bridges, biconnected components, and strongly connected components. In this paper we focus on the most basic application: shortest paths on undirected graphs, and the techniques discussed here can apply to many other applications.

8.6.1.1 The Classic Implementation

Given a graph $G = (V, E)$ with n vertices and m edges, the classic implementation of BFS keeps a queue with size n, and an array of boolean flags with size n indicating that each vertex is visited or not during the search. This implementation requires at most 2 writes for each vertex: one sequential write for adding it to the queue and one random

access for changing the flag of this vertex. Meanwhile, searches along an edge to an already visited vertex require no writes. The overall I/O cost of BFS $Q(n, m) = O(\omega n + m)$ (Chapter 5).

This bound is asymptotic optimal for arbitrary graphs, since the output of BFS, the shortest-path tree, has size $O(n)$. However, a number applications (e.g., s-t shortest-path or connectivity, graph radii estimation or eccentricity estimation) have output size $O(1)$, which allow techniques utilizing the small-memory and reducing the number of writes.

8.6.1.2 BFS Implementation using Rotating Arrays

The rotating arrays are used in many algorithms to reduce the space requirement. For example, the diamond DAG computation in the longest common subsequence (LCS) problem only requires storing two consecutive columns (or rows) at any time during the dynamic programming process, since each DP value only depends on three other nearby vertices in the diamond DAG. The algorithm maintains two arrays each with the size of either input string: one to hold the results in odd columns (rows) while the other for the even ones. We call this structure the *rotating arrays* since the two arrays are rotating and holding computed and uncomputed values. They only require temporal (cache) space of one dimension although the nodes in the entire DAG have two dimensions.

Here we observe that BFS on undirected graphs can apply a similar approach: instead of keeping a queue and a global array of boolean flags with size n, we maintain separate queues, each corresponding to a specific frontier (i.e., the set of vertices with the same distance to the source). This is because, during the BFS on undirected graphs, the outgoing edges from the i-th frontier can only reach the vertices in either the $(i-1)$-th frontier or the $(i+1)$-th frontier. Otherwise, assume an edge reaches a vertex in the $j(< i-1)$-th frontier, then the vertex in j-th frontier will visit this vertex in $(j+1)$-th frontier. As a result, when processing the i-th frontier, we only need to keep three frontiers with distance $i-1$, i and $i+1$.

Since each frontier is separately kept and all vertices in one frontier have the same distances, we no longer need to keep the relative orders of the elements within each frontier. We hence directly use an unordered set to maintain each frontier. Since only three frontiers are useful during the BFS process, we also only allocate and keep three unordered sets and their roles rotate among the previous, the current and the next frontier.

The 2-level hash table introduced in Section 8.4.1.1 works perfectly well here since we have no control of the frontier sizes, which can be either greater or smaller than previous ones. For each frontier, we loop over all vertices in the set, and for each outgoing edge, if the other endpoint is not in either of the three unordered sets then it is added to the next frontier. Therefore, the operations to the sets include lookups and insertions. We do not explicitly delete the sets. Instead, when the searching of one frontier is finished, the new next frontier will reuse this space of the previous frontier.

8.6.1.3 Bidirectional Search

This previous implementation based on rotating arrays can greatly reduce the number of write transfer when the frontier size is much smaller than the number of vertices, like grids, meshes, roadmaps, etc. For other graphs with larger frontier size and smaller diameter (e.g., real-world networks that follow power law), I/O cost is not improved significantly. However for these graphs, the average distance between every pair of vertices is usually very small, and this is called the "small-world phenomenon" [277].

To utilize this property, we employ bidirectional breadth-first search on computing the s-t shortest path. Assuming the distance between this pair of query vertices is d, then the search from each direction will only visit the vertices within distance $\lceil d/2 \rceil$. On these power-law graphs, the frontier size grows exponentially on the first several steps until most of the vertices are reached. The number of vertices on the top-half levels in the shortest-path tree is therefore far less than n. This difference however, is negligible on graphs with bounded frontier size, since the distance between a random pair of vertices is large expectedly, and the bidirectional search eventually reaches about the same amount of vertices unless d is much smaller than the graph diameter.

To implement bidirectional search efficiently, we also use three rotating arrays for the search process. The extra detail here is that, the source, destination, and all intermediate vertices are put together in the same sets, and use one bit (the sign bit) to identify if the vertex is by the search from the source or from the destination. We only keep three instead of six sets, which reduces the number of read transfers approximately by a half.

8.6.1.4 Experiment

In the experiment, we examine whether and how the new algorithms we just discussed improve the I/O cost. We implement four versions of breadth-first search: classic and bidirectional search with and without using rotating arrays. The experiment is run on 8 graph instances with various cache size.

Graph instances. We use graphs of various types from the SNAP datasets [199] as the input of the algorithms. The graphs include the road networks in Pennsylvania and Texas (real-world planar graphs), the web graphs of the University of Notre Dame and Stanford University, the DBLP collaboration network, and the Youtube online social network (4 real-world networks). In the case of web graphs, each edge represents a hyperlink between two web pages. We also use synthetic graphs of 2D and 3D grids. For each of the graphs used, the numbers of vertices and edges are given in the table below. If a graph does not come equipped with weights, we assign to every edge a random integer between 1 and $10,000$. The graphs we used are relatively small due to the overhead in our software and hardware simulator, but we adjust the cache size accordingly.

Overall performance. The numbers of read and write transfers on 8 graph instances with various cache size are given in Table 8.11. Their weighted I/O costs are given in

Type	Graph	# Vertices	# Edges
Sparse Graphs	Grid: 2D	1M	3.96M
	Grid: 3D	1M	5.94M
	Roadmap: Pennsylvania	1.09M	3.08M
	Roadmap: Texas	1.39M	3.84M
Social Networks	Webgraph: Notre Dame	325K	2.20M
	Webgraph: Stanford	281K	3.98M
	DBLP collab. network	317K	2.10M
	Youtube network	1.13M	5.98M

Table 8.12 considering two typical values of ω, and visualized (merged into two categories) in Figure 8.6.

To better understand the performance of different implementations, we also count various statistics of the searching including the depths and frontier sizes, and the results are generalized in Table 8.13.

Unidirectional search. Indicated in Table 8.11, classic BFS requires no more than one sequential write (pushed into the queue) and one random write (marking the distance) per vertex. The random write can be avoided if the associated cache line is cached, so better locality of vertex indices of some graphs (roadmaps and NDU webgraph) and larger cache size reduce writes per vertex. The algorithm also reads the edges of each vertex and checks whether the other endpoint is visited or not, which is also affected by the edges per vertex and the locality of vertex indices.

For the implementation using rotating arrays, the key factor is whether the frontier fits into the cache. Shown in Table 8.13, the sparse graphs (grids and roadmaps) have smaller frontier sizes, so as long as the cache can hold each of them, the writes are largely minimized. This is also true for reads since checking is always in the hash table. However, once the frontier is larger than the cache size, then each insertion to the hash table now becomes a random write and can hardly be cached because of the hash function. The reads are increased even more, since checking whether a vertex is visited can lead to at most 6 cache misses: three rotating arrays each with 2 levels. Summarized in Figure 8.6, the I/O cost is largely improved using the rotating arrays when frontiers fit into the cache, but deteriorates when not.

Bidirectional search. The I/O performance of classic bidirectional BFS is similar to the classic unidirectional BFS since they essentially search in a similar pattern. The bidirectional search requires fewer reads and writes since it strictly searches fewer vertices, especially in social networks since these graphs have smaller diameters and the two searches usually meet earlier before the majority of the vertices are visited.

Algorithm	Classic BFS						BFS using RA					
Cache Size	500		2000		10000		500		2000		10000	
	RT	WT	RT	WT	RT	WT	RT	WT	RT	WT	RT	WT
2D Grid	18.47	6.17	17.60	6.01	10.88	4.59	8.75	0.84	6.52	0.16	6.20	0.01
3D Grid	19.38	5.45	16.42	5.31	13.51	4.51	88.24	4.26	31.03	2.01	6.39	0.38
PA Roadmap	8.73	2.78	8.07	2.54	2.43	1.05	29.61	2.95	5.37	0.65	1.26	0.02
TX Roadmap	6.79	2.16	6.01	1.91	1.95	0.85	23.45	2.29	4.32	0.50	1.00	0.02
Stan Webgraph	39.75	3.97	27.11	3.75	9.63	1.48	232.09	5.99	150.70	4.74	35.19	2.10
NDU Webgraph	3.47	1.04	3.02	0.97	2.52	0.77	84.67	4.80	58.91	3.91	18.33	1.73
DBLP Network	19.70	5.75	15.43	5.07	6.85	1.68	140.01	7.34	104.12	6.27	37.67	3.20
Youtube Network	13.23	2.47	9.64	2.29	5.63	1.84	91.70	6.77	71.98	6.39	39.95	5.08
Algorithm	**Classic Bidirectional BFS**						**Bidirectional BFS using RA**					
Cache Size	**500**		**2000**		**10000**		**500**		**2000**		**10000**	
	RT	**WT**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**
2D Grid	18.14	4.97	17.61	4.94	9.39	4.61	7.60	0.55	4.94	0.13	4.69	0.01
3D Grid	11.66	3.08	10.52	3.06	9.47	2.98	38.24	1.97	8.80	0.69	2.90	0.09
PA Roadmap	7.87	3.19	7.58	3.09	3.37	1.60	27.00	2.62	4.25	0.43	1.06	0.02
TX Roadmap	5.31	2.30	5.03	2.19	1.79	1.06	11.12	1.18	2.30	0.21	0.60	0.01
Stan Webgraph	4.70	1.38	4.24	1.37	2.74	1.29	7.32	0.84	3.00	0.47	0.63	0.13
NDU Webgraph	0.80	0.70	0.77	0.70	0.74	0.68	2.63	0.43	1.12	0.28	0.16	0.06
DBLP network	1.40	1.01	1.29	0.99	1.05	0.89	1.33	0.25	0.37	0.12	0.12	0.04
Youtube network	0.74	0.68	0.71	0.68	0.69	0.67	0.25	0.09	0.08	0.04	0.02	0.01

Table 8.11: Numbers of read and write transfers of BFS implementations on different cache sizes. Numbers are r/w transfers per vertex per **10** queries. We pick the number 10 to fit the numbers in one table.

This special property of social network largely helps our implementation using rotating arrays. Shown in Table 8.13, the two bidirectional searches use 2.7-3.9 rounds on average to meet, which preserves the frontier in the cache during the search even for very small cache sizes. As a result, the bidirectional BFS using rotating arrays has a constantly good performance on all combinations of graphs and cache sizes, which is shown in Table 8.12 and Figure 8.6.

8.6.1.5 Conclusions

We discuss how to efficiently implement BFS in the asymmetric setting and experiment the I/O performance for four implementations on a variety of graphs. We show that if the query is s-t (pairwise) distance, our bidirectional BFS using rotating arrays shows an overwhelming advantage in all cases we tested. If all distances to the source are required,

$\omega = 10$

Algorithm	Classic BFS			BFS using RA			Classic Bidir. BFS			Bidir. BFS using RA		
Cache Size	**500**	**2000**	**10000**	**500**	**2000**	**10000**	**500**	**2000**	**10000**	**500**	**2000**	**10000**
2D Grid	80.18	77.72	56.74	17.13	8.15	6.25	67.80	67.04	55.53	13.14	6.27	4.76
3D Grid	*73.85*	69.49	58.60	130.86	51.16	10.18	*42.44*	41.16	39.29	57.98	15.65	3.82
PA Roadmap	*36.50*	33.42	12.89	59.10	11.84	1.44	*39.76*	38.46	19.39	53.24	8.52	1.24
TX Roadmap	*28.42*	25.14	10.44	46.36	9.36	1.18	28.27	26.95	12.37	22.89	4.37	0.69
Stan Webgraph	*79.41*	*64.64*	*24.39*	291.99	198.08	56.18	18.48	17.93	15.62	15.75	7.71	1.93
NDU Webgraph	*13.84*	*12.70*	*10.20*	132.71	98.04	35.58	7.80	7.73	7.54	6.90	3.90	0.75
DBLP network	*77.19*	*66.16*	*23.67*	213.43	166.84	69.63	11.53	11.22	9.91	3.84	1.61	0.56
Youtube network	*37.92*	*32.59*	*24.06*	159.43	135.92	90.72	7.58	7.50	7.38	1.17	0.51	0.17

$\omega = 100$

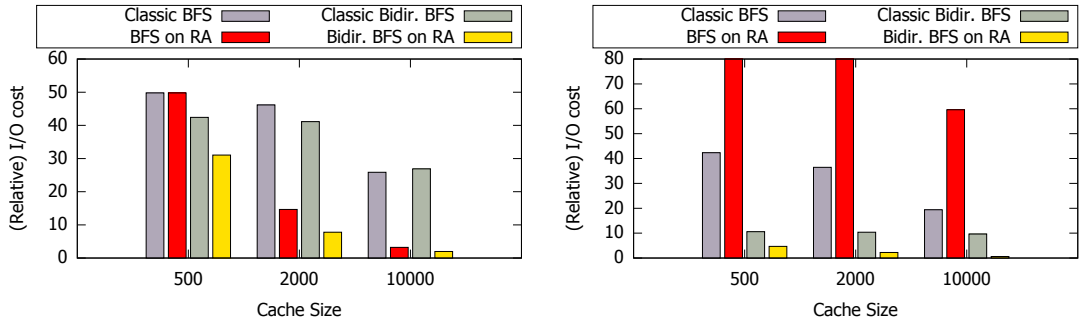
Algorithm	Classic BFS			BFS using RA			Classic Bidir. BFS			Bidir. BFS using RA		
Cache Size	**500**	**2000**	**10000**	**500**	**2000**	**10000**	**500**	**2000**	**10000**	**500**	**2000**	**10000**
2D Grid	635.5	618.7	469.4	92.5	22.7	6.7	514.7	511.8	470.7	62.9	18.2	5.3
3D Grid	564.0	547.1	464.3	514.4	232.3	44.2	319.4	316.9	307.6	235.6	77.4	12.0
PA Roadmap	*286.4*	261.5	107.0	324.4	70.0	3.0	326.7	316.4	163.6	289.4	47.0	2.8
TX Roadmap	*223.1*	197.3	86.8	252.5	54.7	2.8	234.8	224.2	107.6	128.8	23.0	1.5
Stan Webgraph	*436.3*	*402.4*	157.1	831.1	624.5	245.1	142.5	141.2	131.6	91.6	50.1	13.6
NDU Webgraph	*107.1*	*99.8*	79.2	565.0	450.2	190.8	70.8	70.3	68.8	45.3	28.9	6.0
DBLP network	*594.6*	*522.6*	175.1	874.2	731.3	357.3	102.6	100.5	89.7	26.4	12.8	4.5
Youtube network	*260.2*	*239.1*	190.0	769.0	711.4	547.6	69.1	68.5	67.6	9.4	4.3	1.5

Table 8.12: I/O costs of BFS implementations on different cache sizes. Numbers of r/w transfers are from Table 8.11. Italic-font number indicates that the classic implementation has a lower I/O cost in that setting.

the unidirectional BFS using rotating arrays has a better performance if the cache can hold each frontier.

8.6.2 Dijkstra's Algorithm

Dijkstra's Algorithm [114] is a well-known algorithm to compute single-source shortest paths on a non-negative weighted graph $G = (V, E)$. Due to the rapid growth of the data size, real-world graphs nowadays can easily go beyond the size of the order of gigabytes, and they need to be stored on the large non-volatile memory. Running the classic implementation of Dijkstra's algorithm can be costly in this setting. We show that with an appropriate implementation, the algorithm can write much fewer to the non-volatile memory, which further leads to much lower I/O cost.



(a) $\omega = 10$, sparse graphs

(b) $\omega = 10$, social networks

(c) $\omega = 100$, sparse graphs

(d) $\omega = 100$, social networks

Figure 8.6: The trends of the I/O costs of four different implementations of BFS. The new implementations shown in this paper are the BFS and bidirectional BFS based on rotating arrays (red and orange bars). Graphs used in the experiment are shown in Section 8.6.1.4 and categorized into sparse (almost planar) graphs and social networks. We show the relative I/O cost based on varied cache sizes, and each number is geometric mean of the four graphs in that category (the exact numbers are given in Table 8.12). We can see the consistent advantages of the new BFS implementation on sparse graphs, and the improvement of the new bidirectional version in all cases. Notice that in (b) and (d) some values exceed the ranges of vertical axis.

Throughout this section we assume the input graph $G = (V, E)$ contains $n = |V|$ vertices and $m = |E|$ edges.

Dijkstra's algorithm maintains a set of visited vertices associated with their shortest distances to the source (denoted by d_v for $v \in V$), and the unvisited neighbors of these vertices form the "frontier" (d_v for unvisited vertex v is $+\infty$). Each vertex u in the frontier stores a tentative distance to be $\min_{v \in N(u)}\{d_v + e_{u,v}\}$ where $N(u)$ is the incoming neighbor set of vertex u. Initially the visited set contains only one vertex: the source node. The invariant of this algorithm is that, the minimum tentative distance in the frontier set is indeed the shortest distance of this vertex, and thus Dijkstra's algorithm iteratively move

	Unidirectional search			Bidirectional search		
	depth	maximum frontier size	average frontier size	depth	maximum frontier size	average frontier size
2D Grid	1571.4	1460	636.4	370.2	2091	1102.8
3D Grid	229.9	13277	4349.7	49.3	15058	4628.9
PA Roadmap	574.5	5032	1893.1	163.9	7204	2718.3
TX Roadmap	748.6	6058	1804.9	174.7	5752	2102.1
Stan Webgraph	107.1	87614	2383.4	3.5	50094	3782.8
NDU Webgraph	29.2	124519	11155.1	3.8	51562	2053.5
DBLP Network	16.1	129283	19694.4	3.9	36373	1420.1
Youtube Network	15.8	607534	71828.5	2.7	1841	107.4

Table 8.13: The depths (number of levels in the shortest-path tree) and frontier sizes (number of vertices) during the search processes. Note that the numbers are averaged from multiple searches.

this vertex from the frontier to the visited set and update the frontier accordingly. The correctness can easily be proven by induction on the number of visited nodes.

Due to the widespread use of Dijkstra's algorithm, there are plenty of works on the efficient implementations of Dijkstra's algorithm and we refer the readers to some recent works [69, 213] for summaries of the work bounds of different approaches. Different implementations have different costs on the two operations in Dijkstra's algorithm: Extract-Min that finds and removes the vertex with minimum distance in the frontier set, and Decrease-Key that updates the tentative distance of the other endpoint of an edge of this vertex removed from the frontier in this iteration. The data type that supports the queries is abstracted as a priority queue. Specifically when the priority queue is implemented using Fibonacci Heap [128] to maintain the frontier set, the overall time complexity is $O(m + n \log n)$.

8.6.2.1 Classic Implementation using a Binary Heap

Although there are many advanced implementations of the priority queue with lower time complexities, the constant hidden in the asymptotic bound is large. In practice the classic implementation using a binary heap works reasonably well on general sparse real-world graphs, and its wall-clock performance is competitive or even better on modern computer architecture. We hence implement it as a baseline and measure the number of read and write transfers of this algorithm as a comparison to our write-efficient version.

For each vertex, we maintain the shortest distance in a global array with size n. For practical purpose, instead of initializing the priority queue of size n with infinite

distances, we insert a vertex when it is first added to the frontier (so the priority queue needs to support INSERT, which most implementations do). The binary heap only stores the indices of the vertices which optimizes the memory footprint and number of write transfers. To perform DECREASE-KEY in a binary heap efficiently, we keep another global array, an auxiliary structure that maps each vertex to its position in the heap, and is maintained up-to-date as the priority queue changes. This implementation has worst-case time complexity to be $O(m \log n)$. Since this implementation does not take any special optimization on caching, the I/O cost is therefore $O(\omega m \log(nB/M))$, assuming the cache always keeps the first M/B vertices in the priority queue.

8.6.2.2 Phased Dijkstra

Phased Dijkstra (Section 5.4.1) is a specific implementation of Dijkstra's algorithm that the goal is to fully maintain the priority queue in small-memory. It only requires $O(n)$ writes to large-memory, the shortest distances to all vertices. The idea is to partition the computation into phases such that for a parameter M' each phase keeps a priority queue of size at most $(1+\epsilon)M'$ and visits at least M' vertices. By selecting $M' = M/c$ for an appropriate constant c, the priority queue fits in small-memory, and the only writes to large-memory are the final distances.

Algorithm 15: Phased Dijkstra

Input: A connected weighted graph $G = (V, E)$ and a source vertex s
Output: The shortest distances $\delta = \{\delta_1, \ldots, \delta_n\}$ from source s

```
1  Priority Queue P ← ∅
2  Mark vertex s as visited and set δ(s) ← 0
3  while there exists unvisited vertices do
4      if P = ∅ then
5          Scan over all edges in E and store at most M' closest unvisited vertices in P
6          if |P| = M' then
7              Set d_max as the distance to the farthest vertex in P
8          else
9              d_max ← +∞
10     u = P.Extract-Min()
11     Set δ_u as the distance from s to u, and mark u as visited
12     foreach (u, v, dis_{u,v}) ∈ E do
13         if δ_u + dis_{u,v} < d_max then
14             if v ∈ P then
15                 P.Decrease-Key(v, δ_u + dis_{u,v})
16             else
17                 P.insert(v, δ_u + dis_{u,v})
18                 if |P| = (1 + ε)M' then
19                     Remove εM' vertices with larger distances in P
20                     Set d_max as the farthest distance in P
21 return δ(·)
```

The pseudocode of the algorithm is provided in Algorithm 15. Technically the priority queue P can be implemented using an arbitrary heap since it is fully in the small-memory and will not affect the I/O cost. In our experiment we implement it using a binary heap.

In phased Dijkstra, each phase starts and ends with an empty priority queue P. The priority queue is kept small by discarding the $\epsilon M'$ largest elements (vertex distances) whenever $|P| = (1+\epsilon)M'$. To achieve this, P is flattened into an array, the M'-th smallest element d_{max} is found by selection, and the priority queue is reconstructed from the elements no greater than d_{max}, all taking linear time. After such a truncation, all further insertions in a given phase are not added to P if they have a value greater than d_{max}.

The processing of each phase in phased Dijkstra consists of two parts. The first part (line 5–9) of each phase loops over all edges in the graph and relaxes any that go from a visited to an unvisited vertex (possibly inserting or decreasing a key in P). The second part (repeatedly loop over line 10–20) then runs standard Dijkstra's algorithm repeatedly visiting the vertex with minimum distance and relaxing its neighbors until P is empty. Similar to other implementations, to implement relax, the algorithm needs to know whether a vertex is already in P, and if so where in P so that it can do a decrease-key on it. However in phased Dijkstra it is too costly to store this information using a global array. Instead, we use an unordered map introduced in Section 8.4.1.1 for this mapping. Theoretically the hash table can be set with fixed size, but in practice we use a 2-level hash table since it leads to better performance when the frontier size is consistently small, and equal performance otherwise. This map is referred as ***vertex map*** later.

The I/O cost of phased Dijkstra is $Q(n,m) = O\left(n\left(\frac{m}{M}+\omega\right)\right)$. More details on the correctness and complexity analysis can be found in Section 5.4.1.

We make a special optimization that once all outgoing edges of a vertex are visited, we remove this node and all associated edges in the scan in Line 5. This is done by using the sign-bit of the output distances, such that it is more likely being cached. We call the ***active set*** that contains visited but not removed vertices.

For an efficient implementation that optimizes memory footprint and I/O efficiency, each heap element contains the vertex index, the pointer to the vertex map, and the tentative distance. In total it takes 16 bytes. Each element in the vertex map only stores a pointer, and the vertex index can be check via the corresponding heap element. Throughout our experiment we set the maximum occupancy rate of the 2-level hash table to be 0.8, and truncation ratio $\epsilon = 0.25$. M' is chosen such that the hash table and priority queue occupy about 40% of the small-memory, while various values of M' are also discussed.

8.6.2.3 Experiments

The experiments are run on two Dijkstra implementations with different parameter combinations on cache size, cache policy, and the priority queue size. The experiment is run on eight graph instances with various cache size.

Since in phased Dijkstra the number of reads is significantly more than that of writes, to keep the priority queue in the cache, the special cache strategy in Section 8.3.2 is

	Classic Dijkstra using binary heap						Phased Dijkstra								
Cache Size	**2k**		**10k**		**50k**		**2k**			**10k**			**50k**		
Graph Instance	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**		**RT**	**WT**		**RT**	**WT**	
2D Grid	10.37	4.87	4.61	2.34	3.53	1.89	5.93	1.12	(1)	3.62	1.11	(1)	2.74	1.04	(1)
3D Grid	25.87	14.53	17.44	8.54	8.20	3.35	126.56	1.12	(383)	28.16	1.12	(44)	5.90	1.07	(1)
PA Roadmap	7.82	4.14	2.16	0.81	1.12	0.40	30.40	0.99	(202)	1.89	0.52	(1)	1.04	0.28	(1)
TX Roadmap	7.69	4.10	2.16	0.82	1.09	0.40	37.93	0.97	(262)	1.88	0.53	(1)	1.01	0.27	(1)
Stan Webgraph	37.31	18.13	26.73	12.06	10.60	3.91	404.61	1.03	(100)	81.02	1.02	(16)	13.83	0.88	(2)
NDU Webgraph	24.97	15.34	15.14	8.21	5.19	1.65	99.86	1.10	(132)	22.57	1.01	(24)	5.81	0.67	(3)
DBLP Network	32.14	19.66	23.17	13.02	10.08	4.58	341.08	1.12	(131)	64.75	1.10	(24)	11.53	0.94	(4)
Youtube Network	35.19	22.86	26.80	16.53	21.18	12.90	836.64	1.13	(477)	161.36	1.11	(93)	33.08	1.05	(18)

Table 8.14: Number of read and write transfers of different Dijkstra implementations on different cache size. Numbers are normalized to read or write transfers per vertex. We ran SSSP queries on 10 different randomly-chosen source nodes. Numbers in the parentheses are the average phases.

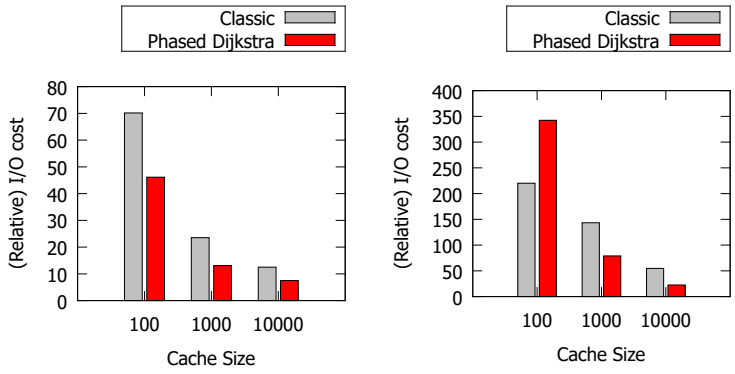
	$\omega = 10$						$\omega = 100$					
	Classic			**Phased Dijkstra**			**Classic**			**Phased Dijkstra**		
Cache Size	**2k**	**10k**	**50k**	**2k**	**10k**	**50k**	**2k**	**10k**	**50k**	**2k**	**10k**	**50k**
2D Grid	59.1	28.0	22.4	17.1	14.7	13.1	497.8	238.9	192.2	118.3	114.3	106.8
3D Grid	171.2	102.8	41.7	137.8	39.3	16.6	1478.9	871.2	343.5	239.0	140.1	113.1
PA Roadmap	49.2	10.2	5.1	40.3	7.1	3.8	421.4	83.1	41.2	129.0	53.9	28.7
TX Roadmap	48.7	10.4	5.1	47.6	7.1	3.7	417.5	84.3	40.9	134.5	54.5	28.3
Stan Webgraph	*218.6*	147.3	49.7	414.9	91.2	22.6	1850.2	1232.2	401.8	507.7	182.8	101.9
NDU Webgraph	178.4	97.2	21.6	110.9	32.7	12.5	1559.4	836.3	169.7	209.9	123.9	72.8
DBLP Network	*228.7*	153.4	55.9	352.3	75.7	20.9	1997.6	1325.2	468.4	453.1	174.7	105.9
Youtube Network	*263.8*	192.1	150.1	847.9	172.5	43.6	2321.0	1679.4	1310.8	949.3	272.8	138.3

Table 8.15: The I/O costs on different Dijkstra implementations on different cache size. The write-read ratio ω are selected to be typical projected values 10 (latency, bandwidth) and 100 (energy). Results are based on the numbers in Table 8.14.

required. In Table 8.16 we show that different cache policies only cause minor differences, so in the majority of this section we use the **Static** policy.

Overall performance. In Table 8.14 we show the number of read and write transfers of two implementations on different graphs with various cache size. Cache size varies from 2,000 to 50,000 cache lines each with 64 bytes. In Table 8.15 the overall I/O costs with different values of ω are listed based on the numbers in Table 8.14.

For the binary-heap implementation, the actual reads and writes mostly match the theoretical bound $O(m \log(nB/M))$. Reads are about slightly less than twice as writes: each edge is read once during Dijkstra (linear scan and requires no modification), and an



(a) $\omega = 10$, sparse graphs (b) $\omega = 10$, social networks

(c) $\omega = 100$, sparse graphs (d) $\omega = 100$, social networks

Figure 8.7: I/O costs of classic Dijkstra and phased Dijkstra. Graphs are categorized into sparse (almost planar) graphs and social networks and the I/O cost is geometric mean of the four. Numbers are from Table 8.15.

update in the binary heap always requires one more read than writes. The only exception is on the roadmaps when the frontier size is consistently small while nearby vertices share contiguous indices. In this case the cache efficiently holds the entire heap and leads to fewer reads and writes to the large-memory.

For phased Dijkstra, we first observed that the number of writes is always no more than 1.3 per vertex: one write per vertex when the distance is finalized, plus some other cost to maintain the active set. The number of read transfers is mainly decided the number of phases, and the size of the active set (the edge lists of active vertices at the beginning of each phase needs to be scanned).

The overall I/O performance (shown in Table 8.15) indicates that phased Dijkstra is consistently better than the binary-heap version except for the only case that both ω and cache size are small and the active set is large. This case can hardly happen in practice

since the cache size is even much smaller than the current L3 cache. The improvement on I/O cost in all cases is up to 3 and 7.6 when ω is 10 and 100. This peak occurs when the small-memory sizes is large and frontier size is larger, since at this time the cost to maintain the binary heap in the classic implementation is costly, while the extra cost to run phased Dijkstra is insignificant.

Cache Policy	SplitPool		Static	
	RT	WT	RT	WT
2D Grid	3.74	1.11	3.62	1.11
3D Grid	27.43	1.12	28.16	1.12
PA Roadmap	1.84	0.88	1.89	0.52
TX Roadmap	1.83	0.87	1.88	0.53
Stan Webgraph	81.40	1.02	81.02	1.02
NDU Webgraph	21.96	1.07	22.57	1.01
DBLP Network	65.20	1.11	64.75	1.10
Youtube Network	163.38	1.12	161.36	1.11

Table 8.16: Number of read and write transfer of phased Dijkstra with two different cache policies: the **SplitPool** policy and the **Static** policy. The cache contains 10,000 cache-lines. Different policies give very similar performance except for the writes on roadmaps.

Priority Queue Size	30%		40%		50%		60%	
	RT	WT	RT	WT	RT	WT	RT	WT
2D Grid	8.4	1.10	8.6	1.11	8.9	1.11	9.3	1.11
3D Grid	33.7	1.12	28.9	1.12	25.6	1.12	23.3	1.12
PA Roadmap	2.7	0.48	2.8	0.52	3.0	0.58	3.2	0.66
TX Roadmap	2.6	0.48	2.7	0.53	2.9	0.59	3.1	0.66
Stan Webgraph	94.4	1.02	82.9	1.02	72.1	1.02	67.1	1.02
NDU Webgraph	26.1	1.01	22.8	1.01	20.2	1.03	18.5	1.04
DBLP Network	75.1	1.10	64.9	1.10	57.3	1.10	51.5	1.11
Youtube Network	183.8	1.11	161.5	1.11	144.4	1.12	134.9	1.12

Table 8.17: Number of read and write transfers of phased Dijkstra with different priority queue's size. The overall percentage of priority queue and vertex map varies from 30% to 60% comparing to the cache size, which is 10,000 cache-lines.

Different cache maintenance policy. We show the number of read and write transfer of phased Dijkstra on two different cache policies in 8.16. Different policies actually give

a very similar performance in the graph instances. More analysis is shown in the full version of this paper.

Picking appropriate priority queue's size M'. In previous experiment we empirically set the overall size of the priority queue and vertex map to be about 40% of the overall cache size. However, this percentage can affect the performance of phased Dijkstra. Larger percentage leads the heap to contain more elements so that the overall number of phases is decreased. Meanwhile the size left for the rest cache is smaller, which decreases the cache performance. Hence, the number of phases and specific graph property lead to different optimality point of the the priority queue's size and no easy conclusions can be drawn. In Table 8.17 we show a snapshot on the cache size of 10,000 and the percentage varies from 30% to 60%. The trend is that, when the priority queue's size is smaller than the average frontier size then larger priority queue helps, and vice visa. In general different priority queue's sizes only make a minor difference on I/O cost and will not affect the relatively lower cost comparing to the binary-heap implementation.

8.6.2.4 Conclusions

We discussed phased Dijkstra and experiment its performance on a variety of graphs. The high-level idea is to fit the computation within the small-memory (i.e., the cache) and thus requires no intermediate writes to the large asymmetry memory. The experimental results show that phased Dijkstra consistently outperforms the binary-heap version on I/O cost except for the combination of small ω (= 10), small cache size, and on social networks. Although phased Dijkstra contains some parameters, we also show that they do not affect the efficiency of phased Dijkstra when they are within a reasonable large range. A similar case also holds for different cache policies.

Notice that the idea here that fits the computation in the small-memory can also be applied to computing minimum spanning tree, sorting, and many other problems.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

This thesis provides a comprehensive study of write-efficient algorithms, from computational models, to lower and upper bounds, experiments, as well as guaranteeing the fault-tolerance of the write-efficient algorithms. These write-efficient algorithms will have improved performance on the new non-volatile main memory with asymmetric read and write costs, or can provide additional functionality with little costs in other recent settings that may lead to write-read asymmetry.

This thesis discusses how to extend the most commonly-used computational models and cost measures used in analyzing classic algorithms to the asymmetric setting in Chapter 2, which provides the theoretical approaches to analyze the algorithms asymptotically. Based on the models, researches can analyze lower bounds of the problems in the asymmetric setting, and a list of known results are shown in Chapter 3 and 7.

The main contribution of this thesis consists of a list of new write-efficient algorithms. They are widely spread from the basic algorithmic building blocks (sorting, reduce, filter, etc.) in Chapter 4, to graph algorithms (e.g., (bi)connectivity, shortest paths, MST, BFS) in Chapter 5, geometric algorithms and data structures (e.g., convex hull, Delaunay triangulation, k-d trees, augmented trees) in Chpater 6, as well as many cache-oblivious algorithms for dynamic programming and linear algebra problems 7. All these algorithms have asymptotically improved cost under the sequential (M, ω)-ARAM or the parallel Asymmetric NP model. Also, most of these algorithms are highly parallelized. Since the new memories provide a much larger (terabyte-level) capacity, parallelism is a necessity to process data with such volume.

It is interesting to point out that, this thesis also introduces many new techniques and analysis frameworks on many types of algorithms. Each of them not only leads to many new write-efficient algorithms, but also is of independent interests since many results in the symmetric settings on these problems can also be improved based on it.

This thesis also contains experiment verifications of the write-efficient algorithms in Chapter 8. This part contains the first experimental framework to analyze the performance of write-efficient algorithms in practice, and experiments on a variety of algorithms which leads to many interesting findings and lessons.

9.2 Future Work

This thesis consolidates the research of write-efficient algorithms, which is a relatively new research topic. It is notable that this is still a very open area, and there are many new problems worth to be investigated. A few of the interesting future topics are listed here.

Actual Performance.

The new hardware has been manufactured at the time when this thesis is being written. As a result, it is reasonable to assume that the new hardware will be available to the public very soon. When it becomes available, implementing the write-efficient algorithms and testing their actual wall-clock performance is definitely one of the works that should be investigated. As discussed in Chapter 8, the software simulator for write-efficient algorithms is still a useful tool even when the hardware is available. However, it is interesting to know the best parameter to set in the simulator that can fit the performance in practice the best. Also, it is of notable importance to have the highly-optimized implementations to be in an available library, so researchers or other programmers whose work is not performance engineering can directly call the functions in the library.

Write-efficient algorithms.

This thesis discusses a few dozens of new write-efficient algorithms, but this is still a relatively small amount compared to the entity of the algorithms that are widely being used in practice. The write-efficient versions of these algorithms remain to be a future research topic.

Other than designing write-efficient algorithms for other problems, the new problems raised by studying the write-efficient algorithms in this thesis can also lead to potential future work. For example, many new problems are proposed by the analysis framework of the k-d grid computation structure in Section 7.9. There are many unanswered questions for geometric algorithms as well.

This thesis discusses the implicit k-decomposition and the compact representations for connectivity and biconnectivity information of a graph. It is interesting to know whether such kinds of compact representations of other graph problems, or even data structure and other problems (like a triangulation). A further question is the tradeoff between the space requirement, the number of writes, and the total algorithmic instructions for each problem. It remains unknown if one can show the lower bounds of a combination of these terms for graph connectivity, and we can ask similar questions to other graph problems as well.

Persistency and fault-tolerance.

As discussed in Section 1.1.2, another major property of these new main memories are their non-volatility or persistency: unlike DRAM, they have the capability of surviving power outages and other failures without losing data. As a result, it is possible to design the special programming model and algorithms that are resilient to either the processor fault or the power lost.

The consistent states of the intermediate data stored in the persistent main memory need to be guaranteed such that either a single processor or the entire program can be restarted from here. On the other hand, standard caches are write-back (write-behind), meaning that a write to a memory location will make it as far as the cache, until at some later point the updated cache line gets flushed out to the persistent memory. Usually the programmers have no control of this process. Therefore, a fault-tolerant algorithm running on the persistent main memory needs to explicitly flush some of the cache lines to guarantee the desired states of the data in the persistent memory, using instructions (such as Intel's `CLFLUSH` instruction) supported by various programming models (e.g., [178, 240, 241]). Such instructions require memory barriers to enforce the ordering in the execution, which leads to extra cost for these writes such as in synchronizing the processors. As a result, the new non-volatile main memory provides the option for algorithms to programs to be fault resilient, but maintaining the memory consistency can cause the costly writes on these memories to be even more expensive. Write-efficient algorithms can be efficient in such settings because of the fewer overall writes that need to be flushed to the persistent main memory.

However, as a first step, we need to first have a programming model to support such fault-tolerant programming. Such a model is introduced by Belleoch et al. [71], which is named as the *Parallel Persistent Memory (Parallel-PM) model*. Based on this model, we can discuss how to make algorithms to be fault-tolerant, with or without considering the more expensive writes.

The Parallel Persistent Memory (Parallel-PM) model, consists of P processors, each with a fast local ephemeral memory of limited size M, and sharing a large slower persistent memory. As in the external memory model [29, 30], each processor runs a standard instruction set from its ephemeral memory and has instructions for transferring blocks of size B to and from the persistent memory. The cost of an algorithm is calculated based on the number of such transfers, and the cost can be either symmetric or asymmetric. A key difference, however, is that the model allows for individual processors to fault at any time. If a processor faults, all of its processor state and local ephemeral memory is lost, but the persistent memory remains. We consider both the case where the processor restarts (soft faults) and the case where it never restarts (hard faults).

The model is motivated by two complimentary trends. Firstly, it is motivated by upcoming non-volatile memories that are nearly as fast as existing random access memory (DRAM), are accessed via loads and stores at the granularity of cache lines, and have large

capacity (more bits per unit area than existing random access memory). For example, Intel's 3D-Xpoint memory technology, currently available as an SSD, is scheduled to be available as such a random access memory in 2019. While such memories are expected to be the pervasive type of memory [210, 221, 283], each processor will still have a small amount of cache and other fast memory implemented with traditional *volatile* memory technologies (SRAM or DRAM). Secondly, it is motivated by the fact that in current and upcoming large parallel systems the probability that an individual processor faults is not negligible, requiring some form of fault tolerance [83].

In this model, a single processor version, the *PM model*, is first discussed, which gives conditions under which programs are robust against faults. In particular, we identify that breaking a computation into "capsules" that have no write-after-read conflicts (writing a location that was read earlier within the same capsule) is sufficient, when combined with our approach to restarting faulting capsules from their beginning, due to its idempotent behavior. We then show that RAM algorithms, external memory algorithms, and cache-oblivious algorithms [131] can all be implemented asymptotically efficiently on the model. This involves a simulation that breaks the computations into capsules and buffers writes, which are handled in the next capsule. However, the simulation is likely not practical. We therefore consider a programming methodology in which the algorithm designer can identify capsule boundaries, and ensure that the capsules are free of write-after-read conflicts.

Then the multiprocessor counterpart, the Parallel-PM described above, is considered, which gives conditions under which programs are correct when the processors are interacting through the shared memory. We identify that if capsules are free of write-after-read conflicts and atomic, in a way that we define, then each capsule acts as if it ran once despite many possible restarts. Furthermore we identify that a compare-and-swap (CAS) instruction is not safe in the PM model, but that a compare-and-modify (CAM), which does not see its result, is safe.

The most significant result of this work is a work-stealing scheduler that can be used on the Parallel-PM. The scheduler is based on the scheduler of Arora, Blumofe, and Plaxton (ABP) [30]. The key challenges in adopting it to handle faults are (i) modifying it so that it only uses CAMs instead of CASs, (ii) ensuring that each stolen task gets executed despite faults, (iii) properly handling hard faults, and (iv) ensuring its efficiency in the presence of soft or hard faults. Without a CAS, and to avoid blocking, handling faults requires that processors help the processor that is part way through a steal. Handling hard faults further requires being able to steal a thread from a processor that was part way through executing the thread.

Based on the scheduler we show that any race-free, write-after-read conflict free multithreaded fork-join program with work W, depth D, and maximum capsule work C

will run in expected time:

$$O\left(\frac{W}{P_A} + kD\left(\frac{P}{P_A}\right)\left\lceil \log_{1/(Cf)} W \right\rceil\right).$$

Here k is the ratio of whether the reads and writes are symmetric (1 for symmetric setting, ω for the asymmetric setting), P is the maximum number of processors, P_A the average number, and $f \leq 1/(2C)$ an upper bound on the probability a processor faults between successive persistent memory accesses. This bound differs from the ABP result only in the $\log_{1/(Cf)} W$ factor on the depth term, due to faults along the critical path.

Based on the models, it is interesting to design Parallel-PM algorithms. A few examples are already discussed, while it remains to be an interesting future work about how to adapt the write-efficient algorithms in this thesis to be fault-tolerant, and whether there are general high-level approaches to make this happen.

Appendix A

Detail Information of Asymmetric Memory

While DRAM stores data in capacitors that typically require refreshing every few milliseconds, and hence must be continuously powered, emerging NVM technologies store data as "states" of the given material that require no external power to retain. Energy is required only to read the cell or change its value (i.e., its state). While there is no significant cost difference between reading and writing DRAM (each DRAM read of a location not currently buffered requires a write of the DRAM row being evicted, and hence is also a write), emerging NVMs incur significantly higher cost for writing than reading. This large gap seems fundamental to the technologies themselves: to change the physical state of a material requires relatively significant energy for a sufficient duration, whereas reading the current state can be done quickly and, to ensure the state is left unchanged, with low energy. Existing research has shown such asymmetry, for example:

- A cell in Spin-Torque Transfer Magnetic RAM can be read in 0.14 ns but uses a 10 ns writing pulse duration, using roughly 10^{-15} joules to read versus 10^{-12} joules to write [117] (these are the raw numbers at the materials level).
- A Memristor Resistive RAM cell uses a 100 ns write pulse duration, and an 8MB Memristor RRAM chip is projected to have reads with 1.7 ns latency and 0.2 nJ energy versus writes with 200 ns latency and 25 nJ energy [280], over two orders of magnitude differences in latency and energy.
- Phase-change memory is the most mature of the three technologies, and early generations are already available as I/O devices. A recent paper [187] reported 6.7 μs latency for a 4KB read and 128 μs latency for a 4KB write. Another reported that the sector I/O latency and bandwidth for random 512B writes was a factor of 15 worse than for reads [176]. As a future memory/cache replacement, a 512MB PCM memory chip is projected to have 16 ns byte reads versus 416 ns byte writes, and

writes to a 16MB PCM L3 cache are projected to be up to 40 times slower and use 17 times more energy than reads [118].

While these numbers are speculative and subject to change as the new technologies emerge over time, there seems to be sufficient evidence that writes will be considerably more costly than reads in these NVMs. Thus, studying write-efficient (parallel) algorithms and systems is of significant, lasting importance.

Note that, unlike SSDs and earlier versions of phase-change memory products, these emerging memory products will sit on the processor's memory bus and be accessed at byte granularity via loads and stores (like DRAM). Thus, the time and energy for reading can be roughly on par with DRAM, and depends primarily on the properties of the technology itself relative to DRAM.

Bibliography

[1] Ittai Abraham and Ofer Neiman. Using petal-decompositions to build a low stretch spanning tree. In *ACM Symposium on Theory of computing (STOC)*, pages 395–406, 2012. 5.3.2

[2] Umut Acar, Guy Blelloch, and Robert Blumofe. The data locality of work stealing. *Theory Comput. Sys.*, 35(3), 2002. 1.2, 2.2.4, 2.3.5

[3] Stephen Adams. Implementing sets effciently in a functional language. Technical Report CSTR 92-10, University of Southampton, 1992. 8.4.2.2

[4] Stephen Adams. Efficient sets—a balancing act. *Journal of functional programming*, 3(04):553–561, 1993. 8.4.2.2

[5] Georgy Adelson-Velsky and Evgenii Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963. 8.4.2, 8.4.2.1

[6] Alok Aggarwal and Maria Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 27(1-2), 1990. 7.8.1

[7] Alok Aggarwal and Jeffrey Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1988. 1, 1.2, 2.1.2, 2.3.1, 3.2, 4.4.3.1, 4.4.3.2, 6.7, 6.7.3, 8.1, 8.3.1

[8] Alok Aggarwal, Ashok Chandra, and Marc Snir. Communication complexity of prams. *Theor. Comput. Sci.*, 71(1):3–28, 1990. 7.2

[9] Alfred Aho and John Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974. 7.2

[10] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974. 7.7.3

[11] Miklos Ajtai and Nimrod Megiddo. A deterministic $poly(\log\log n)$-time n-processor algorithm for linear programming in fixed dimension. In *ACM Symposium on Theory of Computing (STOC)*, pages 327–338, 1992. 6.8.1

[12] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *ACM Symposium on Theory of Computing (STOC)*, 1983. 3.3

[13] Deepak Ajwani, Nodari Sitchinava, and Norbert Zeh. Geometric algorithms for private-cache chip multiprocessors. In *European Symposium on Algorithms (ESA)*, 2010. 6.7

[14] Ameen Akel, Adrian Caulfield, Todor Mollov, Rajech Gupta, and Steven Swanson. Onyx: A prototype phase change memory storage array. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2011. 1.1.1, 8.2

[15] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008. 6.6

[16] Yaroslav Akhremtsev and Peter Sanders. Fast parallel operations on search trees. In *International Conference on High Performance Computing (HiPC)*, Dec 2016. 6.7.3.5

[17] Noga Alon and Nimrod Megiddo. Parallel linear programming in fixed dimension almost surely in constant time. *J. ACM*, 41(2):422–434, March 1994. 6.8.1

[18] Noga Alon and Baruch Schieber. Optimal preprocessing for answering on-line product queries. *Technical Report, Tel Aviv University*, 1987. 16

[19] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *ACM Symposium on Theory of computing (STOC)*, pages 20–29. ACM, 1996. 1.1.3

[20] Mahdi Amani, Kevin Lai, and Robert Tarjan. Amortized rotation cost in AVL trees. *Information Processing Letters*, 116(5):327–330, 2016. 8.4.2.1

[21] Richard Anderson and Gary Miller. A simple randomized parallel algorithm for list-ranking. *Inf. Proc. Letters*, 1990. 4.3.1

[22] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 120–124, 2004. 5.3.2

[23] Alex Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inf. Proc. Letters*, 1979. 6.9

[24] Cecilia Aragon and Raimund Seidel. Randomized search trees. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 540–545, 1989. 8.4.2, 8.4.2.1

[25] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1), 2003. 4.4.3.3, 1, 4.4.3.3, 4.4.3.3, 7.4.2

[26] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6), 2003. 6.7, 6.7.3

[27] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. In *ACM Symposium on Principles of Database Systems (PODS)*, 1999. 6.7, 6.7.1.3, 6.7.3

[28] Lars Arge, Gerth Stølting Brodal, and Rolf Fagerberg. Cache-oblivious data structures. *Handbook of Data Structures and Applications*, 27, 2004. 7.1

[29] Lars Arge, Michael Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *ACM Symposium on Parallelism in algorithms and architectures (SPAA)*, 2008. 9.2

[30] Nimar Arora, Robert Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2), Apr 2001. 2.2.4, 2.3.3, 9.2

[31] Mikhail Atallah and Micheal Goodrich. Deterministic parallel computational geometry. In *Synthesis of Parallel Algorithms*, pages 497–536. Morgan Kaufmann, 1993. 6.5

[32] Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth Ross. Path processing using solid state storage. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2012. 1.1.1, 8.2

[33] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985. 5.3.2

[34] Baruch Awerbuch, Michael Luby, Andrew Goldberg, and Serge Plotkin. Network decomposition and locality in distributed computation. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–369, 1989. 5.3.2

[35] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Low-diameter graph decomposition is in NC. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 83–93. Springer, 1992. 5.3.2

[36] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Communication-optimal parallel and sequential cholesky decomposition. *SIAM J. Scientific Computing*, 32(6):3495–3523, 2010. 7.2

[37] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3): 866–901, 2011. 7.2

[38] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972. 1

[39] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972. 8.4.2, 8.4.2.1

[40] Richard Bellman. *Dynamic programming*. Princeton University Press, 1957. 7.2

[41] Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash-memory algorithms. In *European Symposium on Algorithms (ESA)*, 2006. 1.1.1, 8.1, 8.2

[42] Naama Ben-David, Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016. 1.6

[43] Naama Ben-David, Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2018. 1.6

[44] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 1975. 6.6

[45] Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5), 1979. 6.7.1.2

[46] Petra Berenbrink, Bruce Krayenhoff, and Frederk Mallmann-Trenn. Estimating the number of connected components in sublinear time. *Information Processing Letters*, 114(11), 2014. 5.3.2

[47] Omer Berkman and Uzi Vishkin. Recursive *-tree parallel data-structure. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1989. 6.7.2.1

[48] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993. 5.2

[49] Daniel Blandford, Guy Blelloch, and Clemens Kadow. Engineering a compact parallel Delaunay algorithm in 3D. In *ACM Symposium on Computational Geometry (SoCG)*, pages 292–300, 2006. 6.1

[50] Guy Blelloch and Phillip Gibbons. Effectively sharing a cache among threads. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2004. 1.2, 2.3.5

[51] Guy Blelloch, Jonathan Hardwick, Gary Miller, and Dafna Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3-4): 243–269, 1999. 6.5

[52] Guy Blelloch. Nesl: A nested data-parallel language. Technical report, Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, 1992. 2.2.2

[53] Guy Blelloch. Programming parallel algorithms. *Commun. ACM*, 39, March 1996. 5.4.3

[54] Guy Blelloch and Laxman Dhulipala. Introduction to parallel algorithms. *Lecture note for 15-853: Algorithms in the Real World.* 2.2.4, 2.2.4

[55] Guy Blelloch and Margaret Reid-Miller. Pipelining with futures. *Theory of Computing Systems*, 32(3), 1999. 6.7.3.5

[56] Guy Blelloch, Charles Leiserson, Bruce Maggs, C. Greg Plaxton, Stephen Smith, and Marco Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *SPAA*, 1991. 11, 4.4.3.2

[57] Guy Blelloch, Phillip Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2), March 1999. 2.2.4

[58] Guy Blelloch, Phillip Gibbons, and Harsha Vardhan Simhadri. Low-depth cache oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 189–199, 2010. 2.3.5

[59] Guy Blelloch, Phillip Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 189–199, 2010. 1.2, 1.4.4, 7.1, 7.2, 7.6.1, 7.6.1, 7.8.2, 7.8.2

[60] Guy Blelloch, Jeremy Fineman, Phillip Gibbons, and Julian Shun. Internally deterministic algorithms can be fast. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 181–192, 2012. 1.1.3, 6.1

[61] Guy Blelloch, Jeremy Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 308–317, 2012. 1.4.3, 6.1, 6.2

[62] Guy Blelloch, Anupam Gupta, and Kanat Tangwongsan. Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design. In *ACM Symposium on Parallelism in algorithms and architectures (SPAA)*, 2012. 6

[63] Guy Blelloch, Anupam Gupta, Ioannis Koutis, Gary Miller, Richard Peng, and Kanat Tangwongsan. Nearly-linear work parallel sdd solvers, low-diameter decomposition, and low-stretch subgraphs. *Theory of Computing Systems*, 55(3):521–554, 2014. 5.3.2

[64] Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015. 1.6, 7.1

[65] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264. ACM, 2016. 6.7.2.1, 6.7.3.5, 8.4.2.1, 8.4.2.2, 8.4.2.3

[66] Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *arXiv preprint:1603.03505*, 2016. 1.4.4, 7.1

[67] Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Yan Gu, and Julian Shun. Efficient algorithms with asymmetric read and write costs. In *European Symposium on Algorithms (ESA)*, pages 14:1–14:18, 2016. 1.6

[68] Guy Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in algorithms and architectures (SPAA)*, 2016. 1.6, 4.4, 6.5.3, 6.5.3

[69] Guy Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. Parallel shortest paths using radius stepping. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 443–454, 2016. 1.6, 8.6.2

[70] Guy Blelloch, Yan Gu, and Yihan Sun. Efficient construction of probabilistic tree embeddings. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2017. 1.6, 6

[71] Guy Blelloch, Phillip Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018. 1.6, 9.2

[72] Guy Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in algorithms and architectures (SPAA)*, 2018. 1.6

[73] Robert Blumofe, Matteo Frigo, Christopher Joerg, Charles Leiserson, and Keith Randall. An analysis of DAG-consistent distributed shared-memory algorithms. In *ACM Symposium on Parallelism in algorithms and architectures (SPAA)*, 1996. 1.2

[74] Robert Blumofe and Charles Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM (JACM)*, 46(5), September 1999. 2.2.4, 2.2.4, 2.3.3, 2.3.3

[75] Robert Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Usenix HotPar*, 2009. 6.1

[76] Jean-Daniel Boissonnat and Monique Teillaud. On the randomized construction of the delaunay tree. *Theoretical Computer Science*, 112(2):339–354, 1993. 6.3, 6.3.1, 6.5, 15, 6.5.1

[77] Charles Bouton. Nim, a game with a complete mathematical theory. *The Annals of Mathematics*, 3(1/4), 1901. 7.3

[78] Richard Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974. 2.2.4, 2.3.3

[79] Gerth Stølting Brodal. Cache-oblivious algorithms and data structures. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 3111. Springer, 2004. 7.1

[80] Zoran Budimlić, Vincent Cavé, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Jisheng Zhao, and Vivek Sarkar. The design and implementation of the habanero-java parallel programming language. In *Proceedings of the ACM international Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 185–186, 2011. 2.2.2

[81] Paul Callahan and S Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)*, 42(1), 1995. 6.6.3

[82] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008. 6.1

[83] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov.: Int. J.*, 1(1), April 2014. 1.1.2, 9.2

[84] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. Write-avoiding algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 648–658, 2016. 1.1.1, 1.1.2, 7.1, 7.2, 8.2

[85] Timothy Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4), 1996. 6.9.1, 1

[86] Timothy Chan. Backwards analysis of the Karger-Klein-Tarjan algorithm for minimum spanning trees. *Inf. Proc. Letters*, 67(6), 1998. 5

[87] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 40, pages 519–538, 2005. 2.2.2

[88] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34(6), 2005. 5.3.2

[89] Danny Chen and Jinhui Xu. Two-variable linear programming in parallel. *Computational Geometry*, 21(3):155 – 165, 2002. 6.8.1

[90] Shimin Chen, Phillip Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *Conference on Innovative Data Systems Research (CIDR)*, 2011. 1.1.1, 8.1, 8.2, 8.3.2

[91] Sangyeun Cho and Hyunjin Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009. 1.1.1, 8.2

[92] Rezaul Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley Kuszmaul, Charles Leiserson, Armando Solar-Lezama, and Yuan Tang. Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016. 1.1.3, 1.4.4, 7.1, 7.2, 7.8, 7.8.2, 7.8.4

[93] Rezaul Chowdhury. Cache-efficient algorithms and data structures: Theory and experimental evaluation. *PhD Thesis, UT Austin*, 2007. 7.2

[94] Rezaul Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. In *ACM-SIAM Symposium on Discrete algorithm (SODA)*, 2006. 1.1.3, 1.4.4, 7.1, 7.2, 7.7.2, 7.8, 7.8.1, 7.8.2, 7.8.2

[95] Rezaul Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *ACM Symposium on Parallelism in algorithms and architectures (SPAA)*, pages 207–216, 2008. 7.2

[96] Rezaul Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, 47(4), 2010. 1.4.4, 7.1, 7.2, 7.7.2, 7.8, 7.8.2

[97] Rezaul Chowdhury, Hai-Son Le, and Vijaya Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 7(3):495–510, 2010. 7.2

[98] Paolo Cignoni, Claudio Montani, Raffaele Perego, and Roberto Scopigno. Parallel 3D delaunay triangulation. *Computer Graphics Forum*, 12(3):129–142, 1993. 6.1

[99] Marcelo Cintra, Diego Llanos, and Belén Palop. International conference on computational science and its applications. In *Speculative Parallelization of a Randomized Incremental Convex Hull Algorithm*, pages 188–197, 2004. 6.1

[100] Kenneth Clarkson and Peter Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(5):387–421, 1989. 6.2, 6.5, 6.5.1

[101] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997. 6.1

[102] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988. 4.4, 11, 6.4.2

[103] Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. In *International Colloquium on Automata, Languages, and Programming (ICALP)*. Springer, 2010. 7.4.2

[104] Richard Cole, Philip N. Klein, and Robert Endre Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 243–250, 1996. 5.3.1, 5.3.4, 4, 16

[105] Jeremy Condit, Edmund Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009. 8.3.2

[106] Stephen Cook and Ravi Sethi. Storage requirements for deterministic polynomial time recognizable languages. *Journal of Computer and System Sciences*, 13(1), 1976. 3.4

[107] Don Coppersmith, Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. A divide-and-conquer algorithm for identifying strongly connected components. Technical

Report RC23744, IBM, 2003. 6.1

[108] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009. 5.3.1, 5.3.5.1, 5.3.5.2, 7, 5.4.2, 6.7.2, 6.7.3, 7.2, 7.8.4

[109] Mark de Berg, Otfried Cheong, Mark van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008. 6.5, 6.6, 6.6.1, 6.7.1.1, 6.7.1.3, 7, 6.7.2, 6.7.3

[110] Erik Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4), 2002. 7.1

[111] Xiaotie Deng. An optimal parallel algorithm for linear programming in the plane. *Information Processing Letters*, 35(4):213 – 217, 1990. 6.8.1

[112] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018. 1.4.2

[113] Pedro Diaz, Diego Llanos, and Belen Palop. Parallelizing 2D-convex hulls on clusters: Sorting matters. *Jornadas De Paralelismo*, 2004. 6.1

[114] Edsger Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1), 1959. 5.4.1, 5.4.1, 8.6.2

[115] David Dinh, Harsha Vardhan Simhadri, and Yuan Tang. Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016. 7.1, 7.2, 7.7, 7.8

[116] Brandon Dixon, Monika Rauch, and Robert Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. on Computing*, 21(6), 1992. 5.4.2.2, 16

[117] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Hai Li, and Yiran Chen. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *ACM Design Automation Conference (DAC)*, 2008. 1.1.1, 8.2, A

[118] Xiangyu Dong, Norman Jouupi, and Yuan Xie. PCRAMsim: System-level performance, energy, and area modeling for phase-change RAM. In *ACM International Conference on Computer-Aided Design (ICCAD)*, 2009. 1.1.1, 8.2, A

[119] Martin Dyer. A parallel algorithm for linear programming in fixed dimension. In *Symposium on Computational Geometry (SoCG)*, pages 345–349, 1995. 6.8.1

[120] Herbert Edelsbrunner. *Dynamic data structures for orthogonal intersection queries*. Technische Universität Graz/Forschungszentrum Graz. Institut für Informationsverarbeitung, 1980. 6.7.1.1, 6.7.2, 6.7.3

[121] Herbert Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge University Press, 2006. 6.5

[122] David Eppstein and Zvi Galil. Parallel algorithmic techniques for combinatorial computation. *International Colloquium on Automata, Languages, and Programming (ICALP)*, 1989. 7.2

[123] David Eppstein, Michael Goodrich, Michael Mitzenmacher, and Pawel Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In *ACM International Symposium on Experimental Algorithms (SEA)*, 2014. 1.1.1, 8.1, 8.2

[124] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, near linear time. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2001. 4.5.1

[125] Michael Fischer and Albert Meyer. Boolean matrix multiplication and transitive closure. In *IEEE Symposium on Switching and Automata Theory*, 1971. 7.7.3

[126] Robert Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962. 7.7.3

[127] W Donald Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3), 1970. 11, 4.4.3.2

[128] Michael Fredman and Robert Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3), 1987. 5.4.1, 8.6.2

[129] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *ACM International Conference on Supercomputing*, pages 361–366, 2005. 7.2

[130] Matteo Frigo, Charles Leiserson, and Keith Randall. The implementation of the cilk-5 multithreaded language. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 33(5):212–223, 1998. 2.2.2

[131] Matteo Frigo, Charles Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999. 1, 1.2, 1.4.4, 2.1.3, 2.3.4, 7.1, 7.2, 7.4.1, 7.4.2, 7.5, 12, 7.6.1, 7.6.1, 7.8, 9.2

[132] ME Furman. Application of a method of fast multiplication of matrices to problem of finding graph transitive closure. *Doklady Akademii Nauk SSSR*, 194(3), 1970. 7.7.3

[133] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2), 2005. 1.1.1, 8.1, 8.2

[134] Zvi Galil and Raffaele Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64(1), 1989. 7.2, 7.8.2

[135] Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1), 1992. 7.8.1, 7.8.4

[136] Zvi Galil and Kunsoo Park. Parallel algorithms for dynamic programming recurrences with more than O(1) dependency. *Journal of Parallel and Distributed Computing*, 21(2), 1994. 7.2, 7.8.1, 7.8.2, 7.8.3, 7.8.4

[137] Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, December 1991. 5.3.1, 5.3.4

[138] Hillel Gazit, Gary Miller, and Shang-Hua Teng. *Optimal tree contraction in the EREW model*. Springer, 1988. 4.3, 4.3.2, 4

[139] Mujtaba Ghouse and Michael Goodrich. Fast randomized parallel methods for planar convex hull construction. *Computational Geometry*, 7(4), 1997. 1

[140] Phillip Gibbons, Yossi Matias, Vijaya Ramachandran, et al. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, pages 638–648, 1997. 1.1.2

[141] Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991. 6.8.2

[142] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5), 1987. 6.6.3

[143] Mordecai Golin, Rajeev Raman, Christian Schwarz, and Michiel Smid. Simple randomized algorithms for closest pair problems. *Nordic J. of Computing*, 2(1):3–27, March 1995. 6.8.2

[144] Arturo Gonzalez-Escribano, Diego Llanos, David Orden, and Belen Palop. Parallelization alternatives and their performance for the convex hull problem. *Applied Mathematical Modelling*, 30(7):563 – 577, 2006. 6.1

[145] Michael Goodrich and Edgar Ramos. Bounded-independence derandomization of geometric partitioning with applications to parallel fixed-dimensional linear programming. *Discrete & Computational Geometry*, 18(4):397–420, 1997. 6.8.1

[146] Michael Goodrich. Finding the convex hull of a sorted point set in parallel. *Inf. Proc. Letters*, 26(4), 1987. 6.9

[147] Michael Goodrich. Fixed-dimensional parallel linear programming via relative ϵ-approximations. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 132–141, 1996. 6.8.1

[148] Michael Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *Proc. ACM Symposium on Theory of Computing (STOC)*, 2014. 3.3

[149] Ronald Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Proc. Letters*, 1972. 6.9

[150] Peter Green and Robin Sibson. Computing Dirichlet tessellations in the plane. *The Computer Journal*, 21(2):168–173, 1978. 1.1.3, 6.5

[151] Yan Gu. Improved parallel cache-oblivious algorithms for dynamic programming and linear algebra. *arXiv preprint:1809.09330*, 2018. 1.6

[152] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. Efficient BVH construction via approximate agglomerative clustering. In *High-Performance Graphics Conference (HPC)*, 2013. 1.6, 4

[153] Yan Gu, Yong He, and Guy Blelloch. Ray specialized contraction on bounding volume hierarchies. In *Computer Graphics Forum*, volume 34, pages 309–318, 2015. 1.6

[154] Yan Gu, Julian Shun, Yihan Sun, and Guy Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in algorithms and architectures (SPAA)*, 2015. 1.6, 6.5.3, 6.6.1.3

[155] Yan Gu, Yihan Sun, and Guy Blelloch. Algorithmic building blocks for asymmetric memories. In *European Symposium on Algorithms (ESA)*, 2018. 1.6

[156] Leo Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. 1978. 8.4.2, 8.4.2.1

[157] Leonidas Guibas, Donald Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992. 6.2, 6.3, 6.3.1, 6.5

[158] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984. 4

[159] Torben Hagerup. An even simpler linear-time algorithm for verifying minimum spanning trees. In *Graph-Theoretic Concepts in Computer Science*. Springer, 2010. 5.4.2.2, 16

[160] Shay Halperin and Uri Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. Syst. Sci.*, 53(3):395–416, 1996. 5.3.1, 5.3.4

[161] Shay Halperin and Uri Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests. In *J. Algorithms*, pages 1740–1759, 2000. 5.3.1, 5.3.4

[162] Sariel Har-Peled. *Geometric approximation algorithms*, volume 173. American Mathematical Society, 2011. 6.6, 6.8.2

[163] Sariel Har-Peled. Shortest path in a polygon using sublinear space. *Journal of Computational Geometry*, 7(2), 2016. 6.8.4

[164] William Hasenplaugh, Tim Kaler, Tao Schardl, and Charles Leiserson. Ordering heuristics for parallel graph coloring. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 166–177, 2014. 1.4.3, 6.1

[165] Yong He, Yan Gu, and Kayvon Fatahalian. Extending the graphics pipeline with adaptive, multi-rate shading. *ACM Transactions on Graphics (TOG)*, 33(4):142, 2014. 1.6

[166] John Hennessy and David Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. 1.1.2

[167] Daniel Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, June 1975. 4.5.3

[168] Daniel Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. 7.2

[169] Daniel Hirschberg and Lawrence Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4), 1987. 7.8.1

[170] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. ACM Symposium on Theory of Computing (STOC)*, 1981. 3.1, 3.2, 7.4

[171] HP. HP, SanDisk partner on memristor, ReRAM technology. http://www.bit-tech.net/news/ hardware/2015/10/09/hp-sandisk-reram-memristor, October 2015. 1, 1.1.1, 7.1

[172] Jingtong Hu, Qingfeng Zhuge, Chun Xue, Wei-Che Tseng, Shouzhen Gu, and Edwin Sha. Scheduling to optimize cache utilization for non-volatile main memories. *IEEE Transactions on Computers*, 63(8), 2014. 8.2

[173] S-HS Huang, Hongfei Liu, and Venkatraman Viswanathan. Parallel dynamic programming. *IEEE transactions on parallel and distributed systems*, 5(3), 1994. 7.2

[174] Shou-Hsuan Huang, Hongfei Liu, and Venkatraman Viswanathan. A sublinear parallel algorithm for some dynamic programming problems. *Theoretical Computer Science*, 106(2), 1992. 7.2

[175] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17, 1982. 4.4.3.3

[176] IBM. www.slideshare.net/IBMZRL/theseus-pss-nvmw2014, 2014. 1.1.1, 8.2, A

[177] Intel. Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/ community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology, July 2015. 1, 1.1.1, 7.1

[178] Intel. Intel architecture instruction set extensions programming reference. Technical Report 3319433-029, Intel Corporation, April 2017. 1.1.2, 9.2

[179] Intel Threading Building Blocks. https://www.threadingbuildingblocks.org. 2.2.2

[180] Dror Irony, Sivan Toledo, and Alexandre Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9): 1017–1026, 2004. 7.2

[181] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016. 1.4.4, 7.1, 7.2, 7.8.2, 7.8.4

[182] Rico Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In *Proc. 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017. 1.3, 3, 3.5

[183] Joseph Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992. 2.2.4, 4.3, 4.3.1, 4.3.2, 11, 5.3.1, 5.3.5.1, 5.3.5.2, 7, 6.7.2.1

[184] Java Fork-Join. http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html. 2.2.2

[185] David Karger, Philip Klein, and Robert Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *JACM*, 42(2), 1995. 5.4.2.2, 5, 5.4.2.2

[186] Samir Khuller and Yossi Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118(1):34–37, 1995. 6.8.2

[187] Hyojun Kim, Sangeetha Seshadri, Clement Dickey, and Lawrence Chu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *USENIX Conference on File and Storage Technologies (FAST)*, 2014. 1.1.1, 8.2, A

[188] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2), 1997. 5.4.2.2, 16

[189] David Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM J. on Computing*, 15(1), 1986. 6.9.1, 1, 6.9.2

[190] Stephen Kleene. Representation of events in nerve nets and finite automata. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA, 1951. 7.7.3

[191] Philip Klein, Shay Mozes, and Oren Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$-time algorithm. *ACM Transactions on Algorithms*, 6(2), 2010. ISSN 1549-6325. 4.5.1

[192] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006. 7.8.1

[193] Donald Knuth and Michael Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11), 1981. 7.8.1

[194] Marvin Künnemann, Ramamohan Paturi, and Stefan Schneider. On the fine-grained complexity of one-dimensional dynamic programming. *arXiv preprint arXiv:1703.00941*, 2017. 7.8.1

[195] Gad Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *ACM Symposium on Theory of computing (STOC)*, pages 220–230, 1986. 7.2

[196] Benjamin Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ACM International Symposium on Computer Architecture (ISCA)*, 2009. 1.1.1, 8.2

[197] Benjamin Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009. 8.3.2

[198] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom Keller. Energy management for commercial servers. *Computer*, 36(12): 39–48, 2003. 1

[199] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. 2014. 8.6.1.4

[200] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *European Conference on Computer Systems*, page 4. ACM, 2014. 1

[201] Nathan Linial and Michael Saks. Decomposing graphs into regions of small diameter. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 91, pages 320–330, 1991. 5.3.2

[202] Diego Llanos, David Orden, and Belen Palop. Meseta: A new scheduling strategy for speculative parallelization of randomized incremental algorithms. *International Conference on Parallel Processing Workshops*, pages 121–128, 2005. 6.1

[203] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Efficient parallelization using rank convergence in dynamic programming algorithms. *Communications of the ACM*, 59(10):85–92, 2016. 7.2

[204] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Low-rank methods for parallelizing dynamic programming algorithms. *ACM Transactions on Parallel Computing (TOPC)*, 2(4):26, 2016. 7.2

[205] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. Exploiting NVM in large-scale graph analytics. In *Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. ACM, 2015. 8.2

[206] Krishna Malladi, Ian Shaeffer, Liji Gopalakrishnan, David Lo, Benjamin Lee, and Mark Horowitz. Rethinking DRAM power modes for energy proportionality. In *IEEE/ACM International Symposium on Microarchitecture*, pages 131–142, 2012. 1

[207] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos Papadopoulos, and Yannis Theodoridis. *R-trees: Theory and Applications*. Springer Science & Business Media, 2010. 4

[208] Edward McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical report, 1980. 6.7.1.1, 6.7.2, 6.7.3

[209] Edward McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2), 1985. 6.7.1.3, 6.7.2, 6.7.3

[210] Jagan Meena, Simon Sze, Umesh Chand, and Tseung-Yuan Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9, 2014. 1, 1.1.1, 9.2

[211] Nimrod Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal on Computing*, 1983. 6.8.3

[212] Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *JACM*, 31(1), 1984. 1

[213] Ulrich Meyer and Peter Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1), 2003. 8.6.2

[214] Micro-ARchitectural and System Simulator for x86-based Systems. MARSSx86. http://marss86.org. 1.5, 8.1

[215] Gary Miller and John Reif. Parallel tree contraction part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991. 4.3, 4.3.2, 4

[216] Gary Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 196–203, 2013. 5.3.2, 5.3.4, 5.3.4.1, 1

[217] Gary Miller and John Reif. Parallel tree contraction and its application. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 478–489, 1985. 4.3, 4.3.2, 4, 5.4.3

[218] Ketan Mulmuley. *Computational geometry - an introduction through randomized algorithms*. Prentice Hall, 1994. 6.1, 6.5, 6.8.4

[219] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2), 1971. 7.7.3

[220] Suman Nath and Phillip Gibbons. Online maintenance of very large random samples on flash storage. *VLDB J.*, 19(1), 2010. 1.1.1

[221] Faisal Nawab, Joseph Izraelevitz, Ternece Kelly, Charles Morrey III, and Dhruva Chakrabarti amd Michael Scott. Dali: A periodically persistent hash map. In *International Symposium on Distributed Computing (DISC)*, 2017. 9.2

[222] Jürg Nievergelt and Edward Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1):33–43, 1973. 6.7.1, 8.4.2, 8.4.2.1

[223] Openmp. http://www.openmp.org. 2.2.2

[224] Mark Overmars. *The design of dynamic data structures*, volume 156. Springer Science & Business Media, 1983. (document), 6.6.2.1

[225] Xinghao Pan, Dimitris Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael Jordan. Parallel correlation clustering on big graphs. In *Advances in Neural Information Processing Systems (NIPS)*, 2015. 1.4.3, 6.1

[226] Hyoungmin Park and Kyuseok Shim. FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8), 2009. 1.1.1, 4.4, 4.4.3, 8.1, 8.2

[227] Mike Paterson. Improved sorting networks with $O(\log n)$ depth. *Algorithmica*, 5(1), 1990. 3.3

[228] David Peleg. Distributed computing. *SIAM Monographs on discrete mathematics and applications*, 5, 2000. 1.1.3

[229] Seth Pettie and Vijaya Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6):1879–1895, 2002. 5.3.1, 5.3.4

[230] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *PLDI*, 2011. 6.1

[231] Chung Keung Poon and Vijaya Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 212–222, 1997. 5.3.1, 5.3.4

[232] PTLsim. PTLsim. http://www.ptlsim.org. 1.5, 8.1

[233] Moinuddin Qureshi, Vijayalakshmi Srinivasan, and Jude Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009. 8.3.2

[234] Moinuddin Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. *Phase Change Memory: From Devices to Systems*. Morgan & Claypool, 2011. 1.1.1, 8.2

[235] Michael Rabin. Probabilistic algorithms. *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39, 1976. 6.8.2

[236] Sanguthevar Rajasekaran and John Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989. 11, 11, 6.7.2.1

[237] Alexander Rasmussen, Michael Conley, George Porter, Rishi Kapoor, Amin Vahdat, et al. Themis: an I/O-efficient mapreduce. In *ACM Symposium on Cloud Computing*, page 13. ACM, 2012. 1.1.2

[238] John Reif and Sandeep Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, 7(1-6):91–117, 1992. ISSN 0178-4617. 6.5

[239] John Reif and Sandeep Sen. Parallel computational geometry: An approach using randomization. *Handbook of Computational Geometry*, chapter 18. Elsevier Science, 1999. ISBN 9780080529684. 11

[240] Andy Rudoff. The impact of the NVM programming model. In *Storage Developer Conference*, 2013. 1.1.2, 9.2

[241] Andy Rudoff et al. pmem.io: Persistent memory programming, 2015. 1.1.2, 9.2

[242] Wojciech Rytter. On efficient parallel computations for some dynamic programming problems. *Theoretical Computer Science*, 59(3), 1988. 7.2

[243] Kunihiko Sadakane. Space-efficient data structures for flexible text retrieval systems. In *International Symposium on Algorithms and Computation*, pages 14–24. Springer, 2002. 5.2

[244] Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *International Symposium on Computer Architecture (ISCA)*, volume 41, pages 475–486. ACM, 2013. 1.5, 8.1

[245] Tao Schardl. *Performance engineering of multicore software: Developing a science of fast code for the post-Moore era.* PhD Thesis, 2016. 1.4.4

[246] Jeanette Schmidt. All shortest paths in weighted grid graphs and its application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27, 1998. 4.5.1

[247] Raimund Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6(3):423–434, 1991. 6.8.1

[248] Raimund Seidel. Backwards analysis of randomized geometric algorithms. In *New Trends in Discrete and Computational Geometry*, pages 37–67. 1993. 6.1, 6.2, 6.4.1

[249] Raimund Seidel and Cecilia Aragon. Randomized search trees. *Algorithmica*, 16 (4-5):464–497, 1996. 8.4.2, 8.4.2.1

[250] Joel Seiferas. Sorting networks of logarithmic depth, further simplified. *Algorithmica*, 53(3), 2009. 3.3

[251] S. Sen. A deterministic $poly(\log \log n)$ time optimal CRCW PRAM algorithm for linear programming in fixed dimensions. Technical report, Department of Computer Science, University of Newcastle, 1995. 6.8.1

[252] Julian Shun, Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012. 6.1

[253] Julian Shun, Laxman Dhulipala, and Guy Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 143–153, 2014. 5.3.1, 5.3.4, 1, 5.3.4.2

[254] Julian Shun, Yan Gu, Guy Blelloch, Jeremy Fineman, and Phillip Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 431–448, 2015. 1.4.3, 1.6, 4.3, 4, 5.4.3, 6.1, 8, 6.7.3.2

[255] Nodari Sitchinava. Computational geometry in the parallel external memory model. *SIGSPATIAL Special*, 4(2), 2012. 6.7

[256] Nodari Sitchinava and Norbert Zeh. A parallel buffer tree. In *ACM Symposium on Parallelism in algorithms and architectures (SPAA)*, 2012. 1, 6.7

[257] Daniel Sleator and Robert Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2), 1985. 2.3.4

[258] Daniel Sleator and Robert Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985. 8.4.2.2

[259] Edgar Solomonik, Aydin Buluc, and James Demmel. Minimizing communication in all-pairs shortest paths. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013. 7.2

[260] Yihan Sun and Guy Blelloch. Parallel range and segment queries with augmented maps. *arXiv preprint:1803.08621*, 2018. 6.7.2, 6.7.3.5

[261] Yihan Sun, Daniel Ferizovic, and Guy E Belloch. PAM: parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 290–304, 2018. 6.7.3.5, 8.4.2.2

[262] Robert Swendsen and Jian-Sheng Wang. Nonuniversal critical dynamics in monte carlo simulations. *Physical review letters*, 58(2):86, 1987. 5.3.1

[263] Yuan Tang and Shiyi Wang. Brief announcement: Star (space-time adaptive and reductive) algorithms for dynamic programming recurrences with more than O(1) dependency. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017. 1.1.3, 1.4.4, 7.1, 7.2, 7.8.2, 7.8.2

[264] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul Chowdhury. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2015. 7.1, 7.2, 7.8

[265] Robert E Tarjan and Uzi Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *IEEE Symposium on Foundations of Computer Science(FOCS)*, 1984. 4.3.2

[266] Robert Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985. 5.3.5.1, 5.3.5.2

[267] Robert Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983. 4.1, 8.4.2.1, 8.4.2.2

[268] Task Parallel Library (TPL). https://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx. 2.2.2

[269] Alexandre Tiskin. All-pairs shortest paths computation in the BSP model. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 178–189, 2001. 7.7

[270] Jesmin Jahan Tithi, Pramod Ganapathi, Aakrati Talati, Sonal Aggarwal, and Rezaul Chowdhury. High-performance energy-efficient recursive dynamic programming with matrix-multiplication-like flexible kernels. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015. 1.1.3, 1.4.4, 7.1, 7.2, 7.8, 7.8.2

[271] Sivan Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4), 1997. 1.4.4, 7.1, 7.2

[272] Stratis Viglas. Adapting the B^+-tree for asymmetric I/O. In *East European Conference on Advances in Databases and Information Systems (ADBIS)*, 2012. 1.1.1, 8.1, 8.2

[273] Stratis Viglas. Write-limited sorts and joins for persistent memory. *VLDB Endowment*, 7(5), 2014. 1.1.1, 4.4, 8.1, 8.2, 8.5.1

[274] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG)*, 26(1), 2007. 4

[275] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9 (1), 1962. 7.7.3

[276] Michael Waterman and Temple Smith. Rna secondary structure: A complete mathematical analysis. *Mathematical Biosciences*, 42(3-4), 1978. 7.8.3

[277] Duncan Watts and Steven Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393(6684):440–442, 1998. 8.6.1.3

[278] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, 1991. 6.8.3

[279] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer i/o practical. In *DAGS/PC Symposium*, 1993. 1.4.4, 7.1, 7.2

[280] Cong Xu, Xiangyu Dong, Norman Jouppi, and Yuan Xie. Design implications of memristor-based RRAM cross-point structures. In *IEEE Design, Automation and Test in Europe (DATE)*, 2011. 1.1.1, 8.2, A

[281] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung-Gon Yu. A low power phase-change random access memory using a data-

comparison write scheme. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2007. 1.1.1, 8.2

[282] F Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *ACM Symposium on Theory of Computing (STOC)*, 1980. 7.8.4

[283] Yole Developpement. Emerging non-volatile memory technologies, 2013. 1, 1.1.1, 9.2

[284] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ACM International Symposium on Computer Architecture (ISCA)*, 2009. 1.1.1, 8.2

[285] Omer Zilberberg, Shlomo Weiss, and Sivan Toledo. Phase-change memory: An architectural perspective. *ACM Computing Surveys*, 45(3), 2013. 8.2