The Locality-First Strategy for Developing Efficient Multicore Algorithms

by

Helen Jiang Xu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author . . . .
Department of Electrical Engineering and Computer Science
January 26, 2022

Certified by . . . .
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

The Locality-First Strategy for Developing Efficient Multicore Algorithms

by

Helen Jiang Xu

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

To scale applications on multicores up to bigger problems, software systems must be optimized both for parallelism to take full advantage of the multiple cores and for locality to exploit the memory system for cache-friendliness. Parallelization alone does not suffice to reach peak performance due to the processor-memory gap: the increasing divergence of processor and memory speeds. Locality and parallelism are difficult to optimize for independently — and even more challenging to combine — because they tend to conflict with each other.

I advocate that algorithm developers employ a ***locality-first strategy*** for developing efficient parallel and cache-friendly algorithms for multicores. That is, they should first understand and exploit locality as much as possible before introducing parallelism. I argue that an algorithm developer can achieve high-performing code more easily with the locality-first strategy than with either a parallelism-first strategy or a strategy of trying to optimize both simultaneously.

I present ten artifacts that leverage the locality-first strategy to create fast multicore algorithms that are simple to describe and implement. For example, locality-first data structure design in graph processing achieves about $2\times$ speedup over the state of the art. Additionally, I prove mathematically that multicore cache-replacement algorithms that take advantage of locality outperform all other online algorithms. The other eight artifacts make similar contributions in their respective domains. Together, these artifacts demonstrate that the locality-first strategy provides an effective roadmap for algorithm developers to design and implement theoretically and practically efficient multicore code.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

This thesis would not have been possible without the steadfast support of my mentors, collaborators, friends, and family. I'd like to use this opportunity to thank as many of them as I can.

First and foremost, I want to thank my advisor, Charles E. Leiserson, for inspiring me to strive every day to challenge and improve myself a researcher, teacher, and all-around person. I am grateful to Charles for his guidance and support throughout my PhD as not only a brilliant researcher but also a phenomenal mentor. His performance engineering class enabled me to pursue many new exciting lines of research spanning algorithm design, analysis, and engineering. Since then, he has taught me invaluable lessons not only about research but also about writing, communication, and mentoring. In addition to the technical aspects, Charles has created a social lab environment in the Supertech group through events such as standup comedy classes and apple picking. Thanks to his efforts, my time at MIT has been full of fascinating collaborations and discussions. I look forward to carrying forward everything he has taught me onwards and upwards.

I would like to thank Tao B. Schardl for all of his help and support during my PhD. I first met TB before coming to MIT when he took the time to chat with me about the Supertech group. Since then, I have been lucky to work closely with him on algorithms problems as well as on this thesis, my defense, and endless statements and presentations. His contributions, patience, and openness to chat about any questions have been invaluable to this thesis.

Thanks to Julian Shun and Virginia Vassilevska Williams for their service on my thesis and RQE committees. I thank Julian for sharing his expertise in graph algorithms and processing. His input has shaped the work in this thesis (Terrace, PPCSR, PCSR) on dynamic graph processing. I thank Virginia for her guidance on sparse matrix algorithms and for including me in many fun games of tennis.

I would also like to thank my other mentors without whom I would not be here today.

- My undergraduate advisor, Michael Bender, was the first person who taught me that computer science could be beautiful and fun. As someone from a non-technical background, I was unsure that CS would be a good fit for me, and I was especially skeptical that I could be interested in research. In my first semester, he saw how to connect my other interests with CS and how to get me interested in the topics. Since then, he has supported me in so many other ways: introducing me to Charles and other members of our research community, inviting me to speak at Stony Brook, and sharing food and literature recommendations.

- Thanks to Cynthia Phillips for always inviting me to explore new opportunities. I spent my summer before my PhD working with her at Sandia, during which I got to see many beautiful landscapes as well as beautiful theorems.

- I thank Shahin Kamali for his invaluable career advice and his support throughout my PhD. Our work on parallel cache replacement was my first foray into parallel algorithms research: he helped me discover new topics in online algorithms to build on my theoretical interests. I am also grateful for his help and effort to improve my dissertation, statements, and presentations.

- I would also like to thank Michael Pellauer and Joel Emer for being a joy to work with and for their helpful suggestions on my thesis. Michael hosted me (virtually) at NVIDIA in 2020 as part of the Architecture Research Group. Thanks to his efforts to meet with me every day in the summer and to continue our collaboration, I was able to see a hardware architect's perspective on sparse problems and to incorporate it into multicore algorithm engineering.

Thanks to all of my coauthors for their indispensable contributions to the work presented in this thesis. My collaborators are all not only technically brilliant but also lovely to work with.

- Brian Wheatman coauthored the PCSR, PPCSR, and Terrace papers on dynamic graph processing. Many thanks to Brian for our discussions on cache-efficient parallel algorithms and data structures as well as for sharing his performance engineering expertise.

- Prashant Pandey coauthored the Terrace paper on dynamic graph processing. Prashant deserves thanks not only for sharing his expertise in data structures but also for inviting me along for many awesome hikes. I am also grateful for his and Kavita's hospitality while hosting me in India for their wedding.

- Quanquan Liu coauthored the scan-hiding and smoothed analysis papers. In addition, I also thank Quanquan for our discussions on graph algorithms and data structures. In addition to being an excellent researcher, she is a great tennis partner and fun company to visit local restaurants with (especially Mary Chung's).

- William (Bill) Kuszmaul coauthored the smoothed analysis paper. Thanks to Bill for many interesting discussions about theoretical algorithm design and analysis. Also, I am especially grateful for his and Rose's hospitality during the last part of my PhD.

- Peter Ahrens and Nicholas Schiefer coauthored the PHIL paper. Special thanks to Peter for introducing me to sparse problems and our visits to Tosci's when writing the paper and to Nick for helping with my first move in Cambridge.

- Sean Fraser coauthored the prefix-sums and included/excluded sums papers. I am grateful for his work in developing all of the algorithms in the aforementioned papers.

In addition to the mentors and coauthors mentioned so far, I am grateful to everyone I have worked with and discussed research with. I would like to thank the coauthors I have not yet mentioned: Aydın Buluç, Andrea Lincoln, Jayson Lynch, Rezaul Chowdhury, Rathish Das, Rob Johnson, Simon Mauras, Martín Farach-Colton, Tyler Mayer, James Aimone, Ojas Parekh, Ali Pinar, and William Severa. This thesis would not have been possible without their contributions. Thanks to the members of the Supertech group, especially Tim Kaler, Matthew Kilgore, Billy Moses, and Alexandros Iliopoulos, for fostering a lively and open research environment. Additional thanks to Cree Bruins for supporting the Supertech group and for her infallible cheer and friendliness. I would also to thank Deanna Montgomery for her service as the EECS Communication Lab manager. Being part of the Comm Lab has been one of the highlights of my time at MIT, and I am grateful for her input in all communication-related matters.

Finally, I am grateful to all of my friends and family for their encouragement and support. Thanks to my childhood friends: Nii for being my ride-or-die, Fay for the Shakespeare and adventures, Ash for the fun trips and good times, Kense for always coming through, and Dillon and Mimz for hanging out with me. Thanks to my more recent friends: Helena for the chats and weird snacks, Yen-Ling for carrying me through my TQEs and for the infinite hot pot, Wanrong for the plays, egg tarts, and tea times, and Ryan for always hyping up my aspirations. Special thanks to Ilia for the drinks, meat, and support in this last stretch of my grad school life. I thank my parents, Ye and Bryant, for supporting me in all of my endeavors and always keeping a place to return to, especially during the pandemic. Thanks to my brother, Allan, for the hikes, drives, and karaoke jams. I would not have made it this far without them.

I am grateful for the support of my research sponsors. This research was sponsored in part by: a National Physical Sciences Consortium Fellowship, the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator under Cooperative Agreement Number FA8750-19-2-1000, Los Alamos National Laboratory (LANL) under Subcontract Number 531711, the National Science Foundation (NSF) under grant CCF-1533644, and the Intelligence Advanced Research Projects Activity (IARPA) under grant 138076-5093555.

Contents

Chapter 1

Introduction

This thesis is concerned with scaling applications on multicores by optimizing algorithms both for locality to take advantage of the memory system and for parallelism to make full use of the multiple cores. Parallelism alone is not enough because of the ***processor-memory gap***: the divergence between processor and memory speeds [90, 299]. Unfortunately, as we shall see, locality and parallelism are hard to achieve independently. Furthermore, they are even more difficult to achieve together because they tend to trade off with each other.

The seemingly fundamental tension between locality and parallelism poses challenges to developing efficient codes on multicores. As we shall see, exploiting locality involves computing on the same or similar data over time. Computing on the same or similar data in parallel can result in correctness and performance issues, however. In this chapter, we will see concrete examples of how optimizing for one of these features can disrupt the other.

To overcome these tensions, I advocate a "locality-first strategy" for creating high-performance multicore algorithms that are optimized for both locality and parallelism. Specifically, I advocate the following thesis statement:

> **Thesis statement:** To create efficient parallel algorithms for multicores, algorithm developers should use a ***locality-first strategy***. That is, they should first *understand, enhance, and exploit locality* as much as possible as a first step before introducing parallelism.

To employ the locality-first strategy, let us first understand the different kinds of locality. Algorithm developers take advantage of ***spatial locality*** and ***temporal locality*** in order to fit the underlying hardware. Spatial locality is the tendency of programs to access nearby memory locations in a short time period, whereas temporal locality is the tendency of programs to access the same memory location over time [124, 305]. A program may exhibit spatial locality, temporal locality, both, or neither, depending on the problem.

As we shall see in detail in Chapter 2, locality and parallelism tend to conflict with each other in fundamental ways. At a high level, exploiting locality involves computing on the same or similar data over time. Unfortunately, computing on

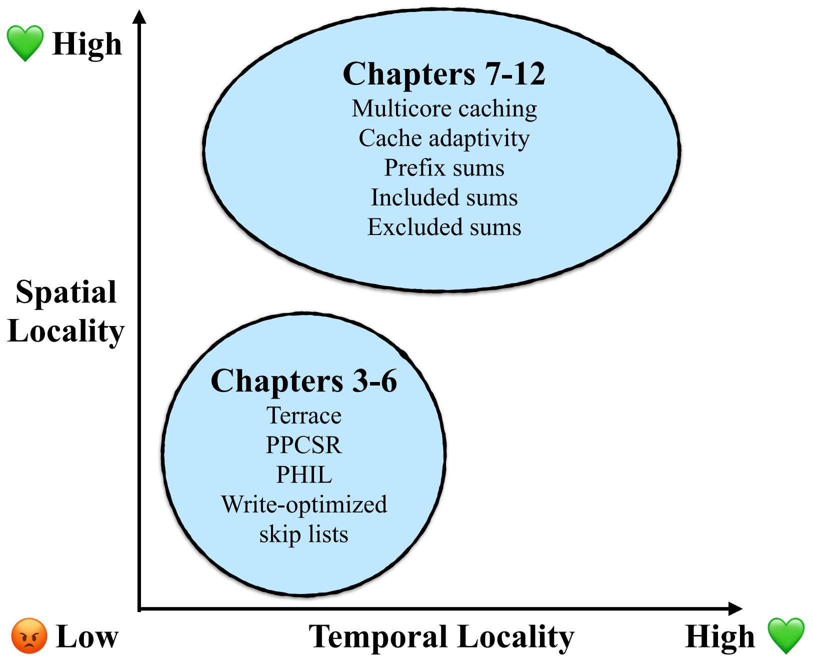


Figure 1-1: Graphical illustration of the main artifacts in this thesis.

the same data in parallel can cause nondeterministic behavior, and computing on similar data over time can cause contention. These issues disrupt correctness and performance in parallel programs.

Contributions

In support of the thesis statement, I present ten intellectual artifacts that demonstrate the potential for the locality-first strategy to overcome seemingly fundamental tensions between parallelism and cache-friendliness. These artifacts fall into two categories based on whether they enhance locality by changing the data layout or whether they exploit naturally-occurring locality in the algorithm's access pattern. Figure 1-1 illustrates the organization of the artifacts into these two categories.

The ten principal artifacts and their contributions are as follows:

- The ***Terrace*** artifact [294]: A dynamic-graph-processing system optimized for skewed dynamic graphs. Terrace enhances spatial locality with a hierarchical data representation that takes advantage of naturally-occurring graph skewness. Terrace is on average 2× faster than Aspen [128], a state-of-the-art dynamic-graph-processing system, on graph queries. Furthermore, it overcomes traditional tradeoffs between queries and updates and supports similar updatability compared to Aspen. I developed the Terrace artifact with Prashant Pandey, Brian Wheatman, and Aydın Buluç.

- The ***Parallel Packed Compressed Sparse Row*** (PPCSR) artifact [379]: A dynamic-graph-processing system optimized for spatial locality. PPCSR enhances spatial locality by storing all of the data contiguously in a "Packed Memory Array" data structure. PPCSR supports graph queries about 1.6× faster than Aspen while maintaining competitive update throughput. I developed the PPCSR artifact with Brian Wheatman.

- The ***PHIL*** artifact [6], an efficient algorithm to speed up sparse tensor algebra via blocked formats. PHIL exploits spatial locality by estimating fill, an important quantity in blocked sparse tensor operations. PHIL estimates the fill with provable time and accuracy guarantees. Its advantage is evident in the empirical evaluation: it estimates the fill at least $2\times$ faster than OSKI [371], a state-of-the-art fill-estimation algorithm, and up to $40-50\times$ faster than OSKI. I developed the PHIL artifact with Peter Ahrens and Nicholas Schiefer.

- The ***write-optimized skip list*** artifact [49]: A cache-optimized randomized data structure. The write-optimized skip list enhances spatial locality in data structure design. It achieves write-optimized bounds in that it supports asymptotically optimal searches, inserts, and deletes. Furthermore, it supports inserts and deletes asymptotically faster than searches. I developed the write-optimized skip list artifact with Michael A. Bender, Martín Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, and Cynthia Phillips.

- The ***cyclic analysis*** artifact [210]: A measure for beyond worst-case analysis of online algorithms applied to parallel cache-replacement. Cyclic analysis mathematically supports the locality-first strategy for algorithms that exploit temporal locality. Specifically, it provides theoretical grounding for the established practical superiority of the Least-Recently-Used (LRU) [342] algorithm because LRU takes advantage of temporal locality. I developed the cyclic analysis artifact with Shahin Kamali.

- The ***lower bounds*** artifact [209]: Worst-case analysis of online algorithms for parallel cache-replacement. The lower bounds artifact motivates the study of beyond-worst-case analysis measures to mathematically explain the empirical superiority of algorithms that take advantage of temporal locality. The lower bounds show that a large class of online algorithms for parallel cache-replacement are equally arbitrarily far from optimal. I developed the lower bounds artifact with Shahin Kamali.

- The ***smoothed analysis*** artifact [41]: Beyond-worst-case analysis of cache-adaptive algorithms. The smoothed analysis for cache-adaptive algorithms provides mathematical grounding for algorithms that take advantage of temporal locality in shared memory. Specifically, it closes the gap between cache-oblivious and cache-adaptive algorithms. I developed the smoothed analysis artifact with Michael A. Bender, Rezaul A. Chowdhury, Rathish Das, Rob Johnson, William Kuszmaul, Andrea Lincoln, Quanquan C. Liu, and Jayson Lynch.

- The ***scan-hiding*** artifact [246]: A technique for converting non-cache-adaptive algorithms into cache-adaptive ones. Scan-hiding enhances temporal locality in a large class divide-and-conquer algorithms. I developed the scan-hiding artifact with Andrea Lincoln, Quanquan C. Liu, and Jayson Lynch.

- The **`CAST_BLK`** and **`PAIR_BLK`** artifacts [151]: Two algorithms for fast-and-accurate parallel prefix sums. These algorithms exploit naturally-occurring spatial locality without changing the data representation. The `CAST_BLK` and `PAIR_BLK` algorithms achieve comparable performance to the state-of-the-art parallel prefix algorithm with at least $5\times$ more accuracy. I developed the `CAST_BLK` and `PAIR_BLK` artifacts with Sean Fraser and Charles E. Leiserson.

- The ***bidirectional box-sum*** and ***box-complement*** artifacts [389]: Two efficient algorithms for multidimensional included and excluded sums. These algorithms naturally achieve good spatial locality without changing the data layout. These artifacts improve the asymptotic work for the multidimensional included-sums and excluded-sums problems from exponential to linear in the number of dimensions. This improvement in higher-dimensions is borne out in the empirical evaluation. I developed the bidirectional box-sum and box-complement artifacts with Sean Fraser and Charles E. Leiserson.

Thesis organization

The rest of this thesis has a simple structure. Chapter 2 provides context for the locality-first strategy as a method for general algorithm engineering on multicores. Chapters 3-12 present the principal artifacts. Chapter 13 summarizes the thesis and takes a step back from the principal artifacts to discuss the broader applicability of the locality-first strategy.

The appendices present auxiliary technical material to several of the principal artifacts. Appendix A introduces Packed Compressed Sparse Row, a serial storage format for dynamic graphs that underlies the artifacts in the domain of dynamic graph processing (Chapters 3 and 4). Appendix B contains additional background material and proofs for the artifacts in the domain of cache-adaptive algorithms (Chapters 9 and 10). Appendix C supplements the proofs and experimental results in the included-sums and excluded-sums chapter (Chapter 12).

Chapter 2

The Locality-First Strategy

This chapter argues that the locality-first strategy provides a principled method for general multicore algorithm engineering in four parts.

Section 2.1 motivates the need for a principled method for algorithm engineering on multicores by describing the challenges involved in optimizing programs on multicores. As we shall see, both parallelism and locality are difficult to achieve separately. They are even more challenging to achieve simultaneously because they often conflict with each other.

Section 2.2 justifies why the locality-first strategy is a good method for general algorithm engineering with practical and principled reasons. For example, the locality-first strategy simplifies multicore algorithm development by focusing on the serial execution first.

Section 2.3 shows how the locality-first strategy created the artifacts that enhance spatial locality — Terrace, PPCSR, PHIL, and the write-optimized skip list artifacts — by changing the data layout.

Section 2.4 explains how the locality-first strategy relates to the artifacts that exploit locality without changing the layout — the cyclic analysis, lower bounds, smoothed analysis, scan-hiding, `CAST_BLK`, `PAIR_BLK`, bidirectional box-sum, and box-complement artifacts.

2.1 Creating efficient multicore algorithms

Why do we need a principled method for creating efficient multicore algorithms? To get good performance for a multicore application, a performance engineer must design algorithms to exploit two principal multicore hardware features: a steep cache hierarchy and multiple cores numbering into the hundreds. Exploiting each of these features is complex and requires theoretical and practical expertise. Without a principled approach, optimizations for one feature may be pessimizations for the other.

To understand the issues involved more deeply, this section is broken into four parts. The first part reviews multicore hardware and argues that the cache hierarchy and the multiple cores are the salient hardware features in algorithm engineering on multicores. The second part explores cache-friendly algorithms and how they enable a performance engineer to take advantage of the cache hierarchy. The third part addresses task-parallel algorithms, which abstract and simplify the programming of multiple cores, and the issues of nondeterminism and scalability that complicate parallel programming. The fourth part illustrates the problems inherent in trying to optimize for both cache-friendliness and parallelism at the same time by providing concrete examples such as false sharing and contiguous data.

Multicore hardware features

To understand in detail the issues involved in cache-friendly algorithms, parallel algorithms, and their interaction, however, it will be helpful to review the characteristics of multicore hardware. Modern multicore processors are characterized by three salient architectural features — the cache hierarchy, multiple cores, and vector units — as well as a host of other architectural features such as prefetching, instruction-level parallelism, and symmetric multithreading, which we won't discuss.

The Amazon `c5.metal` instance [10] provides a good example of a modern multicore. It contains two Intel Cascade Lake [91] processors each with 24 cores for a total of 48 physical cores, and a clock speed of 3.4GHz. Each core is hyperthreaded (Intel's term for symmetric multithreading), so it has a total of 96 virtual cores. Each core has a private 32Kb L1 cache that takes about 0.5 seconds to access [115] and a 1MB L2 cache that takes about 7ns to access. Each processor also has a shared L3 cache of 1.375 MB/core for a total of 33 MB/processor that takes about 25ns to access [115]. The entire system has 192 GB of main memory that takes about 100ns to access [115]. Each core also has vector units capable of 8 simultaneous 64-bit floating point operations. The `c5.metal` instance has a steep cache hierarchy, almost 100 virtual cores, and vector units.

Of the three salient features of a multicore, vector processing is comparatively well-understood and exploited by applications because of advances in compiler technology [233, 248, 268]. In contrast, compilers cannot access a huge space of cache-friendly optimizations because they cannot arbitrarily rearrange data representations. Furthermore, although there has been progress on automatic task parallelism to take advantage of the multiple cores [142, 316, 366], these technologies are still immature and not widely used. For example, the automatic parallelization feature in `gcc` currently only applies to loops that the compiler can determine contain no dependencies [284, 380]. Although there remain many compelling open questions in the area of vectorization, this thesis focuses on the hard problem of creating multicore algorithms that exploit the cache hierarchy and the multiple cores.

Furthermore, although GPUs are a key part of today's computing substrate, GPU algorithm engineering is outside of the direct scope of this thesis because it is a separate subsystem from multicores. Multicore systems such as the Amazon `p2` instance series may offer multicore processors with attached GPUs that rely on the multicore as a host processor to run programs and store data GPUs [314]. The CPU and GPU are separate computing systems: algorithms for CPUs and GPUs are written in different languages and are run on one of the platforms but not both simultaneously. Furthermore, not all multicore platforms have attached GPUs. For example, the `c5.metal` instance, one of Amazon's most powerful multicores, does not have a GPU. Furthermore, although GPUs exhibit more parallelism than multicores can outperform GPUs by orders of magnitude on irregular workloads [89] because GPUs are optimized for regular workloads. These results demonstrate that especially for irregular applications such as dynamic graph processing in Chapters 3 and 4, multicores can achieve comparable or better performance compared to GPUs. Finally, the locality-first strategy also applies to GPU programming because GPUs have hierarchical memory [173]. Specifically, GPUs have on-chip shared memory and global memory. Accessing shared memory on a GPU is about 100× faster than accessing global memory on a GPU [173]. Therefore, optimizing GPU programs for temporal locality has the potential for significant performance improvement by avoiding accesses to global memory [172]. The focus of this thesis is on multicore algorithm engineering because multicores can solve general problems efficiently.

Designing algorithms for both cache-friendliness and parallelism is necessary for peak performance on a multicore. In the Cascade Lake processor, the maximum speedup from cache-friendliness was about 100× (the difference between a L1 and main memory access). Additionally, the maximum parallel speedup was almost 100× (the number of cores). Optimizing algorithms for only one of these features leaves orders of magnitude of performance on the table.

Cache-friendly algorithms

To understand how to optimize programs on multicores, let us first turn our attention to how to exploit locality for cache-friendliness.

To understand in detail the issues involved in creating cache-friendly algorithms, it will be helpful to review the types of locality that algorithms can optimize for. As mentioned in Chapter 1, there are two types of locality: spatial locality — the tendency of programs to access nearby data over time — and temporal locality — the tendency of programs to access the same data over time. Depending on the problem, an algorithm may exploit either spatial locality, temporal locality, or both.

To see why exploiting locality is important for cache-friendliness, let us consider a simplified multicore memory system. The first main component of a memory system is the ***cache hierarchy***: a sequence of memory stores with varying access speeds and sizes. For simplicity, let us consider a cache hierarchy with two levels: a large slow main memory and a small fast cache. Optimizing algorithms for temporal locality exploits the cache hierarchy by maximizing accesses to the cache and avoiding slow accesses to memory. The second main component of a memory system is the ***cache line***: the unit of data transfer between cache and main memory. A computer's memory is divided into contiguous, non-overlapping units called cache lines. Cache lines are usually multiple bytes (e.g. 64 bytes). If a program requests data that is not already in the cache, the memory system brings the entire cache line that data resides in from main memory to cache. Optimizing algorithms for spatial locality exploits cache lines by maximizing the amount of contiguous useful data and minimizing memory block transfers.

The classical ***Disk-Access Machine*** (DAM) model [3] due to Aggarwal and Vitter formalizes this simplified memory system. It models two levels of memory: a small bounded-size cache of size M, and an unbounded-size memory. Any data must be brought to the cache first before it is processed. Data is transferred in blocks of size B between the cache and the memory, and transfers have unit cost. Algorithms that achieve speedup in M in the DAM model take advantage of temporal locality to reuse data as much as possible while it is in cache. On the other hand, algorithms that achieve speedup in B in the DAM model take advantage of spatial locality to minimize cache-line transfers. The DAM model enables algorithm developers to reason about cache-friendliness in a straightforward way without requiring them to know the exact details of the underlying memory system [19, 367].

Unfortunately, the DAM model abstracts away important details of the real-world memory systems. In reality, memory systems have many levels of cache, so maximizing cache-friendliness requires reusing data optimally in all of the levels. These levels of cache and main memory make up Random Access Memory (RAM). Additionally, memory systems contain a disk: a larger and slower level of storage outside of RAM. Memory-block sizes differ between RAM and disk, so a single setting of the block size B may not capture all of the benefits of block transfers between multiple levels of memory.

To understand a concrete example of the challenges that these details pose to engineering real-world cache-friendly algorithms, let us consider the classical B-tree [34], a widely-used data structure in large indexes and databases.

The gap between the theory and practice of B-trees illustrates the general complexity of developing cache-friendly algorithms. In theory, the B-tree [34] achieves asymptotically optimal performance in the DAM model when the tree-node size is set to the block-size parameter B [76]. Programming a theoretically optimal B-tree is difficult because it requires knowledge of the underlying memory-block size, which depends on the architecture. Furthermore, in practice, the optimal tree-node size depends not only on the underlying hardware but also on the workload [42]. The right tree-node size can even vary by orders of magnitude [42]. For example, B-trees in databases and file systems often use a node size of 16Kb [258, 266, 275], while B-trees optimized for range queries use larger node sizes around 1MB [289, 300]. The complexity of engineering efficient B-trees demonstrates that developing efficient cache-friendly algorithms involves expertise in both the hardware and the problem.

Parallel algorithms

Let us turn our attention to how to optimize programs for ***task parallelism***, a type of parallelism that distributes tasks performed by threads across cores [134, 231, 322, 353]. Since the focus of this thesis is on how to take advantage of the multiple cores, the remainder of this thesis will refer to "task parallelism" as ***parallelism***.

Although tremendous progress has been made in mitigating challenges to developing efficient parallel algorithms, creating parallel algorithms is still notoriously difficult because of the issues of nondeterminism and scalability [203, 286, 319, 334]. ***Dynamic multithreading*** [24, 30, 155, 237] platforms have emerged as the dominant way to take advantage of task parallelism. Dynamic multithreading enables a ***processor-oblivious*** programming model that enables algorithm developers to expose parallelism without explicitly scheduling the physical cores. The fundamental challenges of nondeterminism and scalability remain, however, because they stem from the ability of parallel algorithms to perform logically discrete computations at the same time.

Nondeterminism is one of the key difficulties in developing correct and efficient parallel algorithms [238]. Parallel algorithms may exhibit ***nondeterministic behavior*** — different behavior on different runs even on the same input — due to how the operating system schedules and executes threads during any particular run of the program [319]. For example, parallel algorithms may contain determinacy races, or nondeterministic behavior due to two parallel threads accessing the same memory location and at least one updating it. Determinacy races may disrupt correctness and performance [145]. Additionally, determinacy races pose challenges to traditional debugging techniques, because the bug may not occur in every run of the program [145].

Researchers [54, 55, 62, 68, 145] have proposed ***deterministic parallelism*** in response to the problem of nondeterminism, but many practical parallel codes still exhibit nondeterministic behavior. Deterministic parallel programs avoid programmer-observable nondeterminism. They simplify testing and debugging by theoretically reducing reasoning about correctness to reasoning about the corresponding serial program. Additionally, deterministic algorithms can be practically fast [64]. Many real-world parallel codes exhibit nondeterministic behavior, however, because existing parallel-programming technologies do not provide a general framework for deterministic parallel programming [319]. Because of the nondeterminism issue, creating correct and efficient parallel codes requires theoretical and practical expertise.

Furthermore, tremendous research effort has been devoted to addressing the "scalability" issue, but developing general scalable parallel algorithms remains challenging. ***Scalable*** parallel algorithms achieve improved performance by computing with more cores. As we shall see, algorithm developers can use "work-span analysis" [108, Chapter 27] to theoretically characterize scalability by identifying and analyzing logically parallel tasks. Achieving good scalability requires insight, however, to parallelize seemingly serial computations in a clever way [65, 66, 129, 175, 203, 204, 239, 249, 319, 334, 339]. Moreover, profiling the scalability of parallel codes is much more difficult than profiling the runtime distribution in serial codes [177, 320]. To address this issue, researchers have introduced efficient scalability profilers based on "work-span analysis" [177, 320]. These technologies are immature, however, and may not apply to other threading models. Because of the scalability issue, developing efficient parallel codes requires both deep theoretical design and analysis as well as careful engineering.

Work-span analysis[1] [108, Chapter 27] formalizes scalability by analyzing the cost of parallel algorithms in terms of their "work" and "span." For concreteness, let $\mathcal{A}$ be an algorithm. The ***work*** of $\mathcal{A}$, denoted by $\text{Work}(\mathcal{A})$, is the total time to execute the entire algorithm in serial. The ***span***[2], denoted by $\text{Span}(\mathcal{A})$, is the longest serial chain of dependencies in the computation (or the runtime on an infinite number of processors). The ***parallelism*** of an algorithm is defined as the work divided by the span, or $\text{Work}(\mathcal{A})/\text{Span}(\mathcal{A})$. Algorithms with more parallelism are

[1]This thesis omits the theoretical foundations of work-span analysis because the main contributions are not in the analysis of parallel algorithms, but the formal model is presented in tutorial fashion in [319, Chapter 2].

[2]Sometimes called **critical-path length** [67] or **computational depth** [62].

more scalable because more of the work can be performed in parallel. Work-span analysis characterizes scalability in parallel algorithms without the exact details of the underlying machine.

Work-span analysis enables algorithm developers to predict how much speedup a parallel algorithm will achieve on a given machine. Given some program running on P processors, let T_1 be the time taken by the serial execution and T_P be the time taken by the P-processor execution. The ***speedup*** of the program is defined as T_1/T_P. In traditional work-span analysis, the potential speedup is upper bounded by P due to the Work Law, which states that $T_P \geq \text{Work}(\mathcal{A})/P$. If the program achieves speedup P, we say that the application exhibits ***linear speedup***. In practice, to achieve linear speedup, a program should exhibit ***ample parallelism***, or parallelism much larger than the number of processors [155].

Unfortunately, real-world parallel algorithms may not achieve the speedups predicted by work-span analysis due to other constraints on multicores. In practice, memory size and memory bandwidth bottleneck scalability for many applications [334,355]. For example, the parallel scalability of algorithms for the "graph-processing" problem studied in this thesis are limited by memory bandwidth because the problem often requires random memory access [334]. The complexity of interactions between parallelism and the memory system demonstrates the difficulty of achieving good practical scalability.

Tensions between parallelism and cache-friendliness

Cache-friendliness and parallelism are difficult to optimize for independently and even more difficult to combine because they tend to conflict with each other. At a high level, exploiting temporal locality involves computing on the same data over time, and exploiting spatial locality involves computing on similar (nearby) data over time. But computing on the same or similar data in parallel often leads to nondeterminism and cache-line contention, which may affect correctness and performance.

One example of this tension is false sharing: a performance bug that occurs when multiple workers access different locations in the same cache line in parallel, repeatedly invalidating that cache line in each worker's private cache and wasting system bandwidth [362]. One solution to false sharing is to pad each variable with extra bytes to force them onto different cache lines. Padding the elements disrupts cache-friendliness however, because the elements of interest are now further apart and require more cache-line reads.

Another example is the data representational choice of keeping data contiguous (e.g., in a list) or non-contiguous (e.g., in a tree). Keeping data in a list maximizes spatial locality by increasing the number of elements fetched per read and eliminating pointer-chasing. Concurrently modifying a list is much more complicated than concurrently modifying a tree, however, because a tree can support efficient concurrent updates by locking a small part at a time (e.g. with hand-over-hand locking [181]), while a list-based data structure may have to lock the entire data in the worst case [379]. These examples demonstrate the conflicts between cache-friendliness and parallelism.

2.2 The locality-first strategy for general multicore algorithm engineering

This section presents rationale for why the locality-first strategy is a useful method for creating algorithms that achieve both cache-friendliness and parallelism. Specifically, it discusses both practical and principled reasons for a locality-first approach to algorithm optimization for multicores. It also concretizes those reasons by explaining how each one relates to the principal artifacts.

This section justifies the locality-first strategy in four parts. The first part explains that the locality-first strategy simplifies algorithm engineering by focusing on serial optimizations first before parallelism. The second part explains how the locality-first strategy creates efficient algorithms by optimizing for the total work. The third part demonstrates that optimizing for locality takes advantage of other types of hardware features available in multicores such as vector units and GPUs. Finally, the fourth part organizes the principal artifacts into categories based on the type of locality they exhibit and therefore how they relate to and support the locality-first strategy.

Enabling easier practical algorithm engineering

The locality-first strategy makes writing parallel codes easier because it focuses on optimizing the serial execution first. Section 2.1 shows that writing parallel algorithms is strictly more difficult than writing serial algorithms. Parallelism may introduce nondeterministic behavior and race conditions, which affect correctness and performance. Parallelism complicates performance measurement, an integral part of principled performance engineering, because of additional variability from nondeterminism in the thread scheduler [319]. Profiling the scalability of parallel codes is much more difficult than profiling the runtime distribution in serial codes [320]. The locality-first strategy simplifies algorithm design because it first produces a serial, working specification for a parallel code before moving on to parallelization.

The artifacts in this thesis concerning dynamic graph processing reflect this progression. Appendix A presents "Packed Compressed Serial Row", a serial "Packed Memory Array" data structure for dynamic graphs that enhances spatial locality before introducing parallelism. The Parallel Packed Compressed Sparse Row (PPCSR) artifact (Chapter 4) takes the next step and introduces parallelism on top of (serial) Packed Compressed Sparse Row. Since dynamic graph applications have ample parallelism, The Terrace artifact (Chapter 3) further improves performance by trading some of the parallelism in PPCSR for improved locality by taking advantage of problem structure.

A principled approach to creating efficient algorithms

The locality-first strategy draws inspiration from Cilk's ***work-first principle*** of minimizing the work of a serial algorithm, even if it adversely affects parallel scalability, because the work has a more direct impact on performance [155]. Although the Cilk multithreaded language presented the work-first principle in the context of dynamic multithreading, the artifacts in this thesis demonstrate its potential in general multicore software optimization. As we shall see later in this section, optimizing for spatial locality reduces work by taking advantage of other hardware features in multicores. In general, optimizing for cache-friendliness by exploiting locality reduces work in serial, which enables algorithms to reach their peak efficiency after parallelization.

Furthermore, in reality, exploiting temporal locality offers opportunities for continuous speedups due to the multiple levels of the cache hierarchy. For example, in modern multicores, there are multiple levels of cache (e.g. L1, L2, L3) before main memory. The DAM model expresses speedups in terms of a single memory block size B and a single memory size M. In reality, these analyses apply between any two levels of cache, resulting in more continuous speedups from exploiting locality. The maximum improvement available from taking advantage of the cache hierarchy can range up to orders of magnitude. For example, an L1 cache hit takes about 1 nanosecond, while a main memory access takes about 100 nanoseconds [115]. Additionally, a disk seek takes about 10 million nanoseconds. In contrast, speedups from parallelization are maximized at P, the number of processors. Therefore, optimizing for locality first offers multiple levels of improvement at the different levels of cache before parallelization. Chapter 9 concerns cache-oblivious algorithms, which use all levels of the cache hierarchy asymptotically optimally.

Although parallelism and cache-friendliness are in tension, the artifacts in this thesis demonstrate that trading off some parallelism for improved locality can still improve overall performance. The artifacts in this thesis support the observation in the original Cilk-5 presentation of the work-first principle that there is ample parallelism in the common case [155]. For example, "blocking" (e.g. the "blocked formats" in Chapter 5, or the "blocked prefix sums" in Chapter 11) improves overall performance by taking advantage of locality at the cost of some parallelism.

Taking advantage of other multicore hardware features

Optimizing for locality enables efficient usage of hardware features than enable different types of parallelism in multicores. In reality, multicores support other types of parallelism besides task parallelism, such as "instruction-level" and "data-level" parallelism. Optimizing for spatial locality enables better use of ***instruction-level parallelism*** such as cache prefetching, a hardware-supported optimization that brings data before it is needed into cache to reduce future latency [343]. For example, the PMA data structure in Chapter 4 achieves fast scan performance from cache prefetching because it exploits spatial locality by storing all of its data contiguously. Additionally, algorithms with good spatial locality can make better use of ***data-level parallelism*** via SIMD instructions that take advantage of hardware-level vector units [148]. Vectorizing the "Packed Memory Array" (PMA) data structure in Chapter 4 is more straightforward than vectorizing tree-based data structures because the PMA is contiguous in memory [379]. Optimizing for spatial locality by storing data contiguously enables other types of parallelism in multicores such as instruction-level and data-level parallelism.

2.3 Enhancing locality by changing the data layout

The Terrace, PPCSR, PHIL, and write-optimized skip list artifacts apply the locality-first strategy to enhance spatial locality by changing the underlying data layout (Chapters 3-6).

The Terrace, PPCSR, and PHIL artifacts target sparse graph and sparse tensor applications and demonstrate that optimizing for spatial locality first, even at the cost of some parallelism, can improve overall performance. Many sparse computations, such as sparse graph applications, exhibit minimal temporal locality [249,276,334]. Therefore, these chapters focus on improving spatial locality by co-locating data through cache-friendly data structure design.

This section overviews this thesis's contributions regarding enhancing spatial locality in the domain of sparse applications in two steps. First, it presents challenges to locality in sparse applications. Next, it summarizes each of the artifacts in the domain of sparse applications and explains how each uses the locality-first strategy to address challenges to locality.

Finally, this section explains how to use the locality-first strategy to create the write-optimized skip list, a cache-friendly randomized data structure.

Background on sparse applications

Sparse graphs and sparse tensors appear in many fundamental applications that range from scientific computing [87,369] to social networks [141,279]. ***Sparse*** datasets have many more zeroes than nonzeroes. For example, in social networks, most of the users are not connected to most of the other users. If the vertices are users and edges are connections between users, most of the entries representing connections would be empty.

Sparsity disrupts spatial locality because of the presence of zeroes between actually present values, limiting the effectiveness of fetching multiple adjacent elements in a cache line. To illustrate this issue, let us consider an adjacency matrix representation of a graph [108]. In an adjacency matrix, a zero represents a lack of connection between vertices, while a nonzero represents an edge. Therefore, a naive representation of sparse data exhibits poor spatial locality because most of the data is zeroes.

Performance engineers can improve spatial locality and algorithm complexity in sparse applications with compressed representations that store only the nonzeroes and the locations of the nonzeroes [361]. These compressed representations enable fast algorithms with work proportional to the number of nonzeroes. In practice, these data structures reduce the work by orders of magnitude because they allow direct computation over only the existing data.

As we shall see in the artifacts in this section, the introduction of ***metadata***, or the locations of the nonzeroes, raises challenges and opportunities for designing scalable and cache-friendly data structures and algorithms to store and process the metadata.

Storing and processing dynamic graphs

Terrace and PPCSR apply the locality-first strategy to design and implement fast parallel dynamic-graph-processing systems that are optimized for spatial locality first.

Many real-world graphs ranging from machine learning graphs [303] to social networks [141,279] exhibit ***graph irregularity***, or a sparse and skewed structure. These graphs have a skewed distribution of vertex degrees, where there are a few high-degree vertices and many low-degree vertices [279]. Furthermore, these graphs are often ***dynamic***: they change over time. For example, ***dynamic-irregular*** graphs arise naturally in social networks, computational biology, and the Internet. Much larger problems on the order of gigabytes and up to terabytes can be efficiently solved by exploiting irregularity, which is crucial for scaling applications to handle large graphs [335].

Graph updates pose challenges to exploiting irregularity for locality because preprocessing the graph into a cache-friendly representation is infeasible when the graph changes over time. Existing static-graph-processing systems optimized for graph irregularity achieve high performance and low space usage by preprocessing a cache-friendly graph partitioning based on vertex degree [95]. In the dynamic setting, however, finding an optimal partitioning is not feasible in the presence of updates. Therefore, existing high-performance dynamic-graph-processing systems such as Aspen [128] use a ***one-size-fits-all*** representation, which pre-selects one type of data structure upfront for all vertices. The one-size-fits-all approach leaves performance on the table because it disrupts spatial locality by using separate per-vertex data structures.

To address these challenges to locality, the ***Terrace*** artifact applies the locality-first strategy to dynamic graph-processing by using cache-friendly data structures that adapt to naturally occurring irregular graph structure [294]. Terrace uses a hierarchical data structure design to dynamically partition vertices based on their degrees and adapt to skewness in the underlying graph. The evaluations in Chapter 3 show that Terrace supports faster batch insertions for batch sizes up to 1M when compared to Aspen [128]. On graph query algorithms, Terrace is between $1.7-2.6\times$ faster than Aspen. Surprisingly, in some cases Terrace is even faster (up to $1.3\times$ faster) than Ligra [335], a state-of-the-art static-graph-processing system, on graph queries. The reason for these performance gains is that Terrace experiences significantly fewer cache misses (in some cases, up to about $3\times$ fewer than Ligra and about $6\times$ fewer than Aspen) during graph queries because it exploits graph skewness for cache-friendliness.

One of the main components of Terrace is the ***Parallel Packed Compressed Sparse Row*** (PPCSR) artifact, a dynamic-graph-processing system that applies the locality-first strategy to enhance spatial locality by co-locating all of its data [379]. PPCSR is built on top of a parallel Packed Memory Array (PMA) [44, 196] data structure. The PMA is well suited for storing and processing dynamic graphs because it supports updates and efficient scans by storing data in one contiguous block of memory [378]. Concurrently updating a PMA raises challenges, however, because an update requires rewriting the entire structure in the worst case [379]. The difficulty of updating a PMA exemplifies the tension between parallelism and cache-friendliness. Surprisingly, one of the main findings in Chapter 4 is that the PMA can achieve the best of both worlds. Specifically, the PMA is well-suited to concurrent updates despite occasionally requiring a rewrite of the entire structure because 1) most of the updates only write to a small part of the structure and 2) the worst case is highly parallel and cache-efficient [379]. These results demonstrate the performance benefits of optimizing for locality first via cache-friendly data structures in dynamic graph processing.

Finding block structure in sparse tensors

This thesis also studies the locality-first strategy in the domain of blocked formats, which address memory bandwidth issues in sparse tensor kernels by enhancing spatial locality. A common issue that arises in parallel implementations of sparse tensor algebra, such as sparse matrix-vector multiplication (SpMV), is limited memory bandwidth due to irregular memory traffic from the locations of the nonzeros.

To regularize memory traffic and improve memory bandwidth issues, researchers have developed ***blocked storage formats*** to take advantage of natural blocked structure, or clusters of nonzeroes, in sparse matrices [373]. Blocked storage formats store dense blocks of nonzeros instead of storing the nonzeros individually to take advantage of the natural blocked structure of some blocked sparse matrices and tensors. These blocked storage formats simplify memory traffic and enable instruction-level parallelism such as vectorization [212]. For example, choosing the correct blocking can speed up SpMV by more than a factor of 2 on matrices with blocked structure [372]. Blocked storage formats must choose a block size that is carefully tuned to match the structure of a tensor to avoid unnecessary overhead, however.

The ***PHIL*** artifact supports the locality-first strategy by enabling efficient usage of blocked storage formats. PHIL estimates the fill, a metric for block-size quality, with provable guarantees. The fill has important applications in autotuning pipelines, but computing the fill exactly is too expensive. As a result, researchers developed OSKI [371], an empirically fast-and-accurate heuristic for estimating the fill that samples matrix rows. In contrast to OSKI, the number of samples that PHIL requires is independent of the size of the input. This advantage of PHIL over OSKI is evident in the empirical evaluation in Chapter 5: PHIL estimates the fill at least $2\times$ faster than OSKI on small matrices and $40 - 50\times$ faster on large matrices. Since PHIL has provable accuracy guarantees, it also provides useful estimates of the fill even in pathological test cases where OSKI fails to estimate the fill within any useful error. PHIL's empirical success demonstrates the potential for efficient algorithms to enhance spatial locality in sparse tensor applications.

Enhancing spatial locality in data structures

The ***write-optimized skip list*** artifact uses the locality-first strategy to enhance spatial locality in general data structures. Although the write-optimized skip list is a serial data structure, it is a first step towards an efficient parallel and cache-efficient skip list. The skip list is an elegant and simple randomized in-memory data structure that supports efficient searches, inserts, and deletes. As we shall see in Chapter 6, there are theoretical challenges to getting the skip list to generalize well to external memory, however. The intuitive method to convert an in-memory skip list to an out-of-memory skip list does not achieve good high-probability guarantees. The write-optimized skip list overcomes these challenges and to achieve ***write-optimized*** bounds. That is, given a memory-block size B, for $0 < \varepsilon < 1$, a write-optimized skip list on N elements supports queries of size K in $O(\log_{B^\varepsilon} N + K/B)$ I/Os with high probability (w.h.p.) and insertions and deletions in $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ amortized I/Os w.h.p. That is, the write-optimized skip list supports asymptotically optimal searches, inserts, and deletes. Furthermore, it supports inserts and deletes asymptotically faster than searches. The write-optimized skip list applies the locality-first strategy by first exploiting spatial locality in data structure design as a step towards efficient parallel data structures.

2.4 Exploiting locality without changing the data layout

The cyclic analysis, lower bounds, smoothed analysis, scan-hiding, `CAST_BLK`, `PAIR_BLK`, bidirectional box-sum, and box-complement artifacts apply the locality-first strategy to exploit locality in problems without changing the data layout. These artifacts take advantage of naturally-occurring spatial and temporal locality by modifying algorithm access patterns to make them cache-friendly (Chapters 7-12).

The cyclic analysis and smoothed analysis artifacts theoretically ground the locality-first strategy via "beyond-worst-case" analysis [315] and demonstrate that algorithms that are designed to fully exploit temporal locality are mathematically good despite potential disruptions to cache-friendly access patterns due to parallelism. These artifacts study the case where algorithms exhibit temporal locality, but multiple threads compete for space in a shared cache. These artifacts address a potential concern with the locality-first strategy that locality trades off with parallelism.

These artifacts characterize the practical benefits of the locality-first strategy by bringing theory and practice together via "beyond-worst-case" analysis. ***Beyond-worst-case*** analysis is a class of algorithmic techniques that consider algorithm performance outside of a single worst-case input. Beyond-worst-case analysis closes the gap between theory and practice in cases where worst-case analysis fails to capture realistic inputs. For example, it compares algorithms on more realistic inputs, such as those with locality, whereas traditional worst-case analysis compares algorithms on a single worst-case input. As a result, beyond-worst-case analysis presents a more holistic view of algorithm performance.

This section overviews this thesis' contributions in mathematically grounding the locality-first strategy with respect to two problems: "multicore cache-replacement" and "cache-adaptive algorithms." Specifically, it summarizes the cyclic analysis and lower bounds artifacts for "multicore cache-replacement" and the smoothed analysis and scan-hiding artifacts for "cache-adaptive algorithms." For each problem, this section describes challenges to locality due to parallelism in shared caches, summarizes the relevant artifacts, and explains how they support the locality-first strategy.

Finally, this section demonstrates how to apply the locality-first strategy when the problem has spatial locality and not much temporal locality. Since spatial locality is easy to achieve in these problems, the main focus is on other optimizations. By understanding locality in the problem first and exploiting it as much as possible, the locality-first strategy created the `CAST_BLK` and `PAIR_BLK` artifacts for the "prefix sums" problem, the bidirectional box-sum artifact for the "included sums" problem, and the box-complement artifact for the "excluded sums" problem.

Multicore cache replacement

As we shall see in more detail in Chapter 8, multiple parallel threads in a multicore environment disrupt the cache-friendliness of each individual thread's memory access pattern by causing cache evictions due to threads contending for space. Every processor with a cache needs to implement a cache-replacement algorithm, or an algorithm that manages the data in cache by deciding what to evict and what to keep when the cache becomes full. Single-core cache-replacement is a classical problem in online algorithms [342] and has inspired a decades-long line of both theoretical and practical research [195, 207]. The multicore setting differs significantly from the single-core setting, however: for example, López-Ortiz and Salinger [251] demonstrated that competitive ratio of canonical cache-replacement algorithms such as Least-Recently-Used and Furthest-In-Future grows with the length of the input in the multicore setting (as opposed to growing with the size of the cache in the single-core setting). This divergence between multicore and single-core cache-replacement mathematically validates the tension between parallelism and cache-friendliness.

The cyclic analysis and lower bounds artifacts mathematically ground the locality-first strategy for the multicore cache-replacement problem by validating the empirical superiority of the Least-Recently-Used (LRU) algorithm because LRU takes advantage of locality.

The ***cyclic analysis*** artifact mathematically justifies the locality-first strategy by showing the advantage of LRU in the presence of temporal locality via the ***cyclic analysis*** artifact, a new technique for beyond-worst-case analysis [210]. Cyclic analysis compares online algorithms on the entire space of inputs rather than a single worst-case input. Under cyclic analysis, LRU is the single best online algorithm on inputs with (temporal) locality in the multicore setting. These results demonstrate the potential of alternative beyond-worst-case measures to separate multicore cache-replacement algorithms and to capture real-world performance. Specifically, cyclic analysis grounds the observed superiority of LRU in practice because it exploits naturally-occurring locality [8].

The ***lower bounds*** artifact motivates beyond-worst-case analysis for multicore caching algorithms by showing that all known deterministic algorithms are equally arbitrarily far from optimal [209]. This work answers an open question from past work about the existence of a competitive algorithm in the negative. This lower bound fails to capture real-world differences between algorithms due to naturally-occurring locality, however. Despite these negative equivalence results from worst-case analysis, algorithms exhibit different real-world performance due to locality, motivating alternative measures that capture these differences.

Cache-adaptive algorithms

Additionally, multiple programs sharing a cache each experience ***memory fluctuations***, or dynamically changing cache sizes,that potentially disrupt each program's cache-friendliness [94, 122, 123]. Systems that require programs to share a cache such as shared-memory machines, multicore architectures, and time-sharing systems are ubiquitous in modern computing. Memory fluctuations are the common case in shared-memory machines such as multicores, where threads share cache and RAM. Multiple programs in a shared cache pose challenges to taking advantage of locality, however. For example, optimally cache-friendly algorithms in a fixed-size cache may not be optimal when the cache size changes over time. In the worst case, an algorithm may become logarithmically worse when the cache size fluctuates [45]. This divergence between algorithm performance in fixed-size and variable-size caches further reinforces the tension between parallelism and cache-friendliness.

To cope with memory fluctuations, experimentalists have developed heuristics and experimentally fast algorithms that perform well in practice but unfortunately are vulnerable to worst-case inputs [295, 296]. These algorithms include empirically efficient algorithms for major operations such as database sorts and joins. Although these empirical solutions achieve performance improvements on practical workloads, they lack theoretical analysis and worst-case guarantees.

To provide mathematical grounding for algorithms in the face of cache fluctuations, researchers have studied ***cache-adaptive algorithms*** that gracefully handle changes in cache allocation with worst-case guarantees [43,45]. An algorithm is cache adaptive if it achieves optimal utilization of the dynamically changing cache. Prior work on cache adaptivity, used worst-case analysis to separate algorithms, however, which may be overly pessimistic due to carefully constructed worst-case inputs.

The smoothed analysis and scan-hiding artifacts mathematically ground the locality-first approach for cache-adaptive algorithms. These artifacts prove that algorithms that take advantage of temporal locality as much as possible adapt well to cache fluctuations.

The ***smoothed analysis*** artifact supports the locality-first strategy by showing that "cache-obliviousness" provides a solid foundation for adaptivity when the ***memory profile***, or the sequence of fluctuations in the memory size, is not adversarial [41]. Smoothed analysis [348] is a type of beyond-worst-case analysis that considers algorithm performance on a shuffled (randomized) input. Additionally, ***cache-oblivious algorithms*** achieve asymptotically optimal performance on all fixed cache sizes without knowledge of the memory size [153,154]. They may perform poorly on the worst-case memory profile, however, when the available memory changes over time [45]. Smoothed analysis closes this gap between cache-obliviousness and cache-adaptivity: if one takes an arbitrary profile and performs a random shuffle on the location of "significant events" in the profile, then the shuffled profile becomes optimally cache-adaptive in expectation, even if the initial profile is adversarially constructed. The optimal expected performance of cache-oblivious algorithms mathematically supports the locality-first strategy because cache-oblivious algorithms take advantage of temporal locality.

The smoothed analysis artifact builds on the ***scan-hiding*** artifact, which applies the locality-first strategy to convert non-cache-adaptive algorithms to cache-adaptive algorithms by improving their temporal locality [246]. Scan-hiding applies to a large class of non-cache-adaptive algorithms, including all currently known subcubic matrix multiplication algorithms. These results provide guidance for analyzing cache-adaptive algorithms beyond the adversarial worst case. They ground the locality-first strategy by proving that algorithms that exploit temporal locality perform well.

Problems with high spatial locality and low temporal locality

The `CAST_BLK`, `PAIR_BLK`, bidirectional box-sum, and box-complement artifacts study the case where spatial locality is easy to achieve, so the main contributions are in other optimizations. These artifacts differ in their techniques due to the problem domain that each addresses, but they all involve first understanding locality and exploiting it as much as possible.

The **`CAST_BLK`** and **`PAIR_BLK`** artifacts, two fast-and-accurate algorithms for parallel prefix sums, involve first understanding and exploiting locality before optimizing for accuracy. Parallel prefix sums are a fundamental subroutine in a wide range of applications and have therefore been targeted for efficient implementations [61, 187]. Specifically, floating-point prefix sums appear in scientific computing applications such as summed-area table generation [178] and the fast multipole method [118]. Unfortunately, floating-point summation introduces error due to limited machine precision [182]. Therefore, the goal of the ***accurate prefix sums*** problem is to minimize error while maintaining good performance. The `CAST_BLK` and `PAIR_BLK` algorithms achieve comparable performance to the state of the art with significantly higher accuracy. They employ the locality-first strategy by first understanding locality in the accurate prefix sums problem and then optimizing for parallelism and accuracy because locality is relatively easy to achieve in prefix sums.

The ***bidirectional box-sum*** artifact for the "included-sums" problem and the ***box-complement*** artifact for the "excluded-sums" problem apply the locality-first strategy to understand and exploit locality before optimizing the overall work of the algorithms. As we shall see, these problems have opportunities for exploiting naturally-occurring spatial locality, but not much temporal locality. The included-sums and excluded-sums problems underlie scientific computing applications such as summed-area table generation and the fast multipole method. These problems take as input a d-dimensional array, a d-dimensional box size, and a binary associative operator $\oplus$. The ***included-sums*** problem requires that the elements within overlapping boxes cornered at each element within the array be reduced using $\oplus$. The ***excluded-sums*** problem reduces the elements outside each box. The bidirectional box-sum and box-complement artifacts improve the state-of-the-art theoretical and practical performance of included-sums and excluded-sums algorithms, respectively. Specifically, given a d-dimensional tensor of size N, these artifacts reduce the total work for the included-sums and excluded-sums problems from $\Theta(2^d N)$ to $\Theta(dN)$. The bidirectional box-sum and box-complement artifacts take advantage of spatial locality and improve overall efficiency by asymptotically reducing the work required to solve the problems when compared to the state of the art.

Chapter 3

Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs

This chapter presents Terrace, a dynamic-graph-processing framework that overcomes traditional tradeoffs between query and algorithm speed in dynamic graph processing by leveraging the locality-first strategy with cache-friendly data structures. Terrace supports graph queries about 2× faster than Aspen [128], a state-of-the-art high-performance dynamic-graph-processing system, while maintaining similar update throughput. Terrace enhances spatial locality in dynamic graph processing with dynamic cache-friendly data structures that adapt to naturally-occurring skewness in the underlying graph. As we shall see in the exhibits in this chapter, Terrace experiences significantly fewer cache misses than Aspen. With additional optimizations for locality, Terrace even experiences fewer cache misses than Ligra [335], a state-of-the-art static-graph-processing system, on some applications. This improved cache performance translates into improved graph algorithm performance without sacrificing updatability.

This work was conducted in collaboration with Prashant Pandey, Brian Wheatman, and Aydın Buluç [294].

Abstract

Various applications model problems as streaming graphs, which need to quickly apply a stream of updates and run algorithms on the updated graph. Furthermore, many dynamic real-world graphs, such as social networks, follow a skewed distribution of vertex degrees, where there are a few high-degree vertices and many low-degree vertices.

Existing static graph-processing systems optimized for graph skewness achieve high performance and low space usage by preprocessing a cache-efficient graph partitioning based on vertex degree. In the streaming setting, the whole graph is not available upfront, however, so finding an optimal partitioning is not feasible in the presence of updates. As a result, existing streaming graph-processing systems take a "one-size-fits-all" approach, leaving performance on the table.

This chapter presents Terrace, a system for streaming graphs that uses a hierarchical data structure design to store a vertex's neighbors in different data structures depending on the degree of the vertex. This multi-level structure enables Terrace to dynamically partition vertices based on their degrees and adapt to skewness in the underlying graph.

The experiments show that Terrace supports faster batch insertions for batch sizes up to 1M when compared to Aspen, a state-of-the-art graph streaming system. On graph query algorithms, Terrace is between 1.7×–2.6× faster than Aspen. Terrace is also between 0.5×–1.3× as fast as Ligra, a state-of-the-art static graph-processing system. Surprisingly, in some cases, Terrace is even faster than a static system without updates because Terrace exploits locality.

3.1 Introduction

Many real-world sparse graphs, such as social networks or road networks, change over time. Therefore, systems for storing and processing dynamic (i.e. streaming) graphs [83, 128, 139, 144, 168, 229, 254] have been designed to process a stream of updates (e.g., edge weight update, or edge insertions and deletions) and a stream of queries quickly. That is, both query-processing time and graph-update time must be fast.

The ability to quickly apply a batch of updates is critical for efficient streaming graph processing. For example, in incremental triangle counting, insertion (or deletion) time accounted between 25% – 90% of the overall time [255]. Similarly, on 32 cores, updating the graph takes up to 90% of the overall running time in incremental connected components [263]. This chapter focuses on data structure design for dynamic graph processing in order to support both efficient updates and queries.

In practice, dynamic real-world graphs follow skewed vertex degree distributions as shown in Table 3.1. For example, real-world graphs, such as those from social networks [141, 279] or computational biology, contain a few very high-degree vertices and many low-degree vertices. This skewness presents unique challenges for efficiently representing dynamic graphs. However, these diverse distributions also present an opportunity to build cache-efficient graph representations via adaptive data structures that take advantage of degree distributions.

Existing static graph-processing systems that optimize for skewness demonstrate the potential for improved cache locality. For example, PowerLyra [95] partitions vertices based on their degree to improve locality of vertex computations. Other frameworks preprocess the graph into cache-friendly formats to improve locality. For example, Cagra [403] uses segmenting to divide the graph into cache-friendly subgraphs. Similarly, Gridgraph [404] partitions vertices and edges into blocks for locality. These techniques greatly improve locality in computations on static graphs, but do not easily translate to graphs that evolve over time. GPU-based static graph processing systems also exploit the skewness to support fast graph algorithms and use the vertex's degree to decide which scheduler to use to run iterations [157, 247, 293].

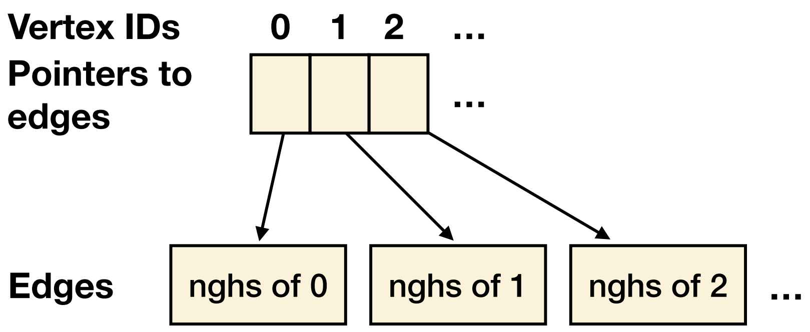


Figure 3-1: A high-level design for graph storage formats. There is a vertex structure that keeps track of where the neighbors (nghs) for each vertex are stored, and a structure for each vertex's edges.

Graph	*% <10 Nghs*	*% <100 Nghs*	*% <1000 Nghs*
LiveJournal	65	97.2	99.98
Twitter	64.56	95.39	99.51
Protein	30.47	61.49	98.80

Table 3.1: Distribution of degree of vertices in three different real-world graphs. Columns show the % of vertices that have less than 10, 100, 1000 neighbors (nghs). The maximum degree in the graphs are: LiveJournal (20333), Twitter (2997487), and Protein (3779).

In contrast, many existing dynamic graph-processing systems take a "one-size-fits-all" approach to data structure design, leaving performance on the table when processing and updating skewed graphs. Figure 3-1 illustrates a classical design for a graph storage format: a list of pointers (one for each vertex) to preselected data structures holding each vertex's neighbors (nghs). For example, the canonical static Compressed Sparse Row [361] (CSR) format stores a list of offsets into an edge list. Dynamic graph systems adopt a similar two-level design: Stinger [139] stores neighbors in a variant of a blocked adjacency list, while Aspen [128] stores each vertex's neighbors in a separate probabilistic balanced tree (C-tree). Since the neighbor data structures can only be accessed after a memory indirection, these dynamic systems must incur at least two cache misses per vertex during a graph traversal. Moreover, in tree-based representations such as Aspen, traversing a vertex's neighbors requires non-sequential memory accesses, which are slower than sequential memory accesses in array-based representations such as CSR.

The ideal structure for storing a vertex's neighbors in a dynamic graph framework depends on the access pattern of graph algorithms and the cost of doing updates. If a vertex has low degree, a simple data structure such as an array incurs minimal indirection and supports efficient traversal and updates. If a vertex has high degree, however, a more complex data structure such as a tree with better asymptotic search and update performance may be more suitable. Even though a balanced tree may have asymptotically better performance than an array, in the context of storing a vertex's neighbors, these data structures exhibit crossover points in their performance depending on the degree of the vertex.

Characterizing graph skewness. Table 3.1 presents the distribution of vertex

Kernel	*Ligra*	*Aspen*	*Terrace*
BFS	3.5M	6.3M	1.1M
PR	174M	197M	128M

Table 3.2: Average cache misses in breadth-first search (BFS) and PageRank (PR) on the LiveJournal graph over 100 rounds. Cache misses are higher in PR as it was run for 10 iterations compared to a single iteration in BFS.

degrees in three different real-world graphs that exhibit skewness. These graphs are picked from three different domains. A major fraction of all the vertices in these graphs have less than 10 neighbors which can be easily packed in a single cache line along with other meta information about the vertex, e.g., the vertex degree. However, there is also high variance between degree of vertices: the maximum degree in these graphs goes up to 2.99 million (in the Twitter graph). Therefore, the high-degree vertices must be stored in a sophisticated structure to enable efficient updates and queries. Furthermore, graph-processing systems must treat low- and high-degree vertices differently to achieve better cache locality and good performance.

Exploiting skewness in streaming graphs. This chapter introduces Terrace, a dynamic graph-processing framework that exploits skewness present in real-world graphs to build a cache-optimized representation. The main idea behind Terrace is a hierarchical data structure design that stores a vertex's incident edges in different data structures based on its degree. That is, a vertex's degree determines what type of data structure its edges will be stored in. The hierarchical design and degree cutoffs can be adapted to the distribution of a particular graph for improved performance and space usage.

A key insight behind Terrace is that neighbors of low-degree vertices can be stored *in place* rather than in a separate data structure, reducing latency and improving locality. That is, a few neighbors of each vertex can be stored directly in the vertex structure. Storing neighbors in-place in the vertex structure avoids cache misses for low-degree vertices during a graph traversal because it avoids following pointers for low-degree vertices.

At a high level, Terrace stores edges in three main types of data structures: a sorted array that stores a few neighbors per vertex in place, a shared Packed Memory Array [44, 196] (PMA) that compactly stores neighbors of medium-degree vertices, and per-vertex B-trees [108, Chapter 18] for high-degree vertices. The PMA and B-tree are cache-efficient structures with asymptotically better update and query costs than traditional packed lists.

Cache miss analysis. Existing static and dynamic graph-processing systems incur a high number of cache misses during graph kernels because they use a uniform out-of-place per-vertex structure regardless of vertex degree. To verify this hypothesis, Table 3.2 reports[1] the number of cache misses during graph kernels in Ligra [335] and Aspen [128], two state-of-the-art graph processing systems, as well as in Terrace. The

[1]This experiment measures the number of cache misses using the `perf` utility in Linux. To compute the average number of cache misses, the experiments measures the total cache misses for 1, 10, 20, 100 rounds of kernel runs and then computes the average for a single run.

experiment measures cache misses during breadth-first search and PageRank [387], as these two kernels have distinct access patterns and can be used as representatives for access patterns in other graph kernels. Ligra is a static graph framework that stores its edges in CSR format, while Aspen supports dynamic graphs using compressed trees. Both Ligra and Aspen incur more cache misses than Terrace because they require indirection to access neighbors for all vertices, while Terrace stores neighbors of low-degree vertices in place. The improved locality in Terrace translates into graph kernel performance: Figure 3-3 summarizes the results of the evaluation.

Contributions

To be specific, the contributions in this chapter are as follows:

- The design of a dynamic graph-processing system using hierarchical data structures for improved locality.
- An implementation of Terrace, a graph-processing system using the hierarchical design in Cilk [191].
- An experimental study of Terrace compared to Aspen [128] and Ligra [335], two state-of-the-art graph-processing frameworks, that demonstrates that Terrace supports faster updates and queries.

The goal of this chapter is to demonstrate how to organize vertex neighbors dynamically in a hierarchical way rather than in a "one-size-fits-all" framework. Although the idea of handling low- and high-degree vertices separately has been introduced in the static setting, this work takes the first step in hierarchical processing for the dynamic setting. Therefore, one of the main contributions is the multilevel design of Terrace and the characterization of desirable data structure properties at each level rather than a new data structure. The simplicity of the design of Terrace is its strength.

In terms of evaluation, this chapter compares Aspen and Terrace on update throughput, and all systems on graph kernel performance. There is an extension of Ligra, called Ligra+ [337], that adds compression on top of the regular graph representation in Ligra. On the tested graphs, Ligra+ was slower than Ligra although more space-efficient, so this chapter only includes the results for Ligra.

The implementation of Terrace extends the interface proposed by Ligra [335] with functionality for updating the graph. Therefore, all algorithms implemented with Ligra and Aspen, such as graph-traversal algorithms, local graph algorithms [338], and others [126, 127] can be run on top of Terrace with minor cosmetic changes.

Figure 3-2 shows that Terrace achieves up to 80 million updates per second and supports faster batch insertions (between 1.1×–3.1×) for batch sizes up to 1M when compared to Aspen. Table 3.6 contains the full results of batch insertions and deletions in Terrace and Aspen. Figure 3-3 shows that Terrace performs the shared graph kernels 1.7×–2.6× faster than Aspen and up to 1.3× faster than Ligra. On

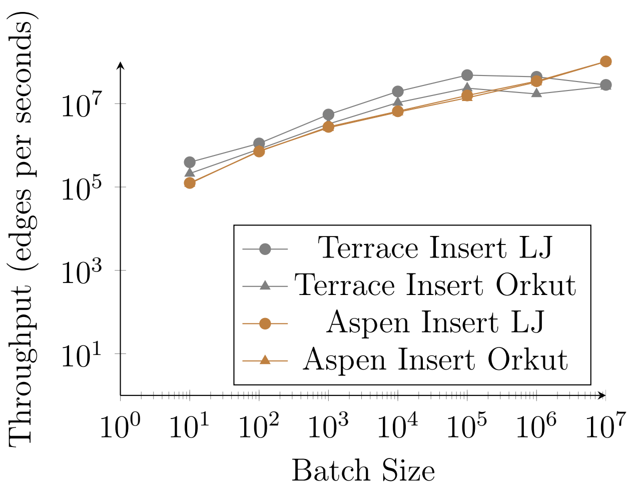


Figure 3-2: Batch insert throughput in Aspen and Terrace as a function of batch size on the LJ and Orkut graphs. The LJ graph has about 85 million edges, while the Orkut graph has about 234 million edges.

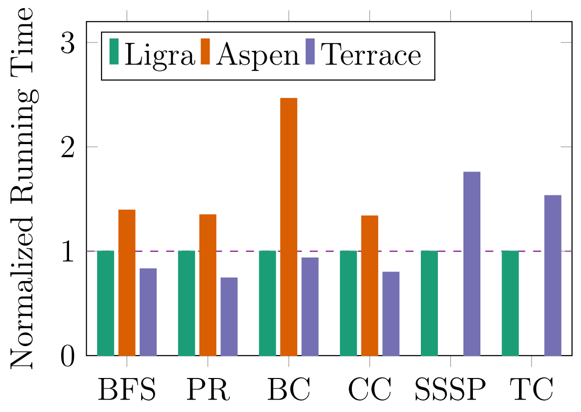


Figure 3-3: Average time to run kernels across all graphs in Ligra, Aspen, and Terrace normalized to Ligra. The four kernels tested for all systems were breadth-first search (BFS), PageRank (PR), single-source betweenness centrality (BC), and connected components (CC). Aspen does not have publicly available implementations of single-source shortest paths (SSSP) or triangle counting (TC), so this plot omits it from SSSP and TC.

the kernels that do not have implementations in Aspen, Terrace is about 1.6× slower than Ligra.

Terrace overcomes traditional tradeoffs between fast updates and locality of graph computations. Existing state-of-the-art systems lie on either end of the spectrum: for example, Ligra is a static system and faster for graph computations, while Aspen is dynamic but slower for graph computations. Terrace shows how to support updates as fast as Aspen while being faster or similar to Ligra for graph computations.

Map. The rest of the chapter is organized as follows. Section 3.2 provides background on graph processing and the data structures underlying Terrace. Section 3.3 discusses the high-level hierarchical design of Terrace, and Section 3.4 concretizes that design with specific data structures and a comparative theoretical analysis. Section 3.5 describes the implementation of Terrace. Section 3.6 provides an empirical evaluation of Terrace compared to Ligra and Aspen. Finally, Section 3.7 reviews related work and Section 3.8 provides concluding remarks.

3.2 Preliminaries

This section introduces reviews graph preliminaries necessary to understand the data stored in graph-processing systems. It also formalizes the B-tree and Packed Memory Array data structures that underlie Terrace.

Graph preliminaries. A graph is a way of storing objects as ***vertices*** and connections between those objects as ***edges***.

Definition 3.1 (Graph) *A graph $G = (V, E, w)$ is a set of vertices V, a set of edges E, and an edge weight function w. This thesis denotes the number of vertices with $|V|$, the number of edges with $|E|$, and the degree*[2] *of a vertex $v \in V$, or the number of edges incident to vertex v, with $deg(v)$. Each vertex $v \in V$ is represented by a unique non-negative integer less than $|V|$ (i.e. $v \in \{0, 1, \ldots, |V| - 1\}$). Each edge is a 2-tuple (u, v) where $u, v \in V$. Finally, the weight function w maps each edge $e \in E$ to a non-zero real weight ($w(e) \in \mathbb{R}, w(e) \neq 0$).*

Dynamic-graph-processing systems must store and process graph vertices and edges.

B-trees. The ***B-tree*** data structure generalizes balanced binary trees to work well in the external-memory model (Chapter 1) and is widely used in databases [108, Chapter 18]. This chapter considers B-trees with node size (fanout) $\Theta(B)$, where B is the cache-line size from the external-memory model. A B-tree on N elements takes $O(N)$ space and supports updates and point queries in $O(\log_B N)$ transfers. Furthermore, a B-tree on N elements supports range queries in $O(\log_B N + k/B)$ transfers, where k is the number of elements in the query range. Since a B-tree takes $O(N)$ space, scanning a B-tree takes $O(N/B)$ transfers. B-trees support fast updates but are slower to traverse than array-based structures because their nodes are not contiguous in memory.

Packed Memory Array. The Packed Memory Array [44, 196] (PMA) data structure is an array-based order-maintenance data structure that keeps spaces between elements. A PMA on N elements takes $O(N)$ space and supports updates in amortized and worst-case $O(\log^2(N/B))$ transfers in the external-memory model. Point queries in a PMA take $O(\log(N/B))$ transfers and range queries that return k elements take $O(\log(N/B)+k/B)$ transfers. Since the PMA takes $O(N)$ space, scanning the entire PMA takes $O(N/B)$ transfers. Since the focus of Chapter 4 is on parallelizing PMAs, Chapter 4 contains additional details about the PMA's structure and bounds.

[2]This chapter focuses only on directed graphs and uses degree to mean out-degree. An undirected graph can be represented by a directed graph with edges in both directions.

Although, B-trees asymptotically dominate PMAs in terms of updates and queries, in practice PMAs are faster to scan because their elements are stored contiguously in memory. Due to these properties, as we shall see in detail in Chapter 4, PMAs are used to efficiently represent sparse graphs [240, 378, 379].

3.3 Hierarchical data structure design

This section proposes the three-level data structure design that Terrace implements to take advantage of skewness in graphs. This hierarchical design represents a vertex's incident edges in different data structures depending on that vertex's degree, in contrast to the classical "one-size-fits-all" design that uses one type of data structure for all vertices. The hierarchical design improves cache-friendliness without sacrificing updatability by choosing cache-friendly data structures as much as possible.

Balancing locality and updatability. The first principle in the design of Terrace is that order-maintenance array-based and tree-based data structures provide different guarantees and exhibit crossover points in terms of updatability and traversal cost. Trees designed for the external-memory model (e.g. B-trees) are quick to update and achieve asymptotically optimal cost to list all elements, but access memory out-of-order. In contrast, ordered array-like structures have asymptotically worse insertion cost than trees, but support fast traversals because they are stored contiguously in memory. In practice, there is a crossover point in the update performance of tree-like and array-like structures based on the number of elements in the structures. Therefore, the choice of structure for a vertex's neighbors should depend on that vertex's degree.

Separating vertices based on degree. The next principle in the design of Terrace is that vertices should share contiguous array-based structures for locality, but only if their degree is not too high. Sharing an array-like structure between vertices avoids cache misses while switching vertices during a traversal through the edges. If the vertices have high degree, however, the effect of saving a single cache miss per vertex is negligible because the cost to traverse all the edges dominates. Furthermore, sharing the data structure between vertices trades improved locality for slower updatability because the update cost depends on the total size of the structure. Storing high-degree vertices in an array-like structure will slow down updates for all vertices in the structure regardless of their degree. Therefore, high-degree vertices should store their neighbors in separate per-vertex data structures so they do not affect the cost of updating smaller-degree vertices. High-degree vertices are more suited to tree-based structures, because they require better asymptotic updatability guarantees.

One size does not fit all. Since the benefit of a contiguous data structure depends on the degree of vertices that use it, this chapter proposes that graph systems store vertex neighbors in either array-like or tree-like structures based on vertex degree. Specifically, it proposes a hierarchical design that stores the neighbors of ***medium-degree*** vertices in a shared array-based structure and the neighbors of ***high-degree*** vertices in per-vertex trees.

Storing the neighbors of medium-degree vertices in an array-based structure improves cache locality during traversals. The hierarchical design limits the maximum degree that any vertex in the array-based structure can have, so the total size of the array-based structure is bounded. In contrast, storing the neighbors of high-degree vertices in per-vertex trees ensures that updating those vertices does not bottleneck the update throughput of the entire system.

Storing neighbors in place. In addition to storing neighbors in different data structures based on vertex degree, one natural optimization is to store some neighbors *in place* because accessing neighbors requires accessing at least one cache line to look up the pointer to the next data structure. Storing each vertex's neighbors in an out-of-place data structure disrupts locality during graph queries and updates. In contrast, storing some neighbors in place in the same cache line can save a cache miss from accessing a separate data structure.

Therefore, this section proposes the following the three-level design:

1. A list of in-place neighbors and any necessary metadata for each vertex,
2. a shared array-based data structure containing neighbors of medium-degree vertices, and
3. individual tree-based data structures for each high-degree vertex.

3.4 Data structure choices

This section describes Terrace, an implementation of the high-level hierarchical design in Section 3.3 using the PMA and B-tree data structures from Section 3.2. To concretize the implementation, this section theoretically analyzes the resulting system. Since the hierarchical design chooses different data structures depending on vertex degree, the resulting theoretical analysis falls into cases depending on vertex degree. Finally, this section concretizes the tradeoffs between updatability and scan performance with a microbenchmark that compares PMAs and B-trees.

Implementation overview. Terrace concretizes the first level of the hierarchical design with an array of ***vertex blocks*** containing metadata and in-place neighbors for each vertex. Next, Terrace implements the second level as a Packed Memory Array (Section 3.2) as an associative structure to store the neighbors of medium-degree vertices. Finally, Terrace implements the third level with individual B-trees [108, Chapter 18] for each high-degree vertex.

This chapter denotes the maximum number of in-place neighbors per vertex with the parameter S and the maximum number of neighbors per vertex in the PMA with the parameter L. A vertex can have all its neighbors stored in place if its degree is less than S, or spread across the in-place and PMA levels or in-place and B-tree levels depending upon whether its degree is greater or smaller than $S + L$. That is, if a vertex v has neighbors only in place, $\deg(v) \leq S$. If a vertex v has neighbors in the in-place and PMA level, $S < \deg(v) \leq S + L$. Similarly, if a vertex v has neighbors in the in-place and B-tree level, $\deg(v) > S + L$.

In-place level. The first level in Terrace consists of a list of vertex blocks designed to store a few neighbors of each vertex in place and avoid a cache miss for accessing the neighbors of in-place vertices. Each vertex has a corresponding vertex block, and vertex blocks are ordered by vertex index. The vertex block corresponding to vertex v stores the degree of v, up to S neighbors of v sorted in place, and a pointer to the root of the corresponding B-tree in the third level (if v has high degree). The number of in-place neighbors S is a configurable parameter and is adjusted so that each vertex block can fit in a cache line or two, if Terrace must store extra attributes (e.g., weights) per edge.

Array-like level. The second level in Terrace stores up to L neighbors per medium-degree vertex in a single shared PMA to support cache-efficient traversals when edges are accessed in order. Chapter 4 details the parallel PMA in Terrace. Storing neighbors in the PMA provides good cache locality since all neighbors of a given vertex are stored in consecutive memory locations, like in the edge list of CSR. The cost of performing an update or query operation in a PMA is asymptotically higher than in a B-tree, however. Since the cost to update the PMA in Terrace depends on the total PMA size, Terrace limits the degree of each vertex that stores its neighbors in the PMA.

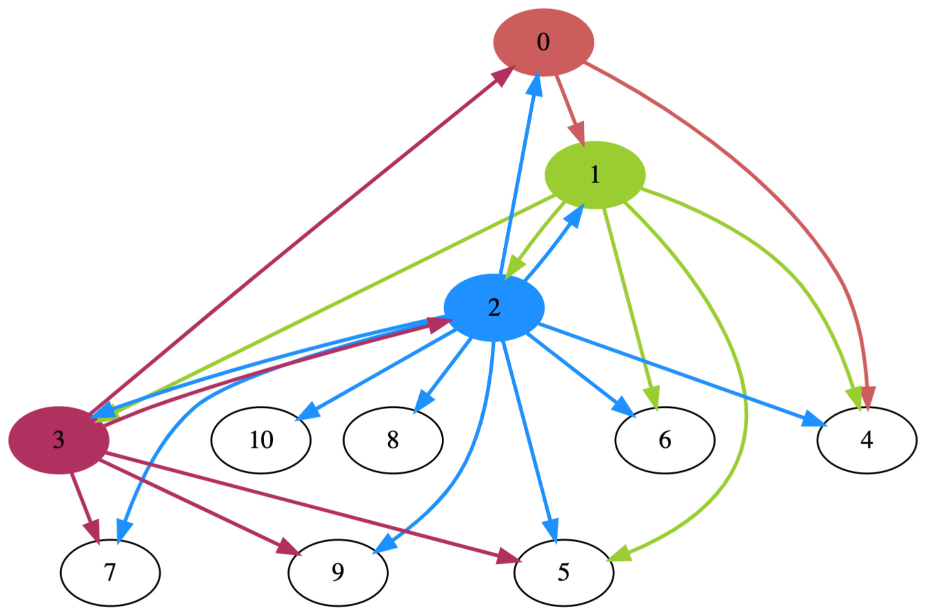
The maximum number of neighbors per vertex in the PMA level, L, is a configurable parameter that balances update throughput and cache locality in Terrace. That is, the parameter L exploits the crossover point between PMA and B-tree insertions in practice: when neighbors of a vertex are stored in a few consecutive pages, insertions in a PMA are competitive with insertions in a B-tree even though B-tree insertions asymptotically dominate PMA insertions.

Tree-like level. The third level in Terrace consists of individual B-trees (one for each vertex with degree $> S + L$). B-trees are a good candidate for storing high-degree vertices because they are quick to modify, have minimal space overhead, and support asymptotically optimal scans.

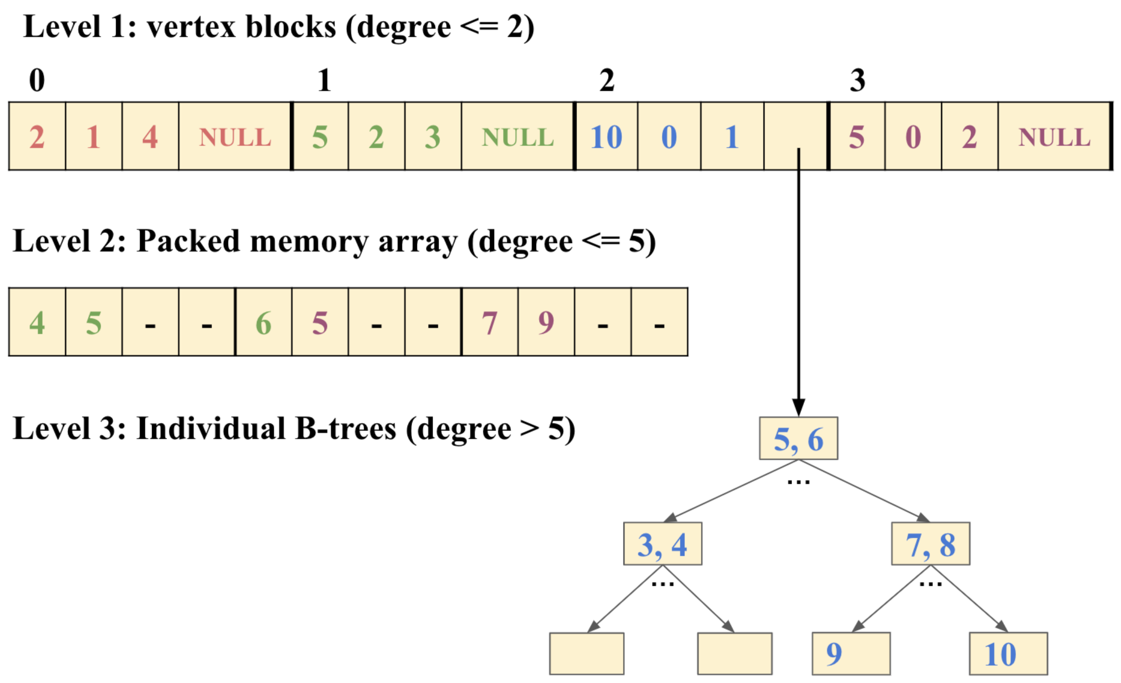
As we shall see, in practice, PMAs and B-trees exhibit a tradeoff between scan and update performance because they have different amounts of spatial locality. PMAs support faster scans while B-trees support faster inserts. The hierarchical design balances each data structure's strengths to overcome traditional tradeoffs that arise from choosing one type of data structure.

Putting it all together. As illustrated in Figure 3-4b, the neighbors of any vertex may be stored in at most two levels in Terrace. Each vertex has a vertex block in the first level. Each vertex block can only store a small number of neighbors, however. If a vertex's neighbors do not fit in its vertex block, its remaining neighbors are stored in either the PMA or B-tree level. Terrace maintains a global order of neighbors for each vertex across different levels, i.e., the in-place neighbors are always in sorted order and the biggest in-place neighbor is smaller than the smallest neighbor in the PMA or B-tree.

Figure 3-4b illustrates how Terrace stores four vertices with different degrees when $S = 2, L = 3$. Vertex 0 has only two neighbors, so all of its neighbors fit in the first level. Vertices 1 and 3 have 5 neighbors each, so their neighbors are distributed between the in-place and PMA level. The first two neighbors are stored in the vertex block and the next three neighbors are stored in the PMA. Vertex 2 has 10 neighbors, so its first two neighbors are stored in its vertex block and the last field in the vertex block contains the pointer to the root of the corresponding B-tree where rest of the neighbors are stored.



(a) A sample directed graph.


Level 1: vertex blocks (degree <= 2)
0
1
2
3
2 1 4 NULL
5 2 3 NULL
10 0 1
5 0 2 NULL
Level 2: Packed memory array (degree <= 5)
4 5 - - 6 5 - - 7 9 - -
Level 3: Individual B-trees (degree > 5)
5, 6
3, 4
7, 8
9
10

(b) An example showing how vertices of different degrees are stored in Terrace. Specifically, Terrace stores vertices 0-3 from the graph in Figure 3-4a when $S = 2, L = 3$. The first level is an array of vertex blocks. The second level is a shared PMA, and the last level consists of individual per-vertex B-trees.

Figure 3-4: An example of a graph stored in Terrace. If a vertex has reasonably high degree, its edges may be stored across multiple data structure levels.

Operation	*Ligra [335]*	*Aspen [128]*		*Terrace*	
				$O(S/B)$	when $\deg(u) \leq S$
`add_edge`(u,v)	$O((\lvert E\rvert + \lvert V\rvert)/B)$	$O(\log \lvert V\rvert + c^2 \log(\deg(u))/B)$	in exp.	$O(S/B + \log^2(\texttt{PMA\_SIZE}/B))$	when $S < \deg(u) \leq S+L$
				$O(S/B + \log_B(\deg(u) - S))$	when $\deg(u) > S+L$
				$O(S/B)$	when $\deg(u) \leq S$
`find_edge`(u,v)	$O(\log(\deg(u)))$	$O(\log \lvert V\rvert + c/B)$	in exp.	$O(S/B + \log((\deg(u) - S)/B))$	when $S < \deg(u) \leq S+L$
		$O(\log \lvert V\rvert + c\log(\deg(u))/B)$	w.h.p.	$O(S/B + \log_B(\deg(u) - S))$	when $\deg(u) > S+L$
`get_neighbors`(u)	$O(\deg(u)/B)$	$O(\log \lvert V\rvert + \deg(u)/B + deg(u)/c)$		$O(\deg(u)/B)$	

Table 3.3: The table lists the theoretical runtime performance of graph representations storing a graph $G(V, E)$. All bounds are $\Omega(1)$, but this table omits the added 1 for ease of notation. The parameter c is expected size of nodes in Aspen (called b in the Aspen paper [128] and set to 2^8). Furthermore, S and L denote the cutoffs for the medium-degree and high-degree structures in Terrace, and `PMA_SIZE` denotes the size of the middle-level PMA in Terrace. The size of the middle-level PMA is bounded by $O(nL)$, where n is the number of vertices in the graph. The theoretical performance is measured in the external-memory model discussed in Chapter 1. The node size in the B-tree is $\Theta(B)$ where B is the memory-block size from the external-memory model.

Theoretical analysis

Table 3.3 shows the asymptotic runtime of operations in Ligra, Aspen, and Terrace in the external-memory model (Chapter 1). Given an edge (u, v) or vertex u, the operations in Table 11.1 are as follows:

- `add_edge`(u, v) adds an edge from vertex u to v.
- `find_edge`(u, v) returns whether the edge (u, v) exists in the graph.
- `get_neighbors`(u) returns all neighbors of vertex u.

The runtime of operations in Terrace depends on the degree of the vertex in question. For an in-place vertex, adding, querying, or listing all neighbors incurs only $O(S/B)$ cache misses.

For a medium-degree vertex, adding an edge requires inserting a new item in the PMA or moving an item from the in-place neighbors and adding the new item in the in-place list. Therefore, the number of cache misses is dominated by the insert operation in the PMA, which in turn depends on the overall size of the PMA. Querying a vertex requires a binary search on that vertex's neighbors, which only depends on the degree of the vertex. Listing all neighbors of a vertex requires a sequential scan through that vertex's neighbors in the PMA, which again only depends on the degree of the vertex. For a high-degree vertex, adding, querying, or listing is dominated by inserting/searching through the B-tree consisting of all the neighbors of the vertex and hence depends only on the degree of the vertex.

Ligra uses CSR as its underlying representation, which is a static graph format designed for queries but not updates. Therefore, adding an edge in Ligra depends on the total number of vertices and edges in the graph. Querying or listing in Ligra only depends on the degree of the vertex.

Aspen is a dynamic representation based on probabilistic balanced C-trees that supports fast concurrent updates and queries. Specifically, it stores a tree per vertex to hold its neighbors as well as a tree of pointers to each of the per-vertex trees. Since Aspen stores the vertex array as a tree, it requires at least $O(\log |V|)$ work per operation [3]. Its insertion cost may improve upon Terrace for medium- and high-degree nodes depending on the expected size of C-tree nodes.

Data structure microbenchmarks

Although the PMA and B-tree have the same (optimal) asymptotic scan cost in the external-memory model, they exhibit significant differences in scan performance in practice due to differences in their structure. The PMA stores all data contiguously for efficient sequential scans, while the B-tree stores its data in cache-line sized blocks connected by pointers for asymptotically faster searches and updates. The external-memory model does not capture the relative performance benefit of accessing sequential cache lines (in the PMA) compared to pointer chasing (in the B-tree) [42].

To illustrate tradeoffs between the PMA and B-tree and guide when to prefer each data structure, Figure 3-5 reports the results of a micro-benchmark that tests insertion, get (point query), and sum (aggregating all values) time in both a PMA and B-tree and report the results in Figure 3-5. The PMA supports scanning over all elements (in the sum benchmark) $2-5\times$ than the B-tree, but is $1.5-5.2\times$ slower for inserts and gets.

Terrace's hierarchical design exploits this tradeoff between the PMA and B-tree. When a vertex's degree is relatively small, Terrace stores its neighbors in a PMA for faster scans. On the other hand, when the number of neighbors is large, Terrace uses the B-tree to balance insertion and scan cost.

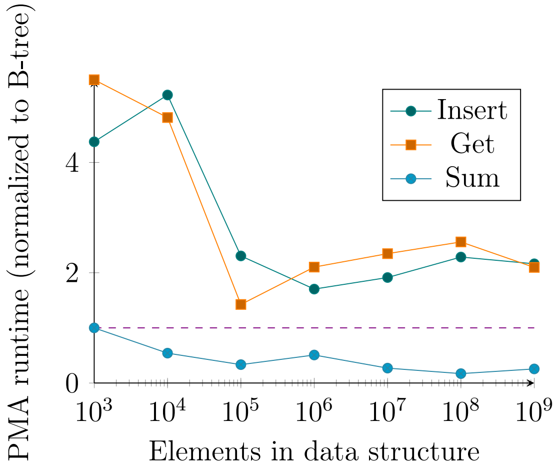


Figure 3-5: Normalized running time of insert, get, and sum in a PMA (normalized to a B-tree).

3.5 Implementation of Terrace

This section explains how to optimize for cache locality to achieve both fast update and queries in Terrace. Specifically, it will describe how to tune the degree cutoff pa-

[3]Aspen may perform an additional optimization called a flat snapshot [128] to flatten the node tree into an array, but this analysis omits it because it relies on amortization of the cost across multiple queries.

rameters between levels in Terrace for cache locality. Next, it will explain how Terrace supports batch updates and multi-threading for fast updates and queries. Finally, it will give a brief description of the VertexSubset/EdgeMap API in Ligra [335], which Terrace uses to implement graph kernels.

Optimizing Terrace for cache locality. For unweighted graphs, vertex blocks in the first level are sized to fit in a single cache line so that accessing in-place neighbors only requires a single cache miss. For concreteness, let us first consider 32-bit (4-byte) neighbors. To support 64-bit (8-byte) neighbors, Terrace would size each vertex block to two cache lines. Since the metadata in each vertex block takes 12 bytes (4 bytes for the degree and 8 bytes for the B-tree pointer), a cache line of B bytes can hold up to $(B - 12)/4$ in-place neighbors. Since a cache line is typically 64 bytes on most x86 machines, Terrace sets the maximum number of in-place neighbors $S = (64 - 12)/4 = 13$.

When the graph is weighted, Terrace uses two consecutive cache lines per vertex block to pack metadata, neighbors, and weights. For concreteness, let us consider 32-bit (4-byte) weights. Similarly to the unweighted case, Terrace would double the vertex block size for 64-bit (8-byte) weights. After accounting for metadata, there is space for 14 neighbors with weights in two cache lines, so Terrace sets $S = 14$ in the weighted case.

Finally, Terrace restricts the maximum number of neighbors for a vertex that can be stored in the second level (PMA) to $S + L = 1024$ throughout the evaluation so that all of the neighbors of a single vertex can fit in a small number of consecutive 4 KB pages. S and L are configurable parameters and the performance of Terrace is not sensitive to slight changes to these parameters. Section 3.6 presents a detailed evaluation to understand Terrace's performance sensitivity to these parameters.

Batch updates. Given the hierarchical design in Terrace, we perform batch updates in phases. The first phase of a batch update sorts all the edges in the batch based on the destination vertex and then based on the source vertex. For each vertex, the second phase merges in-place neighbors and the new incoming neighbors in a new sorted list of neighbors. Finally, the last phase stores the first S neighbors from the merged list in place and insert the rest either in the PMA or the B-tree depending on the degree of the vertex. If the degree of a vertex becomes greater than $S + L$ during a batch insertion, the algorithm removes that vertex's neighbors from the PMA and inserts them in a B-tree along with the new incoming neighbors.

Deletes are implemented symmetrically to insertions. Given a sorted batch of edges to delete, the batch delete algorithm first removes all of those edges that were stored in-place and then deletes the rest either in the PMA or B-tree.

After deletion, if a vertex degree drops from the B-tree to the PMA level, we delete the B-tree and put all of its edges into the PMA level. To fill the new empty spaces in the vertex block, we move the smallest edges from the corresponding vertex's PMA or B-tree to the vertex block.

Multi-threading. Terrace supports updating multiple vertices at once, but only a single thread may update a given vertex at a time.

Since the vertex blocks and B-trees in Terrace are not shared between vertices, multiple threads can concurrently update individual vertices in those levels without contention. Terrace uses lightweight spin locks to synchronize threads trying to update neighbors in the same vertex.

Since the PMA in the second level of Terrace is shared between vertices, multi-threaded updates in the PMA require additional locks. Terrace uses the locking-based thread-safe PMA from Chapter 4 to implement the second level.

VertexSubset and EdgeMap API. Terrace implements the VertexSubset/EdgeMap interface proposed by Ligra [335] to define graph kernels. The ***VertexSubset*** data structure represents a set of active vertices, and the ***EdgeMap*** primitive applies a function to edges incident to a set of vertices.

More formally, an EdgeMap takes as input a graph $G = (V, E, w)$, a VertexSubset U, and two boolean functions F and C. A call to EdgeMap applies function F to a set of edges E' such that an edge (u, v) is in E' if and only if $u \in U$ and $C(v) = true$. It returns a VertexSubset U' such that vertex $u \in U'$ if and only if $(u, v) \in E'$ and $F(u, v) = true$.

The VertexSubset in Terrace has one optimization which can help with some algorithms. Specifically, it has a boolean flag which specifies if the subset includes all of the vertices. If the flag is set, then membership queries into the VertexSubset simply return true instead of performing a lookup. This optimization helps with algorithms that process all of the vertices at each step.

3.6 Evaluation

This section empirically evaluates Terrace and demonstrates that it overcomes traditional tradeoffs to support both fast updates and queries. It compares Terrace to Aspen [128], a state-of-the-art graph-streaming system. It also includes Ligra [335], a static graph-processing system, as a baseline for running graph algorithms. Ligra is static and supports faster graph algorithms compared to dynamic systems. It compares all systems in terms of running time for different graph algorithms and memory footprint, and Terrace and Aspen on edge update (insert/delete) throughput. Finally, it tests different Terrace configurations to investigate the performance effects of the level cutoff parameters and the three-level structure.

Experimental setup. We implemented Terrace as a `c++` library parallelized using `Cilk` [191] and the Tapir/LLVM [322] branch of the LLVM [234,235] compiler (version 9). We compiled Aspen and Ligra with `g++` version 7.5 as recommended by the respective authors. All experiments were run on a 48-core 2-way hyper-threaded Intel® Xeon® Platinum 8275CL CPU @ 3.00GHz with 189 GB of memory from AWS [10]. However, to perform a fair evaluation and avoid non-uniform memory access (NUMA) issues across sockets we ran all experiments on a single socket with 24 physical cores and 48 hyper-threads.

Graph kernels. Table 3.4 details the algorithms we implemented in Terrace: breadth-first search (BFS), PageRank (PR), connected components (CC), single-source be-

Graph kernel	*Input*	*Output*	*Notes*
Breadth-first search (BFS)	Source vertex	$\|V\|$-sized array of parent IDs	
PageRank (PR)		$\|V\|$-sized array of ranks	No early exit
Connected components (CC)		$\|V\|$-sized array of component labels	No shortcut
Triangle counting (TC)		Number of triangles	
Betweenness centrality (BC)	Source vertex	$\|V\|$-sized array of centrality scores	Single source
Single-Source shortest paths (SSSP)	Source vertex	$\|V\|$-sized array of distances	Bellman-Ford

Table 3.4: A list of graph kernels and inputs and outputs used to evaluate graph representation systems.

tweenness centrality (BC), triangle counting (TC), and single-source shortest paths (SSSP). The algorithms are almost exactly the same as in Ligra [335] with minor cosmetic changes. The CC implementation does not have a shortcut, and the PR implementation runs for a fixed number (10) of iterations (i.e. it does not early-exit). Finally, the SSSP algorithm implements Bellman-Ford [108, Chapter 24].

Datasets. Table 3.5 lists the graphs used in the evaluation and their sizes. We tested on real social network graphs, a graph from computational biology, and a synthetic graph. Social network graphs usually have a few very high-degree vertices while the rest of the vertices have low degree according to a power-law distribution [27]. We used the ***LiveJournal*** (LJ) and ***Orkut*** social network graphs from the SNAP dataset [241]. LiveJournal is a directed graph of the LiveJournal social network [69], and Orkut is an undirected graph of the Orkut social network. Additionally, we used the ***Twitter*** social network graph, which is a directed graph of the Twitter network of follower relationships [35].

We also use the ***Protein*** network graph [25]. The protein network graph is an induced subgraph and is available in the data repository of the HipMCL algorithm 4 [25]. It contains 1/8-th of the original vertices of the sequence similarity network that contained all the isolate genomes from the IMG database at the time. Unlike social network graphs, the protein network graph is not heavily skewed and most (98.8%) vertices have degree less than 1000. We also generated an arbitrary graph by sampling edges from an rMAT generator [93] with $a = 0.5; b = c = 0.1; d = 0.3$ to match the distribution from Aspen [128] (we will refer to this graph as the ***rMAT*** graph).

To evaluate SSSP, we generated weighted graphs from unweighted graphs by assigning random integer weights in the range $[0, 256)$.

We used symmetrized versions of all of the graphs for a fair comparison with the publicly available version of Aspen, which supports only unweighted undirected graphs.

Since LiveJournal, Orkut, and Twitter are static graphs which may have been preprocessed with vertex reordering [376], we randomly relabeled the vertices in all of the input graphs to model the dynamic streaming graph setting. Reordering is more difficult in streaming graphs because a good ordering may change with the stream of edges [18].

System descriptions. Terrace and Aspen differ significantly in their underlying data structures and parallelization approaches. Aspen takes a purely functional approach with compressed trees, while Terrace modifies a single hierarchical data structure

Dataset	*Vertices*	*Edges*	*Avg. Degree*
LiveJournal	$4,847,571$	$85,702,474$	17.8
Orkut	$3,072,627$	$234,370,166$	76.2
rMAT	$8,388,608$	$563,816,288$	60.4
Protein	$8,745,543$	$1,309,240,502$	149.7
Twitter	$61,578,415$	$2,405,026,092$	39.1

Table 3.5: A list of (symmetrized) graph datasets, number of vertices, number of edges, and average degree of those graphs used to evaluate graph representation systems.

with locks directly. Aspen allows read-only operations (e.g. queries) during writing transactions, and vice versa (i.e. it does not use locks). It requires that the writer is sequentialized, however. In contrast, Terrace uses locks and allows for concurrent reading and writing in different regions of the data structure.

In this evaluation, we performed updates and queries in a phased manner, so queries did not need to acquire locks. The space overhead of locking still remains and impacts the cache-behavior during graph computations, however. This behavior is the same in Aspen as it uses functional trees and there is no overhead of locking if there are no updates. Section 3.7 further discusses mixing concurrent updates and queries in graph streaming systems.

Ligra is a static graph processing system that uses CSR as its underlying graph representation.

Update throughput

Setup. To evaluate insertion and deletion throughput, we first insert edges from an existing graph in Terrace. We then add a new batch of directed edges (with potential duplicates) to the existing graph and delete the same batch of edges from the graph. The batch insertion and deletion are performed using multiple threads. The graph layout remains the same at the start of every batch of insertions and deletions because the set of edges during insertion and deletion is the same. We perform the update evaluation on the LJ and Orkut graphs. To generate edges for updates, we sample directed edges from the same rMAT generator that we used to generate the synthetic rMAT graph. We report the average of 10 trials.

Results. We show that Terrace achieves throughput up to 48 million edges per second for batch insertions and up to 9 million edges per second for batch deletions. We report our findings in Table 3.6. On LJ, Terrace outperforms Aspen on batches of up to $1,000,000$ edges, while Aspen is faster on a batch size of $10,000,000$. On Orkut, Terrace is faster on batch sizes up to $100,000$, while Aspen is faster on batch sizes of at least $1,000,000$. For edge deletion, Aspen outperforms Terrace for batch sizes greater than 1000 on LJ and 100 on Orkut.

Discussion. Terrace is up to $3\times$ faster than Aspen on batch sizes up until $1,000,000$ on LJ and up to $1.75\times$ faster than Aspen on batch sizes up until $100,000$, but does not scale with larger batch sizes as Aspen does. Aspen scales with large batches because it implements insertions as a per-vertex tree merge. As the batch size increases,

	Insert						*Delete*					
	LJ			*Orkut*			*LJ*			*Orkut*		
Batch Size	*Terrace*	*Aspen*	*T/A*	*Terrace*	*Aspen*	*T/A*	*Terrace*	*Aspen*	*T/A*	*Terrace*	*Aspen*	*T/A*
1E1	3.93E5	1.25E5	3.14	2.11E5	7.28E4	1.75	1.42E6	1.31E5	10.86	7.49E5	1.28E5	5.86
1E2	1.11E6	7.11E5	1.56	8.12E5	4.32E5	1.11	2.41E6	7.62E5	3.16	1.37E6	7.55E5	1.82
1E3	5.48E6	2.77E6	1.98	3.25E6	1.97E6	1.23	4.72E6	2.98E6	1.59	1.97E6	2.83E6	0.69
1E4	1.96E7	6.56E6	2.99	1.06E7	4.93E6	1.70	5.55E6	7.38E6	0.75	2.52E6	7.05E6	0.36
1E5	4.83E7	1.57E7	3.09	2.35E7	1.26E7	1.70	8.68E6	1.61E7	0.54	3.62E6	1.46E7	0.25
1E6	4.40E7	3.46E7	1.27	1.71E7	2.69E7	0.52	9.23E6	3.43E7	0.27	4.36E6	3.32E7	0.13
1E7	2.82E7	1.03E8	0.27	2.59E7	7.76E7	0.25	6.61E6	1.05E8	0.06	4.62E6	1.05E8	0.04

Table 3.6: Throughput for inserting and deleting edges with varying batch sizes in the LJ and Orkut graphs in Terrace and Aspen. T/A denotes the ratio of the respective throughputs (Terrace/Aspen).

the number of edges per vertex increases and the overhead of the merge operation is amortized over a larger number of edges. In contrast, Terrace implements batch updates in phases at each level of the structure and performs updates at the granularity of each vertex. For larger batches, the vertex with the most updates dominates the running time.

Most highly dynamic graphs do not require the throughput that Aspen achieves on huge batches, however. For example, Twitter averages $9,346$ tweets per second [193] and peaked at $140,000$ tweets per second [313]. At its peak, Facebook is estimated to process about 13 million transactions per second [92]. Snapchat, another social network, saw around 210 million snaps per day in 2019 (about 2,500 per second) [347]. Applications in cybersecurity process about 10–15 million edges per second [56].

Terrace is not yet optimized for batch deletions which makes Terrace slower for deletions than Aspen for most batch sizes. Batch deletions are not as straightforward as insertions and require a careful engineering effort. Supporting batch deletions is not an inherent limitation of Terrace's design, however.

Query performance

We evaluate the performance of Terrace, Aspen and Ligra on BFS, PR, (single-source) BC, and CC, and report the results in Table 3.7. We plot the normalized time to Ligra using data from Table 3.7 of the various kernels in Figures 3-6, 3-7, 3-8, and 3-9. Since the publicly available version of Aspen is unweighted, we compare Terrace and Ligra on SSSP in Table 3.8. Finally, we compare Terrace and Ligra on TC, since the intersection primitive that the TC algorithm is based on is not yet optimized in Aspen and performs poorly. Table 3.9 presents the performance of Terrace and Ligra on TC. For each graph kernel, we took the average of 10 trials.

Traversals in graph kernels can be divided into two main categories. Vertices may be accessed in an arbitrary order as in PR, or in an order defined by the graph topology as in BFS. CC follows a similar traversal to PR, and BC follows a similar traversal to BFS. In arbitrary order, systems with more locality such as Ligra and Terrace can iterate over the edges with fewer cache misses than systems that store edges out of place. In topology-defined order, all of the systems are likely to incur a cache miss when accessing the neighbors of an arbitrary vertex.

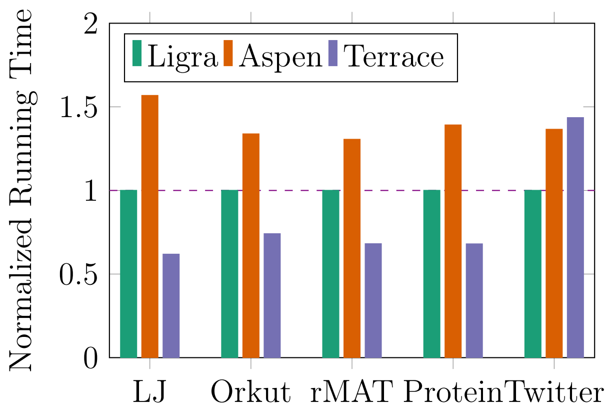


Figure 3-6: Time to run BFS normalized to Ligra.

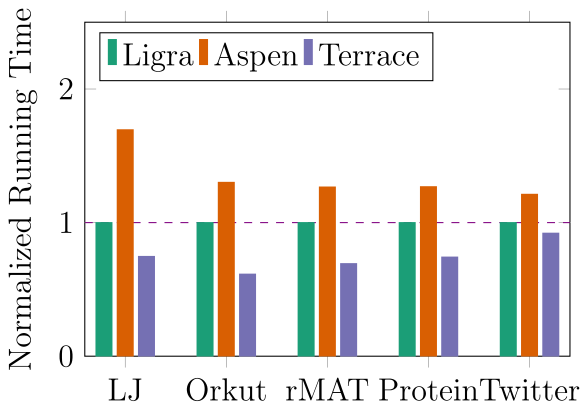


Figure 3-7: Time to run PR normalized to Ligra.

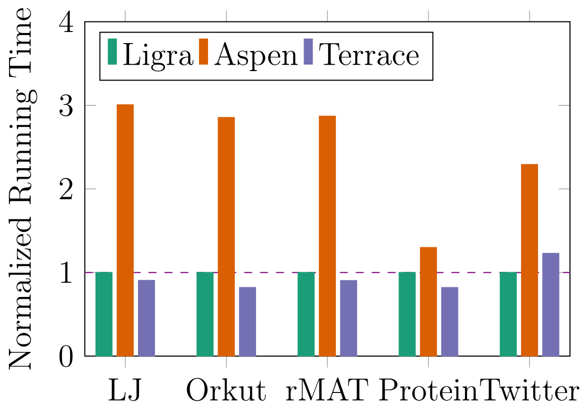


Figure 3-8: Time to run BC normalized to Ligra.

Breadth-first search. Figure 3-6 illustrates the relative speed on BFS of all the systems. On average, Terrace outperforms Ligra and Aspen by 1.2× and 1.6×, respectively. Terrace performs better since it saves cache misses with its in-place level. All of the graphs tested exhibit skewness, so most of their vertices can be stored in place. Terrace performs worse on Twitter than the other graphs because Twitter has much higher maximum degree, so many edges are stored in the relatively unopti-

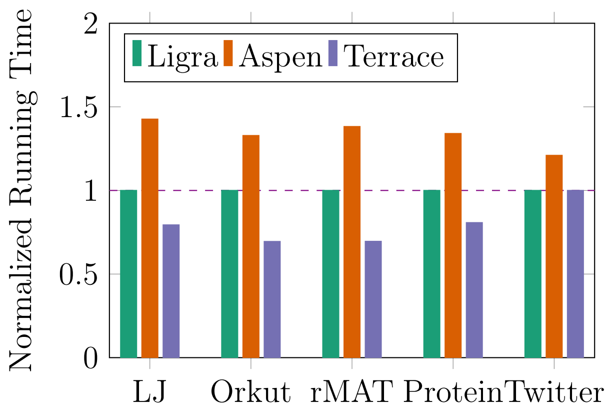


Figure 3-9: Time to run CC normalized to Ligra.

	BFS						*PR*					
	Terrace		*Ligra*		*Aspen*		*Terrace*		*Ligra*		*Aspen*	
Graph	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}
LJ	0.44	0.02	0.85	0.03	1.20	0.05	8.35	0.31	11.90	0.42	21.41	0.71
Orkut	0.44	0.02	0.71	0.03	0.97	0.04	23.65	0.42	26.08	0.80	41.55	1.05
rMAT	0.68	0.04	1.53	0.05	1.91	0.07	63.18	2.16	100.98	3.12	153.18	3.95
Protein	0.57	0.03	0.61	0.04	1.16	0.05	137.28	4.97	278.00	6.70	242.30	8.50
Twitter	X	0.33	X	0.23	X	0.32	X	18.26	X	19.83	X	24.03
	BC						*CC*					
	Terrace		*Ligra*		*Aspen*		*Terrace*		*Ligra*		*Aspen*	
Graph	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}
LJ	2.18	0.09	2.42	0.10	6.38	0.29	2.31	0.09	2.33	0.11	3.63	0.15
Orkut	2.86	0.10	3.29	0.12	5.61	0.34	3.34	0.12	4.16	0.17	6.08	0.22
rMAT	7.74	0.28	7.98	0.31	17.60	0.88	16.34	0.41	15.25	0.59	21.87	0.82
Protein	1.5	0.09	1.43	0.12	2.31	0.15	42.16	1.27	45.17	1.58	62.64	2.11
Twitter	X	2.53	X	2.06	X	4.72	X	4.32	X	4.32	X	5.23

Table 3.7: Running times (in seconds) of Terrace, Ligra, and Aspen on BFS, PR, BC, and CC. T_1 denotes the time on one thread, and T_{96} denotes the time on all (96) threads. Single thread numbers for Twitter graph are omitted due to time constraints.

mized B-tree level of Terrace. Future optimizations include replacing the B-tree with an optimized balanced tree representation, such as Aspen's C-trees [128].

PageRank. Figure 3-7 illustrates the relative speed on PR of all the systems and shows that Terrace achieves between 1.2×–2× speedup over Aspen and outperforms Ligra by 1.3× on average. Terrace shows better performance on PR because it supports faster ordered access of in-place neighbors and neighbors stored in the second level PMA. For most input graphs, a considerable fraction of all edges reside in the in-place and PMA level (see Table 3.11). Moreover, the VertexSubset optimization described in Section 3.5 also helps to improve the PR algorithm running time in Terrace.

Betweenness centrality. Figure 3-8 illustrates the relative speed on BC of all the systems. Terrace achieves similar ($.8\times - 1.1\times$) performance compared to Ligra and outperforms Aspen by 1.6×–3×. BC is similar to BFS in that it follows a topology-defined order and is computationally- and memory-intensive. Therefore, Aspen and Ligra diverge further than in BFS because Aspen incurs relatively more cache misses.

Connected components. Figure 3-9 illustrates the relative speed on CC of all the systems. On average, Terrace achieves 1.2× speedup over Ligra and 1.7× speedup over Aspen. CC starts with all vertices in the frontier, so more in-place neighbors are accessed during larger frontiers in Terrace which helps to avoid unnecessary cache misses.

	Terrace		*Ligra*		*T/L*	
Graph	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}
LJ	9.10	0.39	7.42	0.22	1.22	1.77
Orkut	13.60	0.53	8.00	0.28	1.70	1.89
rMAT	45.95	1.61	35.85	1.01	1.28	1.59

Table 3.8: Running times (in seconds) of Terrace and Ligra on SSSP. T_1 denotes the time on one thread, and T_{96} denotes the time on all (96) threads. T/L denotes the ratio of the respective throughputs (Terrace/Ligra).

	Terrace		*Ligra*		*T/L*	
Graph	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}
LJ	34.06	1.30	21.18	0.60	1.60	2.17
Orkut	191.79	6.52	111.00	3.08	1.72	2.12
rMAT	54.30	1.54	257.80	5.04	0.21	0.31

Table 3.9: Running times (in seconds) of Terrace and Ligra on TC. T_1 denotes the time on one thread, and T_{96} denotes the time on all (96) threads. T/L denotes the ratio of the respective throughputs (Terrace/Ligra).

Single-source shortest paths. Table 3.8 shows that Terrace is between 1.6×–1.9× slower than Ligra on SSSP. The graph traversal in SSSP is similar to that of BFS, so Terrace can take advantage of in-place neighbors. However, Terrace incurs extra overhead for storing weights compared to Ligra because it must store additional empty spaces in the PMA to store the weights array. In the weighted case, accessing neighbors in the PMA level is more expensive than in CSR.

Triangle counting. Table 3.9 shows that Terrace is up to 2.2× slower than Ligra on TC. TC is a computationally intensive kernel that repeatedly loops over vertices and edges, so the smaller representation in Ligra has better locality. Terrace performs well on TC on rMAT because rMAT is more skewed than the other graphs, so almost all vertices can be stored in place. However, for other graphs whenever neighbors are spread across the PMA or the B-tree, looping over neighbors to compute intersections is inefficient and incurs multiple cache misses.

Graph	*Terrace*	*Ligra*	*Aspen*	*T/A*
LJ	1.43	.34	1.18	1.2
Orkut	2.41	.91	1.77	1.3
rMAT	8.73	2.13	4.32	2.02
Protein	19.05	5.27	9.08	2.09
Twitter	43.78	9.87	20.85	2.09

Table 3.10: Memory footprint (in GB) of relabeled and original graphs on the different systems. T/A denotes the ratio of the respective memory footprints (Terrace/Aspen).

Graph	*% In-place*	*% PMA*	*% B-tree*
LJ	20.12	77.20	2.66
Orkut	7.44	84.06	8.49
rMAT	5.72	93.72	0.54
Protein	2.67	83.00	14.32
Twitter	8.38	39.68	51.92

Table 3.11: Percentage space distribution of three layers in Terrace for different graphs.

Memory usage

Table 3.10 reports the memory footprint of the different systems. The space usage of Terrace is up to 2.1× higher than Aspen because Aspen uses data compression techniques, while Terrace uses uncompressed data structures with extra space overhead. Adding data compression to Terrace would decrease space usage and add a small amount of computational overhead.

We present the distribution of the memory in the three levels of Terrace in Table 3.11. For every graph in our evaluation besides Twitter, most of the edges (between $77\% - 94\%$) are stored in the PMA level. The PMA data structure maintains extra space to support fast update operations.

At a high level, there is an inherent tradeoff between the amount of empty space and the speed of updates. We plan to investigate the potential tradeoff between space utilization and update speed in future work.

Terrace configurations

We perform two categories of Terrace micro-benchmarks: we evaluate the performance impact of the 1) cutoffs between levels (S and L), and 2) data structures in different levels of the hierarchy.

Setup. To test the level cutoffs, we vary the values of S (number of in-place neighbors) and L (maximum degree to stay in the PMA). Specifically, we set $S = 29$ (default is 13) to fit the vertex block in two cache lines instead of one. To test the medium-degree cutoff, we fix $S = 13$ and vary L between 2^8 to 2^{12} (default is 2^{10}).

To verify the effects of each level of Terrace, we omit one out of the three levels in Terrace and measure the performance. Specifically, we use three different configurations: Inplace+PMA, Inplace+Btree, and PMA+Btree.

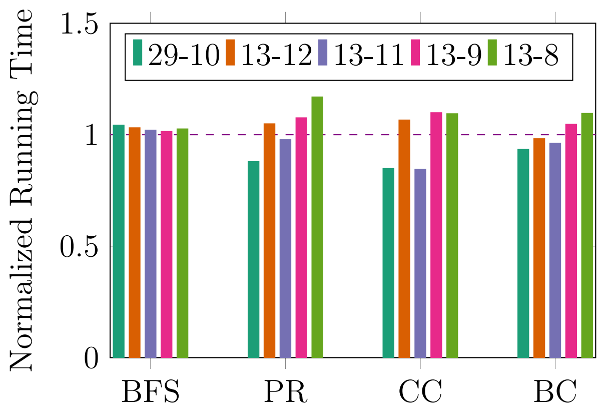


Figure 3-10: Normalized time of Terrace with different level cutoffs. The cutoffs are in the format $S - L$ where $S, 2^L$ are the in-place and PMA cutoffs, respectively. For example, the original Terrace configuration can be denoted $13 - 10$.

We evaluate the performance on four graph kernels BFS, PR, CC, and BC. We use three datasets (***LiveJournal***, ***Orkut***, and ***rMAT***) for both sets of experiments. We also use the ***Twitter*** graph when omitting Terrace levels to evaluate the impact of B-trees on the performance, since B-trees contain a significant fraction of edges in Twitter (see Table 3.11). We report the results by averaging the running times over all datasets and normalized the running time of the modified Terrace with the default configuration.

Discussion. Figure 3-10 illustrates the effect of varying the level cutoff parameters S and L. Terrace is not sensitive to changes in the configuration: the variance in the performance for different graph kernels varies between 1% – 16%. The highest variance is seen in PR and CC, since these both require traversals in an arbitrary order which slightly increases the sensitivity to the change in configuration compared to BFS and BC.

Figure 3-11 presents the results of omitting levels in Terrace. Using only the in-place and PMA levels improves the performance by 15% – 20% for PR and CC because the PMA allows fast sequential access. However, removing the B-tree (and only keeping the in-place and PMA levels) reduces the update throughput by 40% which aligns with the update-query tradeoff described in Section 3.4. Using only the in-place and B-tree reduces the performance by 14% – 88% as the B-tree has poor cache locality compared to the PMA. Therefore, the three-level Terrace design strikes a balance between updatability and graph kernel performance.

3.7 Related work

This chapter focuses on dynamic graphs in the streaming setting [57], but there has been significant research effort devoted to processing graphs in the static setting [112, 163, 252, 256, 280, 293, 308, 335, 375]. For a more detailed survey on static frameworks, see [265, 391].

Many streaming graph systems apply updates in batches [128, 139, 229, 254] to

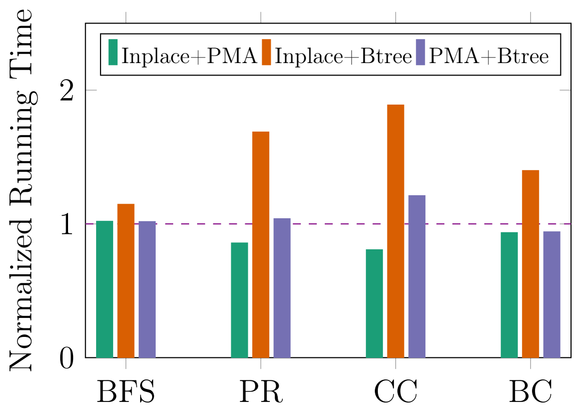


Figure 3-11: Normalized time of Terrace with different hierarchical configurations.

amortize the work of writing to the graph. Batching updates improves update throughput but may delay the time an update appears in the graph because an update may have to wait for a batch to become sufficiently large.

There are two main approaches to applying updates in streaming graph systems. The first and the more popular approach, which this chapter adopts, phases updates and queries separately [11, 83, 85, 139, 144, 168, 277, 327–329, 358, 368, 385]. Separating updates and queries can improve the performance of queries because it removes the need to synchronize writing and reading to the graph data structure. Phasing may delay queries, however, because they must wait until an update phase is finished. The second approach uses snapshotting [97, 197, 198, 227, 254] to enable concurrent updates and queries. Snapshots may even improve query performance by converting the graph storage format into one more amenable to queries [128]. More frequent snapshots are required for a more updated view of the graph, but taking snapshots requires extra processing. Common traversal-based graph operations on dynamic graphs prefer the most up-to-date state of the graph [253]. For more details, refer to a survey on streaming graph systems [57].

Although both update approaches theoretically support incremental graph workloads, many recent works on dynamic graph algorithms model the first approach of applying updates in atomic batches. Specifically, the ***batch-parallel*** model has emerged as the primary theoretical model for design and analysis of incremental graph algorithms [2, 57, 130, 131, 146, 285, 363].

Finally, previous work has also focused on graph databases [79, 137, 218, 226, 306, 331] that support transactions while processing a streaming graph. Unfortunately, support for transactions in graph databases induces significant overhead when compared to state-of-the-art graph-streaming systems such as Stinger [264]. Therefore, the focus of this chapter is on data structure design, which is independent of support for transactions.

3.8 Conclusion

This chapter improves the performance of dynamic graph processing via hierarchical data structure design by taking advantage of the inherent skewness in the degree distribution of real-world graphs. Terrace dynamically adapts to the skewness in the underlying graph. It stores a vertex's incident edges in different data structures based on its degree and support cache-efficient updates and traversals.

We believe Terrace strikes an appropriate balance between batch update speed and graph algorithm performance. It is faster than or competitive with Aspen, a state-of-the-art streaming graph processing system, on batch updates of practical batch sizes. At the same time, Terrace is 2× faster than Aspen on average, and is competitive with or outperforms Ligra, a fast static graph-processing system, on most tested graph kernels.

Future work. The hierarchical design approach offers promise for building high-performance streaming graph representations. In future work, it would be interesting to combine it with incremental graph algorithms that optimize for dynamic graphs [56, 131, 169, 255, 263, 341] to build highly-optimized streaming graph systems.

Future work includes reducing the memory footprint of Terrace using a compressed B-tree implementation and lowering the upper density bound in the PMA to reduce the space overhead to perform a comparison with Aspen with similar memory overheads. By design, the PMA uses a constant fraction of extra slots to support fast inserts. The PMA implementation in Terrace uses twice the space of a packed array, but could easily be changed to use a smaller constant to reduce the space usage at the cost of slightly more expensive insertions. Terrace explores the space-time tradeoff in dynamic graph storage: varying its memory usage would illuminate additional points along the tradeoff.

Locality-first strategy. Terrace applies the locality-first strategy via cache-friendly data structure design for dynamic graph processing to improve performance by enhancing spatial locality. The first step in the locality-first strategy is to understand locality in the problem. Graph processing does not have much temporal locality in any given scan, but has opportunities to exploit spatial locality. Furthermore, the parallelism-first design with separate data structures per neighbor list offers additional opportunities for spatial locality by co-locating as many neighbor lists as possible. Additionally, naturally-occurring graph skewness presents opportunities for improving spatial locality by co-locating low-degree vertices without sacrificing updatability. The next step in the locality-first strategy is to exploit the identified locality: Terrace optimizes for spatial locality with cache-friendly data structures that take advantage of graph skewness. Terrace achieves the best of both worlds in updatability and query speed by leveraging the locality-first strategy.

Chapter 4

A Parallel Packed Memory Array to Store Dynamic Graphs

This chapter presents Parallel Packed Compressed Sparse Row (PPCSR), a dynamic-graph-processing framework that uses the locality-first strategy to enhance spatial locality with the Packed Memory Array data structure [44, 196]. PPCSR supports graph queries about 1.6× faster than Aspen [128], a state-of-the-art high-performance dynamic-graph-processing system, while maintaining competitive update throughput. PPCSR is built on the Packed Memory Array data structure, an array-based data structure that stores all of its data contiguously. As we shall see in this chapter, although tree-based data structures such as the one underlying Aspen asymptotically dominate PMAs, PMAs are fast in practice because they exploit spatial locality. Finally, the parallel PMA that underlies PPCSR is one of the main components in Terrace (Chapter 3) and contributes to Terrace's cache-friendliness.

This work was conducted in collaboration with Brian Wheatman [379]. Appendix A describes an earlier serial version of PPCSR, called Packed Compressed Sparse Row (PCSR) [378].

Abstract

The ideal data structure for storing dynamic graphs would support fast updates as well as fast range queries which underlie graph traversals such as breadth-first search. The Packed Memory Array (PMA) seems like a good candidate for this setting because it supports fast updates as well as cache-efficient range queries. Concurrently updating a PMA raises challenges, however, because an update may require rewriting the entire structure.

This chapter introduces a parallel PMA with intra- and inter-operation parallelism and deadlock-free polylogarithmic-span operations. It shows that the PMA is well-suited to concurrent updates despite occasionally requiring a rewrite of the entire structure because 1) most of the updates only write to a small part of the structure and 2) the worst case is highly parallel and cache-efficient.

To evaluate the parallel PMA, we implemented Parallel Packed Compressed Sparse Row (PPCSR), a dynamic-graph-processing framework based on the parallel PMA. We show that PPCSR is on average about 1.6x faster on graph kernels than Aspen,

a state-of-the-art graph-streaming system. PPCSR achieves up to 80 million updates per second and is $2-5$x faster than Aspen on most batch sizes. Finally, PPCSR is competitive with Ligra and Ligra+, two state-of-the-art static graph-processing frameworks.

4.1 Introduction

As discussed in Chapter 3, there has been significant research effort devoted to systems for storing and processing dynamic graphs [83, 128, 139, 144, 168, 229, 254] because many real-world graphs change in real-time. These systems must process a stream of updates (e.g. edge-weight update, or edge insertions and deletions) and a stream of queries quickly. This chapter focuses on parallel data structure design optimized specifically for fast cache-efficient range queries[1] while still maintaining fast updates.

A suitable data structure for dynamic graphs must support efficient vertex neighbor queries in order to gather a vertex's neighbors for the next phase of the algorithm. Many graph algorithms, such as breadth-first search and betweenness centrality, can be expressed by iteratively processing a set of active vertices and their neighbors [335]. Therefore, efficient data structures for graph processing should store neighbors as close as possible for locality during range queries.

There is a tradeoff between update and range query performance in data structure design. For example, a hash table can achieve $O(1)$ amortized update cost [108, Chapter 11], but a range query $r(u, v)$ must take $O(v-u)$ work. At the other extreme, a range query in a sorted array with n elements takes $O(\log n + k)$ work, where k is the number of elements in the range, but updating a sorted array takes $O(n)$ work.

In the static setting, Compressed Sparse Row (CSR) [361], a canonical storage format for sparse graphs, achieves optimal performance for range queries by storing edges in a contiguous sorted array. Unfortunately, CSR is a static storage format: adding an edge to CSR may require shifting the entire edge array. Inspired by the cache-friendliness of CSR, Appendix A introduces Packed Compressed Sparse Row (PCSR) [378], a dynamic graph storage format that replaces the edge array in CSR with a Packed Memory Array (PMA) [44,196] for (amortized) $O(\log^2 |E|)$ update cost and asymptotically optimal range queries.

There are a couple of factors that make PCSR a good candidate for processing dynamic graphs beyond its theoretical guarantees. First, the observed update cost of PCSR is much better in practice than its theoretical bound might suggest because the worst-case rewrites are cache-efficient [378]. Additionally, PCSR avoids pointer indirections in contrast with non-contiguous data structures such as search trees (e.g. B-trees), which require pointer chasing. Finally, PCSR supports efficient scans and has good cache locality because the elements are laid out contiguously in memory.

[1]A range query $r(u, v)$ in a data structure takes two indices u, v and returns all elements in the range $[u, v]$.

Parallelization strategies

This chapter proposes parallel modifications to augment the PMA with both intra- and inter-operation parallelism to improve the performance. ***Intra-operation parallelism*** exploits logically parallel work present in the operations themselves, while ***inter-operation parallelism*** enables multiple threads to update or query the data structure at the same time.

For example, a PMA could support inter-operation parallelism without intra-operation parallelism by running the operations at the same time, but doing the work of each sequentially [51, 114].

The PMA is well-suited to intra-operation parallelization because the expensive operations are highly parallel. In the worst case, an update in a PMA with n elements may require rewriting the entire structure, which takes $O(n)$ work. This work can be parallelized, however. As we will see, updating a PMA has $O(\log^2 n)$ span in the worst case.

Furthermore, we will use the ***shared-memory multiple-writer / multiple-reader model*** for inter-operation parallelism for generality.

There are several challenges in supporting concurrent updates in a PMA when compared to search trees. In parallel search trees with locking, updates or queries may only need to acquire a few locks at a time (e.g. in hand-over-hand locking) to do an update or a query. Furthermore, purely functional trees may not even require locking because they can take a snapshot without traversing the entire structure [128]. These tree-based locking or snapshotting schemes do not directly translate to a PMA. Furthermore, an update to a search tree requires updating only a few nodes and pointers, while an update to a PMA (in the worst case) may require table doubling and rewriting the entire structure [378], which would seem to put the PMA at a disadvantage in terms of the fraction of the structure that needs to be locked. Previous work confirms this intuition: a PMA with locking and multiple writers achieves much lower update throughput when compared to search-tree variants optimized for writes [114].

This chapter overcomes these challenges to concurrent updates and shows that a parallel PMA with locking can simultaneously achieve high update throughput and fast queries. Past work showed that PMAs have much slower updates than tree-based structures [114]. In contrast, this chapter shows that PMAs can achieve similar or even better update throughput than tree-based structures in many cases. The PMA achieves fast updates in practice because the worst case of rewriting the entire structure during an update not only happens extremely rarely, but is also fast than the worst-case bound suggests because the rewrite is cache-efficient.

Contributions

This chapter introduces ***Parallel Packed Compressed Sparse Row*** (PPCSR), a graph storage format based on a PMA with parallel modifications to support both inter- and intra-operation parallelism. Along the way, it shows how to parallelize a PMA with polylogarithmic span for each operation. Furthermore, it introduces a

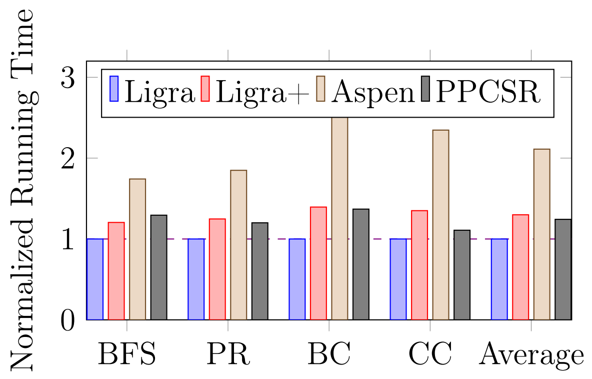


Figure 4-1: Time to run kernels normalized to Ligra averaged across all graphs. The four kernels tested were breadth-first search (BFS), PageRank (PR), betweenness centrality (BC), and connected components (CC).

deadlock-free locking scheme with polylogarithmic span[2].

We implemented PPCSR and found that it enables fast serializable phased updates and queries. That is, multiple writers can update concurrently, or multiple readers can read concurrently, but not both. To enable queries PPCSR extends the interface from Ligra [335], a static graph-processing framework. Therefore, all algorithms implemented with Ligra, such as graph-traversal algorithms, local graph algorithms [338], and others [126, 127] can be run on top of PPCSR with minor cosmetic changes.

We evaluate PPCSR and compare it to Aspen [128], Ligra [335], and Ligra+ [337], three state-of-the-art graph processing frameworks. Aspen is a graph-streaming framework, while Ligra and Ligra+ are static graph-processing frameworks. Although we expect the static graph-processing frameworks to outperform dynamic systems, we compare them on query cost to evaluate the cost of updatability. Therefore, we compare Aspen and PPCSR on update throughput, and all systems on graph kernel performance.

PPCSR supports efficient queries because it takes advantage of spatial locality. As shown in Figure 4-1, PPCSR outperforms Aspen by about 1.6x on average on the four tested graph queries. PPCSR is competitive with Ligra and Ligra+. On average, PPCSR is 1.25x slower than Ligra.

Furthermore, PPCSR achieves up to 80 million updates per second. As shown in Figure 4-2, PPCSR is $2-5$x faster than Aspen on small-batch updates but between $2-5$x slower on batch sizes of at least 10 million.

To be specific, our contributions are as follows:

- The design and theoretical analysis of a parallel PMA that supports intra- and inter-operation parallelism.
- An implementation of PPCSR on top of the parallel PMA using Cilk [191].
- An experimental study of PPCSR compared to Aspen, Ligra, and Ligra+ that demonstrates that PPCSR supports efficient updates and queries.

[2]Assuming grabbing a lock takes $O(1)$ work.

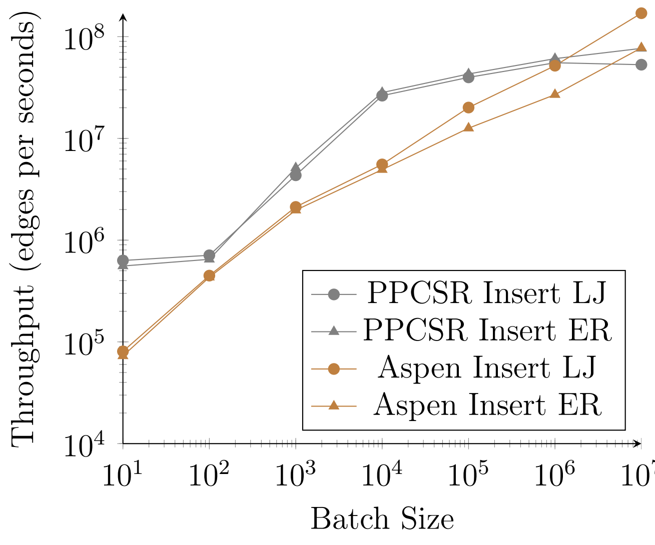


Figure 4-2: Insert throughput as a function of batch size on the LJ and ER graphs. The LJ graph is about 85 million edges, while the ER graph is about 1 billion edges.

Map. The rest of the chapter is organized as follows. Section 4.2 details the Packed Memory Array data structure that underlies PPCSR. Section 4.3 describes modifications to the PMA to make parallelization easier Sections 4.4 and 4.5 show how to exploit intra- and inter-operation parallelism in the PMA. Section 4.6 introduces PPCSR and describes how to augment the serial PCSR with locks to enable multiple writers. Section 4.7 describes how to implement graph operations with the PMA operations from Section 4.4. Section 4.8 presents the results from the experimental evaluation. Finally, Section 4.9 presents concluding remarks.

4.2 Packed Memory Array

This section reviews details of the Packed Memory Array (PMA) data structure that underlies PPCSR. Specifically, this section describes the operations that the PMA supports, its asymptotic guarantees, and how it maintains its structure.

A Packed Memory Array [44, 196] maintains elements in order in an array with (a constant fraction of) spaces between its elements. A PMA holds n elements in $N = O(n)$ cells and supports updates with amortized $O(\log^2 n)$ work. Point queries in a PMA take $O(\log n)$ work, and range queries $r(s, t)$ that return k elements have $O(\log n + k)$ work.

The PMA is composed of a contiguous implicit complete binary tree with leaves of size $\log N$. That is, the implicit tree has $N/\log N$ leaves and height $\log(N/\log N)$. Each leaf $i \in \{0, \ldots, N/\log N - 1\}$ encompasses cells in the ***region*** $[i \log N, (i + 1) \log N)$, and each internal node encompasses all of the cells of its descendants. The height of a node is the distance from that node to a leaf.

Each node of the PMA tree has an upper and lower bound on its ***density***, or the fraction of occupied cells in its region, that defines the number of empty cells allowed in that node. The upper and lower density bounds in each node are related to the

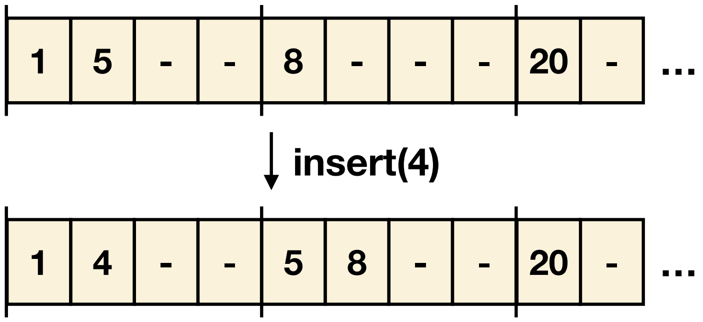


Figure 4-3: An example of inserting into a PMA with a leaf size of 4 and a leaf density bound of 0.5.

height of that node.

Operations. A PMA implements three ***external operations***:

- INSERT: inserts an element into the PMA.
- DELETE: deletes an element from the PMA.
- SEARCH: finds an element in the PMA.

Range queries in a PMA can be implemented by searching for the start of the range and doing a forward scan until the end of the range.

In order to implement the external operations, a PMA also supports the following ***internal operations*** as subroutines:

- COUNT_NON_NULLS: given a region, returns a list of counts of the number of elements in each PMA leaf in that region.
- REDISTRIBUTE: given a PMA node, spreads elements from that node evenly among the leaves in the subtree rooted at that node.
- DOUBLE_PMA: doubles the size of the PMA.
- HALVE_PMA: halves the size of the PMA.

The functions COUNT_NON_NULLS and REDISTRIBUTE take start and end indices s, t that must be at the beginning and end of PMA nodes, respectively (i.e. $s, t \mod \log N = 0$). Additionally, $(t - s)/\log N = 2^x$ for some non-negative integer x.

During insertions, a PMA enforces its density bounds by ***redistributing*** elements to neighbor nodes whenever a node violates its density bound so that the densities of both siblings are equal. Figure 4-3 illustrates an example of an insert and a redistribution in a PMA. Deletions are symmetric to insertions: if a deletion violates any node's density bound, it redistributes elements in that node.

In practice, PMAs support updates much faster than their update bound of $O(\log^2 n)$ might suggest because the update cost comes from the amortization cache-efficient redistributes [378]. These redistributes occur in contiguous memory and exhibit high spatial locality.

Scanning a PMA. The PMA supports cache-efficient scans, i.e., reading S sequential elements takes $O(1+S/B)$ cache-line transfers in the external-memory model [3]. In practice, a PMA supports fast scans because all of the data is stored contiguously in memory, so it has good spatial locality.

4.3 PMA modifications

This chapter details modifications to a PMA that aid in parallelization without impacting the PMA's theoretical guarantees. These modifications enable efficient parallelization of updates in PPCSR while maintaining fast scans.

Density bound To ensure that parallel threads can always insert without waiting or blocking, the parallel PMA enforces a stricter upper density bound on its leaves that ensures leaves are never completely full. Given an original upper density bound at the leaves d_{leaf}, the new upper density bound at the leaves of the PMA is $\min(d_{\text{leaf}}, (\log N - 1)/\log N)$. Since

$$\lim_{N\to\infty} (\log N - 1)/\log N = 1,$$

the additional density requirement does not impact the asymptotic behavior of the PMA. The extra bound ensures that a thread can always place an element immediately into the PMA and will only wait in the redistribute phase of an insert.

Packed-left property To parallelize locking, the parallel PMA enforces a ***packed-left*** property of the nodes in the PMA so that inserts into one region do not spill over into others. Instead of evenly distributing elements in the PMA leaves, the parallel PMA puts them all contiguously at the beginning of the leaf. The packed-left property along with the non-full density bound ensure that a thread will never shift elements into another node's region, which facilitates locking. Similarly, a delete would re-compress elements to the left at the beginning of each leaf.

Scanning over a PMA with the packed-left property asymptotically reduces the number of wasted accesses. When scanning over a standard PMA, each cell is checked to see if it is null or not. The packed-left property reduces the number of empty cells evaluated to $O(N/\log N)$ from $O(N)$ because a pass through each leaf evaluates at most one empty cell.

The packed-left property maintains the work bounds of the original PMA because the original PMA evenly distributes elements in a leaf after inserting into that leaf [44, 52], which requires reading and writing to each cell in that leaf. In the worst case, inserts into a PMA with the packed-left property also require reading and writing to each cell in the associated leaf. Furthermore, a PMA with the packed-left property maintains the cache-efficiency of the original PMA.

4.4 Intra-operation parallelism

This section provides theoretical grounding for the parallel PMA's practical performance by proving that PMA insertions have polylogarithmic span. Along the way, the section proves that the internal operations that insert uses also have polylogarithmic span. It begins by describing the the parallel primitives prefix sum and `memcpy`, which the PMA uses to implement its operations. It then proves that the internal operations well as the external operation SEARCH, which insert relies on, have polylogarithmic span.

Parallel primitives

First, let us review core parallel primitives necessary to implement the PMA operations.

Parallel prefix sum. The `prefix_sum`(A, N) operation takes as input a list A of N numbers and outputs a list A' where $\forall i \in \{0, 1, \ldots, N-1\}$,

$$A'[i] = \sum_{j=0}^{i} A[i].$$

Parallel implementations of prefix sum [61] can be done in place in $O(N)$ work and $O(\log N)$ span.

Parallel memcpy. The `memcpy(src, dest, size)` operation copies `size` bytes of data from location `src` to location `dest`. It can be implemented in parallel using a single parallel for loop in $O($`size`$)$ work and $O(\log($`size`$))$ span.

Internal operations

The COUNT_NON_NULLS(s, t) function returns the number of non-nulls in each leaf in a region in the PMA defined by start and end indices s, t.

Lemma 4.1 COUNT_NON_NULLS(s, t) *has work* $O(t - s)$ *and span* $O(\log(t - s))$.

PROOF. We can count the number of non-empty cells in each leaf in parallel using the parallel prefix operation. There are $t - s$ cells in the range, for work $O(t - s)$ and span $O(\log(t-s)) = O(\log(t-s))$. □

```
def redistribute(s, t):
  counts = count_non_nulls(s, t)
  temp[t - s] # create array
  parallel_prefix_sum(counts)
  # copy and pack all edges to temp
  parallel_for k in [s, t); k += log(N):
    if i == s: start = 0
    else: start = counts[i-1]
    for j in [k*log(N), (k+1)*log(N)):
      if pma[j] is not null:
        temp[start] = pma[j], start++
      pma[j] = null

  num_leaves = (t - s) / log(N)
  end_idx = counts.size - 1
  leaf_avg = counts[end_idx] / num_leaves
  extra = counts[end_idx] % count_per_leaf

  parallel_for i in [0, num_leaves):
    # number of items for this leaf
    for_leaf = leaf_avg + (i < extra)
    # start of leaf in temp and in PMA
    tmp_start = leaf_avg*i + min(i, extra)
    leaf_start = s + (i * log(N))

    # copy edges into PMA
    memcpy(&pma[leaf_start],
           &temp[tmp_start], for_leaf)
```

Figure 4-4: Pseudocode for REDISTRIBUTE(s, t).

The REDISTRIBUTE function enforces the density bound of a region in the PMA. Specifically, the REDISTRIBUTE(s, t) function guarantees that all nodes in the region defined by s, t respect their density bounds.

Theorem 4.2 REDISTRIBUTE(s, t) *has* $O(t-s)$ *work and* $O(\log(t-s))$ *span.*

PROOF. The pseudocode[3] for REDISTRIBUTE(s, t) can be found in Figure 4-4.

By Lemma 4.1, the call to COUNT_NON_NULLS(s, t) has $O(t-s)$ work and $O(\log(t-s))$ span. The function then prefix sums all the leaves in the range in $O(t-s)$ work and $O(\log(t-s))$ span.

The first `parallel_for` has $O(t-s)$ work and $O(\log(t-s))$ span. The second `parallel_for` iterates over the number of leaves, which is $(t-s)/\log N$, so the span of the second `parallel_for` is

$$O(\log((t-s)/\log N)) = O(\log(t-s)).$$

[3]Unless otherwise specified, all divisions in pseudocode are integer division (rounded down).

Therefore, the work and span of this `parallel_for` are $O(t-s)$ and $O(\log(t-s))$, respectively.

The total work and span of REDISTRIBUTE(s,t) are therefore $O(t-s)$ and $O(\log(t-s))$, respectively. $\square$

Resizing the PMA. If the PMA becomes too dense or sparse, it may have to be resized with the DOUBLE_PMA and HALVE_PMA functions. Given a PMA of N cells, both subroutines take $O(N)$ work and $O(\log N)$ span. At a high level, the functions densify the data, resize the PMA, and redistribute the data into the new size.

Lemma 4.3 *The* DOUBLE_PMA *procedure has work $O(N)$ and span $O(\log N)$.*

PROOF. PMA doubling requires initializing a new PMA of size $2N$, copying over the old PMA into the new one, and redistributing in the new PMA. Initializing the new PMA of size $2N$ and copying over the old data has work $O(N)$ and span $O(\log N)$ since these operations take $O(1)$ work per cell. As shown in Theorem 4.2, redistribute also has work $O(N)$ and span $O(\log N)$. Therefore, DOUBLE_PMA has work $O(N)$ and span $O(\log N)$. $\square$

Finally, the HALVE_PMA function is the inverse of the DOUBLE_PMA function and requires initializing a new PMA of half the size and copying over the elements into the new PMA. It has the same asymptotic behavior as DOUBLE_PMA with $O(N)$ work and $O(\log N)$ span.

External operations

Next, we will turn our attention to the SEARCH function, a key subroutine in the INSERT function. The search function SEARCH(v) checks a sorted region of the PMA and returns the location of the smallest element that is at least v in that region.

```
# returns the index of the first element with value at least v
def search(v):
  lo = 0
  hi = N
  while (lo < hi):
    mid = (hi - lo) / 2
    if pma[mid] is null:
      # gets beginning of next leaf
      mid = ((mid / log(N)) + 1) * log(N)
      # do a linear scan of size O(log N)
      if mid > hi:
        for i in [lo, hi):
          if pma[i] >= v: return i
    # pma[mid] guaranteed to be non-null
    if pma[mid] is v: return mid
    elif pma[mid] > v: hi = mid
    else: lo = mid
  return lo
```

Figure 4-5: Pseudocode for SEARCH(v).

Lemma 4.4 SEARCH(v) *has* $O(\log(N))$ *work and span.*

PROOF. The pseudocode for the SEARCH function can be found in Figure 4-5. The PMA uses a modified binary search to deal with null values. If the midpoint `pma[mid]` is null, the search algorithm sets the midpoint to the beginning of the next *PMA leaf* in $O(1)$ instructions. Since the parallel PMA enforces the *packed-left* property in its leaves, the beginning of each leaf is guaranteed to be non-null. Checking whether a cell is null and computing the beginning of the next leaf take constant time. Suppose that at some level of the binary search $\texttt{hi} - \texttt{lo} = \ell$. The maximum size of the next step is $\ell/2 + \log N$. If $\log N \approx \ell/2$, meaning that the search does not decrease the size of the next step by a constant fraction, then the algorithm can just look at all the cells serially with work and span $O(\log N)$. Otherwise, $\ell/2 + \log N = O(\ell/2)$, so the next search steps decrease the size of the search space by a constant fraction for at most $\log N$ binary search steps. □

```
# inserts the element v in sorted order
def insert(v):
  depth = log(N / log(N)), height = depth
  index = search(v)
  # slide elements to the right until a null space is found
  slide_right(index)
  pma[index] = v
  # range of this leaf we inserted into
  start = (index / log(N)) * log(N)
  end = start + log(N)
  counts = count_non_nulls(start, end)
  # non-integer division
  density = float(counts[0]) / log(N)
  while density > density_bound(depth):
    # get start and end of parent nodes
    start = get_parent_start(start, depth)
    end = get_parent_end(end, depth)
    count = get_element_count(start, end)
    density = float(count) /
      (log(N) >> (height - depth))
    depth = depth - 1
    if depth < 0:
      double_pma()
      return
  redistribute(start, end)
```

Figure 4-6: Pseudocode for inserting into a PMA.

The INSERT(v) function inserts an element v into a sorted PMA in amortized $O(\log^2 N)$ work [44] with the parallel modifications as described in Section 4.3. Each insert in a PMA requires a search to find the location to insert the element, which has $O(\log N)$ span because it is a binary search on the PMA.

Theorem 4.5 INSERT(v) *has* $O(\log^2 N)$ *worst-case span.*

PROOF. The pseudocode for the INSERT(v) function can be found in Figure 4-6. By Lemma 4.4, the SEARCH(v) function has $O(\log N)$ span. The slide-right function touches at most $O(\log N)$ cells of the PMA, so it also has $O(\log N)$ span. There are at most $O(\log N)$ calls to COUNT_NON_NULLS(s, t) and parallel prefix sum, which each have $O(\log N)$ span. Lastly, there is one call to either DOUBLE_PMA or REDISTRIBUTE(s, t), which have $O(\log N)$ span by Theorem 4.2 and Lemma 4.3. □

The bound in Theorem 4.5 is tight for the worst case when the entire PMA must be redistributed. The worst case is rare, however, and only happens once every $O(N)$ operations. To more accurately characterize the average case, we will now consider the ***amortized span***, or the total span of a set of parallel operations performed one at a time.

Theorem 4.6 INSERT(v) *has* $O(\log N)$ *amortized span.*

PROOF. The pseudocode for the INSERT(v) function can be found in Figure 4-6. The bulk of this proof will focus on analyzing the cost of the COUNT_NON_NULLS(s, t) function, which varies over inserts.

As before, SEARCH and SLIDE_RIGHT have $O(\log N)$ span.

Each insertion requires counting the elements in the corresponding leaf to check its density, which has $O(\log\log N)$ span. This is done by the helper routine GET_ELEMENT_COUNTS which returns the number of elements in a region by counting them in parallel with logarithmic span. For every $N/\log N$ insertions, the PMA has to redistribute a larger section. Specifically, it has to redistribute 2^i leaves every $N/(2^i \log N)$ insertions for positive integers i.

Let $H = \log(N/\log N)$, the height of the PMA. We calculate the "extra" span $T(N)$ of these redistributes over N insertions.

$$\begin{aligned}
T(N) &= \frac{N}{\log N}\sum_{j=1}^{H}\frac{1}{2^j}\Big(\sum_{i=1}^{j}\log(2^i\log N)\Big)\\
&= \frac{N}{\log N}\sum_{j=1}^{H}\frac{1}{2^j}\Big(j\log\log N + \sum_{i=1}^{j} i\Big)\\
&= \frac{N}{\log N}\Big(\log\log N\sum_{j=1}^{H}\frac{j}{2^j} + \sum_{j=1}^{H}\frac{j(j+1)}{2^{j+1}}\Big)\\
&\le \frac{N}{\lg N}\left(2\log\log N + 4\right) = O\Big(\frac{N\log\log N}{\log N}\Big).
\end{aligned}$$

The "total span" of counting the non-nulls over N insertions is therefore $O\left((N\log\log N)/\log N + N\log\log N\right)$ Dividing the "total span" by N yields $O(\log\log N)$ amortized span for the calls to COUNT_NON_NULLS(s, t) over N insertions.

There is one call to either DOUBLE_PMA or REDISTRIBUTE(s, t) on each insertion, which both have $O(\log N)$ span by Lemma 4.3 and Theorem 4.2. □

Deleting an element from the PMA is symmetric to inserting an element and has $O(\log^2 N)$ work, $O(\log^2 N)$ worst-case span, and $O(\log N)$ amortized span. A delete requires a search to find the element, a slide left to overwrite it, and a REDISTRIBUTE with lower density bounds to maintain the density requirements.

4.5 Inter-operation parallelism

This section describes a locking scheme to support theoretically-efficient concurrency with multiple parallel writers. Grabbing all the locks naively in serial disrupts parallelism because it makes the span $O(N)$ in the worst case. Therefore, this section explains how to grab locks in parallel to maintain the worst-case bounds from Section 4.4.

Description of locks The scheme augments the PMA with one lock per leaf[4] of the PMA. The proofs in this section assume grabbing a lock takes $O(1)$ work.

The parallel PMA uses reader-writer locks with a ranking system for prioritizing REDISTRIBUTE. When unlocking a lock, a thread can mark the lock so that the lock can only be taken by another thread with higher rank.

Grabbing locks in parallel Next, this section will show how to grab locks in parallel without deadlock and with polylogarithmic span. The only time a PMA will need need to grab multiple locks at once is on a REDISTRIBUTE, when a thread will grab all the locks in the subtree rooted at the node it is redistributing.

This section describes an algorithm called LOCK_ORDER for grabbing contiguous sequences of locks on leaves in parallel. The LOCK_ORDER algorithm grabs locks according to implicit priorities of each leaf in the PMA. It then serially iterates over each priority in order and grabs the locks with that priority in parallel.

The LOCK_ORDER algorithm first assigns implicit priorities to each leaf in the PMA depending on its index. The ***priority*** of a leaf with index i is POPCOUNT(i). The POPCOUNT function returns the number of ones in the bit representation of a number. For example, since $5 =$ `0b101`, POPCOUNT$(5) = 2$. We provide an example of how to assign priorities to nodes in Figure 4-7.

Remark 4.7 *The height of the root of any subtree defines the "pattern" of* POPCOUNTS*, but not the minimum popcount of the leaves in that subtree. For example, consider leaves 0-3 and 4-7 in Figure 4-7, which correspond to two subtrees with roots at the same level. The* POPCOUNTS *of consecutive leaves in each subtree have the same differences between them but have different minimums in the different subtrees. The unique minimum priority of any leaf in a subtree is the priority of the first leaf in that subtree. Consider leaves 4-7 in the second: their minimum* POPCOUNT *is 1 because the upper bit must be set (4 =* `0b100`*).*

The LOCK_ORDER algorithm is deadlock-free and has polylogarithmic span.

[4]Locking each leaf is equivalent to locking nodes at any set depth in the tree, which trades off between locking overhead and parallelism.

Theorem 4.8 *Grabbing locks for any two nodes in the PMA using* LOCK_ORDER *is deadlock-free.*

Proof. This proof will proceed with case analysis. Suppose two threads are trying to grab locks for two nodes a and b. We denote the set of leaves in the subtree rooted at some node γ with leaves(γ).

Case 1: leaves(a) $\cap$ leaves(b) $= \emptyset$. Since the regions have no locks in common, grabbing them in parallel will not cause deadlock.

Case 2: leaves(a) $=$ leaves(b). If $a = b$, there will be a unique leaf with lowest priority according to Remark 4.7. The thread that grabs it first will grab the rest of the region while the other one waits for it, avoiding circular wait.

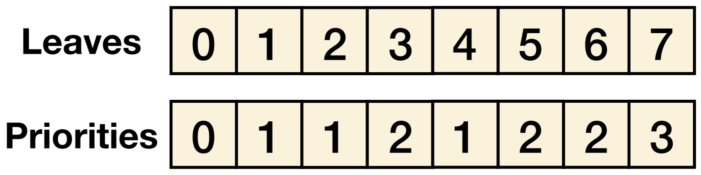
Case 3: leaves(a) $\subset$ leaves(b) (w.l.o.g.). Let left$_a$ be the leftmost leaf in leaves(a). Since left$_a$ has smaller priority than all the other leaves in leaves(a), both threads will attempt to grab it before any other leaf in leaves(a). Therefore, whoever grabs left$_a$ will be able to grab leaves(a) first. There is no circular wait because the thread trying to grab the locks of a need no locks outside of leaves(a).

In all cases, there is no circular wait and therefore no deadlock. □

Lemma 4.9 *Grabbing all the locks for any node in the PMA according to* LOCK_ORDER *has polylogarithmic span assuming $O(1)$ work to grab a lock.*

Proof. There are at most $\log N$ distinct priorities because there are at most $\log N$ bits required to represent the priority of a node. Furthermore, there are at most N locks with each priority, so grabbing all the locks with a given priority in parallel has $O(\log N)$ span. Therefore, the total span is $O(\log^2 N)$ in the worst case. □

Since most operations take locks for a small region of the PMA (e.g. inserts or small redistributes), it is rare to have to wait on another thread with a lock.



Leaves	0	1	2	3	4	5	6	7
Priorities	0	1	1	2	1	2	2	3

Figure 4-7: The indices of leaves in a PMA and the associated priorities.

4.6 Parallel Packed Compressed Sparse Row

This section introduces Parallel Packed Compressed Sparse Row (PPCSR), a parallel dynamic graph representation based on the parallel PMA. Along the way, it reviews the serial Packed Compressed Sparse Row [378] (PCSR) data structure based on the PMA as a basis for PPCSR. Next, it describes a locking protocol for PPCSR in order to enable multiple parallel writers. Section 4.7 describes how to implement graph operations in parallel using the operations described in Section 4.4.

Compressed Sparse Row. Compressed Sparse Row (CSR) is a common storage format for sparse graphs [317,361]. It stores a graph as a set of three dense arrays: a vertex array, an edge array, and a weights array. The edge array holds the edges first sorted by source, then by destination. The weights array stores the weights according to the order of edges in the edge array. The vertex array has one entry for each vertex corresponding to the start of its region in the edge and weight array.

Packed Compressed Sparse Row. PCSR replaces the dense edge and weight array of CSR with a PMA. Each cell in the vertex list stores pointers to the beginning and end of the region in the edge PMA corresponding to the edges of that vertex.

PCSR also stores sentinels at the beginning of a vertex's region in the edge PMA. ***Sentinels*** are special elements that hold pointers to the region's source in the vertex array. These sentinels facilitate updates to the vertex list when elements are shifted in the edge PMA.

One important difference between PCSR and a traditional PMA is that the PMA in PCSR stores multiple sorted lists (one for each vertex in the graph). Under the hood, the insert and search operations in PCSR are augmented with `lo` and `hi` variables that denote the beginning and end of the sorted neighbor list of interest.

PCSR requires a constant factor more space than CSR. The vertex array takes twice as much space because it stores a pointer to the beginning and end of each region. The edge PMA takes $O(m+n)$ cells compared to the m cells in CSR.

Figure 4-8 contains an example of a graph stored in PPCSR.

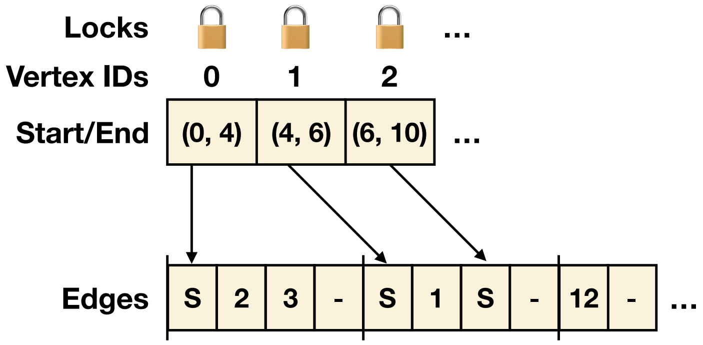


Figure 4-8: An example of a graph stored in PPCSR format. "S" denotes a sentinel at the beginning of a vertex's region in the edge PMA. The tall lines denote leaf boundaries and elements are packed left in leaves.

Section 4.5 describes how to lock a traditional PMA with one lock per node. In PPCSR, where there may be more than one lock per node from multiple vertices, grabbing all the associated vertex locks can be done sequentially. A vertex lock may also encompass multiple PMA leaves. Figure 4-9 presents an example of how vertex regions might be distributed among PMA nodes.

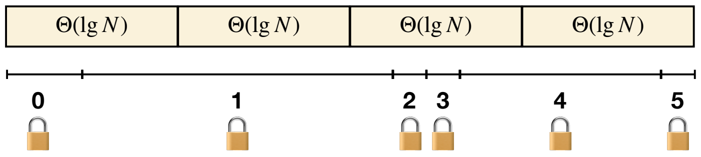


Figure 4-9: An example of the edge PMA in PPCSR with locks on vertices. The boxes represent the leaf boundaries of the PMA and the lines under the PMA represent regions associated with vertices in the graph (with their corresponding locks).

4.7 Parallel graph operations

This section shows how to implement the PPCSR system from Section 4.6 using the parallel PMA from earlier sections. Specifically, it will show how to implement graph operations using the parallel PMA operations from Section 4.4.

A graph represents data as n vertices and m edges. Section 3.2 presents graph preliminaries and notation necessary to understand graph processing systems.

Graph operations

Graph storage formats support the following read operations:

- FIND_WEIGHT: returns the weight of an edge or 0 if it is not in the graph.
- FIND_NEIGHBORS: returns the neighbors of a vertex.

Additionally, graph storage formats support the following write operations:

- ADD_EDGE: sets the weight of an edge if it is already in the graph, or adds the edge and its weight it if it is not yet in the graph.
- DELETE_EDGE: removes an edge from the graph.
- ADD_VERTEX: adds a vertex to the graph.
- DELETE_VERTEX: removes a vertex from the graph.

Read operations

Let us first consider the read-only operations FIND_WEIGHT and FIND_NEIGHBORS.

The FIND_WEIGHT routine corresponds directly with SEARCH in the PMA. From Lemma 4.4, FIND_WEIGHT has $O(\log(u))$ work and span.

The FIND_NEIGHBORS function finds the neighbors of a vertex in the graph. More formally, given a vertex $u \in V$, FIND_NEIGHBORS(u) returns a new set S_u of vertices such that for all $v \in V$, $v \in S_u$ if and only if $(u, v) \in E$. The pseudocode for FIND_NEIGHBORS can be found in Figure 4-10.

Lemma 4.10 FIND_NEIGHBORS(u) *has* $O(deg(u))$ *work and* $O(\log(deg(u)))$ *span.*

PROOF. Each `parallel_for` loop that iterates over $O(u))$ cells has work $O(deg(u))$ and span $O(\log(u))$ because it iterates through $O(deg(u))$ cells in parallel. As mentioned in Section 4.4, a parallel prefix sum on an array of length N can be implemented with span $O(\log N)$ [61]. The rest of the function takes $O(1)$ work. □

```
def find_neighbors(u):
  start = vertices[u].start
  end = vertices[u].end
  counts[end - start]
  # end - start = O(deg(u))
  parallel_for i in [start, end):
    if edges[i] is not null:
      counts[i - start] = 1
    else:
      counts[i - start] = 0

  parallel_prefix_sum(counts)
  output[counts[end - start - 1]]
  parallel_for i in [start, end):
    if counts[i] > counts[i-1]:
      output[counts[i-1]] = edges[i]
  return output
```

Figure 4-10: Pseudocode for FIND_NEIGHBORS in PPCSR.

Write operations

We begin by describing ADD_EDGE and showing how to implement it with parallel PMA operations. ADD_EDGE$(u, v, w(u, v))$ sets the value of the edge $(u, v) = w(u, v)$. If the edge $(u, v) \notin E$, ADD_EDGE adds it to the graph with weight $w(u, v)$.

Theorem 4.11 ADD_EDGE$(u, v, w(u, v))$ *has amortized* $O(\log^2(m+n))$ *work,* $O(\log^2(m+n))$ *worst-case span, and* $O(\log(m+n))$ *amortized span.*

PROOF. The ADD_EDGE function updates the edge structure in PPCSR. First, PPCSR performs a search to check if the edge already exists: if so, it updates the edge's weight. This takes $O(\log(\deg(u))$ work and span by Lemma 4.4.

Otherwise, PPCSR needs to insert a new edge using INSERT. It performs a modified version of INSERT to handle moving sentinels (in SLIDE_RIGHT and REDISTRIBUTE). This modification takes $O(1)$ work per edge because it checks if each cell contains a sentinel and if so, modifies the pointer to that sentinel in the vertex array. The INSERT function takes amortized work and worst-case span $O(\log^2(m+n))$ by Theorem 4.5 and amortized span $O(\log(m+n))$ by Theorem 4.6. □

The DELETE_EDGE procedure is just the inverse of ADD_EDGE and has amortized $O(\log^2(m+n))$ work, $O(\log^2(m+n))$ worst-case span, and $O(\log(m+n))$ amortized span.

Finally, PPCSR implements ADD_VERTEX with ADD_EDGE.
The ADD_VERTEX function adds a new vertex with index n to a graph with n vertices and updates the edge structure with a sentinel.

Lemma 4.12 ADD_VERTEX *has amortized* $O(\log^2(m+n))$ *work,* $O(\log^2(m+n))$ *worst-case span, and* $O(\log(m+n))$ *amortized span.*

PROOF. The ADD_VERTEX function updates both the vertex and the edge structure in PPCSR. First, ADD_VERTEX appends a new vertex to the end of the vertex array in amortized $O(1)$. If adding a new vertex triggers an $O(n)$ work copy, the copy has $O(\log n)$ span. PPCSR then inserts the sentinel in the same way we inserted an edge using a call to ADD_EDGE. $\square$

The DELETE_VERTEX function can be implemented with amortized $O(\log^2(m + n))$ work, $O(\log^2(m + n))$ worst-case span, and $O(\log(m + n))$ amortized span by keeping track of which vertices are deleted and rewriting the entire structure once half of them have been deleted.

4.8 Empirical evaluation

This section empirically evaluates PPCSR and compares it with Aspen [128], a state-of-the-art graph-streaming system, and Ligra [335]/Ligra+ [337], two static graph-processing systems. The evaluation shows that PPCSR achieves fast updates and queries and improves upon query speed compared to Aspen because it takes advantage of locality in PMA the data structure.

This section evaluates all systems on algorithm speed and memory usage, and Aspen and PPCSR on update throughput. We implemented four algorithms in PPCSR: breadth-first search (***BFS***), single-source betweenness centrality (***BC***), PageRank (***PR***), and connected components (***CC***).

Experimental setup. We implemented PPCSR as a `c++` library parallelized using `Cilk Plus` [191] and the Tapir [321, 322] branch of the LLVM [234, 235] compiler. We compiled Aspen, Ligra, and Ligra+ with `g++` version 7.5.

All experiments were run on a 48-core 2-way hyper-threaded Intel® Xeon® Platinum 8275CL CPU @ 3.00GHz with 189GB of memory from AWS [10].

Types of graphs. We tested on both real social network graphs and synthetic graphs. Social network graphs usually have a few very high-degree vertices while the rest of the vertices have low degree according to a power-law distribution [27]. We used the ***LiveJournal*** (LJ) and ***Orkut*** social network graphs from the SNAP dataset [241]. LiveJournal is a directed graph of the LiveJournal social network [69], and Orkut is an undirected graph of the Orkut social network. We also generated a random (***rMAT***) graph by sampling edges from an rMAT generator [93] with $a = 0.5; b = c = 0.1; d = 0.3$ to match the distribution from Aspen [128]. Finally, we generated a random Erdős-Rényi (***ER***) graph [140] with $n = 10,000,000$ and $p = 0.000005$ which was then symmetrized.

We used symmetrized versions of all the graphs for a fair comparison with the publicly available version of Aspen, which supports only unweighted undirected graphs. To store undirected unweighted graphs in PPCSR, we store directed edges both ways with weight 1. The sizes of all the graphs can be found in Table 4.1.

Since LiveJournal and Orkut are static graphs which may have been pre-processed with vertex reordering [376], we randomly relabel the vertices in all the input graphs to model the dynamic setting. Reordering is more difficult in streaming graphs because a good ordering may change with the stream of edges [18].

System descriptions. PPCSR and Aspen differ significantly in their underlying

Name	Vertices	Edges	Avg. Deg.
LiveJournal (LJ)	4, 847, 571	85, 702, 474	17.8
Orkut	3, 072, 627	234, 370, 166	76.2
rMAT	8, 388, 608	563, 816, 288	60.4
Erdős-Rényi (ER)	10, 000, 000	1, 000, 009, 436	100

Table 4.1: Sizes of (symmetrized) graphs used.

data structures and parallelization approaches. Aspen takes a purely functional approach with compressed trees, while PPCSR modifies a single parallel PMA with locks directly. Aspen is a compressed tree with difference encoding [346], whereas PPCSR is uncompressed. Aspen allows read-only operations (e.g. queries) during writing transactions, and vice versa (i.e. it does not use locks). It requires that the writer is sequentialized, however. In contrast, PPCSR supports concurrent readers or writers but uses locks, which prevents concurrent reading and writing in the same region of the data structure.

For simplicity, we implemented a locking scheme that grabs locks in a serial forward pass in PPCSR rather than according to the priority-based scheme described in Section 4.5. Since there is still an order to the locks, the forward-pass method is also deadlock-free. Although this method is not logarithmic in the worst case, almost all the operations only modify a small region of the PMA, so a thread usually only has to grab a constant number of locks.

Ligra is a static graph processing system that uses CSR as its underlying graph representation. Ligra+ adds data compression on top of the Ligra CSR representation.

Updates

We show that the batch insertions in PPCSR achieves up to 80 million edges per second for batch insertions and report our findings in Figure 4-2 and Table 4.2. To further optimize for large batches, PPCSR supports merging in a batch of edges. PPCSR outperforms Aspen on batches of up to 1, 000, 000 edges, while Aspen is faster on batch sizes of at least 10, 000, 000.

Setup. To generate our edges, we sample directed edges from the same rMAT generator that we used to generate the synthetic rMAT graph. To evaluate our insertion and deletion throughput, we add batches of directed edges to the LJ and ER graph in parallel (with potential duplicates). We report the average of 20 trials on small batches and the average of 5 trials on large batches.

Discussion. PPCSR is $2-5$x faster than Aspen on batch sizes up until 100, 000, competitive with Aspen on batches of 1, 000, 000, but does not scale with larger batch sizes as Aspen does. However, most highly dynamic graphs require much less throughput for huge batches. For example, Twitter averages 5, 700 tweets per second, and peaked at 140, 000 tweets per second [313].

Aspen implements batch insertions as a per-vertex merge, while PPCSR implements batch insertions as concurrent point insertions. For a batch size of B edges

	Insert						Delete					
	LJ			ER			LJ			ER		
Batch Size	PPCSR	Aspen	P/A	PPCSR	Aspen	P/A	PPCSR	Aspen	P/A	PPCSR	Aspen	P/A
1.00E1	6.31E5	8.07E4	7.82	5.57E5	7.28E4	7.65	1.59E6	8.39E4	18.9	2.06E6	7.49E4	27.5
1.00E2	7.10E5	4.48E5	1.59	6.48E5	4.32E5	1.50	1.52E6	4.71E5	3.22	1.56E6	4.28E5	3.65
1.00E3	4.34E6	2.12E6	2.05	5.13E6	1.97E6	2.61	7.80E6	2.24E6	3.49	8.24E6	2.12E6	3.89
1.00E4	2.63E7	5.55E6	4.74	2.82E7	4.93E6	5.71	3.03E7	6.25E6	4.85	3.18E7	5.44E6	5.83
1.00E5	3.98E7	2.01E7	1.98	4.30E7	1.26E7	3.42	4.74E7	2.02E7	2.35	5.10E7	1.18E7	4.31
1.00E6	5.54E7	5.18E7	1.07	6.08E7	2.69E7	2.26	7.64E7	5.15E7	1.48	7.90E7	2.66E7	2.97
1.00E7	5.30E7	1.70E8	0.31	7.67E7	7.76E7	0.99	7.98E7	1.70E8	0.47	8.29E7	7.97E7	1.04
1.00E8	2.08E8	4.56E8	0.46	4.68E7	2.50E8	0.19	2.43E8	4.98E8	0.49	7.87E7	2.72E8	0.29

Table 4.2: Throughput for inserting and deleting edges with varying batch sizes in the LJ and ER graphs in PPCSR and Aspen. P/A denotes the ratio of the respective throughputs (PPCSR/Aspen).

and a PPCSR representation with $|E|$ edges, merging in the batch takes $O(B + |E|)$ work. Since it is theoretically better to perform a merge when the batch size is very large ($B \approx O(|E|/\log^2|E|)$), PPCSR supports merging in very large batches. Since insert and delete throughput in both systems were comparable, we illustrate only the insert throughput in Figure 4-2.

In practice, memory bandwidth is the main bottleneck in insertions in PPCSR because every insert requires a cache-inefficient binary search. Although theoretically insertions into the ER graph should be slower than insertions into the LJ graph because the ER graph is much bigger, in practice they are similar because the size of the binary search each insertion requires is similar.

PPCSR supports insertions much faster than its worst-case theoretical bound of $O(\log^2|E|)$ would suggest. The theoretical bound is given by an amortization of the rebalances, but in practice the rebalances are extremely cache-efficient.

Query performance

We evaluate the performance of PPCSR, Aspen, Ligra, and Ligra+ on BFS, PR, (single-source) BC, CC and report the exact runtimes in Table 4.3.

Algorithm setup. In order to run all algorithms using the same API as the other systems, we implemented the `EdgeMap` / `VertexSubset` interface proposed by Ligra in PPCSR. The `VertexSubset` in PPCSR has an additional optimization for the case when the frontier is all the vertices, which improves PR and CC. We keep track of whether the frontier is full and skip membership queries if it is.

For PR, we removed the early exit and damping from the Ligra implementation, ported it into PPCSR, and verified the correctness of the translation into Aspen in private communication. For BFS and BC, we ported the Ligra implementation into PPCSR and ran the native Aspen implementations. For CC, we converted the Ligra implementation into PPCSR and Aspen. For BFS and BC, we ran all systems starting from the same vertex.

We implemented all kernels in PPCSR assuming undirected graphs to compare with Aspen. For each graph kernel, we took the average of ten trials.

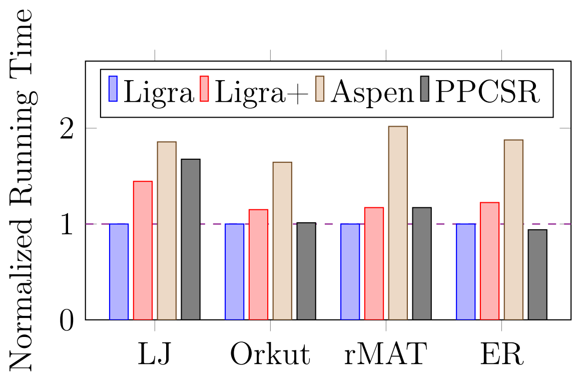


Figure 4-11: Time for all systems to calculate PR normalized to Ligra.

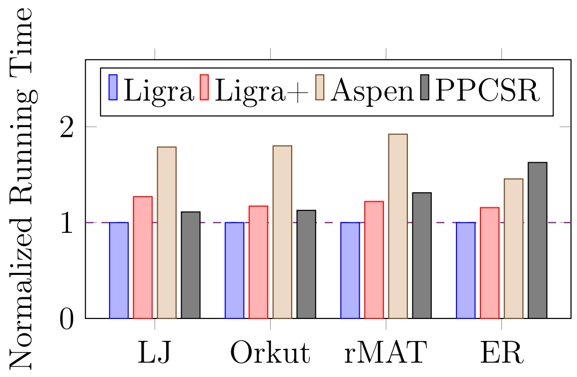


Figure 4-12: Time for all systems to calculate BFS normalized to Ligra.

PageRank. Figure 4-11 illustrates the relative speed on PR of all the systems. On all the graphs we tested, PPCSR achieves between $1.2-2$x speedup over Aspen on PR because PPCSR supports fast ordered traversals. Furthermore, PPCSR is competitive with Ligra and Ligra+ on PR (between $1-1.6$x slower). PR is essentially a linear scan through the PMA because it iterates through all vertices.

Breadth-first search. Figure 4-12 illustrates the relative speed on BFS of all the systems. PPCSR is competitive ($0.6-1.1$x) with Aspen and is $1.1-1.6$x slower than Ligra on BFS. We hypothesize that Aspen may experience extra work overheads due to compression. Furthermore, when the number of vertices in the BFS frontier is large, processing the frontier requires an efficient ordered scan through PPCSR.

Betweenness centrality. Figure 4-13 illustrates the relative speed on BC of all the systems. On BC, PPCSR is about 2x faster than Aspen, and competitive (about 1.3x) with Ligra. Since BC is more computationally- and memory-intensive than BFS, it requires more passes through the structures. PPCSR and Ligra support efficient ordered passes.

Connected components. Figure 4-14 illustrates the relative speed on CC of all the systems. PPCSR exhibits about 2x speedup over Aspen and achieves similar performance with Ligra on CC. Since CC starts with all vertices in the frontier, it has more iterations with many vertices, which PPCSR can traverse efficiently.

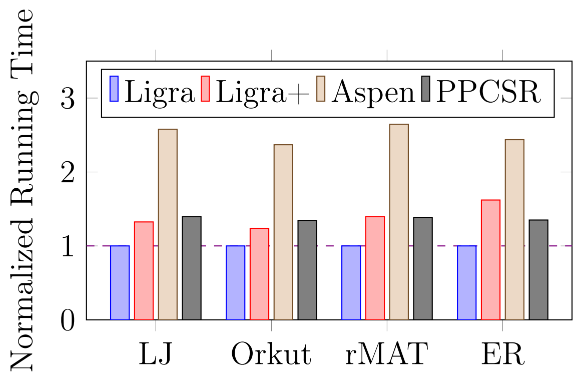


Figure 4-13: Time for all systems to calculate BC normalized to Ligra.

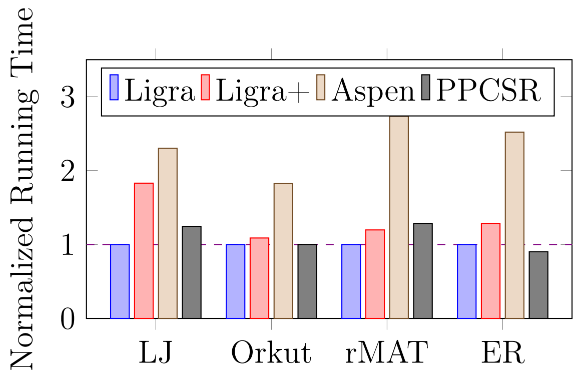


Figure 4-14: Time for all systems to calculate CC normalized to Ligra.

	BFS								PR							
	PPCSR		Ligra		Ligra+		Aspen		PPCSR		Ligra		Ligra+		Aspen	
Graph	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}
LJ	0.57	0.025	0.74	0.022	1.14	0.028	1.60	0.040	4.78	0.19	4.25	0.11	6.26	0.16	10.03	0.21
Orkut	0.53	0.021	0.65	0.019	0.94	0.022	1.55	0.033	7.31	0.20	8.14	0.20	8.61	0.23	15.73	0.32
rMAT	0.65	0.048	1.36	0.037	1.95	0.045	3.32	0.071	34.64	1.05	38.21	0.90	45.70	1.05	95.63	1.81
ER	1.26	0.054	1.11	0.033	1.59	0.038	2.01	0.048	76.49	1.68	70.56	1.79	85.55	2.19	155.34	3.36

	BC								CC							
	PPCSR		Ligra		Ligra+		Aspen		PPCSR		Ligra		Ligra+		Aspen	
Graph	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}	T_1	T_{96}
LJ	2.59	0.09	2.26	0.07	3.54	0.09	7.72	0.17	2.65	0.07	1.97	0.06	3.12	0.10	5.60	0.13
Orkut	3.24	0.11	3.01	0.08	4.31	0.10	9.46	0.19	5.84	0.09	4.05	0.09	4.15	0.10	10.57	0.17
rMAT	7.58	0.25	7.36	0.18	12.01	0.25	25.88	0.48	11.93	0.41	14.10	0.32	18.20	0.38	45.08	0.86
ER	8.85	0.22	6.76	0.16	12.32	0.27	24.93	0.40	35.19	0.55	28.00	0.61	38.65	0.78	82.76	1.53

Table 4.3: Running times of PPCSR, Ligra, Ligra+, and Aspen on BFS, PR, BC, and CC. T_1 denotes the time on one thread, and T_{96} denotes the time on all (96) threads

Memory usage

By design, PPCSR should use about 2× the space of an unoptimized CSR representation to store the empty spaces of the PMA. It can store the billion-edge ER graph in about 16 GB.

PPCSR uses between $1.3 - 2.3$x the space of Aspen, between $2 - 2.5$x the space of

Name	PPCSR	Ligra	Ligra+	Aspen
LJ	1.3	.34 (.66)	.23 (.55)	0.66 (.98)
Orkut	4.4	.91 (1.78)	.53 (1.4)	1.04 (1.91)
rMAT	8.79	2.13 (4.23)	1.51 (3.61)	3.93 (6.03)
ER	16.2	3.76 (7.49)	2.82 (6.55)	7.06 (9.16)

Table 4.4: Memory footprint (in GB) of the graphs on the different systems. PPCSR was run with weights, and all other systems were run without weights. To compare weighted and unweighted, we add the ideal $4 \times |E|$ (each weight is 4 bytes) to all structures which do not store weights, shown in parentheses.

Ligra, and $2.3 - 3.2$x the space of Ligra+. We report the memory usage of all systems in Table 4.4. One reason for the space difference is that Aspen and Ligra+ use data compression techniques (e.g. delta compression), while PPCSR is uncompressed.

4.9 Conclusion

Dynamic sparse graphs appear in applications from social networks to network routing and often see thousands of updates per second. This chapter introduced Parallel Packed Compressed Sparse Row, a dynamic graph data structure which has parallel operations with polylogarithmic span and allows for concurrent updates and queries. In practice, PPCSR supports about 80 million updates per second while maintaining fast queries and traversals and performs updates much faster than its worst-case theoretical bounds would suggest. PPCSR is especially well-suited to graph traversals scan through all of the edges (e.g. PageRank).

Locality-first strategy. PPCSR takes the first step in applying the locality-first strategy to dynamic-graph processing via cache-friendly data structures. First, the locality-first strategy involves understanding opportunities for locality in the problem. There are opportunities for spatial locality via co-locating as much data as possible. Therefore, the locality-first strategy exploits those opportunities with a single PMA data structure to hold all of the data contiguously. This chapter shows that surprisingly, concurrently updating a PMA is still fast because most of the operations 1) operate only on a small part of the data structure and 2) are cache-friendly and therefore efficient. The results in this chapter demonstrate the improved query and fast update performance due to optimizing for locality first.

Chapter 5

A Fill Estimation Algorithm for Sparse Matrices and Tensors in Blocked Formats

This chapter presents PHIL, a fill estimation algorithm that efficiently enable the use of blocked formats, which apply the locality-first strategy by enhancing spatial locality in sparse matrix and tensor computations. Sparse matrices and tensors from a variety of formats exhibit natural blocked structure where the nonzeroes are grouped together spatially. Blocked formats store the nonzeroes of sparse matrices and tensors in nonempty blocks rather as individual elements to enable spatial-locality-based optimizations such as vectorization. Each matrix has a unique blocked structure, however, so the blocked format must fit the matrix structure to minimize overhead. PHIL estimates the *fill*, a metric for block-size quality, with provable guarantees. These theoretical guarantees translate into improved practical performance: PHIL estimates the fill at least $2\times$ faster than OSKI on small matrices and $40-50\times$ faster on large matrices. Additionally, PHIL always produces useful estimates of the fill because of its accuracy guarantees. PHIL facilitates efficient usage of the locality-first strategy by minimizing the overhead necessary for blocked storage formats.

This work was conducted in collaboration with Peter Ahrens and Nicholas Schiefer [6]. This work also appears in Peter Ahrens' [7] and my [388] S.M. theses. We would like to thank Jiajia Li, Richard Vuduc, Charles E. Leiserson, Saman Amarasinghe, and David Karger for the helpful discussions.

Abstract

Many sparse matrices and tensors from a variety of applications, such as finite element methods and computational chemistry, have a natural aligned rectangular nonzero block structure. Researchers have designed high-performance blocked sparse operations which can take advantage of this sparsity structure to reduce the complexity of storing the locations of nonzeros. The performance of a blocked sparse operation depends on how well the block size reflects the structure of nonzeros in the tensor. Sparse tensor structure is generally unknown until runtime, so block size selection must be efficient. The *fill* is a quantity which, for some block size, relates the number

of nonzero blocks to the number of nonzeros. Many performance models use the fill to help choose a block size. However, the fill is expensive to compute exactly.

This chapter presents a sampling-based algorithm called PHIL to estimate the fill of sparse matrices and tensors in any format. We provide theoretical guarantees for sparse matrices and tensors, and experimental results for matrices. The existing state-of-the-art fill estimation algorithm, which we will call OSKI, runs in time linear in the number of elements in the tensor. The number of samples PHIL needs to compute a fill estimate is unrelated to the number of nonzeros and depends only on the order (number of dimensions) of the tensor, desired accuracy of the estimate, desired probability of achieving this accuracy, and number of considered block sizes. The empirical evaluation compares PHIL and OSKI on a suite of 42 matrices. On most inputs, PHIL estimates the fill at least 2 times faster and often more than 20 times faster than OSKI. PHIL consistently produced accurate estimates — in all cases that we tested PHIL was faster and/or more accurate than OSKI. Finally, the evaluation shows that PHIL and OSKI produce comparable speedups in multicore blocked sparse matrix-vector multiplication (SpMV) when the block size was chosen using fill estimates in a model due to Vuduc *et al.*

5.1 Introduction

In the spring of 2017, Peter Ahrens came to me and Nicholas Schiefer with the "fill-estimation problem" and an idea for a randomized sampling-based algorithm (which we later named PHIL) for approximating a property of blocked sparse matrices called the "fill". Practitioners developed blocked sparse storage formats to exploit the natural blocked structure of some sparse matrices for performance optimizations. Im *et al.* [189] introduced a quantity called the ***fill***, or the ratio of introduced zeros to the original number of nonzeros, to determine an optimal blocking for a given sparse matrix. The fill measures how well each blocking captures the natural blocked structure of a given sparse matrix. Vuduc *et al.* [372] then showed that choosing the correct matrix blocking can speed up sparse matrix-vector multiplication, a common numerical kernel, by more than a factor of 2 on matrices with blocked structure.

Since computing the fill exactly may take hundreds of times the cost of one sparse matrix-vector multiplication, researchers developed heuristics for estimating the quantity with reasonable accuracy. Vuduc *et al.* [369] proposed a randomized algorithm for estimating the fill of a sparse matrix. We call this fill-estimation algorithm OSKI since Vuduc *et al.* implemented the algorithm in the Optimized Sparse Kernel Interface (OSKI) [371]. OSKI approximates the fill much more quickly than exact algorithms and demonstrates the potential for randomized algorithms in computing the fill. Vuduc *et al.* [369] showed that OSKI empirically approximates the fill with reasonable error but lacks theoretical guarantees about either its accuracy or runtime.

Peter, Nicholas, and I decided to work on the "fill-estimation problem" and explore the potential for a fill-estimation algorithm with provable guarantees about its accuracy and runtime. We devised PHIL, a sampling-based fill-estimation algorithm that requires a number of samples independent of the input size and has both accu-

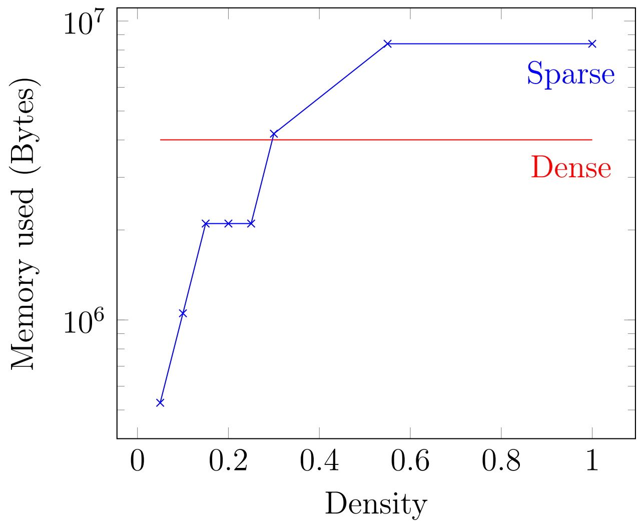


Figure 5-1: Size of a random sparse matrix $\mathcal{A}$ with $n = 1000$ and varying sparsity. For comparison, the size of a dense representation is shown as well. We used a full n^2 matrix as the dense representation and Compressed Sparse Rows as the sparse matrix representation. The x-axis represents the matrix density (i.e., $k(\mathcal{A})\ /\ n^2$), while the y-axis represents the size of the matrix representation.

racy and runtime guarantees. We then showed empirically that PHIL estimates the fill faster than OSKI and generated pathological inputs for OSKI where it does not provide any useful estimate of the fill.

Sparse matrices

Sparse matrices allow performance engineers to write fast algorithms and efficient data structures with complexity proportional to the number of nonzero entries. But sparse matrices introduce substantial storage and computational overhead per element. In contrast, dense formats have almost no computational overhead but may require much more space in total than sparse formats because they must store zeros. That is, ***the number*** $\boldsymbol{k(\mathcal{A})}$ ***of nonzero entries*** in an $m \times n$ sparse matrix $\mathcal{A}$ may be much smaller than $m \times n$. For example, Figure 5-1 compares the memory footprint of a matrix stored in a common sparse matrix format (Compressed Sparse Rows) and a matrix stored in a dense format, as a function of matrix density. Although sparse storage formats require extra space, they still may have an advantage over dense representations if the matrix has enough sparsity. Since sparse matrices have far more zeros than nonzeros, algorithms for sparse matrices may admit substantial performance improvements in performance over algorithms for dense matrices.

For example, sparse matrix-vector multiplication (SpMV) is one of the most heavily used numerical kernels in scientific computing because of its performance compared to dense implementations. Unfortunately, parallel implementations of SpMV are usually limited by memory bandwidth [82, 382]. Sparse matrix-vector multiplication on purely sparse matrix formats that store nonzeros individually usually results in irregular memory traffic due to the locations of the nonzeros.

Blocked formats

Blocked matrices and tensors (multidimensional generalizations of matrices) often appear in scientific computing. Specifically, sparse matrices from finite element methods [369] and sparse tensors from quantum chemistry [87] both exhibit regular block structure.

Since blocked structure varies across different sparse tensors, storage formats that take advantage of natural blocked structure must choose "blocking schemes" according to the structure of a tensor to avoid unnecessary overhead.

Definition 5.1 (Blocking scheme) *Suppose that $\mathcal{A}$ is a tensor of with R dimensions, or an **R-tensor**. Given a maximum block size B, a **blocking scheme** for $\mathcal{A}$ is a vector $\mathbf{b}$ of R block sizes $(b_1, b_2, \ldots, b_R)$ such that for all $i = 1, 2, \ldots R$, $b_i \in \mathbb{N}$ and $1 \leq b_i \leq B$. A blocking scheme $\mathbf{b} = (b_1, b_2, \ldots, b_R)$ applied to a tensor $\mathcal{A}$ tiles $\mathcal{A}$ into blocks of size $b_1 \times b_2 \times \ldots \times b_R$.*

For convenience, blocking schemes are sometimes called blockings.

Figure 5-2 shows an example of a blocking scheme $\mathbf{b} = (2, 3)$ on a sparse matrix. If any entry b_i does not divide the corresponding tensor dimension evenly, one can pad the tensor to the nearest next multiple of b_i.

Researchers have developed ***blocked formats*** which store dense blocks of nonzeros instead of storing the nonzeros individually to take advantage of the natural blocked structure of some blocked sparse matrices and tensors. Blocked formats may also represent some zeros explicitly if they appear in nonempty blocks as shown in Figure 5-2. Several storage formats and tensors reduce the complexity of storing individual entries by taking advantage of structural patterns in the locations of nonzeros [26, 82, 211, 304, 399]. The exact representation of a tensor in a blocked format depends on the selected blocking scheme.

Blocked storage formats are hybrid storage formats between fully sparse and dense storage formats and therefore take advantage of both sparsity and dense subarrays while reducing overhead. They simplify memory traffic and admit performance optimizations such as vectorization [211].

Whether a blocking scheme captures the structure of a sparse tensor determines the performance of a blocked sparse operation. Since zeros in the dense blocks must be stored explicitly, an ideal blocking scheme would perform well on a given architecture while minimizing the "filling in," or explicit representation, of zeros. The quality of a given blocking scheme depends on how well it captures the structure of the sparse tensor. A blocking scheme that fails to capture the structural patterns of a sparse matrix may introduce storage overhead because of introduced zeros without yielding any performance benefits. Vuduc *et al.* [372] shows that choosing the correct blocking can speed up sparse matrix-vector multiplication by more than a factor of 2 on matrices with blocked structure.

The fill in performance modeling

The benefits of blocked sparse formats raise a natural question: how do we choose an optimal blocking scheme for a sparse matrix or tensor?

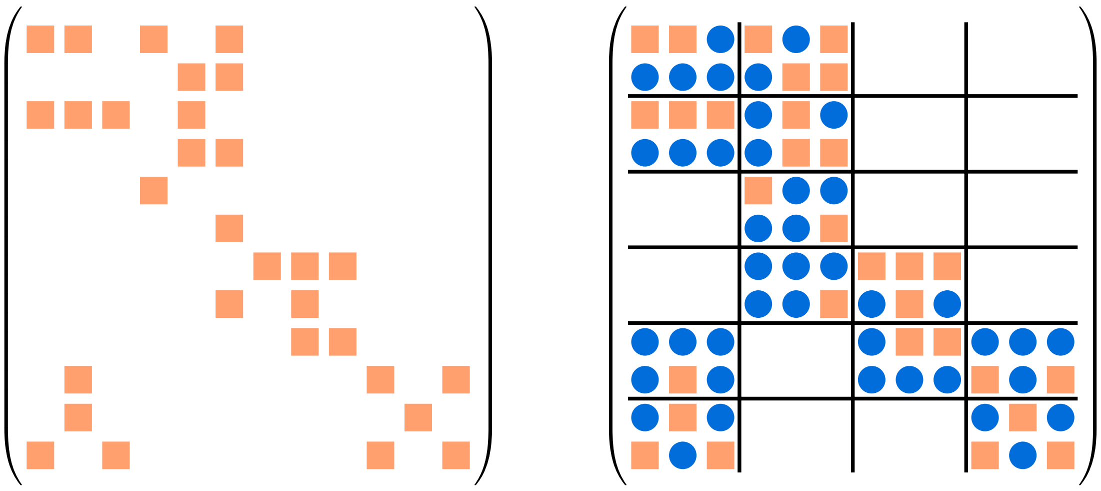


Figure 5-2: On the left, a sparse matrix before blocking. On the right, the same sparse matrix after blocking. The squares denote nonzero elements and circles are explicit zeros that are introduced due to the storage format. In this example, the blocking scheme $\mathbf{b} = (2, 3)$ and $k_{\mathbf{b}}(\mathcal{A}) = 12$. The number of nonzero elements $k(\mathcal{A}) = 30$, so the *fill* $f_{\mathbf{b}}(\mathcal{A}) = (2 \times 3 \times 12)/30 = 2.4$.

To measure how well a blocking scheme captures the structure of a sparse tensor, Im *et al.* [189] introduced a quantity called the fill. Given a sparse tensor $\mathcal{A}$ and a blocking $\mathbf{b}$, the ***fill*** $f_{\mathbf{b}}(\mathcal{A})$ is the ratio of introduced zeros to the original number $k(\mathcal{A})$ of nonzeros. Intuitively, a blocking scheme captures the structure of a sparse tensor well when it introduces relatively few explicit zeros. Since the fill is directly proportional to the number of filled-in zeros, it measures how well a blocking matches the blocked structure of a sparse matrix. Figure 5-2 shows the fill of a sparse matrix under blocking scheme $\mathbf{b} = (2, 3)$.

Researchers have developed "performance models" to determine the performance of blocked sparse operations based on the structure of a sparse matrix $\mathcal{A}$ and a blocking scheme $\mathbf{b}$. A ***performance model*** of a tensor $\mathcal{A}$ under blocking scheme $\mathbf{b}$ on a machine M is a function $P : \mathbb{R} \rightarrow \mathbb{R}$ that maps the fill $f_{\mathbf{b}}(\mathcal{A})$ to the expected performance in FLOP/s of a blocked sparse operation on $\mathcal{A}$ under $\mathbf{b}$.

The fill appears in performance models for a wide variety of blocked sparse kernels. Notably, it appears in several BCSR matrix-vector multiply performance prediction models [84, 188–190, 369, 371, 372] and performance models for for sparse triangular solve and sparse $\mathcal{A}^T\mathcal{A}\mathbf{x}$ [369]. The number of nonzero blocks (proportional to the fill) has been used in performance models for general blocked format sparse matrix-vector multiply [98, 213, 382]. Finally, an estimate of the fill can easily be added as an additional feature in feature-based machine learning approaches to sparse kernel performance modeling [243].

Example: SPARSITY *performance model for blocked SpMV*

As an example, let us examine the SPARSITY performance model for blocked sparse matrix-vector multiply due to Vuduc *et al.* [372]. We call the model SPARSITY because it appears in the SPARSITY library. There are more accurate performance models which still depend on the fill, but we shall focus on computing the fill and not performance modeling. It was later shown that, when the fill is known exactly, performance of the resulting blocking scheme was optimal or within 5% of optimal [369].

The SPARSITY performance model P_{SPARSITY} is an empirical model that is computed once per machine type and then used many times for different tensors and blocking schemes. It takes as input a profile of how a given machine M performs on dense blocks over all blockings, as well as an estimate of the fill $f_{\mathbf{b}}(\mathcal{A})$ of a matrix $\mathcal{A}$ under blocking scheme $\mathbf{b}$. Once per machine, we compute a profile of how the machine performs for each blocking scheme. Let $\text{PERF}(\mathbf{b})$ be the performance of the machine (in FLOP/s) on a dense matrix stored with blocking scheme $\mathbf{b}$. The measure $\text{PERF}(\mathbf{b})$ indicates how efficiently we can process nonzeros when nonzeros are stored under $\mathbf{b}$. The SPARSITY model estimates the expected performance of a blocked SpMV (in FLOP/s) of $\mathcal{A}$ under $\mathbf{b}$, as $\text{PERF}(\mathbf{b})/f_{\mathbf{b}}(\mathcal{A})$, then chooses a blocking scheme that maximizes the estimated performance.

Computing the fill in practice

Computing the fill exactly over all blocking schemes often takes hundreds of times as long as a single sparse matrix-vector multiplication. Since the structure of the sparse tensor is generally not known before runtime, blocking scheme selection must occur at runtime and must therefore be efficient. Thus, our problem is to quickly compute an estimate of the fill over all blocking schemes with reasonable accuracy. Recently, Langr, Šimeček, and Dytrych [232] attempted to parallelize exact computation of the fill for matrices. They were only able to provide competitive results, however, by computing a much smaller number of quantities. Since blocking scheme selection remains a difficult problem for tensors as it is costly to compute the fill exactly, developers have adopted empirical search techniques [345].

Although we limit the limited number of blockings in the case of sparse-matrix vector multiplication, computing the fill exactly over all possible blockings is still too costly. For dense blocks in matrices, let us focus on blocking schemes $\mathbf{b} = (b_1, b_2)$ that are small enough to fit b_1 elements of the input vector, b_2 elements of the output vector, and at least one input matrix element in registers. In practice [369], this requirement usually limits our attention to $b_1, b_2 \leq 12$.

OSKI*: a fill-estimation algorithm*

Vuduc *et al.* [188, 369] introduced the OSKI algorithm, which is the first and (to our knowledge) only existing algorithm that estimates the fill instead of computing it exactly. OSKI is the first known algorithm to produce an empirically accurate approximation of the fill over all blocking schemes in reasonable time.

Given a maximum block size B, OSKI uses randomization to compute the fill over a subset of a sparse matrix. For each block row size $b_1 = 1, 2, \ldots, B$, OSKI samples a fraction of block rows. For each sampled block row, OSKI computes the fill exactly for all block column sizes $b_2 = 1, 2, \ldots, B$ simultaneously. OSKI does this by iterating through coordinates (i, j) of nonzeros in the block row and using a perfect hash table for each block column size to record the number of unique block column coordinates $(\lceil j/b_2 \rceil)$ seen. The fraction of block rows evaluated is specified by a parameter σ which is usually set to 0.02.

Property	OSKI	PHIL
Described for	Sparse matrices	Arbitrary sparse tensors
Implemented for	Sparse matrices	Sparse matrices
What it samples	Block rows	Nonzeros
Estimates fill over	All blockings	All blockings
Number of samples	$\sigma(m/B)$	$B^{2R}\ln(2B^R/\delta)/(2\varepsilon^2)$
Operations to process a sample	$O(\sigma \cdot k(\mathcal{A}))$ (on average)	$(R+1)(2B)^R + B^R$
Error guarantee	None	Within a factor of ε

Figure 5-3: A comparison of OSKI and PHIL. OSKI requires the probability of sampling a block row σ and a sparse $m \times n$ matrix. PHIL computes an (ε, δ)- approximation of the fill of an R-tensor over all blockings with maximum block dimension B.

Although OSKI can estimate the fill of most matrices, it does not give predictable results. Notably, OSKI randomly samples block rows but may fail on matrices where the nonzeros are concentrated in a few rows because it may not evaluate those rows. In our work, we show that it is vulnerable to special cases. To our knowledge, there are no theoretical guarantees on the accuracy of OSKI, and no existing algorithm which estimates the fill of arbitrary tensors beyond matrices.

Moreover, OSKI lacks runtime guarantees. It samples random block rows and computes the fill based on all the nonzeros in those block rows. If OSKI samples block rows with probability σ, it evaluates $\sigma \times k(\mathcal{A})$ nonzeros on average, where $k(\mathcal{A})$ is the number of nonzeros in the matrix $\mathcal{A}$. If most of the nonzeros were concentrated in the selected block rows, however, OSKI's runtime would be linear in the number of nonzeros.

Approximation algorithms

PHIL does not guarantee to find the exact solution to the fill-estimation problem. It achieves theoretical guarantees on its accuracy based on the parameters ε and δ where ε is a multiplicative error bound and δ is a failure probability. We call such an algorithm an (ε, δ)-approximation algorithm.

An (ε, δ)-approximation algorithm guarantees concentration of an estimator around the actual quantity x we are trying to estimate.

Definition 5.2 *Let $\varepsilon > 0, 1 > \delta > 0$. An (ε, δ)-approximation algorithm produces an approximation x^* to a quantity x such that*

$$(1-\varepsilon)x \le x^* \le (1+\varepsilon)x$$

with probability $1-\delta$.

Contributions

The main contribution in this chapter is PHIL, the first fill-estimation algorithm with provable guarantees for sparse matrices and tensors. PHIL is a sampling-based,

(ε, δ)-approximation algorithm that randomly chooses a subset of the nonzeros in a tensor. PHIL uses prefix sums [61] to efficiently compute an estimate of the fill for all blocking schemes around each chosen nonzero.
PHIL takes as input the following parameters:

- a sparse R-tensor $\mathcal{A}$,
- the error bound ε,
- the failure probability δ,
- and the maximum block size B.

For an R-tensor (a tensor with R dimensions), the maximum block volume is therefore B^R.

Figure 5-3 summarizes the differences between PHIL and OSKI. We provide an exact bound on the number of samples that PHIL requires that *does not depend* on the number of nonzeros in the tensor. In contrast, OSKI runs in time linear in the number of nonzeros and is described only for matrices in one sparse format (CSR). As long as the tensor storage format allows fast (sublinear in the size of the input) access to elements of the tensor, PHIL runs in time sublinear in the number of nonzeros. Moreover, PHIL does not require a specific tensor storage format.

PHIL requires a number of samples and a total runtime independent of the size of the input tensor. Given an R-tensor and a maximum block size B, PHIL only needs $B^{2R}\ln(2B^R/\delta)/(2\varepsilon^2)$ samples to compute an (ε, δ)-approximation. In addition to the time taken to find the neighboring nonzeros, each sample (for all B^R blocking schemes) can be processed with $(R+1)(2B)^R$ integer additions and B^R floating point divisions and additions.

We experimentally evaluated the runtime, accuracy, and resulting SpMV times of PHIL and OSKI on a large suite of sparse matrices. We demonstrated experimentally that PHIL provides more accurate estimates than OSKI, while requiring only half the time, and often outperforming OSKI by more than a factor of 20. PHIL consistently provided accurate results even when OSKI produced results with a complete loss of accuracy. In all cases we tested, PHIL was faster and/or more accurate than OSKI. PHIL and OSKI produced fill estimates that resulted in almost identical sparse matrix-vector multiplication times when we used the SPARSITY performance model to select a blocking scheme.

Our contributions are as follows:

- PHIL, the first provably accurate fill-estimation algorithm for arbitrary sparse tensors.
- A theorem proving that PHIL requires exactly $B^{2R}\ln(2B^R/\delta)/(2\varepsilon^2)$ samples to compute an (ε, δ)-approximation of the true fill of an R-tensor over all block sizes given a maximum block dimension B.
- A scheme involving prefix sums that requires at most $(R + 1)(2B)^R$ integer additions to process each sample.

- An implementation of PHIL in `C`.
- An empirical evaluation of PHIL and OSKI on a large suite of sparse matrices that shows PHIL estimated the fill over ten times faster than OSKI and yielded almost identical SpMV speedups.
- The construction, theoretical analysis, and empirical evaluation of pathological inputs for PHIL and OSKI.
- A parallel implementation of PHIL in `Cilk` [67], which demonstrates that PHIL can be efficiently parallelized.

Map. The remainder of this chapter is organized as follows. Section 5.2 formalizes the mathematical preliminaries used in PHIL. Section 5.3 describes how PHIL samples nonzeros to estimate the fill. Section 5.4 proves worst-case error bounds on the fill estimate. Section 5.5 shows empirically that PHIL performs much better than its worst-case error bound. Section 5.6 concludes with open problems and extensions of PHIL.

5.2 Background

This section formalizes mathematical preliminaries required to understand PHIL. Since PHIL operates on sparse tensors, it reviews tensor notation. This tensor notation is required to represent the location of the nonzeroes that PHIL randomly samples. Next, it reviews various sparse tensor storage formats. Although PHIL does not require a specific storage format, this chapter describes PHIL in terms of the common Blocked Compressed Sparse Rows (BCSR) for concreteness. Finally, it formally defines the ***fill-estimation problem*** as the problem of computing an (ε, δ)-approximation of the fill.

Tensor notation

Tensors are multidimensional arrays over some field. Specifically, an R-tensor (tensor of order or rank R) is an array with R dimensions with elements from some field $\mathbb{F}$ (usually the real or complex numbers). We denote tensors by capital script letters $\mathcal{A}$ and vectors by lowercase boldface letters $\mathbf{a}$.

We now define how to index coordinates and ranges of coordinates in tensors. Let I_r be the size of the rth dimension of an R-tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$. A ***coordinate*** $\mathbf{i}$ is a list of R indices $(i_1, i_2, \ldots, i_R)$ where $1 \leq i_r \leq I_r$. We denote the element of $\mathcal{A}$ addressed by coordinate $\mathbf{i}$ as $\mathcal{A}[i_1, i_2, \ldots, i_R]$. For compactness of notation, we sometimes specify a coordinate as an R-component vector $\mathbf{i} = (i_1, i_2, \ldots, i_R)$. We represent the range of indices $i, i+1, \ldots, i'$ with the syntax $i : i'$. We represent a range of coordinates as $\mathbf{i} : \mathbf{i}'$, meaning $(i_1 : i'_1) \times \cdots \times (i_R : i'_R)$. Subtensors are formed when we fix a subset of coordinates. We also use ":" without bounds to indicate all elements along a particular dimension.

For convenience, we occasionally redefine the starting coordinate of a tensor. For example, the middle $n/2$ columns of a matrix $\mathcal{A} \in \mathbb{F}^{n \times n}$ are written $\mathcal{A}[:, n/4 : 3n/4]$. Thus, $\mathcal{A} \in \mathbb{F}^{\mathbf{I}:\mathbf{I}'}$ is an $(I'_1 - I_1 + 1) \times \cdots \times (I'_R - I_R + 1)$ tensor whose smallest coordinate is $\mathbf{I}$ and largest coordinate is $\mathbf{I}'$.

We denote the number of nonzero entries in a tensor $\mathcal{A}$ as $k(\mathcal{A})$.

When we compare a vector to a scalar, our comparison is true if and only if the comparison is true for each entry of the vector pointwise. For example, a blocking scheme $\mathbf{b} \leq B$ if and only if for all $i = 1, 2, \ldots, R, b_i \leq B$.

Sparse tensor representations

Although we mention a few specific sparse formats, PHIL applies to any sparse tensor format which admits iteration over nonzero coordinates. Since most sparse formats store only the coordinates which correspond to nonzeros and the nonzero values themselves, PHIL applies to many different sparse storage formats.

The simplest sparse matrix and tensor format is ***Coordinate (COO)*** [26]. In this format, all coordinates which correspond to nonzeros are stored in an unordered list. Entries are stored in sorted order of their coordinates. Figure 5-4 shows an example of a matrix and its COO representation.

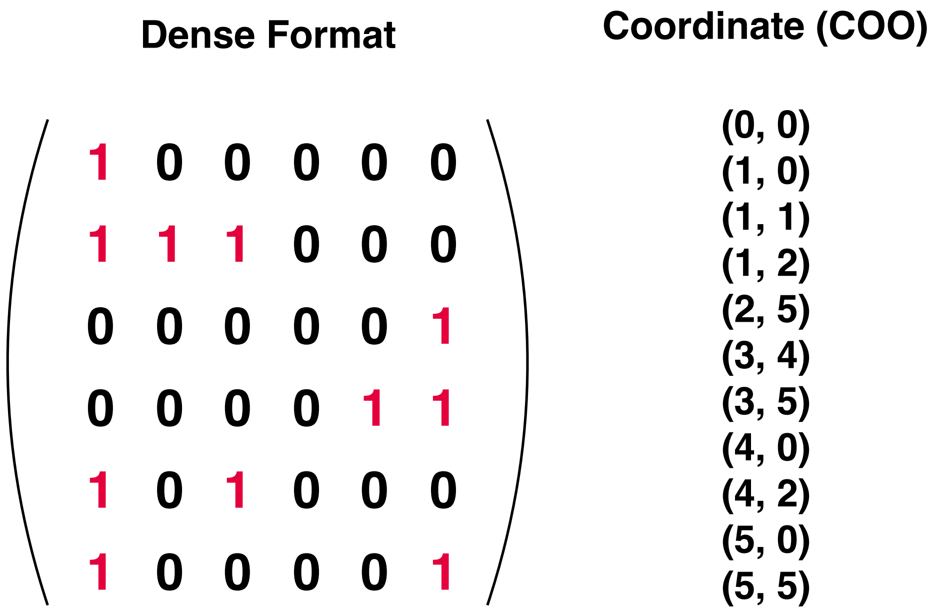


Figure 5-4: An example of a matrix (left) stored in coordinate (COO) format. COO stores the nonzeros in sorted order of their coordinates.

Perhaps the most popular sparse matrix format is ***Compressed Sparse Rows (CSR)*** [304]. In CSR format, the indices of nonzeros in each row are stored in sorted order. Each row has an associated list of coordinates of nonzeros. The nonzeros are stored in a single array with the same ordering as their coordinates. Figure 5-5 shows the same matrix from Figure 5-4 in CSR format.

CSR extends to tensor formats in many ways [26], such as ***Compressed Sparse Fibers (CSF)*** [220, 344]. In CSF format, each coordinate $\mathbf{i}$ is stored in a tree structure where a node in level r represents an index i_r that corresponds to a set of nonzeros. CSR is the matrix case of CSF.

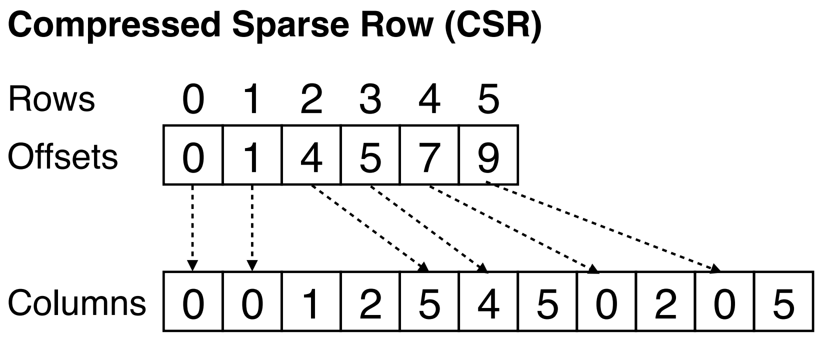


Figure 5-5: The same matrix from Figure 5-4 in CSR format. CSR stores a row array of offsets and a separate list of column indices.

Performance engineers use ***blocked storage formats*** to store blocks of nearby nonzeros together and therefore decrease the complexity of storing the coordinates of individual nonzeros. Blocked storage formats can reduce the memory usage of sparse operations by reducing the complexity of locating nonzeros. Programmers and compilers can optimize linear algebra on small dense blocks using standard techniques such as loop unrolling, register and cache blocking, and instruction-level parallelism. The effectiveness of these optimizations depends heavily on the structure of the tensor and the blocked storage format [211, 283].

Proposed blocked storage formats are diverse, altering parameters such as the size and alignment of blocks, or the storage format for locations of blocks and nonzeros within blocks [211]. Some formats [304, 399] involve reordering to improve the block structure of the tensor (in this case, blocks may not represent contiguous entries in the original tensor).

Regular blocking

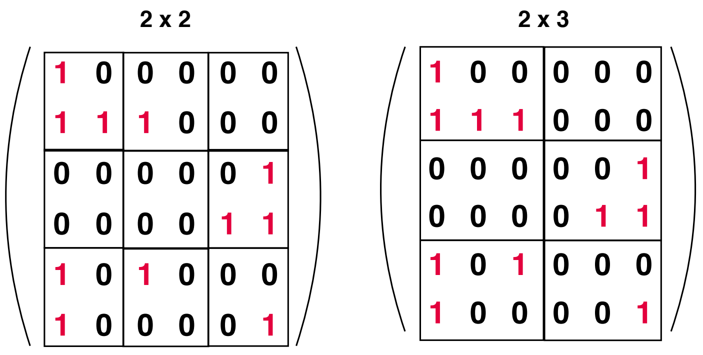
This chapter focuses on "regular blocking" for simplicity. In ***regular blocking***, all nonzero blocks are aligned rectangular blocks of equal size. Each block represents contiguous entries in the original tensor. We formally define regular blocking in Definition 5.3.

We used a blocked extension of CSR called ***Blocked Compressed Sparse Rows (BCSR)*** [304] in our experiments. The locations of the nonzero blocks in BCSR are recorded using CSR format. Figure 5-6 shows an example of the same matrix from Figure 5-4 in BCSR format under different blocking schemes. The BCSR format generalizes naturally to ***Blocked Compressed Sparse Fiber (BCSF)*** format [220, 345] for arbitrary tensors. In BCSR and BCSF, each block is stored in a dense format, with zeros represented explicitly, and only blocks which contain nonzeros are stored.

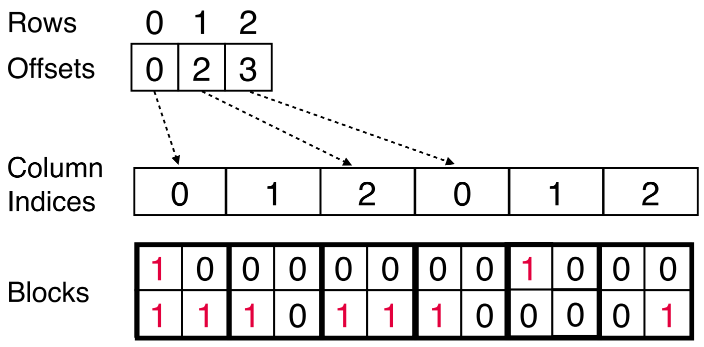
Definition 5.3 (Regular blocking scheme) *Let $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$ be an R-tensor. A (regular) blocking scheme $\mathbf{b}$ of $\mathcal{A}$ is a vector $\mathbf{b} = (b_1, b_2, \ldots, b_R)$ that partitions $\mathcal{A}$ into R-dimensional aligned subtensors of equal size with b_r entries along the r^{th} dimension. Each component of $\mathbf{b}$ is a block size.*

Figure 5-6: Examples of different blockings on the same matrix from Figure 5-4 and their representation in blocked compressed sparse row (BCSR).

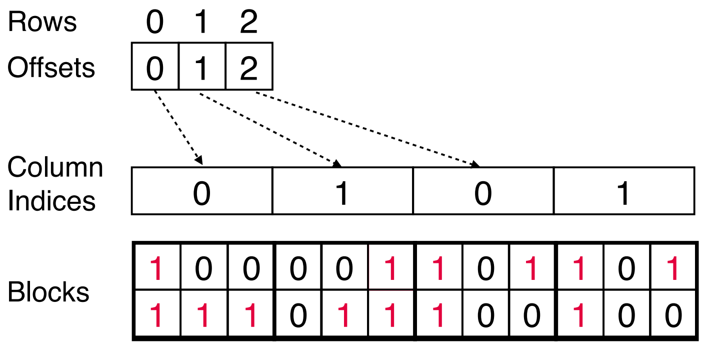
(a) Different blockings of the same matrix.



(b) BCSR representation of the matrix under a 2×2 blocking.



(c) BCSR representation of the matrix under a 2×3 blocking.



Each coordinate of $\mathcal{A}$ has a corresponding ***block coordinate*** *under blocking scheme* **b**. *Specifically, a nonzero at coordinate* **i** *has block coordinate*

$$\left(\left\lceil\frac{i_1}{b_1}\right\rceil, \left\lceil\frac{i_2}{b_2}\right\rceil, \ldots, \left\lceil\frac{i_R}{b_R}\right\rceil\right).$$

Fill-estimation problem

Since the performance of blocked sparse tensor operations depends on the blocking scheme and the structure of the tensor, our goal is to choose the blocking scheme that achieves the best performance for our given tensor. Larger blocks generally admit more opportunities for performance optimizations in blocked sparse formats with dense blocks. If the blocks do not capture the structure of the tensor, however, larger blocks hurt performance because they require computing over more explicitly represented (filled-in) zeros.

At a high level, a "good" blocking scheme includes all of the nonzero entries of a tensor in as few blocks as possible while minimizing the number of explicitly represented zeros.

Definition 5.4 *Supposed we have an R-tensor $\mathcal{A}$ and a regular blocking scheme $\mathbf{b}$. We define the number $k_{\mathbf{b}}(\mathcal{A})$ of blocks containing a nonzero under $\mathbf{b}$.*

Notice that $k_{\mathbf{1}}(\mathcal{A}) = k(\mathcal{A})$, since tiling $\mathcal{A}$ into unit-size blocks will have exactly one non-empty block for every nonzero.

Specifically, a "good" blocking scheme $\mathbf{b}$ for a tensor $\mathcal{A}$ minimizes the number $k_{\mathbf{b}}(\mathcal{A})$ of nonempty blocks while also minimizing the number of introduced zeros.

We now formally define the ***fill*** as a metric which uses the number of nonzero blocks to formally express this notion of blocking scheme quality:

Definition 5.5 (Fill [189]) *The* fill *of an R-tensor $\mathcal{A}$ with respect to a particular blocking scheme $\mathbf{b}$ is the ratio*

$$f_{\mathbf{b}}(\mathcal{A}) = \frac{b_1 \times b_2 \times \cdots \times b_R \times k_{\mathbf{b}}(\mathcal{A})}{k(\mathcal{A})}\,.$$

That is, the fill is the ratio of the number of entries in nonempty blocks of $\mathcal{A}$ under $\mathbf{b}$ to the number $k(\mathcal{A})$ of nonzeros in $\mathcal{A}$. Where it is clear which tensor we refer to, we often write the fill as $f_{\mathbf{b}}$.

The fill $f_{\mathbf{b}}(\mathcal{A})$ is directly proportional to the number of nonzero blocks $k_{\mathbf{b}}(\mathcal{A})$.

Exact computation of the fill for many blocking schemes is costly in comparison to the cost of a sparse matrix-vector multiplication. Instead of exactly computing the fill, our problem is to compute an estimate of the fill.

Problem 5.6 (Fill estimation) *Given an R-tensor $\mathcal{A}$ and a maximum block size B, the **fill-estimation problem** is the problem of computing an (ε, δ)-approximation $F_{\mathbf{b}}(\mathcal{A})$ to the true fill $f_{\mathbf{b}}(\mathcal{A})$ for all (square or rectangular) regular blocking schemes $\mathbf{b} \leq B$.*

Equivalently, we want to compute a random variable $F_{\mathbf{b}}(\mathcal{A})$ such that

$$\Pr\left[\max_{\mathbf{b} \leq B} \frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} > \varepsilon\right] \leq \delta\,.$$

Since $f_{\mathbf{b}}(\mathcal{A})$ differs from $k_{\mathbf{b}}(\mathcal{A})$ by a multiplicative factor of $b_1 b_2 \cdots b_R / k(\mathcal{A})$ (which can easily be computed in constant time), estimating the fill with respect to a blocking scheme is equivalent to estimating the number of nonzero blocks under that blocking scheme.

We will use these formal definitions of tensor notation and regular blocking to exactly define our PHIL algorithm in Section 5.3. Moreover, we show that PHIL solves the fill-estimation problem in Section 5.4.

5.3 PHIL

This section describes the PHIL algorithm for fill estimation and details its important subroutines. At a high level, PHIL randomly samples nonzeros. First, this section shows that PHIL's random sampling results in an accurate estimate of the fill. Next, it explains how PHIL efficiently estimates the fill over all block schemes for each sampled nonzero with a function called COMPUTE$\mathcal{X}$ that evaluates the entire neighborhood of each sample. It concludes by explaining a key step in processing each sample: finding all the nonzeros around a sample in time sublinear in the input size.

PHIL solves the fill-estimation problem by randomly sampling nonzero entries and counting the number of nonzero entries around each sampled nonzero. Suppose we want to estimate the fill of a sparse tensor $\mathcal{A}$ given a maximum block size B. PHIL repeatedly samples a coordinate $\mathbf{i}$ of a nonzero with replacement from $\mathcal{A}$. For each blocking scheme $\mathbf{b} \leq B$, it computes the number $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ of nonzero entries in the block that $\mathbf{i}$ appears in under the blocking scheme $\mathbf{b}$. Next, we show how PHIL uses $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ to estimate the fill.

Unbiased estimation of the fill

PHIL computes an accurate estimate of the fill by counting the number of nonzeros in each block for each sample. Let $\mathcal{A}$ be a tensor and $\mathbf{i}$ be a randomly chosen nonzero from $\mathcal{A}$. We define $F_{\mathbf{b}}$, a quantity proportional to the average of the reciprocals $1/z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$, and show that $F_{\mathbf{b}}$ is an ***unbiased estimator*** for the fill $f_{\mathbf{b}}$ (a random variable with expectation equal to the fill). We give a concentration bound for $F_{\mathbf{b}}$ in Theorem 5.7 and formally prove it in Theorem 5.13.

Theorem 5.7 (Maximum number of samples) *Suppose we want to estimate the fill $f_{\mathbf{b}}$ for all blocking schemes $\mathbf{b} \leq B$ where B is the maximum block size. If* PHIL *samples at least*

$$S \geq S_0 = \frac{B^{2R}}{2\varepsilon^2} \ln\left(\frac{2B^R}{\delta}\right)$$

samples with replacement, then it produces a fill estimate $F_{\mathbf{b}}$ over all blockings such that

$$\Pr\left[\max_{\mathbf{b} \leq B} \frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \leq \varepsilon\right] \geq 1 - \delta\,.$$

Notably, the number of samples PHIL requires to compute an (ε, δ)-approximation to the fill over all blocking schemes depends only on the maximum block size, desired accuracy, and failure probability. The required number of samples S_0 is independent of the input size, which is a clear advantage on large tensors where performance matters the most.

We describe how PHIL computes an unbiased estimator for the fill. First, we introduce the concept of the ***head*** and ***tail*** of a block because we will use it in later definitions.

Definition 5.8 (Head and tail of blocks) *The **head** of a block is the unique coordinate in the block with the lowest index along all dimensions. Let* $\mathbf{b}$ *be a regular* *blocking scheme and* $\mathbf{i}$ *be the coordinate in a tensor* $\mathcal{A}$*. We use* $h_{\mathbf{b}}(\mathbf{i})$ *to denote the head of* $\mathbf{i}$*'s block under the blocking scheme* $\mathbf{b}$*. Similarly, the* ***tail*** $t_{\mathbf{b}}(\mathbf{i})$ *of a block is the unique coordinate in the block containing* $\mathbf{i}$ *under* $\mathbf{b}$ *with the highest index along all dimensions.*

Next, we formally define the "fill component" of a nonempty block under some blocking. The ***fill component*** of a block is the reciprocal of the number of nonzeros in that block.

Definition 5.9 *Suppose we want to estimate the fill of a tensor* $\mathcal{A}$ *under a blocking scheme* $\mathbf{b}$*. Let* $\mathbf{i}$ *be the coordinate of a nonzero of* $\mathcal{A}$*. The* ***fill component*** *is the reciprocal of the number of nonzeros in the block of* $\mathcal{A}$ *containing* $\mathbf{i}$ *under* $\mathbf{b}$*.*

Formally, the fill component $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ *with respect to a nonzero* $\mathbf{i}$ *of* $\mathcal{A}$ *under a blocking* $\mathbf{b}$ *as*

$$x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}) = \frac{1}{z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})} = \frac{1}{k(\mathcal{A}[h_{\mathbf{b}}(\mathbf{i}) : t_{\mathbf{b}}(\mathbf{i})])},$$

where $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ *the number of nonzeros in the block of* $\mathbf{i}$ *under blocking scheme* $\mathbf{b}$*.*

The number of nonzeros in a block is not directly proportional to the fill. The average of the fill component over all nonzeros, however, is exactly the number of nonempty blocks, which is proportional to the fill. PHIL therefore estimates the fill by averaging $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over S coordinates $\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_S$ sampled with replacement from the set of coordinates of nonzeros in $\mathcal{A}$.

We show in Definition 5.10 that the fill estimate $F_{\mathbf{b}}$ is closely related to the average of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over all coordinates $\mathbf{i}$. We explain in Theorem 5.11 how the fill estimate $F_{\mathbf{b}}$ is an unbiased estimator of the fill.

Definition 5.10 (Fill estimate) *For all* $\mathbf{b} \leq B$*:*

$$F_{\mathbf{b}} := \frac{b_1 b_2 \cdots b_R}{S} \sum_{j=1}^{S} x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_j)$$

Theorem 5.11 (Unbiased estimator of the fill) *For any blocking scheme* $\mathbf{b}$*, the random variable* $F_{\mathbf{b}}$ *is an unbiased estimator for the fill: that is,* $\mathbb{E}[F_{\mathbf{b}}] = f_{\mathbf{b}}(\mathcal{A})$*.*

Proof. By definition, the sum over all nonzeros $\mathbf{i}$ within a particular block of fill components $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ is 1 if the block is not empty. Thus, the sum of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over all nonzeros $\mathbf{i}$ in $\mathcal{A}$ is equal to $k_{\mathbf{b}}(\mathcal{A})$, the number of blocks that contain nonzeros. Thus, we may multiply the average of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over $\mathbf{i}$ by $b_1 b_2 \cdots b_R$ to obtain an estimator of $f_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$, by Definition 5.5. $\square$

EstimateFill

The remainder of this section provides details about how PHIL computes a fill estimate. Figure 5-7 shows the highest level of PHIL and abstracts away how to process samples into a subroutine called Compute$\mathcal{X}$. Figure 5-9 shows how to efficiently process each sample to compute the fill over all blocking schemes. Since Compute$\mathcal{X}$ requires finding all nonzeros in a range, we conclude by explaining how to quickly find nonzeros in a range.

```
EstimateFill($\mathcal{A}$, $B$, $\varepsilon$, $\delta$)
    Require: $0 \le \delta \le 1\,, \quad \varepsilon > 0\,, \quad B \ge 1$
1   $\mathcal{Y} \in \mathbb{R}^{B \times \cdots \times B}$
2   $\mathcal{F} \in \mathbb{R}^{B \times \cdots \times B}$
3   $S = \left\lceil \frac{B^{2R}}{2\varepsilon^2} \ln\left(\frac{2B^R}{\delta}\right) \right\rceil .$
4   $\mathcal{Y} = 0$
5   for $\mathbf{i} \in$ sample of size $S$ with replacement from the nonzero coordinates of $\mathcal{A}$
6       $\mathcal{Y} = \mathcal{Y} + \textsc{Compute}\mathcal{X}(\mathcal{A}, B, \mathbf{i})$
7   for $\mathbf{b} \in B \times \cdots \times B$
8       $\mathcal{F}[\mathbf{b}] = \frac{b_1 b_2 \cdots b_R \mathcal{Y}[\mathbf{b}]}{s}$
9   return $\mathcal{F}$
    Ensure: $(1-\varepsilon) f_{\mathbf{b}}(\mathcal{A}) \le \mathcal{F}[\mathbf{b}] \le (1+\varepsilon) f_{\mathbf{b}}(\mathcal{A})$ with probability at least $(1-\delta)$.
```

Figure 5-7: Pseudocode for the EstimateFill routine. Given a sparse tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$, $\mathbf{i}$, and B, EstimateFill computes an approximation to $f_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all blocking schemes $\mathbf{b} \le B$.

Compute$\boldsymbol{\mathcal{X}}$

PHIL estimates the fill efficiently over all blocking schemes using prefix sums in a routine called Compute$\mathcal{X}$. Let $\mathbf{i}$ be a nonzero that PHIL randomly sampled from an R-tensor $\mathcal{A}$. PHIL computes the number $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ of nonzeros in each block that $\mathbf{i}$ appears in for each blocking scheme $\mathbf{b} \le B$. The first step of Compute$\mathcal{X}$ is to find the coordinates of all nonzeros near $\mathbf{i}$ in a routine called NonzerosInRange. Once we find the coordinates of all nonzeros near $\mathbf{i}$, we use multidimensional prefix sums (cumulative sums) to compute $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all blocking schemes $\mathbf{b} \le B$ in less than $(R+1)(2B)^R$ integer additions. Note that we expect both B and R to be small, and that the Compute$\mathcal{X}$ subroutine computes B^R separate quantities simultaneously.

We now describe how PHIL efficiently computes the number of nonzeros in all possible blockings around a sample $\mathbf{i}$ using prefix sums. A naive implementation of computing $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for a sample coordinate $\mathbf{i}$ by might take time B^R in an R-tensor by looking up all the nonzeros in a block corresponding to $\mathbf{i}$. In contrast, PHIL reuses the computations of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for the same $\mathbf{i}$ over different blocking schemes $\mathbf{b}$. Suppose PHIL samples a nonzero at coordinate $\mathbf{i}$. After finding the locations of all the nonzeros within a $2B$ radius of $\mathbf{i}$, PHIL computes $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all $\mathbf{b} \le B$ at the same time.

We describe the details of this routine in Figure 5-9 and provide an example in Figure 5-9. We abstract the process of finding the nonzeros in a range of a tensor into a subroutine NonzerosInRange and discuss potential efficient implementations after Figure 5-9.

The main idea behind Compute$\mathcal{X}$ is to count the number of nonzeros in blocks containing a sampled nonzero over all blocking schemes. Specifically, Compute$\mathcal{X}$ outputs a tensor $\mathcal{Z}_0$ corresponding to the number of nonzeros of an R-tensor $\mathcal{A}$ in subtensors surrounding a sampled nonzero $\mathbf{i} = (i_1, i_2, \ldots, i_R)$. Each entry of the tensor $\mathcal{Z}_0$ has the number of nonzeros in a corresponding blocking. We take the differences between relevant entries to find the number of nonzeros in all blockings around a sample $\mathbf{i}$. More formally, we construct an R-tensor $\mathcal{Z}_0 \in \mathbb{N}^{\mathbf{i}-B:\mathbf{i}+B-1}$ such that for all coordinates $\mathbf{j} = (j_1, j_1, \dots, j_R)$ within a $2B$ radius of $\mathbf{i}$, $\mathcal{Z}_0[\mathbf{j}]$ is equal to the number of nonzeros in the subtensor $\mathcal{A}[\mathbf{i} - B : \mathbf{j}]$. In one dimension, we can compute $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ as $\mathcal{Z}_0[t_{\mathbf{b}}(\mathbf{i})] - \mathcal{Z}_0[h_{\mathbf{b}}(\mathbf{i}) - 1]$. In two dimensions, we can compute $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ as $\mathcal{Z}_0[t_{\mathbf{b}}(\mathbf{i})] - \mathcal{Z}_0[t_{b_1}(i_1), h_{b_2}(i_2) - 1] - \mathcal{Z}_0[h_{b_1}(i_1) - 1, t_{b_2}(i_2)] + \mathcal{Z}_0[h_{\mathbf{b}}(\mathbf{i}) - 1]$.

We briefly describe how to use prefix sums to efficiently construct $\mathcal{Z}_0$ over all blocking schemes. We initialize $\mathcal{Z}_0[\mathbf{j}]$ to 1 if $\mathcal{A}[\mathbf{j}] \neq 0$ and 0 otherwise. Next, we take a prefix sum along each dimension in turn. After the first prefix sum, $\mathcal{Z}_0[\mathbf{j}]$ is the number of nonzeros in $\mathcal{A}[i_1 - B : j_1, j_2, \dots, j_R]$. After the r^{th} prefix sum, $\mathcal{Z}_0[\mathbf{j}]$ is the number of nonzeros in $\mathcal{A}[i_1 - B : j_1, \dots, i_r - B : j_r, j_{r+1}, \dots, j_R]$. After the R^{th} prefix sum (one along each dimension), we have computed $\mathcal{Z}_0$.

We find the number $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ of nonzeros in each block using differences between elements of $\mathcal{Z}_0$. Let $\mathbf{b} = (b_1, b_2, \dots, b_R) \leq B$ be a blocking scheme. For each value of b_1, we set $\mathcal{Z}_1[j_2, \dots, j_R]$ to the number of nonzeros in the subtensor $\mathcal{A}[h_{b_1}(i_1) : t_{b_1}(i_1), i_2 - B : j_2, \dots, i_R - B : j_R]$ as $\mathcal{Z}_0[t_{b_1}(i_1), j_2, \dots, j_R] - \mathcal{Z}_0[h_{b_1}(i_1) - 1, j_2, \dots, j_R]$.

We now show how to generalize COMPUTE$\mathcal{X}$ to arbitrary dimensions. After computing $\mathcal{Z}_1$ for a particular value of b_1, we take the difference between elements of $\mathcal{Z}_1$ for each value of b_2 to compute $\mathcal{Z}_2$, where $\mathcal{Z}_2[j_3, \dots, j_R]$ is the number of nonzeros in the subtensor $\mathcal{A}[h_{b_1}(i_1) : t_{b_1}(i_1), h_{b_2}(i_2) : t_{b_2}(i_2), i_3 - B : j_3, \dots, i_R - B : j_R]$. We do a similar computation for all R dimensions of the tensor until $\mathcal{Z}_R$ is just the scalar $z_{\mathbf{b}}(\mathcal{A}, \mathbf{j})$.

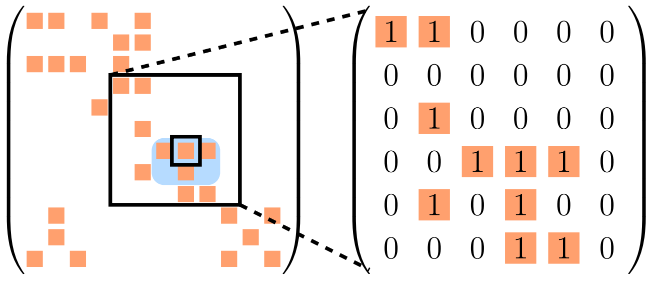
We conclude by analyzing how many operations we need to process each sample. PHIL takes prefix sums in each of the R dimensions where each prefix sum takes at most $(2B)^R$ additions to compute, and we compute R prefix sums. In the final loop, $\mathcal{Z}_r$ is of size $(2B)^{R-r}$. We must compute $\mathcal{Z}_r$ exactly B^r times. Therefore, the block difference computation incurs $\sum_{r=1}^{R} 2^{-r}(2B)^R$ subtractions. Thus, COMPUTE$\mathcal{X}$ uses at most $(R+1)(2B)^R$ integer additions to compute $\mathcal{Z}$.

```
Compute$\mathcal{X}$($\mathcal{A}$, $\mathbf{i}$, $B$)
    Require: $\mathcal{A}[\mathbf{i}] \neq 0\,, \quad B \geq 1$
 1  $\mathcal{Z}_0 \in \mathbb{N}^{\mathbf{i}-B:\mathbf{i}+B-1}$
 2  $\mathcal{Z}_0 = 0$
 3  for $\mathbf{j} \in$ NonzerosInRange($\mathcal{A}, \mathbf{i} - B, \mathbf{i} + B - 1$)
 4      $\mathcal{Z}_0[\mathbf{j}] = 1$
 5  for $r \in 1 : R$
 6      for $j \in i_r - B + 1 : i_r + B - 1$
 7          $\mathcal{Z}_0[\underbrace{:, \ldots, :, j}_{r}, :, \ldots, :] = \mathcal{Z}_0[\underbrace{:, \ldots, :, j}_{r}, :, \ldots, :] + \mathcal{Z}_0[\underbrace{:, \ldots, : j-1}_{r}, :, \ldots, :]$
 8  for $b_1 \in 1 : B$
 9      $\mathcal{Z}_1 = \mathcal{Z}_0[t_{b_1}(i_1), \underbrace{:, \ldots, :}_{r-1}] - \mathcal{Z}_0[h_{b_1}(i_1) - 1, \underbrace{:, \ldots, :}_{r-1}]$
10      for $b_2 \in 1 : B$
11          $\mathcal{Z}_2 = \mathcal{Z}_1[t_{b_2}(i_2), \underbrace{:, \ldots, :}_{r-2}] - \mathcal{Z}_1[h_{b_2}(i_2) - 1, \underbrace{:, \ldots, :}_{r-2}]$
12          $\vdots$
13          for $b_R \in 1 : B$
14              $\mathcal{Z}_R = \mathcal{Z}_{R-1}[t_{b_R}(i_R)] - \mathcal{Z}_{R-1}[h_{b_R}(i_R) - 1]$
15              $\mathcal{X}[\mathbf{b}] = \frac{1}{\mathcal{Z}_R}$
    Ensure: $\mathcal{X}[\mathbf{b}] = x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$
```

Figure 5-8: Pseudocode for the Compute$\mathcal{X}$ routine. Given a sparse tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$, $\mathbf{i}$, and B, the function computes $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all blocking schemes $\mathbf{b} \leq B$. Note that $\mathcal{A}$ may be stored in a sparse format, whereas all other tensors are stored in a dense format.

Figure 5-9: Here we visualize the execution of COMPUTE$\mathcal{X}$ as it computes one element of its output X. Specifically, we show how it computes $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}) = \mathcal{X}[\mathbf{b}]$. In this example, our maximum block size is $B = 3$ and our nonzero of interest is $\mathbf{i} = (7, 8)$. Continuing our example in Figure 5-2, we will show computation of $\mathcal{X}$ only for the blocking scheme $\mathbf{b} = (2, 3)$. Our goal is to compute the reciprocal of the number of nonzero elements in $\mathbf{i}$'s block (depicted by the shaded region).

(a) First, COMPUTE$\mathcal{X}$ uses NONZEROSINRANGE to find the nonzeros within a box of size $2B$ around $\mathbf{i}$. Then, it creates a matrix of the same size as the box and fills it with 0 where there are zeros in the original matrix and 1 where there are nonzeros.

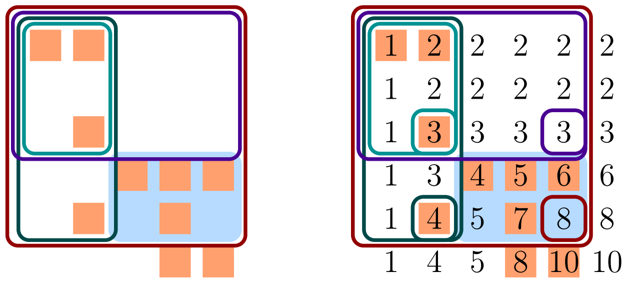


$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

(b) Next, COMPUTE$\mathcal{X}$ performs a prefix sum on the rows and then columns of the matrix. Notice that element $\mathbf{j}$ of the matrix is now equal to the number of nonzero elements in the box extending from the upper left of the matrix to element $\mathbf{j}$.

$$\begin{pmatrix} 1 & 2 & 2 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 & 3 & 3 \\ 0 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 2 & 2 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 2 & 2 & 2 & 2 \\ 1 & 2 & 2 & 2 & 2 & 2 \\ 1 & 3 & 3 & 3 & 3 & 3 \\ 1 & 3 & 4 & 5 & 6 & 6 \\ 1 & 4 & 5 & 7 & 8 & 8 \\ 1 & 4 & 5 & 8 & 10 & 10 \end{pmatrix}$$

(c) Finally, COMPUTE$\mathcal{X}$ computes the number of elements in the desired block by subtracting the number of nonzeros in each medium sized box from the large box, and adding back in the small box to avoid double-counting. Since all of these boxes begin in the upper left corner of our matrix, the number of nonzeros in these boxes are given by the prefix sum results in their lower right corners. The difference operation tells us that the shaded region contains $8-4-3+3 = 4$ nonzeros. Thus, $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}) = 1/4$. At this point, it is easy to compute $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for different $\mathbf{b}$ by repeating the difference operation with different blocks.



NonzerosInRange

Since $\mathcal{A}$ may be stored in an arbitrary sparse format, we abstract the process of finding the coordinates of nonzeros within a certain range into an algorithm called NonzerosInRange. NonzerosInRange$(\mathcal{A}, \mathbf{j}, \mathbf{j}')$ returns a list of all $\mathbf{i} \in \mathbf{j} : \mathbf{j}'$ such that $\mathcal{A}[\mathbf{i}] \neq 0$.

The implementation of NonzerosInRange depends on the initial format of the sparse matrix $\mathcal{A}$. We discuss two implementations to show why this routine should not be costly in theory or practice.

If $\mathcal{A}$ is a matrix in CSR format (where coordinates of nonzeros in each row are stored in sorted order of their column index), we do not need any preprocessing to quickly query nonzeros. Specifically, using a binary search within each row yields an $O(B \log_2(I_2) + B^2)$ time implementation, where the B^2 term is the maximum number of coordinates that may need to be returned. This search technique generalizes to arbitrary tensors in CSF format, yielding an $O\left(\sum_{r=2}^{R} B^{r-1} \log_2(I_r) + B^R\right)$ time implementation.

If $\mathcal{A}$ is stored in any other format (e.g. COO), we can preprocess the tensor such that we can query for nonzeros in a range in time independent of the input size. Before we run EstimateFill, we block the entire R-tensor $\mathcal{A}$ into blocks of size B^R (i.e. with blocking $\mathbf{b} = (B, B, \ldots, B)$) and store the blocks in a sparse format (without explicit zeros). We store each block that contains at least one nonzero in a hash table. Since PHIL only calls NonzerosInRange with ranges of size $2B \times \cdots \times 2B$, there are at most 3^R blocks which might contain zeros in the target range. To find all nonzeros in a range, we scan through these blocks to find nonzeros which are actually in the target range, and return the relevant nonzeros. This implementation of NonzerosInRange has a setup time of $O(k(\mathcal{A}))$ and an individual query time of $O(3^R B^R)$. After preprocessing, the time to complete query of NonzerosInRange is independent of the size of the input.

5.4 Theoretical analysis

This section proves that PHIL produces an accurate estimate of the fill with a number of samples independent of the input size. It first shows concentration bounds on the accuracy of PHIL's estimate using Hoeffding's inequality [185]. The number S of samples required for an accurate estimate only depends on the desired accuracy and probability of that accuracy. Notably, S is constant with respect to the input size, which is especially advantageous when $S \ll k(\mathcal{A})$. Finally, it proposes solutions in case the number of required samples exceeds the number of nonzeros in a tensor, which may occur if the tensor or matrix is small.

Concentration bounds on PHIL*'s error*

Theorem 5.12 (Hoeffding's inequality) *Let $X_1, X_2, \ldots, X_M$ be M independent random variables bounded such that $0 \leq X_j \leq 1$. Let $\overline{X} = \frac{1}{M}\sum_{j=1}^{M} X_j$ be their mean.*

Then for any $t \geq 0$,

$$\Pr\left[\left|\overline{X} - \mathbb{E}[X]\right| \geq t\right] \leq 2\exp(-2Mt^2)\,.$$

We can directly apply Hoeffding's inequality to PHIL's estimate to bound the error given the number of samples. Given a sparse tensor $\mathcal{A}$, a blocking scheme $\mathbf{b}$, and a tensor element $\mathbf{i}$, the fill component $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ is a random variable bounded between 0 and 1. Furthermore, since the samples $\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_S$ are chosen independently from among the nonzeros, the random variables $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_1), x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_2), \ldots, x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_S)$ are independent. Therefore, we obtain our concentration bound from Theorem 5.12.

Theorem 5.13 (Restatement of Theorem 5.7) *Suppose we want to estimate the fill $f_{\mathbf{b}}$ for all blocking schemes $\mathbf{b} \leq B$ where B is the maximum block size. If* PHIL *samples at least*

$$S \geq S_0 = \frac{B^{2R}}{2\varepsilon^2}\ln\left(\frac{2B^R}{\delta}\right)$$

samples with replacement, then it produces a fill estimate $F_{\mathbf{b}}$ over all blockings such that

$$\Pr\left[\max_{\mathbf{b}\leq B}\frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \leq \varepsilon\right] \geq 1 - \delta\,.$$

PROOF. By Definition 5.10, $F_{\mathbf{b}} = b_1 b_2 \cdots b_R (1/S)\sum_{j=1}^{S} x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_j)$. By Theorem 5.11, $\mathbb{E}[F_{\mathbf{b}}] = f_{\mathbf{b}}$. Since each examined block contains at least 1 and at most B^R nonzeros, $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_1), x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_2), \ldots, x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_S)$ are independent and bounded between $1/B^R$ and 1. Similarly, $k_b(\mathcal{A})/k(\mathcal{A})$ in Definition 5.5 is bounded to the same range. By Theorem 5.12,

$$\Pr\left[\frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \geq \varepsilon\right] = \Pr\left[\left|\frac{F_{\mathbf{b}} - \mathbb{E}[F_{\mathbf{b}}]}{b_1 b_2 \cdots b_R}\right| \geq \varepsilon\frac{f_{\mathbf{b}}}{b_1 b_2 \cdots b_R}\right]$$
$$\leq 2\exp\left(-2S\left(\frac{\varepsilon k_b(\mathcal{A})}{k(\mathcal{A})}\right)^2\right) \leq 2\exp\left(\frac{-2S\varepsilon^2}{B^{2R}}\right),$$

since $F_{\mathbf{b}}$ is $b_1 b_2 \cdots b_R$ times an average of S values, each of which is at least $1/B^R$. By the union bound over the B^R possible blocking schemes $\mathbf{b}$,

$$\Pr\left[\max_{\mathbf{b}\leq B}\frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \geq \varepsilon\right] \leq 2B^R\exp\left(\frac{-2S\varepsilon^2}{B^{2R}}\right).$$

Therefore, if $S \geq S_0 = \frac{B^{2R}}{2\varepsilon^2}\ln\left(\frac{2B^R}{\delta}\right)$,

$$\Pr\left[\max_{\mathbf{b}\leq \mathbf{B}}\frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \geq \varepsilon\right] \leq \delta\,.$$

□

The bound S on the number of samples PHIL needs to compute an (ε, δ)-approximation to the true fill is dependent only on the maximum block size, the order of the input tensor, and the desired approximation accuracy. Let $\mathcal{A}$ be an R-tensor. PHIL requires a number of samples that is only only dependent on B, R, ε, and δ. If ε and δ are independent of the number $k(\mathcal{A})$ of nonzeros, the bound S on the number of samples is also constant with respect to $k(\mathcal{A})$. Sampling is therefore especially advantageous when $S \ll k(\mathcal{A})$.

Obtaining a high probability bound with $\delta \leq 1/k(\mathcal{A})^w$ for some w would indeed require dependence on $k(\mathcal{A})$, albeit only logarithmically. In practice, however, a small constant δ such as 0.01 suffices.

Sampling for high accuracy or small tensors

PHIL may require more samples than the number of nonzeros in a small or very sparse tensor if one requests strong guarantees on its fill estimate. For example, a run of PHIL on a matrix ($R = 2$) may set the parameters $B = 12$, $\varepsilon = 0.1$ and $\delta = 0.01$. The number of required samples (10,645,998) may exceed the number of nonzeros in smaller matrices.

We can avoid this issue by sampling without replacement. If we sample without replacement, we can apply a variant of the Hoeffding-Serfling inequality [29] to obtain a bound which scales with the number of nonzeros. This bound is more complicated to describe, and requires the implementation to generate samples without replacement. Furthermore, this bound would still require sampling a significant fraction of the nonzeros.

Instead, we suggest that practitioners who need strong guarantees on small problems use an efficient exact algorithm or lower the maximum block size B. In our example, $B = 4$ needs only 103,308 samples. We show in Section 5.5 that PHIL empirically provides far more accurate estimates than the worst-case guaranteed theoretical bound. In practice, for $B = 12$, running PHIL with $\varepsilon = 3$ and $\delta = 0.01$ (11,829 samples) results in a mean maximum relative error of at most 0.05 for all cases we tested.

5.5 Experimental results

We tested PHIL and OSKI on a large suite of sparse matrices and found that PHIL estimates the fill more accurately in much less time than OSKI for many of the matrices in our test suite. There were no cases in PHIL was both less accurate and slower than OSKI.

Since OSKI lacks theoretical guarantees on its accuracy, we generated a pathological input matrix where OSKI produces useless fill estimates whereas PHIL produces accurate estimates. PHIL computes a provably accurate estimate of the fill for all inputs. We also generate a worst-case input for PHIL and show in Figure 5-10 that PHIL still produces a more accurate estimate than OSKI on this input.

We also found that when using optimized BCSR matrix-vector multiplication routines generated by the Tensor Algebra Compiler (TACO) [220] and the SPARSITY performance model (described in Section 5.1), the estimates produced by PHIL yield

BCSR matrix-vector multiply performance comparable to the performance obtained using estimates from OSKI.

We also chose a few matrices and ran PHIL and OSKI with multiple parameter settings on those matrices. Different parameter settings correspond to different runtimes. For example, the runtime of PHIL increases as ε and δ decrease. Figure 5-10 shows that the return on (time) investment for PHIL is better than OSKI on four matrices, including on synthetic matrices designed to bring out the worst in our PHIL algorithm.

Pathological inputs for PHIL *and* OSKI

We describe two pathological cases we invented to induce worst-case behavior in PHIL and OSKI, respectively. We generated these pathological matrices and call them `pathological_`PHIL and `pathological_`OSKI, respectively. We will show that `pathological_`PHIL is indeed a worst-case input for PHIL.

Definition 5.14 (Pathological PHIL **matrices)** *Pathological* PHIL *matrices are worst-case inputs for* PHIL*. These matrices have an equal number of completely full blocks and blocks with only one nonzero.*

We first try to provide some intuition about why pathological PHIL matrices are the worst-case inputs for PHIL. At a high level, pathological PHIL matrices maximize the variance of the PHIL estimator $F_{\mathbf{b}}(\mathcal{A})$. Let $\mathcal{A}$ be a worst-case tensor for a blocking scheme $\mathbf{b}$. Assume for contradiction that there are nonzero blocks which are not completely full and contain more than one nonzero. We can add nonzeros to more than half full blocks and remove nonzeros from more than half empty blocks to increase the *variance* of each of each fill component $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$. This reassignment increases the variance of the PHIL estimator $F_{\mathbf{b}}(\mathcal{A})$, which increases the probability that it will deviate farther from its mean. Thus, our worst case matrix has only completely full blocks and blocks with only one nonzero.

We formalize this intuition that the variance of the fill estimate $F_{\mathbf{b}}$ is maximized if full blocks and blocks with only one nonzero occur in equal number by showing that such matrices are maximally likely to cause a deviation between the true fill $f_{\mathbf{B}}$ and the PHIL estimator $F_{\mathbf{b}}$.

Theorem 5.15 *Consider a matrix $\mathcal{M}$ with an even number T of nonzero blocks under a particular blocking scheme $\mathbf{b}$, such that precisely $T/2$ of the nonzero blocks are completed filled with nonzeros and $T/2$ of the nonzero blocks contain only one nonzero. Then for any $\varepsilon > 0$ and matrix $\mathcal{M}'$ with T nonzero blocks under blocking scheme $\mathbf{b}$,*

$$\begin{aligned}&\Pr\left[|f_{\mathbf{b}}(\mathcal{M}') - F_{\mathbf{b}}(\mathcal{M}')|/f_{\mathbf{b}}(\mathcal{M}') > \varepsilon\right]\\ &\leq \Pr\left[|f_{\mathbf{b}}(\mathcal{M}) - F_{\mathbf{b}}(\mathcal{M})|/f_{\mathbf{b}}(\mathcal{M}) > \varepsilon\right]\end{aligned}$$

Proof. Given a matrix $\mathcal{M}'$ with T nonzero blocks, exactly one of the following statements must hold:

1. Every block in $\mathcal{M}'$ is either completely filled with nonzeros, or contains a single nonzero.

2. There are some blocks S that are not completely filled but contain more than one nonzero.

For any matrix for which (2) holds, we may pick a block in S and add a nonzero to it (if it more than half full) or remove a nonzero from it (if it is more than half empty). This increases the *variance* of each of each value $x_{\mathbf{b}}(\mathcal{M}', \mathbf{i})$, and therefore also increases the variance of the PHIL estimator $F_{\mathbf{b}}(\mathcal{M}')$. Increasing the variance increases the probability $\Pr\left[|f_{\mathbf{b}}(\mathcal{M}') - F_{\mathbf{b}}(\mathcal{M}')|/f_{\mathbf{b}}(\mathcal{M}') > \varepsilon\right]$. By induction on the number of applications of this procedure, there exists a matrix $\mathcal{A}$ where every block is either completely filled or contains a single nonzero such that $\mathcal{A}$ has a higher failure probability (i.e. is "more pathological") than $\mathcal{M}'$.

Suppose that $\mathcal{A}$ has pT blocks filled completely with ℓ nonzeros and $(1-p)T$ blocks containing a single nonzero, for some $0 \le p \le 1$. Therefore, every $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ is either $1/\ell$ or 1, in the case where $\mathbf{i}$ is in a completely filled block or a nearly-empty block, respectively. The variance of the PHIL estimator $F_{\mathbf{b}}(\mathcal{A})$ is given by $p(1-p)/\ell$, which is maximized when $p = 1/2$. Thus, $\Pr\left[|f_{\mathbf{b}}(\mathcal{A}) - F_{\mathbf{b}}(\mathcal{A})|/f_{\mathbf{b}}(\mathcal{A}) > \varepsilon\right]$ is maximized when $\mathcal{A}$ is $\mathcal{M}$. □

For our concrete test case, we create a $10,000 \times 10,000$ matrix called `pathological_`PHIL with $10,000$ full 12×12 blocks and $10,000$ sparse 12×12 blocks. PHIL should perform poorly on this matrix.

We also devised an empirically pathological matrix called `pathological_`OSKI to bring out the worst in the OSKI algorithm. Since OSKI samples rows with equal probability, hiding many blocks which look different from the rest of the matrix in a single row should cause OSKI to perform poorly. We tested PHIL and OSKI on a `pathological_`OSKI matrix of size $100,000 \times 100,000$ where the first 6 rows are dense, while all other rows have only a single nonzero in the first column.

Evaluation metrics

Since program autotuning algorithms typically run at runtime before execution of the tuned operation, the speedups gained by autotuning must be weighed against the execution time of the algorithm. Because we tested an example of autotuning blocked SpMV, we normalize the time OSKI and PHIL take to estimate the fill by the duration of an unblocked parallel CSR SpMV.

We use the SPARSITY performance model to select a blocking scheme. Since the estimated performance is proportional to the fill, we judge the quality of a fill estimate using the maximum relative error.

Definition 5.16 *The maximum relative error of a fill estimate f over all blockings $\mathbf{b} \le B$ is*

$$\max_{\mathbf{b} \le B} \frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}}.$$

Note that a maximum relative error greater than 1 represents a complete loss of accuracy, as a bogus algorithm that returns 0 for the estimated fill of all blocking schemes would achieve a better maximum relative error.

Empirical study with fixed parameters

We tested PHIL and OSKI on almost all of the matrices with more than one million nonzeros from the sparse matrix collection using the default recommended settings of both algorithms. All but two are from the University of Florida Sparse Matrix Collection (Suitesparse) [113]. These matrices were chosen to represent a variety of application domains and block structures.

My S.M. thesis [388] contains all of the results from our comparison of PHIL and OSKI with fixed parameters. The default parameters to PHIL are $\varepsilon = 3$ and $\delta = 0.01$ when $B = 12$, and they are $\varepsilon = 0.25$ and $\delta = 0.01$ when $B = 4$. The parameters to OSKI are $\sigma = 0.02$ (the recommended setting) for all cases.

These extensive experiments show that for a fixed setting of parameters, the runtime and relative error of our fill estimation algorithms varies substantially from matrix to matrix (although the relative error of PHIL is consistently small).

We compare PHIL and OSKI with fixed settings in terms of runtime, mean maximum relative error, and the resulting BCSR SpMV time. Figure 5-11 shows an example of our with study with fixed parameters on our two synthetic matrices.

Our results show that in most cases, PHIL was more accurate and much faster than OSKI. PHIL always produced results with a mean maximum relative error less than .05, while in a few cases OSKI produced results with a mean maximum relative error which was worse or much worse than 1. Finally, we test PHIL and OSKI on the synthetic pathological matrices and report our findings in Figure 5-11.

Since PHIL uses a fixed number of samples, PHIL's normalized runtime appears higher for small matrices because PHIL takes longer relative to the parallel CSR matrix-vector multiplication time on smaller matrices. On larger matrices (when autotuning is most important), however, PHIL usually takes at most 10 matrix-vector multiplies, outperforming OSKI by factors of 10 to 40.

Both the PHIL and OSKI estimates led to remarkably similar BCSR matrix-vector multiplication times. It may be possible to improve the chosen blocking schemes with a more complex performance model [84], but our focus is on estimating the fill and not on modeling the performance of sparse kernels.

Accuracy return on time investment

Since running both algorithms under fixed settings is only one way to execute PHIL and OSKI, we compared the algorithms using a range of parameters on a selection of matrices in Figure 5-10. Figure 5-10 shows the mean maximum relative error as a function of the runtime of the estimation algorithm on four different matrices.

We chose four matrices as a representative sample of inputs. We compared PHIL and OSKI on the matrices `ct20stif` and `gupta1` from Suitesparse because Vuduc *et al.* [369] used them to measure OSKI. We also tested PHIL and OSKI on our pathological inputs.

We found that PHIL provides better estimates of the fill than OSKI for any amount of time invested. On these four matrices, PHIL is both more efficient and more accurate than OSKI. On `pathological_PHIL`, PHIL performs better than OSKI, but the performance difference is smaller than the difference between PHIL and OSKI on `ct20stif` and `gupta1`. On `pathological_OSKI`, OSKI fails to estimate the fill in any reasonable time.

Experimental setup

We now explain how we generated our empirical results. We implemented[1] both PHIL and OSKI for sparse matrices in CSR format in `C`, which can efficiently execute the dense integer and floating point operations in COMPUTE$\mathcal{X}$ (Figure 5-9). Finally, both implementations run serially and use the `mt19937` random number generator from the `C++` Standard Library.

We chose blocking schemes to maximize estimated performance of blocked SpMV according to the SPARSITY performance model. To create the performance matrix PERF for the SPARSITY performance model, we timed BCSR matrix-vector multiplication performance for 100 trials on a 1000×1000 dense matrix. We used TACO to generate parallel BCSR kernels for each blocking scheme, which we ran on one socket with 12 threads.

We ran all of our experiments on a node with two sockets, each with a 12-core Intel® Xeon™ Processor E5-2695 v3 “Ivy Bridge” at 2.4 GHz. Each core has 32 KB of L1 cache and 256 KB of L2 cache. Each socket has 30 MB of shared L3 cache.

5.6 Conclusion

This chapter introduced PHIL, the first fill-estimation algorithm with provable guarantees. PHIL computes an (ε, δ)-approximation to the fill and requires a number of samples independent of the input size.

It also showed empirically that PHIL estimates the fill of a sparse matrix at least 2 times faster than OSKI on most of our real-world inputs and provides useful estimates of the fill even in pathological test cases. PHIL and OSKI produced comparable speedups in blocked sparse matrix-vector multiply in most cases using their recommended parameters. PHIL produced far more accurate estimates of the fill than its worst-case accuracy guarantee.

Sampling techniques are useful in program autotuning since we can often sacrifice some accuracy in the heuristics for a faster autotuner. As libraries for numerical

[1]Our serial code is available under the BSD 3-clause license at `https://github.com/peterahrens/FillEstimation/releases/tag/IPDPS2018.`

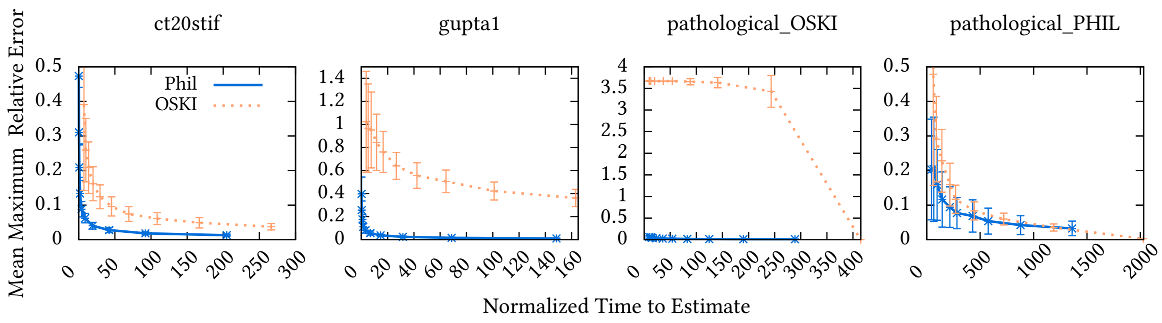


Figure 5-10: Mean maximum relative error (Definition 5.16) as a function of mean estimation time (normalized to the mean time it takes to perform a parallel sparse matrix-vector multiplication in CSR format using TACO [220]) for four matrices. Both axes use logarithmic scale. All means are the average of 100 trials. The error bars reflect one standard deviation above and below the mean. The blue solid line represents PHIL and the orange dotted line represents OSKI. Each point is a separate setting for the parameters. `ct20stif` is the stiffness matrix arising from the application of finite element methods to a structural problem with some block structure. `gupta1` is the matrix representation of a linear programming problem, and has no obvious block structure. The pathological matrices are described in more detail in [388]. Note that errors above 1 represent a complete loss of accuracy.

			$B = 12$						$B = 4$					
Matrix Information			Normalized Time to Estimate Fill		Mean Maximum Relative Error		Normalized TACO SpMV Time		Normalized Time to Estimate Fill		Mean Maximum Relative Error		Normalized TACO SpMV Time	
Name	NNZ (k)	Size (m + n)	PHIL	OSKI	PHIL	OSKI	PHIL	OSKI	PHIL	OSKI	PHIL	OSKI	PHIL	OSKI
Domain: Synthetic														
pathological_PHIL	72,356	23,989	695.7	177.4	0.046	0.383	1.0*	1.0*	2.769	90.79	0.092	0.037	1.0*	1.0*
pathological_OSKI	69,994	20,000	164.0	33.30	0.012	3.666	0.635	0.635	0.793	17.05	0.060	1.800	0.713	0.809

Figure 5-11: On the pathological synthetic matrices, we show the mean estimation time, mean maximum relative error (Definition 5.16), and the resulting mean parallel sparse matrix-vector multiply (SpMV) time in BCSR format with the optimal blocking scheme according to the SPARSITY performance model. Times are normalized to the mean time taken to perform one parallel sparse matrix-vector multiply (SpMV) on the unblocked CSR matrix. All means are the average of 100 trials. All blocked and non-blocked matrix-vector multiplies are performed using TACO. Highlighted cells show the better result between PHIL and OSKI. The left group of columns corresponds to a maximum block size $B = 12$. The right group of columns corresponds to a maximum block size of $B = 4$. * Results with an asterisk are cases where a slowdown was observed when the performance model was used with the given estimates. Since most autotuners will try both an unblocked CSR format and the predicted best blocking scheme with BCSR format, they may choose to use CSR if no speedup is observed and so these results are listed as 1.0.

computation evolve and autotuning moves from compile-time to run-time implementations, developers will need efficient heuristics [133]. PHIL's empirical success suggests broader potential for sampling techniques in the design of autotuned numerical software. Faster sampling algorithms with provable guarantees will allow library developers to write software that can more accurately specialize to user data and provide the best possible performance for their application and hardware.

Future work

Future work includes an optimized, vectorized implementation of PHIL and an extension to handle sparse tensors in multiple storage formats. COMPUTE$\mathcal{X}$ should benefit from instruction-level parallelism. One of our goals in the design of PHIL was to express the fill-estimation problem as a dense set of operations that can be computed efficiently.

We found that the blocked SpMV times due to blocking schemes chosen according to the SPARSITY performance model were similar for both PHIL and OSKI. Perhaps a more complex performance model [84] would lead to different choices of blocking schemes and therefore different blocked SpMV performance.

Coarse fill estimation

Some blocked formats [82,399] store their blocks in a sparse format. These blocks are usually much larger than the blocks we considered in this thesis, but we can extend any algorithm (e.g. PHIL) for Problem 5.6 to estimate the fill of larger blocks by limiting our attention to multiples of some base block size.

Problem 5.17 (Coarse fill estimation) *Given a tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$, a base block size $\mathbf{q}$, and a maximum multiplier B, compute an approximation $F_{\mathbf{b}}(\mathcal{A})$ accurate to within a factor of ε for all $\mathbf{b}$ where $b_r = b'_r q_r$ and $1 \leq \mathbf{b}' \leq B$ with probability $1 - \delta$.*

Let $\mathcal{A}' \in \mathbb{F}^{I'_1 \times I'_2 \times \cdots \times I'_R}$ be a tensor. We first set $\mathcal{A}'[\mathbf{j}]$ to the number of nonzeros in block $\mathbf{j}$ of $\mathcal{A}$ under the blocking scheme $\mathbf{q}$. Notice that $f_{\mathbf{b}'}(\mathcal{A}') = f_{\mathbf{b}}(\mathcal{A})$, so a solution to Problem 5.6 on $\mathcal{A}'$ is a solution to Problem 5.17 on $\mathcal{A}$. Since $k(\mathcal{A}') \leq k(\mathcal{A})$, $\mathbf{I}' \leq \mathbf{I}$, and we can construct $\mathcal{A}'$ in $O(k(\mathcal{A}))$ time, most algorithms (including PHIL) that solve Problem 5.6 can solve Problem 5.17 with an addition of $O(k(\mathcal{A}))$ to their asymptotic running time.

Locality-first strategy. PHIL efficiently supports blocked formats, which apply the locality-first strategy to sparse matrix and tensor operations. The first step in the strategy is to understand locality in the problem. Sparse operations often have low temporal locality, but there are opportunities for spatial locality. Blocked formats enhance spatial locality by storing nonzeroes in nonempty blocks rather than individually. These blocked formats trade off some task parallelism at the individual nonzero level for other types of parallelism such as data-level parallelism such as vectorization. PHIL estimates the fill, an important quantity in block size selection, to efficiently enable blocked formats with minimal overhead and high performance.

Chapter 6

Write-Optimized Skip Lists

This chapter presents an write-optimized external-memory skip list that takes the first step in the locality-first strategy by understanding and optimizing for spatial locality in data structure design. The write-optimized skip list is a serial data structure that takes the first step towards an efficient parallel write-optimized data structure by optimizing for locality first. It achieves asymptotically optimal performance for all operations in the DAM model by exploiting spatial locality.

This work was conducted in collaboration with Michael A. Bender, Martín Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, and Cynthia Phillips [49].

Abstract

The skip list is an elegant dictionary data structure that is commonly deployed in RAM. A skip list with N elements supports searches, inserts, and deletes in $O(\log N)$ operations with high probability (w.h.p.) and range queries returning K elements in $O(\log N + K)$ operations w.h.p.

A seemingly natural way to generalize the skip list to external memory with block size B is to "promote" with probability $1/B$, rather than $1/2$. However, there are practical and theoretical obstacles to getting the skip list to retain its efficient performance, space bounds, and high-probability guarantees.

This chapter gives an external-memory skip list that achieves write-optimized bounds. That is, for $0 < \varepsilon < 1$, range queries take $O(\log_{B^\varepsilon} N + K/B)$ I/Os w.h.p. and insertions and deletions take $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ amortized I/Os w.h.p.

The write-optimized skip list inherits the virtue of simplicity from RAM skip lists. Moreover, it matches or beats the asymptotic bounds of prior write-optimized data structures such as B^ε trees or LSM trees. These data structures are deployed in high-performance databases and file systems.

The main technical challenge in proving the write-optimized bounds comes from the fact that there are so few levels in the skip list, an aspect of the data structure that is essential to getting strong external-memory bounds. This chapter uses extremal-graph coloring to show that it is possible to decompose paths in the skip list into uncorrelated groups, regardless of the insertion/deletion pattern. Thus, this chapter achieves write-optimized bounds by averaging over these uncorrelated paths rather than by averaging over uncorrelated levels, as in the standard skip list.

6.1 Introduction

The skip list [310] is an elegant randomized dictionary data structure built from cascading linked lists of geometrically decreasing sizes.

A skip list with N elements supports searches, inserts, and deletes in $O(\log N)$ operations with high probability[1] (w.h.p.) and range queries returning K elements in $O(\log N + K)$ operations w.h.p. [125, 219, 297]. Skip lists have found broad application [12, 21, 23, 149, 164, 180, 199, 291, 332], and they are widely deployed in production [223, 301, 330].

This chapter proposes a ***write-optimized skip list***. The write-optimized skip list is a randomized ***external-memory*** dictionary that offers asymptotically optimal point-query and insertion performance in the external-memory model while inheriting many of the practical and theoretical advantages of a traditional skip list.

By external-memory, we mean that the data structure resides on a large external storage device, such as a disk or SSD. The external storage device is accessed via I/Os that transfer blocks of size B to a smaller cache (e.g. RAM) of size M.

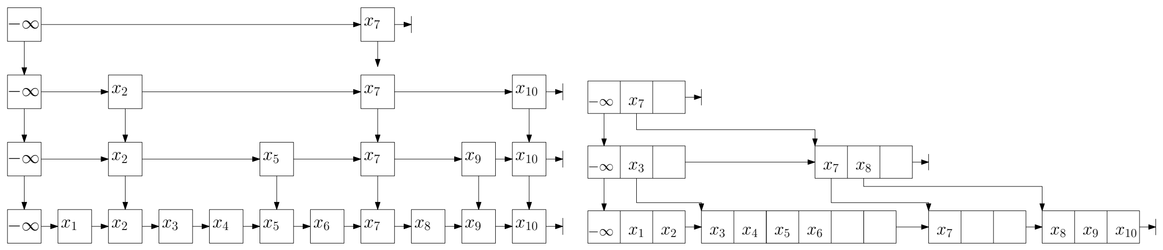
By write-optimized, we mean that the data structure has asymptotically better insertion performance than a B-tree [34] and query performance at or near that of a B-tree. In practice, the best write-optimized dictionaries match B-trees in terms of query speed while performing insertions and deletions one or two orders-of-magnitude faster. Over the past two decades, researchers have developed write-optimized dictionaries for databases and file systems [20, 46–48, 75, 76, 81, 165, 166, 201, 244, 288, 324, 325, 392], several of which have been shown to be asymptotically optimal [76, 392].

Skip list structure. A skip list consists of $h = O(\log N)$ lists $\{\mathcal{L}_0, \mathcal{L}_1, .., \mathcal{L}_h\}$, called ***levels***, where the base level $\mathcal{L}_0$ is a linked list of all items of the set, in sorted order. Each item in level $\mathcal{L}_i$ also appears in (i.e., ***is promoted to***) level $\mathcal{L}_{i+1}$ with probability $1/2$. All elements that have been promoted to $\mathcal{L}_{i+1}$ are ***pivots*** with respect to $\mathcal{L}_i$ because they partition $\mathcal{L}_i$ into ranges for searches. An element promoted to level $\mathcal{L}_{i+1}$ has a pointer to its successor in level $\mathcal{L}_{i+1}$ as well as a pointer to its own occurrence in level $\mathcal{L}_i$ (see Figure 6-1).

A query for element y begins at the first node on level $\mathcal{L}_h$ and ends on level $\mathcal{L}_0$ at the smallest element no greater than y. At level i, the search performs a sequential scan until it finds the last element, e, that is less than or equal to y in $\mathcal{L}_i$. At that point, the search follows the pointer to e in level $\mathcal{L}_{i-1}$ and resumes its sequential scan from that point.

An insertion of element e first performs coin tosses to compute the height h_e of e. The insertion then searches for e and inserts it into lists $\mathcal{L}_0, \ldots, \mathcal{L}_{h_e}$, with appropriate pointer adjustments.

[1]An event E_n on a problem of size n occurs ***with high probability*** if $\Pr\{E_n\} \geq 1 - 1/n^c$ for some constant c.



(a) RAM skip list. **(b)** B-skip list.

Figure 6-1: An in-memory (RAM) skip list (a) and external-memory B-skip list (b). In the B-skip list, the node size varies by a factor of $O(\log^2 N)$. While the B-skip list achieves asymptotically better bounds than the RAM skip list in expectation, they both achieve the same high-probability bounds [39]. In contrast, the write-optimized skip list has better bounds than the B-skip list both in expectation and w.h.p.

Inheriting the desirable properties of skip lists. Skip lists have desirable algorithmic properties, which the write-optimized skip list inherits.

For example, it is an advantage to be built from a collection of linked lists. Practitioners generally like to make concurrent lock-free dictionaries as lock-free skip lists [149, 179, 309, 357, 364] because it is attractive to build on top of existing, production-quality lock-free linked lists [260, 271, 349].

Moreover, skip lists are elegant and have an easy-to-understand randomized balancing mechanism. Finally, skip-lists are weight balanced [282] in a probabilistic sense, which makes them useful as an algorithmic tool.

It is these desirable properties that makes us particularly excited to have another optimal write-optimized data structure at our disposal, even though (a few) other optimal structures already exist [46, 75, 76, 81, 392].

See Section 6.7 for speculation how a write-optimized skip list may make it easier to implement concurrency and perhaps lock-freedom. Given that the community is only now exploring how to make full-featured, scalable, acid-compliant, write-optimized indexing structures, it is worth having many options in an implementer's arsenal.

Adapting to external memory

We now articulate the subtleties in adapting skip lists to work in external memory. We review the external-memory model, which is used to analyze disk-resident indexing structures in databases and file systems.

External-memory model. The external-memory or disk-access model (DAM) [3] consists of two levels of memory: a fast memory (RAM) of size M and a slow arbitrarily large external memory, such as a disk. Block transfers, or I/Os, between disk and RAM occur in blocks of size B. Performance is measured in terms of the number of I/Os.

External-memory skip lists. Given the success of the skip list in internal memory, it is natural to extend it to external memory. Indeed, such a data structure exists and is called a B-skip list [1, 39, 51, 86, 104, 161].

The straightforward way to extend the skip list to external memory is to promote elements with probability $1/B$ rather than $1/2$. At a given level, each promoted element is stored in a contiguous chunk along with the run of nonpromoted elements that follow it. These chunks define the ***nodes*** of the B-skip-list. Since disk blocks have size B, each node consumes at least B space, regardless of how many elements it contains (see Figure 6-1).

This B-skip list retains the simplicity of the original RAM skip list but unfortunately has optimal search performance only in expectation, not with high probability [39]. Each node has an expected $\Theta(B)$ elements, but w.h.p. there exist nodes with as many as $\Theta(B \log N)$ elements and nodes with as few as $\Theta(B/\log N)$ elements. Large nodes cause problems because we want searches to take $O(\log_B N)$ I/Os, but performing a linear scan of a node of size $\Theta(B \log N)$ requires $\Theta(\log N)$ I/Os.

We can obtain high-probability bounds on the cost of searches by changing the promotion probability to $1/\sqrt{B}$, rather than $1/B$ [39]. Even with this larger promotion probability, there are only $O(\log_{\sqrt{B}} N) = O(\log_B N)$ levels. Each node now has $\sqrt{B}$ elements in expectation, with the actual number of elements ranging from $\Theta(\sqrt{B}/\log N)$ to $\Theta(\sqrt{B}\log N)$ w.h.p. No matter how big B is relative to $\log N$, this version of the skip list has a search cost of $O(\log_B N)$.

However, now most nodes are mostly empty, so this version wastes space.

Write-optimized skip list

The write-optimized skip list uses the random and variable amount of extra space in each node to store a buffer, similar to a B$^\varepsilon$-tree [47, 76]. By buffering elements within nodes, we can move (or "flush") inserted items down the skip list in batches. This speeds up insertions on average, similar to buffer use in other write-optimized data structures. However, unlike deterministic write-optimized structures, the number of pivots in a node can vary by a factor of as much as $O(\log^2 N)$, which changes the effectiveness of the buffer substantially, and threatens the attainability of optimal high-probability amortized insert bounds.

The main contribution of this chapter lies in the analysis. We show that the write-optimized skip list has an asymptotically optimal search-insert tradeoff [76,392], similar to the B$^\varepsilon$-tree [76, 81, 201], the COLA [46], or the xdict [75]. These search-insert bounds hold both in expectation and with high probability.

The write-optimized skip list has an additional technical complication at the leaves to ensure good range-queries and space consumption. We promote with probability $1/B^{1-\varepsilon}$ at the leaf level and $1/B^\varepsilon$ at all other levels. We delay the promotion of elements from buffers at the leaf level as a simple mechanism to guarantee that leaf nodes remain $\Theta(B)$ full.

Challenges in attaining high-probability bounds. A particularly troubling aspect of this data structure is that the ratio of a node's buffer size to number of children can vary by a factor of $\Theta(\log^2 N)$. For example, the root itself might be one of these outlier nodes, an $O(\log N)$ factor larger than average. In that case, the large number of pivots (and low amortized per-child buffer size) would affect all insertions.

In data structures with depth $O(\log N)$ such local variation would even out, both on average and w.h.p. But, the write-optimized skip list has only $O(\log_B N)$ depth, which is insufficient to overcome unlucky coin tosses. The surprising result is that this buffered skip list meets the desired I/O goals.

To prove high-probability bounds, we need to find, for any workload, sets of $\Omega(\log B^{\varepsilon})$ insert paths whose I/O complexity is uncorrelated. This appears to be challenging for some workloads. For example, in a sequential-insert workload, any insert path lies on the rightmost spine of the data structure. Furthermore, since all insertions pass through the top level of the data structure, a large node at the top of the skip list can affect the I/O performance of a substantial fraction of insertions.

Fortunately, we are operating in external memory: we can assume that the top few levels of the data structure are cached. Traversing cached levels incurs no I/Os. We show that the remaining levels of the tree offer enough disjoint root-to-leaf paths so that we can prove the desired bounds for write optimization. Indeed, even if all insert paths seem to follow the same root-to-leaf path (e.g., the rightmost spine), the insert path changes structure sufficiently frequently that we can find disjoint root-to-leaf paths.

This proof assumes an optimal paging algorithm. However, the performance bounds in this chapter still hold in systems that use LRU, since LRU with constant resource augmentation is constant competitive with the optimal paging algorithm.

For ease of presentation, we first give a proof of high-probability bounds that applies when there are insertions, but no deletions. The proof relies on a coloring argument of the insert paths.

Deletions destroy this first proof: paths that are independent at some point can be moved together by deletions of intervening elements so that they become correlated. We show, via an extremal graph-coloring argument, that there is always a good partitioning of the paths into uncorrelated sets, no matter what the deletion pattern is. This allows us to prove high-probability bounds under any mix of insertions and deletions.

Contributions

This chapter proves the following theorem establishing the performance of write-optimized skip lists.

Theorem 6.1 *Consider an N-element write-optimized skip list running in external memory. Let memory-hierarchy parameters B and M obey the "tall-cache" assumption that $M = \Omega(B^2 \log^4 B)$. Let the block size B be large enough that $\min\{B^{\varepsilon}, B^{1-\varepsilon}\} \geq \log N$.*

A write-optimized skip list that performs insertions, deletions, and queries achieves the following I/O bounds for tunable performance parameter $0 < \varepsilon < 1$:

- *Insertions and deletions take $O\big((\log_{B^{\varepsilon}} N)/(B^{1-\varepsilon})\big)$ amortized I/Os in expectation and w.h.p.*
- *Range queries returning K elements take $O(\log_{B^{\varepsilon}} N + K/B)$ I/Os in expectation and w.h.p. (Point queries are range queries with $K = 1$.)*
- *The structure takes $O(N)$ space, in expectation and w.h.p.*

The write-optimized skip list's guarantees (like those of a regular skip list) are based on an oblivious adversary. The oblivious adversary can issue arbitrary insert and delete operations, but does not have access to the heights of the elements in the structure (i.e., the random tosses).

Map. Section 6.2 explains how to build and use the write-optimized skip list. Section 6.3 proves several structural properties of the write-optimized skip list. Section 6.4 proves performance bounds for point queries and range queries whp. It also proves bounds on insertion and deletion in expectation. Section 6.5 proves amortized insertions bounds w.h.p. and Section 6.6 adjusts the argument to include deletions w.h.p. Section 6.7 concludes with some extensions and implementation issues.

6.2 Structure and operations of a write-optimized skip list

This section explains how to build the write-optimized skip list. It also sets up notation that will be used throughout the rest of the chapter.

Overall structure. The write-optimized skip list has pointer structure similar to that of the B-skip list [39, 161]. It is composed of a sequence of hierarchical ***levels*** $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_h$, where h is the height of the data structure. We will show $h = O(\log_{B^\varepsilon} N)$ w.h.p.

Each level consists of a linked list of ***nodes*** (which will have size $\Theta(B)$ w.h.p.), where each node is partially filled with ***pivots***. Nodes at level 0 are ***leaves***. Each pivot element e on level $\mathcal{L}_{i\geq 1}$ has a pointer to the ***child*** node containing its occurrence on level $\mathcal{L}_{i-1}$. (We will see that sometimes there may temporarily be no node that contains e on level $\mathcal{L}_{i-1}$; in this case, the pointer points to the node that would contain e based on the sort order.) The smallest pivot in a node is called its ***leader***. Each node at level i contains a pointer to the next node at that level (see Figure 6-2).

Write-optimized skip list nodes are similar to nodes in a B$^\varepsilon$-tree [47, 76] in that each node also contains a ***buffer***. All the elements in a node's buffer are greater than or equal to the node's leader and smaller than the leader of the next node on that level. Inserted items are stored in nodes' buffers and are ***flushed*** in batches from parents to children. Thus, all elements move towards $\mathcal{L}_0$, where they remain (until they are deleted).

Randomized balancing. Each element e in the data structure has an integer ***height*** h_e determined by a sequence of biased coin flips. Coin flips are implemented by hashing e, meaning that even if an element is inserted, deleted, and reinserted, h_e does not change. To determine h_e, flip a biased coin until the first tail and set h_e to the length of the run of heads. For the first flip, the probability of heads is $1/B^{1-\varepsilon}$, and on subsequent flips the probability of heads is $1/B^\varepsilon$. We say that an element e has been ***promoted*** to level $i > 0$ if $h_e \geq i$.

The promotion probabilities are set such that each node on levels $\mathcal{L}_{i\geq 1}$ has $\Theta(B^\varepsilon)$ pivot elements in expectation and each node on level $\mathcal{L}_0$ has $\Theta(B^{1-\varepsilon})$ elements in

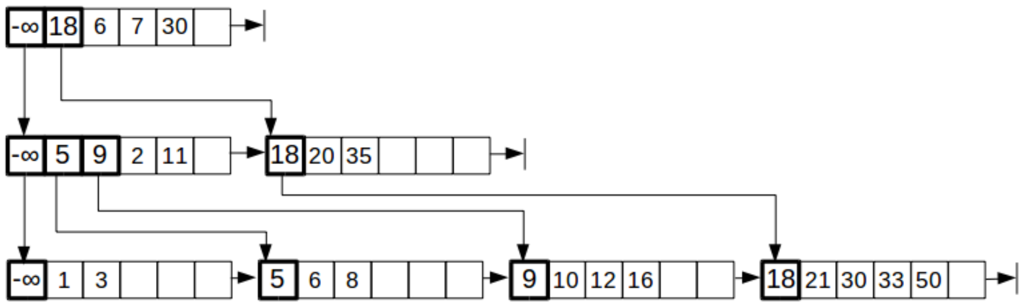


Figure 6-2: Structure of a write-optimized B-skip list with block size $B = 6$. We illustrate the pointer structure of the skip list as well as the pivot and buffer structure of nodes. Each node has size $O(B)$ w.h.p. Any extra space in the nodes is used as buffer space. The number of children at any (internal nonroot) node varies by an $O(\log^2 N)$ factor, meaning that the contribution to the amortized I/O cost for insertions and deletions from that node also varies by an $O(\log^2 N)$ factor. This large variation is an obstacle for designing external-memory skip lists with high-probability performance bounds.

expectation. The variable (random) amount of extra space in each node serves as the buffer space in our insertion algorithm and enables us to achieve amortized high-probability write-optimized update bounds, as discussed in Section 6.5 and Section 6.6.

As with a regular skip list, to ensure that there is a root for the entire structure, there is a special element $-\infty$ that is defined to have the largest height of any element.

Insertions and deletions. When a new element is inserted, store it in the root's buffer. When an element e is deleted, store a ***tombstone*** $\bar{e}$ in the root's buffer.

Buffer-flushing mechanism. When the buffer in node D at level $\mathcal{L}_{i\geq 1}$ becomes full (i.e., it ***overflows***), perform a ***flush*** operation. During a flush, distribute the elements in D's buffer among the buffers of D's children. This may require an I/O per child to bring the children nodes into main memory.

Whenever any one child has $B^{1-\varepsilon}$ delete messages destined for it, flush those delete messages to the appropriate child immediately. (This extra rule for flushing deletes helps us achieve the desired range-query bounds; see Theorem 6.9).

Pivots and leaders. When an element e gets flushed out of the buffer of a node D of height $i \leq h_e$, e becomes a pivot of D in addition to being flushed to the buffer of one of D's children. This new pivot will point to the node to which e is being flushed. This means that D now has two (or more) pivots that point to the same child.

If $i < h_e$, then split D into two nodes D' and D'', making the current leader of D the leader of D' and e the leader of D''. Since D may have multiple pivots pointing to the same child, splitting D may result in some of D's children having more than one parent.

Whenever a node D that has multiple parents is split, update all of D's parents to point to the newly created nodes. Splitting a node D will not change the size of any of D's parents so that, unlike a B-tree, splitting can proceed in a purely top-to-bottom fashion. This is because, whenever a node D is split to create a new node D'' with leader e, element e must already be a pivot in D's parents.

When a delete message $\bar{e}$ is flushed from a node, delete e as a pivot of that node, if it happens to be one. If e is also the leader of that node, then merge that node with its predecessor on that level. Thus, merges are the reverse of splits.

Leaves require special handling. Whenever there is a flush from a parent D at level 1 to all of its leaves, rebalance all the leaves as follows: greedily choose the breaks between leaves so that each leaf approximately fills a block and each leaf begins with a pivot of D (but not every pivot of D begins a leaf).

Queries. To search for element e, traverse the root-to-leaf path to e, and retain all these nodes in memory until the query is done. Our assumptions on the size of memory imply that $M > B \log_{B^\varepsilon} N$, so that a complete root-to-leaf path fits in memory.

The leaf may or may not contain e itself. Insertions and deletions of e may reside in buffers on the root-to-leaf path. Find the messages in the highest buffer that affects e: if it is an insert, then e is present. If it is a delete, then e is absent. The I/O complexity is $O(\log_{B^\varepsilon} N)$ w.h.p.

Each buffer could be checked on the way down, until the first message that affects e is found. But the above method generalizes to richer queries. Consider finding the successor of e. First, find the successor of e in every root-to-leaf buffer and return the min-value of these that is currently in the dictionary. A trivial way to do this is to sort all the messages in all the buffers under consideration by (f, i, t), where f is the key, i is the height, and t is the type (insertion or deletion), then to remove all but the first occurrence of each key. This yields the current state of each key. Finally, search for e's successor by finding the smallest $f > e$ and then scanning to the first insertion.

There is one missing detail. If e is the largest element in its leaf and is larger than everything in the root-to-leaf buffers, then the successor of e will reside in the root-to-leaf path of the successor leaf. This does not increase the I/O complexity of successor, which is $O(\log_{B^\varepsilon} N)$.

A range query is implemented by repeated successor queries. Once the beginning of the range is found, successive leaves contain $\Theta(B)$ values in the range, and the I/Os for fetching internal nodes is dominated by that of fetching leaves. Thus, a K-element range query takes $O(K/B + \log_{B^\varepsilon} N)$ I/Os.

Top-down splits and merges: another advantage of write-optimized skip lists. Splits, merges, and promotions are performed in a *top-to-bottom* fashion. As we describe briefly in Section 6.7, this artifact of using a randomized rebalancing scheme may, in fact, turn out to be another hidden advantage of the write-optimized skip list over other data structures.

In particular, it may make it easier to implement concurrent write-optimized skip lists. There may be advantages both for lock-based implementations as well as lock-free versions. See Section 6.7 for details.

6.3 Structural bounds

This section establishes structural properties of the write-optimized skip list, establishing both expected and high-probability bounds.

We assume throughout that $\min\{B^{\varepsilon}, B^{1-\varepsilon}\} \geq \log N$.

Local structure

Lemma 6.2 (Pivots in an internal node) *An internal node has B^{ε} pivots in expectation and $O(B^{\varepsilon} \log N) = O(B)$ pivots w.h.p.*

Proof. By construction, we begin a new internal node when we see a promotion to the next level. Therefore, the number of pivots in each internal node can be modeled as the number X of tails before the first heads in a sequence of independent coin flips with a head probability of $B^{-\varepsilon}$. The expectation of X is B^{ε}. The high probability bounds follow from the Chernoff bounds. □

The following lemma implies that accessing any node requires $O(1)$ I/Os w.h.p.

Lemma 6.3 (Node size) *For $0 < \varepsilon < 1$, a node is comprised of $O(1)$ blocks w.h.p.*

Proof. For levels greater than 0, nodes contain pivots and $\Theta(B)$ buffer space. By Lemma 6.2, nodes have $O(B)$ pivots w.h.p., so the total size of an internal node is $O(1)$ disk blocks.

By the same argument, even though the promotion probability at the leaves is $1/B^{1-\varepsilon}$, every run of $\Theta(B)$ elements at the leaf level has a promoted element w.h.p. Thus, when packing elements at the leaf level into blocks, we can create a new leaf every $\Theta(B)$ blocks w.h.p. Hence, every node at level 0 consumes $O(1)$ blocks w.h.p.
□

The following lemma bounds the number of neighbors—parents, children, successors and predecessors—of a node. This will help us bound the cost of performing flushes, since flushes may have to access all of a node's neighbors.

Lemma 6.4 (Neighbor bounds) *Let D be a node at height at least 1. The number of parents of D is $O(1)$ w.h.p. The expected number of children of D is $O(B^{\varepsilon})$. If the height of D is exactly 1, then D has $O(\log N) = O(B^{\varepsilon})$ children w.h.p.*

Proof. The bound on children breaks into two cases:

- If D is at level $i > 1$ then, by Lemma 6.2, its expected number of pivots is $O(B^{\varepsilon})$, and therefore so is the expected number of children.

- Nodes at level 1 are split whenever an element is promoted to level 2. Each element in level 0 has a $1/B$ chance of being promoted to level 2. By Chernoff bounds, any run of $\Omega(B \log N)$ elements at level 0 has at least 1 element promoted to level 2 w.h.p. Thus, w.h.p. no node at level 1 has more than $O(B \log N)$ elements in its children. Since each child has $\Theta(B)$ elements, nodes at level 1 have $O(\log N) = O(B^{\varepsilon})$ children w.h.p.

The number of parents of D is at most the number of messages in D's buffer that have height at least 2 larger than the height of D. Since D has height at least 1, the probability that any particular item in D's buffer has height 2 greater than the height of D is at most $1/B^{1+\varepsilon}$. Since D's buffer contains $O(B)$ items, the expected number of such elements in D's buffer is $O(1/B^{\varepsilon})$. Thus, by the Chernoff bounds, the number of such elements is $O((\log N)/B^{\varepsilon})$ w.h.p. Since $\log N < B^{\varepsilon}$, the number of such elements, and hence the number of parents of D, is $O(1)$. $\square$

Global structure

Theorem 6.5 (Linear space) *A write-optimized skip list on N elements uses $O(N/B)$ blocks in expectation and w.h.p.*

PROOF. Each leaf holds $\Theta(B)$ items by construction and from Lemma 6.3 consumes $O(1)$ blocks w.h.p. Thus, the total space consumed by leaves is $O(N/B)$ w.h.p.

The number of blocks at $\mathcal{L}_1$ is also $O(N/B)$ since it is not more than the number of leaves.

For levels 2 and above, the space consumption follows the same analysis as the B-skip list. $\square$

The following two lemmas help us bound the I/O costs of queries and inserts.

Lemma 6.6 (Height upper bound) *For constant $0 < \varepsilon < 1$, the height of the write-optimized skip list is $O(\log_{B^{\varepsilon}} N)$ both in expectation and w.h.p.*

PROOF. The probability that any given element has height at least $h \geq 1$ is $1/B^{1-\varepsilon+(h-1)\varepsilon}$.

Let $c \geq 2$ be a constant. The probability that a given element has height at least $h = 1 + c\log_{B^{\varepsilon}} N$ is at most

$$\frac{1}{B^{1+(h-2)\varepsilon}} \leq \frac{1}{B^{\varepsilon(h-1)}} \leq \frac{1}{B^{\varepsilon c \log_{B^{\varepsilon}} N}}.$$

The probability that any given element has height at least $1 + c\log_{B^{\varepsilon}} N$ is at most $1/N^{c}$. By the union bound, the probability that any of the N elements has height at least $1 + c\log_{B^{\varepsilon}} N$, is at most $1/N^{c-1}$. $\square$

Lemma 6.7 (Pivots on a search path) *The total number of pivots at level 2 or higher touched by any root-to-leaf search path in the data structure is $O\left(B^{\varepsilon} \log N\right)$ w.h.p.*

PROOF. Consider the search path "backwards." That is, start from the element x_i in the leaf level, and consider the unique trajectory from x_i back to the root following pointers backwards. The search path is comprised of some number of horizontal pointers (point to pivots on the same level) and $O(\log_{B^{\varepsilon}} N)$ vertical pointers (from Lemma 6.6).

We can model the length of this search path mathematically as the number of coin flips until $O(\log_{B^{\varepsilon}} N)$ heads have been seen with high probability. At levels 1 and above, the probability of a head is $1/B^{\varepsilon}$. Using Chernoff bounds, one can prove that $O(B^{\varepsilon} \log N)$ coin flips are enough to go back from level 1 to the root w.h.p. $\square$

6.4 Simple runtime bounds

This section gives high-probability bounds on the query performance. It also gives expected bounds on the amortized cost of insertion and deletion.

The amortization in the insertion bound is similar to the analysis of flushes in a B$^\varepsilon$-tree [76]. One interesting difference is that, with B$^\varepsilon$-trees, one must analyze the cost of splitting separately from the cost of flushes, since splitting is a non-local operation. In the write-optimized skip list, on the other hand, splitting and merging is performed locally as part of flushing, so we can bound its cost as part of the analysis of the flushing cost.

Query performance

Next, we show bounds for point queries with constant tunable performance parameter ε.

Theorem 6.8 (Point queries) *A point query has a worst case I/O complexity $O(\log_{B^\varepsilon} N)$ w.h.p.*

Proof. From Lemma 6.7, each search path contains $O(B^\varepsilon \log N)$ elements w.h.p. Furthermore, the height of the tree is $O(\log_{B^\varepsilon} N)$ w.h.p. (from Lemma 6.6). For any search path, we must pay at most a single random I/O each time we descend a level. However, elements of the same level are stored contiguously in blocks (nodes), therefore we can make a linear scan over a level reading $O(B)$ elements per I/O.

Thus, the cost to read all elements in a particular search path is $O(\log_{B^\varepsilon} N + (B^\varepsilon \log N)/B) = O(\log_{B^\varepsilon} N)$ w.h.p. □

Theorem 6.9 (Range queries) *The I/O complexity of range queries is $O(\log_{B^\varepsilon} N + K/B)$ w.h.p. where K is the number of elements in the requested interval.*

Proof. The cost for range queries can be analyzed using the search paths of the left and right ends of the requested interval. The complexity of a range query is bounded by the number of leaf nodes holding the elements in the range plus $O(\log_{B^\varepsilon} N)$ (the cost of a point query w.h.p.). Between the two search paths is a small write-optimized skip list of the K items returned by the range query. By Theorem 6.5, the total space consumed by the nodes in this mini skip list is $O(K/B)$ w.h.p. □

Insert and delete bounds in expectation

Theorem 6.10 (Write-optimized updates) *For $0 < \varepsilon < 1$, the amortized cost of inserting or deleting an element in the data structure is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ in expectation.*

Proof. We first analyze the expected cost of a flush. A flush of a node D must access all the children and parents of D, in addition to writing any new nodes that result from splitting or merging D. By Lemma 6.4, there are $O(B^{\varepsilon})$ parents and children in expectation. If we do a merge, we may have to access D's predecessor and its parents, but this is $O(1)$ additional nodes in expectation. Thus, the total number of nodes accessed during a flush is $O(B^{\varepsilon})$ in expectation. From Lemma 6.3, each node fits in $O(1)$ blocks w.h.p., so the total number of I/Os required by a flush is $O(B^{\varepsilon})$ in expectation.

We now analyze the expected amortized insertion/deletion cost. By Lemma 6.6, each element (or tombstone) must be flushed $O(\log_B N)$ times w.h.p. Thus, the total number of element flushes we must perform during any sequence of N insertions and deletions is $O(N \log_B N)$ with high probability. Each flush performs $\Theta(B)$ element-flushes with high probability. Thus, the total number of flushes performed is $O((N \log_B N)/B)$ with high probability. Since each flush costs $O(B^{\varepsilon})$ I/Os in expectation, the amortized insertion cost is $O((\log_B N)/B^{1-\varepsilon})$ I/Os in expectation.
□

6.5 High probability insertion-only bounds

This section establishes expected and high-probability bounds on the amortized insertion cost for a write optimized skip list that only handles insertions. We prove these bounds for a skip list that also handles deletes in Section 6.6.

Unfortunately, Theorem 6.10 does not obviously generalize to give matching high probability bounds on the amortized insertion cost. This is because, although there are many node flushes, many are not independent, preventing us from applying Chernoff bounds. We may flush a node many times before it gets split or merged with one of its siblings.

To overcome this problem, we partition the elements inserted into the skip list into ***color classes***, where all the elements of the same color follow (mostly) disjoint flushing paths. As a result, all the flushes (and flushing costs) involving these elements are independent. As long as the number of elements in a color class is large enough, we can use Chernoff bounds on the total cost of all the flushes of all the elements in that class.

The main challenge is that we are not guaranteed enough disjoint paths near the root of the skip list.

We use caching to address this problem. Flushes between nodes in cache incur no I/O, and hence can be ignored. As long as enough levels at the top of the skip list are cached, we can find large classes of elements that all follow disjoint paths through the uncached portion of the skip list.

Caching assumptions and structural bounds

Caching assumptions. Our high-probability bounds assume that the top $\Omega(1)$ levels of this data structure (those closest to the root) are permanently pinned in cache.

An optimal cache-replacement policy will outperform these results, but an optimal policy requires prescience, rendering it unimplementable. However, the LRU (least-recently used) cache-replacement strategy is a 2-approximation to optimal, given a cache of twice the size [342], implying that our bounds still hold asymptotically with LRU. We account for the doubled memory in the asymptotics of our tall cache assumption.

For the analysis of the skip list assuming only insertions, we need the cache size to be $\Omega(B^2 \log^2 B)$; for the analysis with insertions and deletions, $\Omega(B^2 \log^4 B)$. Therefore, we generalize the analysis to a cache of size $M = \Omega(B^2 X)$.

Structural properties. We now establish preliminary lemmas about the ***cached region*** (i.e., levels stored in cache).

We first give a lower bound on the number of levels that can be cached and the size of the largest cached level.

Lemma 6.11 (Height of cached region) *Suppose that internal memory has size $M = \Omega(XB^2)$ and let h' be the height of the lowest level with $O(XB^\varepsilon \log N)$ nodes. Then every node in level at least h' fits in memory w.h.p.*

Proof. The number of nodes at level h' is $O(XB^\varepsilon \log N)$. Each node requires $\Theta(B)$ space w.h.p. The amount of space for all nodes at height h' is order the following:

$$BXB^\varepsilon \log N \leq XB^{1+\varepsilon}B^{1-\varepsilon} = XB^2.$$

Thus, the size needed to store nodes of level h' is $O(XB^2)$ w.h.p. By Chernoff bounds, the number of nodes in higher levels decreases exponentially. Once the expected number of elements at a level is at most $\log N$, w.h.p, that level consumes at most one block. So the total number of nodes in all levels at or above h' is $O(XB^\varepsilon \log N + \log_{B^\varepsilon} N)$. Given the tall cache assumption that $M = \Omega(XB^2)$, there is sufficient space to store all levels with height at least h'. □

Lemma 6.12 (Pivots in last level cached) *With a cache of size $\Omega(B^2 X)$, the lowest level that fits in cache has $\omega(X \log N)$ pivots with separate children w.h.p.*

Proof. Let h be the height of the highest level that does not fit into internal memory. (Since $M < N$, h exists.) Then, from Lemma 6.11, the number of nodes at that level is $\omega(XB^\varepsilon \log N)$. This means that the number of pivots at level $h + 1$ that have separate children is $\omega(XB^\varepsilon \log N)$. □

The following theorem will help us argue the existence of independent paths through the disk-resident region.

Theorem 6.13 (Cached element frequency) *If $M = \Omega(XB^2)$, then, in any set of $\Omega(N/X)$ contiguous distinct elements at least one element is promoted into a cached level w.h.p.*

Proof. Let p be the probability that an element has been promoted to the cached region. Using Lemma 6.12, $p \geq (cX \log N)/N$ with high probability for all constants $c > 0$.

Now, let q be the probability that no element is promoted to the cached region in a group of $\Omega(N/X)$ elements.

$$\begin{aligned} q = (1-p)^{\Omega(N/X)} &= \exp\left(\Omega\left(\frac{N}{X}\right)\log(1-p)\right) \\ &\leq \exp\left(-\Omega\left(\frac{pN}{X}\right)\right) \\ &\leq \exp\left(-\Omega\left(\frac{cNX\log N}{NX}\right)\right) \\ &= \frac{1}{\Omega(N^c)}. \end{aligned}$$

Thus, in any such such group of $\Omega(N/X)$ elements, w.h.p. at least one element is promoted to a cached level. □

Element coloring algorithm and analysis

We use the aforementioned bounds on the size of the cached region and the frequency of cached elements to present an element coloring algorithm for insertion analysis.

We describe why normal Chernoff-bound analysis is insufficient and then use a coloring argument on disjoint root-to-leaf paths in a skip list with a large enough cache to establish the amortized cost of insert operations w.h.p.

The obstacle to using Chernoff bounds as above is that insertions that pass through the same node of the skip list will have correlated flushing costs. However, flushes between nodes in cache require no I/Os. Thus, if enough levels at the top of the tree are cached, then many insertions will follow independent paths through the uncached levels of the skip list enabling us to use Chernoff bounds to bound their overall cost.

The ***rank*** of an element is its position in the sorted list of all elements in the data structure regardless of whether or not it has reached the leaves. That is, the ith smallest element in the data structure has rank i, even if it is still making its way through the internal nodes due to the buffer-flushing scheme described earlier.

The insertion paths of two elements a, b are ***independent*** if they are node-disjoint in the part of the skip list that is not cached in memory. The following lemma proves the existence of a coloring of elements into such independence classes.

Lemma 6.14 *There exists a coloring of elements inserted in the data structure such that elements in the same color class experience disjoint root-to-leaf paths w.h.p.*

Proof. The following algorithm colors elements such that after every operation, the difference in rank between any two elements of the same color is at least $N/2X$.

Coloring algorithm:

- Insert the first N/X elements with distinct colors, establishing the set of colors C.

- When a new element e is inserted at rank k, let C_I be the set of colors of the elements at ranks $[k - N/2X, k + N/2X]$. Assign e the color in $C \setminus C_I$ that currently has the fewest elements.

If the difference in rank between two elements a and b (where $a < b$) is $\Omega(N/X)$, then there exists at least one element (greater than a and at most b) between them promoted into the cached region w.h.p. (from Theorem 6.13). This element will be a leader at every level not cached in internal memory. The presence of such an element is enough to isolate the insertion paths of a and b.

Later insertions in the data structure do not affect this property, because the difference in rank between two elements can only increase over time. □

Now we prove, using the above coloring scheme, that the expected amortized cost of insertions holds w.h.p. We use the union bound on the amortized insertion costs for each color class as described in Theorem 6.15.

Theorem 6.15 *The amortized insertion cost for $\Omega(\log B)$ elements with independent insertion paths in the data structure is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ w.h.p.*

PROOF. We show that the amortized cost of flushes is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ w.h.p. Recall from Lemma 6.4 that the amortized cost of flushes from level 1 to level 0 is $O(1/B^{1-\varepsilon})$ I/Os w.h.p. Thus, we need to bound the amortized cost of flushing elements only to levels 1 and above.

As an element moves down the $\log_{B^\varepsilon} N$ levels, there is a flush at each level. Each flush moves $\Theta(B)$ elements down one level. The total number of I/Os for all these flushes is the total number of pivots on the path, since each pivot has a child at the next lower level which could receive elements in a flush. Lemma 6.7 shows that the total number of pivots on levels 1 and above along any root-to-leaf search path of length $\log_{B^\varepsilon} N$ is $O(B^\varepsilon \log N)$ w.h.p., not matching the expected bounds.

For the amortized analysis to hold w.h.p., we need $\Theta(B^\varepsilon \log_{B^\varepsilon} N)$ pivots at levels two and above per path when averaged over all paths. At a high level, when we only consider one path there are not enough trials for us to avoid paying an additional asymptotic cost.

By identifying $\Omega(\log B)$ disjoint paths through the above coloring scheme, we can "concatenate" them. We model the total number of pivots at level 2 and above along this grouped search path mathematically as the number of coin flips needed until $\Omega(\log N)$ heads have been seen with high probability.

By Chernoff bounds, we need $O(B^\varepsilon \log N)$ coin flips in total for $\Omega(\log B)$ heads. Therefore, the amortized number of pivots at level 2 or above per path is $O((B^\varepsilon \log N)/\log B) = O(B^\varepsilon \log_{B^\varepsilon} N)$ w.h.p.

Since we transfer B elements with each I/O, the amortized cost of inserting an element along each of these disjoint paths is $O((B^\varepsilon \log_{B^\varepsilon} N)/B) = O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$. □

Recall the "tall cache" assumption that the size of memory $M = \Omega(XB^2)$ for some memory parameter X. We now show that $X = \Omega(\log^2 B)$ is sufficient to achieve the desired write-optimized bounds in an insert-only data structure.

Theorem 6.16 (Write-optimized insertions) *If the size of memory $M = \Omega(XB^2) = \Omega(B^2 \log^2 B)$ in an insert-only skip list, the insertion cost per element is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ w.h.p.*

Proof. After an insertion sequence, the elements have been divided into color classes. Consider the following two cases for the color classes:

Case 1: A color class has at least $\log B$ elements in it. We can apply the Chernoff bound analysis for concatenated paths from Theorem 6.15 and obtain $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ amortized insert cost per element w.h.p.

Case 2: A color class has fewer than $\log B$ elements in it. If $X = \Omega(\log^2 B)$, there are $O(N/X) = O(N/\log^2 B)$ color classes by construction. Therefore, there are at most $O(N/\log^2 B)$ "bad" color classes for which we cannot apply Chernoff bounds to acquire the same asymptotic bound w.h.p. as in Theorem 6.15. Furthermore, there are strictly fewer than $\log B$ elements in each of these "bad" classes. Therefore, there are at most $O(N/\log B)$ "bad" elements. For a "bad" element we may have to pay the naïve (single path analysis) cost of $(\log N)/B^{1-\varepsilon}$ (w.h.p.). However, the total cost amortized for these elements is at most $O((N \log_{B^\varepsilon} N)/B^{1-\varepsilon})$ w.h.p.

Finally, we calculate the amortized insertion cost per element over N inserts. That is,

$$\left(O(N) \frac{\log_{B^\varepsilon} N}{B^{1-\varepsilon}} + O\left(\frac{N}{\log B}\right) \frac{\log N}{B^{1-\varepsilon}} \right) /N = O\left(\frac{\log_{B^\varepsilon} N}{B^{1-\varepsilon}}\right).$$

□

6.6 High probability bounds with insertions and deletions

Since the previously described coloring scheme in Section 6.5 does not easily extend to allow deletions, this section resolves the deletion issue with an additional coloring argument. The previously described coloring scheme allows us to build groups of $\Omega(\log B)$ independent paths. However, we cannot perform delete operations, because the proofs do not allow the difference in rank between elements to decrease.

As the coloring algorithm is only a theoretical tool, we can assume that the adversary has no knowledge of the chosen colors. Equivalently, we can assume that we know all the requests from the beginning creating an "offline" coloring problem. We show that this allows us to extend the w.h.p. update bounds to include deletes.

We begin by describing a modified coloring scheme based on building a conflict graph of keys less than X apart in rank. Next, we show that this analysis technique allows us to prove the desired write-optimized bounds for both insertions and deletions.

Specifically, we describe a scheme to color $\Theta(N)$ updates on a skip list of size N. We introduce an undirected graph $G = (V, E)$, with the set of vertices V being the keys in the skip list. An edge $\{u, v\}$ is added to E if and only if at some point the difference in rank between keys u and v is smaller than N/X.

- When inserting an element at rank k, we add at most $2N/X$ edges to the graph, binding all the keys for which the difference in rank with k is smaller than N/X not to be of the same color.
- When removing an element at rank k, we add at most N/X edges between the key of rank $k + i - N/X - 1$ and key of rank $k + i$ for $0 \le i \le N/X$.

The total number of edges is therefore $O(N^2/X)$. We will use this information with Lemma 6.17.

Recall from Theorem 6.13 that if $M = \Omega(B^2X)$, then at least one element is promoted into the cached region in a block of N/X elements w.h.p.

Therefore, two elements with the same color will have at least one "splitting element" between them that causes them to have disjoint paths outside of the cached region.

At a high level, assume that we have a write-optimized skip list with some N active elements. In the very beginning (startup stage), allow the data structure to fill to some chosen constant size $C_{\min}$— all operations in this stage therefore have constant cost. We set this as our first N in the following analysis.

We build a conflict graph for some sequence of N' operations such that even if all N' operations are deletes, we still have $\Theta(N)$ active elements in the data structure for our amortized analysis. Thus, N' is some constant fraction of N, e.g., $N/4$. We also stop the sequence if the size of the data structure falls below $C_{\min}$.

Coloring the conflict graph. The analysis of the update cost based on the conflict graph is done in stages, or **epochs**, based on the length of the sequence of operations. At the beginning of a sequence we have N active elements. After N' operations, we have some N_0 active elements such that $N_0 = \Theta(N)$. At the end of this epoch, we set N_0 as the new N and repeat.

Lemma 6.17 *Given an undirected graph $G = (V, E)$, let $\chi(G)$ be the smallest number of colors needed to color the vertices (chromatic number). Then $\chi(G) = O(\sqrt{|E|})$.*

Proof. Assume that we have fewer than $\binom{\chi(G)}{2}$ edges. There must be two colors that can be merged together, contradicting the fact that $\chi(G)$ is optimal. Therefore, $\chi(G) = O(\sqrt{|E|})$. □

We can therefore color the keys using $O(N/\sqrt{X})$ colors.

Analysis using color classes. The analysis is similar to the proof of Theorem 6.16. We use the grouping technique again with the coloring of the conflict graph and bound the number of color classes with fewer than $\log B$ elements.

If a color class has at least $\log B$ elements, we can do the analysis using Chernoff bounds within this color.

We can bound the number of elements for which the "grouped" analysis is not feasible by $O((N \log B)/\sqrt{X})$. For those elements, we can apply Chernoff bounds naïvely for a total cost of $O\left(\frac{N \log B}{\sqrt{X}} \cdot \frac{\log N}{B^{1-\varepsilon}}\right)$.

Since $X = \Omega(\log^4 B)$, the amortized complexity for the "bad" elements is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$.

At the end of each epoch, start a new conflict graph from scratch. That is, start with a vertex for each of the N'' keys currently in the data structure and add edges between each pair of keys whose ranks differ by at most N/X. Now repeat the analysis process with $N = N''$. We thus obtain the following:

Theorem 6.18 (Write-optimized inserts/deletes) *If the size of memory $M = \Omega(B^2 \log^4 B)$ in a skip list with inserts and deletes, then with high probability, the amortized insertion and deletion cost per element is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ w.h.p.*

6.7 Conclusion

The write-optimized skip list achieves the asymptotically optimal I/O bounds of the best write-optimized data structure while retaining the elegance and simplicity of skip lists. The high probability bounds are established via extremal graph-coloring arguments based on the elements' root-to-leaf paths through the data structure.

We are hopeful that the skip list's randomized rebalancing will have practical (as well as theoretical) impact in the burgeoning area of write-optimization (e.g., as a basis for full-featured production data structures). We briefly try to articulate why we are so hopeful.

Regular skip lists have formed the basis for concurrent and lock-free production dictionaries. Part of the reason why is that they have simpler implementations because they are built out of regular linked lists.

We believe that write-optimized skip lists will benefit from these advantages as well. Consider, for example, the node-splitting mechanisms in write-optimized skip lists versus a B-tree or B$^\varepsilon$-tree. In a write-optimized skip list, node splits and merges are triggered "on the way down," i.e., as insert and delete messages make their way deeper into the structure.

In contrast, in B-trees and B$^\varepsilon$-trees, splits are triggered "on the way up," once inserts and deletes have reached the leaves. In a concurrent data structure based on locks, it is important to grab locks according to a prespecified partial order in order to avoid deadlock. Naïve "hand-over-hand" locking (grab and release locks as you walk down the tree to avoid throttling concurrency) is insufficient to design concurrent B-trees or B$^\varepsilon$-trees. If an insert reaches a leaf and triggers some splits higher up in the tree, the data structure no longer has the necessary locks higher up in the tree. Industrial B-trees and B$^\varepsilon$-trees generally deal with this concurrency issue by implementing delayed splitting mechanisms.

For example, when we built TokuDB, we built a mechanism for delaying splits, letting node sizes grow, and letting future inserts or deletes take care of the splits.

Clearly, this locking issue is solvable, but the coding seems simpler in a write-optimized skip list.

Similarly, skip lists have been the data structure of choice for theoretically good, in-production lock-free dictionaries. This is not only because they are built out of separate linked lists, but also because of other structural properties (such as level pointers along the entire level). Perhaps write-optimized skip lists will become the easiest-to-implement lock-free write-optimized data structures.

Future research includes an implementation study to explore whether the theoretical advantages revealed in this chapter can lead to benefits for implementers and users.

Locality-first strategy. The write-optimized skip list leverages the locality-first strategy to exploit spatial locality in data structure design as a first step towards an efficient parallel data structure. Specifically, the skip list in this chapter achieves write-optimized bounds. That is, for any $0 < \varepsilon < 1$, range queries take $O(\log_{B^\varepsilon} N + K/B)$ I/Os w.h.p. and insertions and deletions take $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ amortized I/Os w.h.p. The write-optimized skip list asymptotically matches the search performance of the classical B-tree while supporting faster insertions and deletions. It achieves write-optimized update bounds by taking advantage of memory blocks to buffer elements. Furthermore, for reasons explained earlier in this section, the write-optimized skip list may be a good candidate for parallelization because it is a pointer-based data structure.

Chapter 7

Beyond Worst-Case Analysis of Multicore Caching Strategies

This chapter mathematically grounds the locality-first strategy for the multicore caching problem by showing that the Least-Recently-Used (LRU) algorithm is better than all other online algorithms on inputs with locality of reference. As we shall see in Chapter 8, a large class of "lazy" algorithms are equivalently arbitrarily far from optimal under worst-case analysis. To go beyond worst-case analysis and compare online algorithms directly, this chapter introduces cyclic analysis, a new measure for beyond worst-case analysis for online algorithms. The cyclic analysis artifact takes the first step beyond-worst-case analysis for multicore cache replacement and validates the locality-first strategy by theoretically supporting the established superiority of the Least-Recently-Used algorithm in practice [9].

This work was conducted in collaboration with Shahin Kamali [208, 210].

Abstract

The divergence between multicore and single-core caching under worst-case analysis motivates a more in-depth comparison of multicore caching algorithms via alternative analysis measures. For example, Chapter 8 showed that the competitive ratio of a large class of online algorithms, including Least-Recently-Used (LRU), grows with the length of the input. Furthermore, even offline algorithms like Furthest-In-Future, the optimal algorithm in single-core caching, cannot compete in the multicore setting. These negative results arise from the increased power of the adversary to adapt to multicore caching algorithms. Therefore, these results suggest the need for a direct comparison of online algorithms to each other.

This chapter introduces cyclic analysis, a generalization of bijective analysis introduced by Angelopoulos *et al.* [13]. Cyclic analysis captures the advantages of bijective analysis while offering flexibility that makes it more useful for comparing algorithms for a variety online problems. In particular, this chapter takes the first steps beyond worst-case analysis for analysis of multicore caching algorithms. This chapter uses cyclic analysis to establish relationships between multicore caching algorithms, including the advantage of LRU over all other multicore caching algorithms in the presence of locality of reference.

7.1 Introduction

Despite the widespread use of multiple cores in a single machine, the theoretical performance of even the most common cache eviction algorithms is not yet fully understood when multiple cores simultaneously share a cache. Caching algorithms for multicore architectures have been well-studied in practice, including dynamic cache-partitioning heuristics [312, 350, 354] and operating system cache management [143, 311, 386]. There are very few theoretical guarantees, however, for performance of these algorithms. Furthermore, most existing guarantees on online multicore caching algorithms are negative [209, 251], but resource augmentation may be helpful in some cases [4, 5].

This chapter explores the ***multicore caching***[1] ***problem*** in which multiple cores share a cache and request pages in an online manner. Upon serving a request, the requested page should become available in the shared cache. If the page is already in the cache, a hit takes place; otherwise, when the page is not in the cache, the core that issues the request incurs a miss. In case of a miss, the requested page should be fetched to the cache from a slow memory. Fetching a page causes a ***fetch delay*** in serving the subsequent requests made by the core that incurs the miss. Such delay is captured by the free-interleaving model of multicore caching [216, 251]. Under this model, when a core incurs a miss, it spends multiple cycles fetching the page from the slow memory while other cores may continue serving their requests in the meantime. Therefore, an algorithm's eviction strategy not only defines the state of the cache and the number of misses, but also the order in which requests are served. That is, a caching algorithm implicitly defines a "schedule" of requests served at each timestep through its previous eviction decisions.

Divergence between multicore and single-core caching. Previous work [209, 251] leveraged the scheduling aspect of multicore caching to demonstrate that guarantees on competitive ratio[2] of algorithms in the single-core setting do not extend to multicore caching. In particular, López-Ortiz and Salinger [251] focused on two classical single-core caching algorithms, LEAST-RECENTLY-USED (LRU) [342] and FURTHEST-IN-FUTURE (FIF) [37], and showed these algorithms are unboundedly worse than the optimal algorithm OPT in the free-interleaving model[3]. In the free-interleaving model, FIF evicts the page furthest in the future in terms of the number of requests. In the single-core setting, LRU is k-competitive (where k is the size of the cache) [342], and FIF is the optimal algorithm [37]. Chapter 8 further confirms the intuition that multicore caching is much harder than single-core caching and showed that all lazy algorithms are equivalently non-competitive against OPT. An online caching algorithm is ***lazy*** [257] if it 1) evicts a page only if there is a miss 2) evicts no

[1]This problem is also called "paging" in the literature [251]. We use "multicore caching" because it more accurately reflects the problem studied in this thesis.

[2]For a cost-minimization problem, an online algorithm has a competitive ratio of c if its cost on any input never exceeds c times the cost of an optimal offline algorithm for the same input (up to an additive constant).

[3]Both LRU and FIF evict pages only when the cache is full and there is a request to a page not in the cache. In the multicore setting, ties can happen; both LRU and FIF break ties arbitrarily.

more pages than the misses at each timestep, 3) in any given timestep, does not evict a page that incurred a hit in that timestep, and 4) evicts a page only if there is no space left in the cache[4]. Lazy algorithms capture natural and practical properties of online algorithms. Common caching strategies such as LRU and First-In-First-Out (FIFO) are clearly lazy. Unfortunately, the competitive ratio of this huge class of algorithms is bounded and grows with the length of the input.

At a high level, the divergence between performance of algorithms for multicore and single-core caching stems from the power of the adversary to adapt to online algorithms and to generate inputs that are particularly tailored to harm the "schedule" of online algorithms. For these adversarial inputs, the implicit "scheduling" of lazy algorithms causes periods of "high demand" in which the cache of the algorithm is congested (cores request many different pages). Meanwhile, an optimal offline algorithm avoids these high-demand periods by delaying cores in an "artificial" way. These adversarial inputs highlight the inherent pessimistic nature of competitive analysis.

Beyond worst-case analysis. The highly-structured nature of the worst-case inputs suggests that competitive analysis might not be suitable for studying multicore caching algorithms and motivates the study of alternatives to competitive ratio. There are two main reasons to go beyond competitive analysis for analysis of multicore caching algorithms. First, competitive analysis is overly pessimistic and measures performance on worst-case sequences that are unlikely to happen in practice. In contrast, measures of typical performance are more holistic than worst-case analysis, which dismisses all other sequences. Second, competitive analysis does not help to separate online algorithms for multicore caching because no practical algorithm can compete with an optimal offline algorithm [209]. Therefore, other measures are required to establish the advantage of one online algorithm over others. Many alternative measures have been proposed for single-core caching [38, 71, 73, 214, 224, 395–398]. For a survey of measures of online algorithms, the reader may consult [72, 136, 222].

In particular, bijective analysis [14, 16, 17] is a natural measure that directly compares online algorithms and has been used to capture the advantage of LRU over other online single-core caching algorithms on inputs with "locality of reference" [14, 16]. Despite these results, as we shall see, bijective analysis has restrictions when it comes to multicore caching.

Contributions

This chapter takes the first steps beyond competitive analysis for multicore caching by extending bijective analysis to a stronger measure named cyclic analysis and demonstrating how to apply cyclic analysis to analyze multicore caching algorithms. The pessimistic nature of competitive analysis demonstrates the need for alternative measures of online algorithms.

Cyclic analysis. This chapter introduces cyclic analysis, a measure that captures the benefits of bijective analysis and offers additional flexibility. Cyclic analysis gen-

[4]Lazy algorithms are often called "demand paging" in the systems literature [318]. Algorithms with properties 1-3 (but not necessarily 4) are called "honest" algorithms [251].

eralizes bijective analysis by directly comparing two online algorithms over *all inputs*. Traditional bijective analysis compares algorithms by partitioning the universe of inputs based on input length and drawing bijections between inputs in the same partition [13, 14, 16, 135]. Cyclic analysis relaxes this requirement by allowing bijections between inputs of different lengths. This flexibility allows for alternative proof methods for showing relationships between algorithms.

This chapter shows that all lazy [251, 257] algorithms are equivalent under cyclic analysis, but the the strict advantage of any lazy algorithm over Flush-When-Full (FWF) under cyclic analysis (FWF evicts all pages upon a miss on a full cache). In the single-core setting, the advantage of lazy algorithms over FWF is strict and trivial: for any sequence, the cost of LRU is no more than FWF. In the multicore setting, however, such separation requires careful design and mapping with a bijection on the entire universe of inputs (Theorem 7.8) under cyclic analysis.

Separation of LRU via cyclic analysis. The main contribution is to show the strict advantage of a variant of LRU over all other lazy algorithms under cyclic analysis combined with a measure of locality (Theorem 7.17). Although LRU is equivalent to all other lazy algorithms without restriction on the inputs under cyclic analysis, it performs strictly better in practice [340]. This is due to the locality of reference that is present in real-world inputs [8, 103, 122]. In order to capture the advantage of LRU, this chapter applies cyclic analysis on a universe that is restricted to inputs with locality of reference [8] and show that LRU is strictly better than any other lazy algorithm.

Map. The remainder of the chapter is organized as follows. Section 7.2 concretizes the "scheduling" aspect of multicore caching and defines the cost model used in this chapter. Section 7.3 introduces cyclic analysis and establishes some useful properties of this measure. Section 7.4 applies cyclic analysis to establish the advantage of lazy algorithms over non-lazy FWF. Section 7.5 shows the advantage of LRU over all other lazy algorithms under cyclic analysis on inputs with locality of reference. Section 7.6 reviews related models of multicore caching, and Section 7.7 includes a few concluding remarks.

7.2 Preliminaries

This section presents necessary preliminaries to understand the later contributions in this chapter. First, it reviews the free-interleaving model [216, 251] of multicore caching and the cost model used in this chapter. The free-interleaving model is inspired by real-world architectures and captures the essential aspects of the multicore caching problem. Next, it concretizes the difficulty in multicore caching due to "scheduling" and defines the cost model used in this chapter to measure algorithm cost.

Problem definition

Assume we are given a multicore processor with p cores labeled $P_1, P_2, \ldots, P_p$ and a shared cache with k pages ($k \gg p$).

Input description. An input to the ***multicore paging problem*** is formed by p online request sequences $\mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_p)$. Each core P_i must serve its corresponding ***request sequence*** $\mathcal{R}_i = \langle \sigma_{i,1}, \ldots, \sigma_{i,n_i} \rangle$ made up of n_i page requests. For all i, we assume $n_i \gg k, \tau$. The total number of page requests is therefore $n = \sum_{1 \le i \le p} n_i$.

We assume that for all values of i, the length of the request sequence n_i is arbitrarily larger than k. That is, we assume that $k \in \Theta(1)$, which is consistent with the common assumption that machine parameters like k are constant compared to the size of the input.

All requests $\sigma_{i,j}$ are drawn from a finite universe of possible pages U. Throughout this chapter and Chapter 7, we assume that request sequences for different cores may share requests to the same page. In practice, cores may share their requests because of races, or concurrent reads and writes to the same memory location.

Serving requests. Page requests arrive at discrete timesteps. The requests issued by each core should be served in the same order that they appear and in an online manner. More precisely, for all $i, j \ge 1$, core P_i must serve request $\sigma_{i,j}$ before $\sigma_{i,j+1}$, and $\sigma_{i,j+1}$ is not revealed before $\sigma_{i,j}$ is served. The multicore processor may serve at most p page requests in parallel (up to one request per core[5]). Each page request must be served as soon as it arrives. To serve a request to some page $\sigma_{i,j}$ in sequence $\mathcal{R}_i$, core P_i either has a ***hit***, when $\sigma_{i,j}$ is already in the cache, or incurs a ***fault*** when $\sigma_{i,j}$ is not present in the cache. In case of a fault, the requested page should be fetched into the cache. The multicore caching problem is also parametrized by a ***fetch delay*** τ, or the (integer) number of timesteps it takes to fetch a page into the cache. During these timesteps, P_i cannot see any of its forthcoming requests, that is, $\sigma_{i,j+1}$ is not revealed to P_i before $\sigma_{i,j}$ is ***fully fetched***. In case some other core $P^* \neq P_i$ is already fetching the page when the fault occurs, P_i waits for less than τ timesteps until the page is fully fetched to the cache. Given an algorithm $\mathcal{A}$ and an input $\mathcal{R}$, the ***cost*** $\mathcal{A}(\mathcal{R})$ is the number of faults algorithm $\mathcal{A}$ incurs while serving input $\mathcal{R}$.

A multicore caching algorithm $\mathcal{A}$ reads requests from request sequences in parallel and is defined by its eviction decisions at each timestep. If a core faults while the cache is full, $\mathcal{A}$ must evict a page to make space for the requested page before fetching it. We continue the convention [176, 251] that when a page is evicted, the cache cell that previously held the evicted page is unused until the replacement page is fetched. Finally, the processor serves requests from different request sequences in the same timestep in some fixed order (e.g., by core index).

Timestep (t)	Cache before t	$\mathcal{R}_1$, $\mathcal{R}_2$	Status	Schedule $S_{\mathcal{R},\text{LRU}}[t]$
0	$\perp\perp\perp\perp$	$\underline{a_1}a_2a_1a_5$	P_1 misses, starts fetching a_1	(a_1, a_3)
		$\underline{a_3}a_4a_5a_2$	P_2 misses, starts fetching a_3	
1	$\perp\perp\perp\perp$	$\underline{a_1}a_2a_1a_5$	P_1 is fetching a_1	(a_1, a_3)
		$\underline{a_3}a_4a_5a_2$	P_2 is fetching a_3	
2	$\perp\perp\perp\perp$	$\underline{a_1}a_2a_1a_5$	P_1 completes fetching a_1	(a_1, a_3)
		$\underline{a_3}a_4a_5a_2$	P_2 completes fetching a_3	
3	$a_1a_3\perp\perp$	$a_1\underline{a_2}a_1a_5$	P_1 misses, starts fetching a_2	(a_2, a_4)
		$a_3\underline{a_4}a_5a_2$	P_2 misses, starts fetching a_4	
4	$a_1a_3\perp\perp$	$a_1\underline{a_2}a_1a_5$	P_1 is fetching a_2	(a_2, a_4)
		$a_3\underline{a_4}a_5a_2$	P_2 is fetching a_4	
5	$a_1a_3\perp\perp$	$a_1\underline{a_2}a_1a_5$	P_1 completes fetching a_2	(a_2, a_4)
		$a_3\underline{a_4}a_5a_2$	P_2 completes fetching a_4	
6	$a_1a_3a_2a_4$	$a_1a_2\underline{a_1}a_5$	P_1 has a hit for a_1	(a_1, a_5)
		$a_3a_4\underline{a_5}a_2$	P_2 misses, starts fetching a_5	
			(a_3 is the least-recently-used page and evicted)	
7	$a_1\perp a_2a_4$	$a_1a_2a_1\underline{a_5}$	P_1 misses, waits for a_5	(a_5, a_5)
		$a_3a_4\underline{a_5}a_2$	P_2 is fetching a_5	
8	$a_1\perp a_2a_4$	$a_1a_2a_1\underline{a_5}$	P_1 completes serving a_5	(a_5, a_5)
		$a_3a_4\underline{a_5}a_2$	P_2 completes fetching (and serving) a_5	
9	$a_1a_5a_2a_4$	$a_1a_2a_1a_5$	P_1 has completed $\mathcal{R}_1$	$(\perp, a_2)$
		$a_3a_4a_5\underline{a_2}$	P_2 has a hit for a_2, completes $\mathcal{R}_2$	

Figure 7-1: Example of execution of LRU on the input $\mathcal{R} = (\mathcal{R}_1, \mathcal{R}_2)$, with $\mathcal{R}_1 = \langle a_1a_2a_1a_5\rangle$ and $\mathcal{R}_2 = \langle a_3a_4a_5a_2\rangle$. The cache size is $k = 4$ and the fetch delay is $\tau = 3$. We use $\perp$ in the cache to denote an empty slot or slot reserved for a page currently being fetched.
If a request incurs a miss, we repeat it in the schedule at most τ times (or however long it takes to be fetched, if some other processor already requested it but it has not yet been fetched). For example, in timestep 7, we wait two timesteps for a_5 to be fetched for P_1 because there were two more steps until a_5 was brought to the cache by P_2.
In the "Cache before t" column, we keep track of the state of the cache before each timestep. The rightmost column is the schedule generated by LRU serving $\mathcal{R}$.
The schedules for the two cores P_1 and P_2 are defined respectively with $\langle a_1, a_1, a_1, a_2, a_2, a_2, a_1, a_5, a_5, \perp\rangle$ and $\langle a_3, a_3, a_3, a_4, a_4, a_4, a_5, a_5, a_5, a_2\rangle$.

Scheduling in multicore caching

Multicore caching differs from single-core caching because of the scheduling component as a result of the fetch delay. The fetch delay slows down cores at different rates depending on the misses they experience, and requests with the same index on different cores may be served at different times depending on previous evictions. In other words, the eviction strategy implicitly defines a ***schedule***, or an ordering in which the requested pages are served by an algorithm. Given an input $\mathcal{R}$ defined by p sequences, the schedule of a caching algorithm can be represented with a copy of $\mathcal{R}$ in which some requests are repeated. These extra requests capture the timestep at which the processor serves requests from that input sequence using the caching algorithm. That is, a schedule has all the same requests as the corresponding input, but repeats page requests upon a miss until the page has been fully fetched.

[5]In practice, a single instruction of a core may involve more than one page, but we assume that each request is to one page in order to model RISC architectures with separate data and instruction caches [251].

Figure 7-1 contains an example of serving an input with LEAST-RECENTLY-USED (LRU) [176, 251, 342] under free interleaving. The schedule produced by LRU in the example input in Figure 7-1 is the underlined request at each timestep (a formal definition of a schedule can be found in Section 7.5).

Cost model. We use the ***total time*** to measure algorithm performance and denote the cost that an algorithm $\mathcal{A}$ incurs on input $\mathcal{R}$ with $\mathcal{A}(\mathcal{R})$. The non-competitiveness results from prior work in terms of the number of misses also hold under the total time [209, 251].

Definition 7.1 (Total time) *The total time an algorithm $\mathcal{A}$ takes to serve an input $\mathcal{R}$ is the sum of the timesteps it takes for all cores to serve their respective request sequences. That is, the total time $\mathcal{A}(\mathcal{R}) = \sum_{1 \leq i \leq p} \mathcal{A}(\mathcal{R}_i)$ where $\mathcal{A}(\mathcal{R}_i)$ denotes the timesteps P_i took to serve $\mathcal{R}_i$ with algorithm $\mathcal{A}$.*

Total time combines aspects from both makespan and the number of misses, the two cost measures in previous studies of multicore caching [216, 251]. The makespan is the maximum time it takes any core to complete its request sequence, and hence is bounded above by total time. Specifically, the total time is monotonically increasing with respect to both the number of misses and the makespan.

The total time is a more realistic measure of performance than the number of misses because it determines performance in terms of the time that it takes to serve the input. In contrast, the number of misses does not directly correspond with the time to serve an input because a miss may take less than τ steps to fetch the page if it is already in the process of being fetched by another core. The total time also captures aspects of algorithm performance that are not addressed by makespan. In particular, makespan does not capture the overall performance of all cores. For example, a solution in which all cores complete at timestep t has a better makespan than a solution in which one core completes at timestep $t + 1$ while the rest complete much earlier, e.g. at timestep $t/2$. The second solution is preferred in practice (and also under the total time) as most cores are freed up earlier.

7.3 Cyclic analysis for online problems

This section defines a new analysis measure called ***cyclic analysis*** inspired by bijective analysis [13–16, 135] and explores alternative paths to showing relationships between algorithms under Cyclic analysis extends the advantages of bijective analysis to online problems with multiple input sequences.

Overview. Although traditional bijective analysis has been applied to compare single-core caching algorithms, it requires modification to capture the notion of "input length" in multicore caching. Since each request sequence in an input for multicore caching may have a different length in terms of the number of requests, there are multiple ways to define the length of an input. It is not clear which definition of length is most natural or correct for multicore caching.

Furthermore, partitioning the input space based on the number of requests in an input as in bijective analysis for single-core paging may be overly restrictive for multicore caching, because the time it takes to serve inputs of the same length (in terms of the number of requests) may differ depending on the algorithm. In multicore caching, the time depends on the interleaving of the multiple request sequences. Cyclic analysis addresses these issues by removing the restriction that bijections should be drawn between inputs of the same length.

At a high level, in order to show a relationship between two algorithms $\mathcal{A}$ and $\mathcal{B}$ under bijective analysis or cyclic analysis, one must define a mapping between inputs and their costs under different algorithms. One way to model mappings between inputs with different costs is with a ***input-cost graph***. Given algorithms $\mathcal{A}$ and $\mathcal{B}$, an input-cost graph is an infinite directed graph where the nodes represent inputs and there exists an edge from input $\mathcal{R}_1$ to input $\mathcal{R}_2$ if and only if $\mathcal{A}(\mathcal{R}_1) \leq \mathcal{B}(\mathcal{R}_2)$. In order to show the advantage of algorithm $\mathcal{A}$ over $\mathcal{B}$, traditional bijective analysis partitions the (infinite) graph of inputs into finite subgraphs, each formed by inputs of the same length. Within each partition, the bijection relating $\mathcal{A}$ to $\mathcal{B}$ defines a set of cycles such that each vertex is in exactly one cycle of finite length (cycles may have length one, i.e. they may be self-loops). Cyclic analysis relaxes the requirement that all subgraphs in the partition must be finite, but also requires that each node in each induced subgraph must have an in-degree and out-degree of one. That is, each node in the induced subgraph is part of a cycle.

Measure definition and discussion. Let $\mathcal{I}$ denote the (infinite) set of all inputs, and for an algorithm $\mathcal{A}$ and input $\mathcal{R} \in \mathcal{I}$, let $\mathcal{A}(\mathcal{R})$ denote the cost $\mathcal{A}$ incurs while serving $\mathcal{R}$. The notation in our discussions of cyclic analysis is inspired by [13].

Definition 7.2 (Cyclic analysis) *We say that an online algorithm $\mathcal{A}$ is **no worse** than online algorithm $\mathcal{B}$ under **cyclic analysis** if there exists a bijection $\pi : \mathcal{I} \leftrightarrow \mathcal{I}$ satisfying $\mathcal{A}(\mathcal{R}) \leq \mathcal{B}(\pi(\mathcal{R}))$ for each $\mathcal{R} \in \mathcal{I}$. We denote this by $\mathcal{A} \preceq_c \mathcal{B}$. Otherwise we denote the situation by $\mathcal{A} \not\preceq_c \mathcal{B}$. Similarly, we say that $\mathcal{A}$ and $\mathcal{B}$ are the same according to cyclic analysis if $\mathcal{A} \preceq_c \mathcal{B}$ and $\mathcal{B} \preceq_c \mathcal{A}$. This is denoted by $\mathcal{A} \equiv_c \mathcal{B}$. Finally we say $\mathcal{A}$ is better than $\mathcal{B}$ according to cyclic analysis if $\mathcal{A} \preceq_c \mathcal{B}$ and $\mathcal{B} \not\preceq_c \mathcal{A}$. We denote this by $\mathcal{A} \prec_c \mathcal{B}$.*

Bijective analysis is defined similarly, except that the input universe is partitioned based on the length of inputs, and bijections need to be drawn between inputs inside each partition. In contrast, cyclic analysis allows mapping arbitrary sequences to each other. Bijective analysis and cyclic analysis have several benefits over competitive analysis [16]. Specifically, they:

- *capture overall performance.* If $\mathcal{A} \preceq_c \mathcal{B}$, every "bad" input for algorithm $\mathcal{A}$ corresponds to another input for algorithm $\mathcal{B}$ which is at least as bad. Hence, the performance of algorithms is evaluated over all request sequences rather than a single worst-case sequence.
- *avoid comparing to an offline algorithm.* Competitive analysis is inherently pessimistic as it compares online algorithms based on their worst-case performance

against a powerful adversary. This pessimism is especially pronounced in multicore caching where an offline algorithm can "artificially" miss on some pages in order to schedule sequences in a way to minimize its total cost. This scheduling power is a great advantage for OPT as shown in [251]. Instead, we use cyclic analysis because it compares online algorithms directly without involving an offline algorithm.

- *can incorporate assumptions about the universe of inputs.* Bijective and cyclic analysis can also define relationships between algorithm performance on a subset $S \subset \mathcal{I}$ of inputs. For example, applying bijective analysis to a restricted universe of inputs with locality of reference has been used to separate LRU from other algorithms in the single-core setting [13, 16]. Since LRU exploits locality of reference, analyzing inputs with locality may yield a better understanding of the performance of algorithms. Most other measures such as competitive ratio are unable to separate LRU from other lazy algorithms [13].

As mentioned above, bijective analysis, as defined for single-core caching [13, 16], requires partitioning the universe of inputs $\mathcal{I}$ into finite sets of inputs of the same length. For multicore caching, however, this partitioning is not necessary nor well-defined. In fact, for many online problems, the length of input is not necessarily a measure of "difficulty", as trivial request (e.g., repeating requests to a page) can artificially increase the length. As such, there is no fundamental reason to draw bijections between sequences of the same length.

For problems such as single-core caching and list update [16], where the input is formed by a single sequence, the length of the input is simply the length of the sequence. In multicore caching, however, the length of inputs is not well-defined as multiple sequences are involved. Should the length be the sum of the number of requests or a vector of lengths for each request sequence? To address these issues, cyclic analysis generalizes the finite partitions of bijective analysis to the entire universe of inputs. This would give cyclic analysis a flexibility that makes it possible to study other problems under this measure. We note that, the restrictive nature of bijective analysis not only makes it hard to study algorithms under this measure, but also can cause situations that many algorithms are not comparable at all. The following example illustrates the restriction of bijective analysis when compared to cyclic analysis:

Example. Consider two algorithms $\mathcal{A}$ and $\mathcal{B}$ for an online problem P (with a single sequence as its input). Assume the costs of $\mathcal{A}$ and $\mathcal{B}$ are the same over all inputs, except for four sequences. Among these four, suppose that two sequences σ_1 and σ_2 have the same length m and we have $\mathcal{A}(\sigma_1) = 10$ and $\mathcal{A}(\sigma_2) = 40$ while $\mathcal{B}(\sigma_1) = 20$ and $\mathcal{B}(\sigma_2) = 30$. For inputs of length m, there is no way to define a bijection that shows advantage of one algorithm over another. So, the two algorithms are incomparable under bijective analysis. Next, assume for sequences σ_3 and σ_4 we have $\mathcal{A}(\sigma_3) = 20, \mathcal{A}(\sigma_4) = 30, \mathcal{B}(\sigma_3) = 40$, and $\mathcal{B}(\sigma_4) = 20$. The following mappings shows $\mathcal{A} \prec_c \mathcal{B}$: $\sigma_1 \to \sigma_1, \sigma_2 \to \sigma_3, \sigma_3 \to \sigma_4$, and $\sigma_4 \to \sigma_2$.

Bounding inputs with the same cost. In order for cyclic analysis to be a meaningful measure, there must not be an infinite number of inputs that achieve the same cost. To be more precise, for the universe of inputs $\mathcal{I}$ and an algorithm $\mathcal{A}$, let $\mathcal{A}(\mathcal{I})$ be the corresponding multiset of costs associated with inputs in $\mathcal{I}$.

Definition 7.3 (Bounded-shared-cost property) *A cost measure for an online problem satisfies the bounded-shared-cost property if and only if for any algorithm $\mathcal{A}$ and for all unique costs $m \in \mathcal{A}(\mathcal{I})$, the set of inputs that achieve that cost is bounded.*

If a cost measure does not satisfy the bounded-shared-cost property, it is possible to prove contradicting results under cyclic analysis. That is, if there are infinitely many inputs that achieve each cost, for any algorithms $\mathcal{A}, \mathcal{B}$, it is possible to define bijections such that $\mathcal{A} \prec_c \mathcal{B}$ and $\mathcal{B} \prec_c \mathcal{A}$.

In the case of multicore caching, the total time and makespan cost models both have the bounded-shared-cost property while the miss count and the closely related miss rate do not. For example, the infinitely many sequences that only request some page α (e.g. α, $\alpha\alpha$, $\alpha\alpha\alpha$, $\dots$) all have cost one under miss count, but all have different costs under total time and makespan.

The following lemma guarantees that cyclic analysis has the "to-be-expected" property that if algorithm $\mathcal{A}$ is better than $\mathcal{B}$, then $\mathcal{B}$ is not better than $\mathcal{A}$. In the case of bijective analysis, this property easily follows from the fact that bijections are drawn in finite sets (formed by inputs of the same length). Since the bijections in cyclic analysis are defined in an infinite space, a more careful analysis is required.

Lemma 7.4 *Given algorithms $\mathcal{A}, \mathcal{B}$ for a problem satisfying the bounded-shared-cost property, it is not possible that $\mathcal{A} \prec_c \mathcal{B}$ and $\mathcal{B} \prec_c \mathcal{A}$ at the same time.*

Proof. If $\mathcal{A} \prec_c \mathcal{B}$, by Definition 7.2, there must exist an input $\sigma \in \mathcal{I}$ such that $\mathcal{A}(\sigma) < \mathcal{B}(\pi(\sigma))$. Let σ be the input with the smallest cost under $\mathcal{A}$ that differs between $\mathcal{A}, \mathcal{B}$, and let $\mathcal{I}_{\mathcal{A}}^{\mathcal{A}(\sigma)}, \mathcal{I}_{\mathcal{B}}^{\mathcal{A}(\sigma)} \subset \mathcal{I}$ be the sequences that have cost at most $\mathcal{A}(\sigma)$ in $\mathcal{A}(\mathcal{I}), \mathcal{B}(\mathcal{I})$, respectively. By the bounded-shared-cost property, $|\mathcal{I}_{\mathcal{A}}^{\mathcal{A}(\sigma)}|$ and $|\mathcal{I}_{\mathcal{B}}^{\mathcal{A}(\sigma)}|$ are both bounded and $|\mathcal{I}_{\mathcal{A}}^{\mathcal{A}(\sigma)}| > |\mathcal{I}_{\mathcal{B}}^{\mathcal{A}(\sigma)}|$. It is impossible to define another function ϕ such that $\mathcal{B} \preceq_c \mathcal{A}$ because there are not enough inputs in $\mathcal{I}_{\mathcal{B}}^{\mathcal{A}(\sigma)}$ to map to all inputs in $\mathcal{I}_{\mathcal{A}}^{\mathcal{A}(\sigma)}$ such that the cost of each input under $\mathcal{B}$ is at most the cost of the corresponding input under $\mathcal{A}$. $\square$

Similarly, if $\mathcal{A} \prec_c \mathcal{B}$, then $\mathcal{A} \not\equiv_c \mathcal{B}$ for problems with the bounded-shared-cost property. Additionally, cyclic analysis has the transitive property: if $\mathcal{A} \preceq_c \mathcal{B}$ and $\mathcal{B} \preceq_c \mathcal{C}$, then $\mathcal{A} \preceq_c \mathcal{C}$. The bounded-shared-cost property guarantees that each node in the input-cost graph has infinite out-degree but finite in-degree because each input has infinitely many inputs that cost more than it and finitely many inputs that cost less than it.

Relation of surjectivity to cyclic analysis. In the remainder of the section we will discuss the role of surjective mappings as an intermediate step before defining a bijective mapping between infinite sets. In traditional bijective analysis, since the input set is finite because of the length restriction, any surjective mapping must also

$$\begin{tikzcd}
X_2 & Y_2 & & & X_2 & Y_2 \\
x_{1,1} \arrow[r] & y_1 & & & x_{1,1} \arrow[r] & y_1 \\
x_{1,2} \arrow[ur] & & \arrow[r, Rightarrow, "\text{unzip}"] & {} & x_{1,2} \arrow[r] & y_2 \\
x_{2,1} \arrow[r] & y_2 & & & x_{2,1} \arrow[r] & y_3 \\
x_{2,2} \arrow[ur] & & & & x_{2,2} \arrow[r] & y_4 \\
\dots & \dots & & & \dots & \dots
\end{tikzcd}$$

Figure 7-2: Example of unzipping X_2, Y_2 in a natural surjective mapping.

be bijective. In some problems, including multicore caching, it may be easier to define a surjective mapping between the inputs. We will first show that a class of surjective mappings can be converted into bijective mappings.

Suppose we have a surjective but not necessarily injective mapping between two infinite sets $f : X \to Y$. For all positive integers $m \in \mathbb{N}$, let $X_m \subseteq X, Y_m \subseteq Y$ be subsets of the pre-image and image respectively such that exactly m elements in X_m map to one element in Y_m. That is, given some m, $x \in X_m$ implies that there are $m-1$ other elements $x_1, x_2, \dots, x_{m-1} \neq x$ such that for $i = 1, \dots, m-1$, $f(x) = f(x_i)$. Each X_m, Y_m is an element of a partition of the pre-image and image, respectively.

Definition 7.5 (Natural surjective mapping) *Given a surjective function $f : X \to Y$, f is **natural** if and only if for all $m \in \mathbb{N}$, the partitions X_m and Y_m are either empty or infinite.*

For example, the function $f : \mathbb{N} \to \mathbb{N}, f(x) = \lfloor x/2 \rfloor$ is a natural surjective mapping (assuming $0 \in \mathbb{N}$) because exactly two elements in the pre-image map to each element in the image. In contrast, $g : \mathbb{Z} \to \mathbb{N}, g(x) = |x|$ is not natural because there is only one element in X_1 and Y_1 at $x = y = 0$.

We introduce ***natural surjective (NS) analysis***, a technique to compare algorithms under cyclic analysis using an intermediate surjective *but not injective* mapping. The formalization is almost identical to Definition 7.2, but the function π needs only to be a natural surjective function. We use $\preceq_s$ to denote the relation between two algorithms under NS analysis. In the rest of the chapter, we will refer to natural surjective functions and natural surjective analysis as surjective functions and surjective analysis, respectively.

Lemma 7.6 ("Unzipping" equivalence) *Let algorithms $\mathcal{A}, \mathcal{B}$ be algorithms for a problem with the bounded-shared-cost property. If $\mathcal{A} \preceq_s \mathcal{B}$ under a natural surjective mapping, then $\mathcal{A} \prec_c \mathcal{B}$.*

Proof. At a high level, we will describe how to convert a natural surjective function f into a bijective mapping f_b by "unzipping" any many-to-one mappings in each partition. At a high level, the new mapping f_b "remaps" elements in the preimage to elements in the image.

Let X_m, Y_m be the pre-image and image of a non-empty mapping-based partition for any fixed $m \in \mathbb{N}$. Suppose we order the elements in Y_m from lowest to highest and let y_i be the i-th largest element in Y_m. The elements in any set Y_m can be ordered because of the bounded-shared-cost property. Given an element $y_i^m \in Y_m$, let the corresponding elements in the pre-image be $x_{i,j} \in X_m$ for $j = 1, 2, \ldots, m$ in some order. Since $\mathcal{A}(x_{i,j}) \leq \mathcal{B}(y_i^m)$ for all i, j (by the definition of surjective analysis), for any i, j, $\mathcal{A}(x_{i,j}) \leq \mathcal{B}(y_z^m)$ for $z > i$. Therefore, we define a new bijective mapping f_b based on f such that $f_b(x_{i,j}) = y_{mi+j-1}^m$. The new mapping f_b satisfies the property that for all $\sigma \in \mathcal{I}$, $\mathcal{A}(\sigma) \leq \mathcal{B}(f_b(\sigma))$. □

As shown in the example in Figure 7-2, we can convert a natural surjective mapping to a bijective one by "unzipping" the mapping and maintaining the relative order of inputs.

The relationship between surjective analysis and cyclic analysis allows for different paths to proving relationships between algorithms. In traditional bijective analysis, we had to define a direct bijection between two algorithms because all surjections are bijections in finite sets of the same size. Natural surjective analysis is a potentially easier proof technique that is equivalent to cyclic analysis.

7.4 Cyclic analysis for multicore caching

It is straightforward to show that all lazy multicore caching algorithms are equivalent under cyclic analysis. Therefore, to show a separation between two algorithms, this section analyzes a variant of FWF that flushes (empties) the entire cache if it incurs a miss when the cache is full. In what follows, this section shows the advantage of lazy algorithms over FWF. While this result is not surprising, the techniques used in the proofs in this section prepare the reader for the more complicated proof in the next section.

Lemma 7.7 *Assume $p = 2$. Consider two lazy caching algorithms $\mathcal{A}$ and $\mathcal{B}$ which have the same eviction policy starting at the same timestep t_0 and have the same cache contents at t_0 except for one page x that is present in the cache of $\mathcal{A}$ and absent in the cache of $\mathcal{B}$. If $\mathcal{A}$ and $\mathcal{B}$ incurred the same cost up until timestep t_0, we have $\mathcal{A} \prec_c \mathcal{B}$.*

Proof. At a high level, we will define a surjective cyclic mapping on the input space with cycles of length 2. For inputs where x is never requested before being evicted, $\mathcal{A}$ and $\mathcal{B}$ perform similarly. We assume these inputs are mapped to themselves and ignore them (the cycles associated with these inputs are self-loops). In the remainder of the proof, we assume x is requested for the first time at timestep $t \geq t_0$ before being evicted. At timestep t, $\mathcal{A}$ has a hit on the request to x while $\mathcal{B}$ incurs a miss. As a result, the schedule of the two algorithms (i.e., the order at which they serve the requests) becomes different after serving x and hence there is no guarantee that $\mathcal{A}$ has less cost that $\mathcal{B}$.

We define a bijection b in a way that the schedule of $\mathcal{A}$ for any input $\mathcal{R}$ is similar to that of $\mathcal{B}$ for serving $b(\mathcal{R})$. The bijection that we define creates cycles of length 2: if $\mathcal{R}' = b(\mathcal{R})$ then $\mathcal{R} = b(\mathcal{R}')$; we denote this by $\mathcal{R} \leftrightarrow \mathcal{R}'$.

Let P_1 and P_2 denote the two cores and let $\begin{cases} .. \ \sigma_1 \\ .. \ \sigma_2 \end{cases}$ denote the ***continuation*** of a sequence where P_1 asks for sequence σ_1 and P_2 asks for σ_2 from time t onward.

We define the bijection based on two cases. In both cases, one of the cores, say P_2, has a request to page x at time t and hence $\mathcal{A}$ and $\mathcal{B}$ perform differently on the continuation of the sequence. Assume the contents of the caches of $\mathcal{A}$ and $\mathcal{B}$ at time t are respectively $H \cup \{x\}$ and H. **Case 1:** P_1 requests a page $q \notin H$.

Recall that P_2 asks for x at time t, so the input can be written as $\mathcal{R} = \begin{cases} ..\ q\sigma \\ ..\ x\sigma' \end{cases}$ for some σ and σ'. We define $\mathcal{R}' = \begin{cases} ..\ x^\tau\sigma \\ ..\ q\sigma' \end{cases}$. To show the mapping $\mathcal{R} \leftrightarrow \mathcal{R}'$ is a valid mapping we need to show $\mathcal{A}(\mathcal{R}) \leq \mathcal{B}(\mathcal{R}')$ and $\mathcal{A}(\mathcal{R}') \leq \mathcal{B}(\mathcal{R})$. First, we show $\mathcal{A}(\mathcal{R}) \leq \mathcal{B}(\mathcal{R}')$. On input $\mathcal{R}$, $\mathcal{A}$ has a miss on q and a hit on x at time t; so, $\mathcal{A}$ starts serving σ and σ' at timesteps $t + \tau$ and $t + 1$, respectively; it serves σ exactly $\tau - 1$ timesteps later than σ'. On input $\mathcal{R}'$, $\mathcal{B}$ has a miss on both x and q at time t. It incurs an additional $\tau - 1$ hits on x after fetching it. So, $\mathcal{B}$ starts serving σ and σ' at timesteps $t + \tau + (\tau - 1)$ and $t + \tau$, respectively. In other words, it serves σ exactly $\tau - 1$ timesteps later than σ'. The content of the cache of $\mathcal{A}$ and $\mathcal{B}$ is the same for serving σ and σ'. We conclude that the number of misses (and hence total time) of $\mathcal{B}$ in serving σ and σ' in $\mathcal{R}$ is the same as $\mathcal{A}$ in $\mathcal{R}'$. For the first requests to q and x in $\mathcal{R}$, $\mathcal{A}$ incurs one miss (and total time $\tau+1$) while $\mathcal{B}$ incurs two misses (and total time $3\tau - 1$) for the first requests to x^τ and q in $\mathcal{R}'$. We conclude that $\mathcal{A}(\mathcal{R}) < \mathcal{B}(\mathcal{R}')$. To complete the proof in Case 1, we should show $\mathcal{A}(\mathcal{R}') \leq \mathcal{B}(\mathcal{R})$. When $\mathcal{A}$ serves $\mathcal{R}'$, it incurs τ hits on x^τ and one miss on q; as such, it starts serving σ and σ' at the same time $t+\tau$. On the other hand, when $\mathcal{B}$ serves $\mathcal{R}$, it incurs a miss on both q and x and starts serving σ and σ' at the same time $t + \tau$. So, the two algorithms incur the same cost for serving σ and σ'. Moreover, $\mathcal{A}$ incurs one miss and τ hits (and total time 2τ) for serving x^τ and q while $\mathcal{B}$ incurs two misses (and total time 2τ) for serving q and x, so $\mathcal{A}(\mathcal{R}') = \mathcal{B}(\mathcal{R})$.

Case 2: P_1 asks for a page $a \in H$.

So, the input can be written as $\mathcal{R} = \begin{cases} ..\ a\sigma \\ ..\ x\sigma' \end{cases}$ for some sequence of requests σ and σ'. We define $\mathcal{R}' = \begin{cases} ..\ x\sigma \\ ..\ a^\tau\sigma' \end{cases}$. To show the mapping $\mathcal{R} \leftrightarrow \mathcal{R}'$ is a valid mapping, we first show $\mathcal{A}(\mathcal{R}) \leq \mathcal{B}(\mathcal{R}')$. $\mathcal{A}$ starts serving both σ and σ' in $\mathcal{R}$ at $t + 1$ because $\mathcal{A}$ has hits on both a and x. On the other hand, $\mathcal{B}$ has a miss on x and a hit on a when serving all copies of τ. That means, it starts serving both σ and σ' in $\mathcal{R}'$ at the same time $t + \tau$. The content of the cache of the two algorithms is also the same (x is now in the cache of $\mathcal{B}$). So, $\mathcal{A}$ and $\mathcal{B}$ incur the same number of misses (and total time) for both σ and σ'. For the prefixes a and x in $\mathcal{R}$, $\mathcal{A}$ incurs 0 misses (and total time 2); for the prefixes a^τ and x in $\mathcal{R}'$, $\mathcal{B}$ incurs 1 miss (and total time 2τ). We conclude $\mathcal{A}(\mathcal{R}) < \mathcal{B}(\mathcal{R}')$. Next, we show $\mathcal{A}(\mathcal{R}') \leq \mathcal{B}(\mathcal{R})$. $\mathcal{A}$ has hits on all requests in a^τ and x in $\mathcal{R}'$, i.e., it serves σ and σ' at timesteps $t + 1$ and $t + \tau$, respectively. That is, it serves σ exactly $\tau - 1$ units later than σ'. $\mathcal{B}$, on the other hand, has a hit at a and a miss at x in $\mathcal{R}$, i.e. it serves σ and σ' at times $t + 1$ and $t + \tau$, respectively. So, the two algorithms incur the same cost for σ and σ'. For the prefixes a^τ and x, $\mathcal{A}$ incurs 0 misses and total time $\tau + 1$. For the prefixes a and x, $\mathcal{B}$ incurs 1 miss and total time $\tau + 1$. We conclude that $\mathcal{A}(\mathcal{R}') \leq \mathcal{B}(\mathcal{R})$. □

We show the advantage any lazy algorithm $\mathcal{A}$ over non-lazy FWF by comparing their cache contents at each timestep.

Theorem 7.8 *Any lazy algorithm $\mathcal{A}$ is strictly better than* FWF *under cyclic analysis for $p = 2$, that is, $\mathcal{A} \prec_c$* FWF.

Proof. Let FWF_i be a variant of FWF which, instead of flushing the cache, evicts i pages from the cache; these i pages are selected according to $\mathcal{A}$'s eviction policy. That is, the algorithm evicts i pages that $\mathcal{A}$ evicts when its cache is full (as an example, if $\mathcal{A}$ is LRU, the algorithm evicts the i least-recently-used pages). We will show $\mathcal{A} \prec_c \text{FWF}$ by transitivity of bijection. In particular, we show

$$\mathcal{A} = \text{FWF}_1 \prec_c \ldots \prec_c \text{FWF}_{k-1} \prec_c \text{FWF}_k = \text{FWF}.$$

Let FWF_i^t be an algorithm that applies FWF_i for the first t timesteps and FWF_{i+1} for timesteps after and including $t+1$. If we can show $\text{FWF}_i^{t+1} \prec_c \text{FWF}_i^t$ for all t, again by transitivity of bijection, we get $\text{FWF}_i \prec_c \text{FWF}_{i+1}$. We note that FWF_i^{t+1} and FWF_i^t differ in serving at most one request at time t, and they have the same eviction strategy for the remainder of the input. If the cores do not incur a miss at time t, both algorithms perform similarly. For sequences for which there is a miss at time t, there will be one less page in the cache of FWF_i^t compared to FWF_i^{t+1}. Therefore, $\text{FWF}_i^{t+1} \prec_c \text{FWF}_i^t$ by Lemma 7.7. □

As the bijection in the proofs illustrates, the main insight of cyclic analysis is the direct comparison of algorithms by drawing mappings between inputs of different lengths. In contrast to the single-core setting, inputs of the same length (in the number of requests) in multicore caching may take different amounts of time. So we define bijections based on the schedule (and therefore length in time) rather than the number of requests.

7.5 Advantage of LRU with locality of reference

To demonstrate how to use cyclic analysis to separate algorithms, this section proves the separation of LRU from all other lazy algorithms on inputs with locality of reference via cyclic analysis. Along the way, it demonstrates how to use surjective analysis to establish relations between algorithms under cyclic analysis. In practice, LRU (and its variants) are empirically better than all other known caching algorithms [340] because sequences often have temporal locality.

Preliminaries

First, this section formalizes the notion of a schedule from Section 7.2, which represents an algorithm's eviction decisions by repeating requests in an input on a miss. It will use the schedule to later define locality of reference. Throughout this section, let $\mathcal{A}$ be a caching algorithm and $\mathcal{R}$ be an input.

Definition 7.9 (Schedule) *The* ***schedule*** $\mathcal{S}_{\mathcal{R},\mathcal{A}} = \{\mathcal{S}_{\mathcal{R}_1,\mathcal{A}}, \ldots, \mathcal{S}_{\mathcal{R}_p,\mathcal{A}}\}$ *is another input where each request sequence is defined as the implicit schedule that* $\mathcal{A}$ *generated while serving* $\mathcal{R}$*. That is,* $\mathcal{S}_{\mathcal{R}_i,\mathcal{A}}[t]$ *is the request that core* P_i *serves at timestep* t *under* $\mathcal{A}$*. Also,* $\mathcal{S}_{\mathcal{R},\mathcal{A}}$ *is the same as* $\mathcal{R}$ *with each miss repeated at most* $\tau - 1$ *times (as many repetitions as it takes to resolve the given miss, which might be less than* $\tau - 1$ *if the page was already in the process of being fetched). We use* $\mathcal{S}_{\mathcal{R}_i,\mathcal{A}}[t_1, t_2]$ *(for all* i*) to denote all requests (including repetitions due to misses) made by* P_i *between timesteps* t_1 *and* t_2 *(inclusive).*

We use the formal definition of schedule to discuss dividing up an input under $\mathcal{A}$ based on its schedule up until some timestep.

Definition 7.10 (Schedule prefix and suffix) *Let* $n_{\mathcal{R},\mathcal{A}}$ *be the time required for* $\mathcal{A}$ *to serve* $\mathcal{R}$*. Given an integer timestep* $j < n_{\mathcal{R},\mathcal{A}}$*, we define parts of the schedule that will be served before, after, and during timestep* $j + 1$*.*

Informally, the ***schedule prefix*** $\mathcal{S}^{pre}_{j,\mathcal{R},\mathcal{A}}$ *is all the requests served up to timestep* j *with repetitions matching scheduling delay, the schedule at timestep* $j+1$*,* $\mathcal{S}_{\mathcal{R},\mathcal{A}}[j+1]$*, is all requests served at timestep* $j + 1$*, and the* ***schedule suffix*** $\mathcal{S}^{suf}_{j,\mathcal{R},\mathcal{A}}$ *is all requests served after timestep* $j+1$ *with repetitions matching scheduling delay. Note that* $\mathcal{S}^{pre}_{\mathcal{R},\mathcal{A}}$ *or* $\mathcal{S}^{suf}_{\mathcal{R},\mathcal{A}}$ *may be empty. When the timestep* j *and/or algorithm* $\mathcal{A}$ *are clear from context, we will drop them from the schedule notation.*

Definition 7.11 (Request prefix and suffix) *Let* $\mathcal{R}^{\leq j,\mathcal{A}}$ *be all subsequences from* $\mathcal{R}$ *served up to timestep* j*,* $\mathcal{R}^{>j,\mathcal{A}}$ *be all subsequences from* $\mathcal{R}$ *served after timestep* j*, and* r_{j+1} *be the requests at timestep* $j+1$*. For simplicity, we define the* ***request prefix*** *as* $\mathcal{R}^{pre} = \mathcal{R}^{\leq j,\mathcal{A}}$ *and* ***request suffix*** *as* $\mathcal{R}^{suf} = \mathcal{R}^{>j+1,\mathcal{A}}$ *when* $j, \mathcal{A}$ *are understood from context.*

The request prefix and suffix formalizes the analysis technique from Section 7.4 of defining mappings based on the continuation of the input after some timestep.

Using the LRU example in Figure 7-1 when $j = 4$, $\mathcal{R}_1^{\text{pre}} = a_1a_2$, $\mathcal{R}_2^{\text{pre}} = a_3a_4$ because those are the pages that have been requested until timestep 4. Similarly, $\mathcal{R}_1^{\text{suf}} = a_1a_5$ and $\mathcal{R}_2^{\text{suf}} = a_5a_2$ because those are the requests remaining after timestep 4. Additionally, $\mathcal{S}^{\text{pre}}_{\mathcal{R}_1} = a_1a_1a_1a_2$ and $\mathcal{S}^{\text{pre}}_{\mathcal{R}_2} = a_3a_3a_3a_4$. At timestep 4, both cores are fetching, so the requests at that timestep $r_{j+1} = (a_2, a_4)$. The suffix is the schedule for timesteps after 4, so $\mathcal{S}^{\text{suf}}_{\mathcal{R}_1} = a_2a_1a_5a_5$ and $\mathcal{S}^{\text{suf}}_{\mathcal{R}_2} = a_4a_5a_5a_5a_2$.

Locality of reference and the Max-Model. We will restrict the space of all inputs with the "Max-Model", an experimentally-validated model of locality of reference that limits the number of distinct pages in subsequences of an input with a concave function [8].

We define a ***window*** of size w in the multicore setting as p runs of consecutive requests of length w (one for each core). The Max-Model for multicore caching is the same as in single-core caching except that it considers windows over all cores.

In the Max-Model for multicore caching, an input $\mathcal{R}$ is ***consistent*** with some increasing concave function f if the number of distinct pages in any window of size

w is at most $f(w)$, for any $w \in \mathbb{N}$ [8]. That is, a function $f : \mathbb{N} \to \mathbb{R}^+$ is ***concave*** if $f(1) = p$, and $\forall n \in \mathbb{N} : f(n+1) - f(n) \leq f(n+2) - f(n+1)$. In the Max-Model, we also require that f is surjective on the integers between p and its maximum value.

It is easy to adapt cyclic analysis to the Max-Model by restricting to inputs consistent with a concave function f (denoted by $\mathcal{I}^f$). Let $\mathcal{A} \preceq_c^f \mathcal{B}$ denote that $\mathcal{A}$ is no worse than $\mathcal{B}$ on $\mathcal{I}^f$ under cyclic analysis. Similarly, let $\mathcal{A} \preceq_s^f \mathcal{B}$ denote that $\mathcal{A}$ is no worse than $\mathcal{B}$ on $\mathcal{I}^f$ under surjective analysis.

Advantage of LRU on inputs with locality

In the rest of the section, we will show that LRU is no worse than sequences with locality under cyclic analysis by establishing a surjective mapping (Definition 7.5) and converting it into a bijective mapping (Lemma 7.6). The main technical challenge in the proof is that sequences with the same number of requests may have different schedules and therefore may differ significantly in their cost, even if they only differ in one request. We use cyclic analysis to avoid the restriction of comparing inputs of the same length and instead define a function to relate inputs of the same cost.

Along the way, we demonstrate how to use surjective analysis as a proof technique for comparing algorithms via cyclic analysis on the entire space of inputs as described in Section 7.3. The construction of the surjective mapping is inspired by a similar argument in the single-core setting by Angelopoulos and Schweitzer [16] which establishes a bijective mapping within finite partitions, but requires a more complex mapping based on schedules.

We will show that for every algorithm $\mathcal{A}$, LRU $\preceq_s^f \mathcal{A}$. An arbitrary algorithm $\mathcal{A}$ may be very different from LRU. Therefore, instead of defining a direct bijection, we will use intermediate algorithms $\mathcal{B}_1, \ldots, \mathcal{B}_\ell$ such that $\mathcal{A} \equiv \mathcal{B}_1 \succeq_s^f \ldots \succeq_s^f \mathcal{B}_i \succeq_s^f \ldots \succeq_s^f \mathcal{B}_\ell \equiv$ LRU. The result follows from the transitivity of the "$\preceq_s^f$" relation. Intuitively, we construct algorithms "closer" to LRU at each step in the series as we will explain in Lemma 7.14. We formalize the notion of an algorithm $\mathcal{A}$'s "closeness" to LRU in terms of the evictions that it makes. An algorithm $\mathcal{A}$ is **LRU-*like*** at timestep t if after serving all requests up to time $t-1$, it serves all requests at time t as LRU would.

Defining a surjective mapping between inputs. At a high level, the proof proceeds by defining a surjection between similar sequences with two pages swapped. We define a "complement" of a sequence as a new sequence with certain pages swapped, and show properties of complements of sequences with locality required for our main proof.

Definition 7.12 (Complement [16]) *Let β, δ denote two distinct pages in U, the universe of pages. Let $\mathcal{R}_i[j]$ denote the j-th request in the ith request sequence of an input $\mathcal{R}$. The **complement** of $\mathcal{R}_i[j]$ with respect to β and δ, denoted by $\overline{\mathcal{R}_i[j]}^{(\beta,\delta)}$, is the function that replaces β with δ, and vice versa. Formally, $\overline{\mathcal{R}_i[j]}^{(\beta,\delta)} = \delta$, if $\mathcal{R}_i[j] = \beta$; $\overline{\mathcal{R}_i[j]}^{(\beta,\delta)} = \beta$, if $\mathcal{R}_i[j] = \delta$; and $\overline{\mathcal{R}_i[j]}^{(\beta,\delta)} = \mathcal{R}_i(j)$, otherwise.*

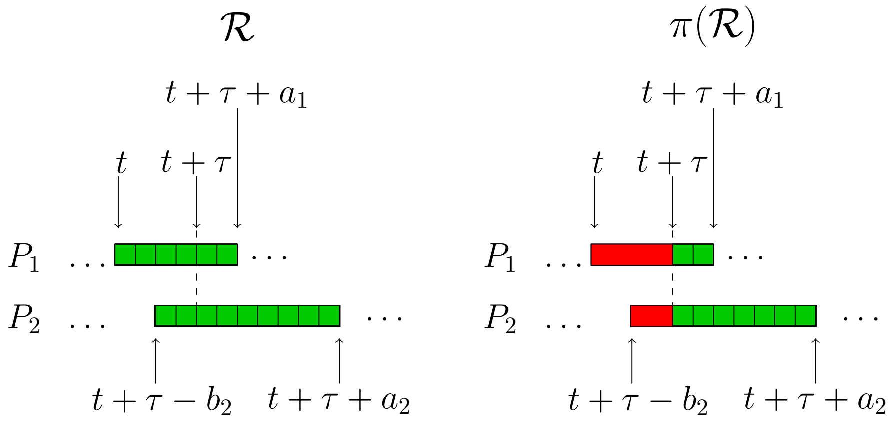


Figure 7-3: An example of the mapping of an input $\mathcal{R}$ under the algorithm $\mathcal{B}$ to $\pi(\mathcal{R})$ under with $\tau = 4$. On the left, an input $\mathcal{R}$ where $a_1 = 2, b_2 = 2, a_2 = 7$. The green boxes indicate hits on a page σ in $\mathcal{B}$'s cache but not in the algorithm $\mathcal{A}$'s cache. On the right, we show the corresponding $\pi(\mathcal{R})$. The red boxes denote misses on σ.

We use $\overline{\mathcal{R}_i[j]}$ when β, δ are clear from context. We denote each request sequence $\mathcal{R}_i = \sigma_1^i \ldots \sigma_{n_i}^i$, where $\mathcal{R}_i$ has n_i requests. For any sequence for a single core $\mathcal{R}_i$, $\overline{\mathcal{R}_i} = \overline{\mathcal{R}_i[1]}, \ldots, \overline{\mathcal{R}_i[n_i]}$. For any multicore sequence $\mathcal{R}$, $\overline{\mathcal{R}} = \{\overline{\mathcal{R}_1}, \ldots, \overline{\mathcal{R}_p}\}$. For any sequence $\mathcal{R}_i$, let $\mathcal{R}_i[j_1, j_2]$ denote the (contiguous) subsequence of requests $\sigma_{i,j_1}, \ldots, \sigma_{i,j_2}$. Also, we use $\mathcal{R}_\alpha \cdot \mathcal{R}_\gamma$ to denote the concatenation of two sequences $\mathcal{R}_\alpha, \mathcal{R}_\gamma$.

We now extend a lemma from [16] about sequences with locality that we will use in our main theorem later. The lemma says that if a sequence $\ldots\delta\ldots\beta\ldots\delta\ldots\beta\ldots$ exhibits locality of reference, then $\ldots\delta\ldots\beta\ldots\beta\ldots\delta$ does as well.

Lemma 7.13 *Let $\mathcal{R}$ be a sequence of requests consistent with f, $\mathcal{A}$ be a caching algorithm, and $n_{\mathcal{R},\mathcal{A}}$ be the time that it takes $\mathcal{A}$ to serve $\mathcal{R}$. Let $j \leq n_{\mathcal{R},\mathcal{A}}$ be an (integer) timestep such that $\mathcal{S}_{\mathcal{R},\mathcal{A}}[1,j]$ contains a request to β, and in addition, δ does not appear in $\mathcal{S}_{\mathcal{R},\mathcal{A}}^{pre} = \mathcal{S}_{\mathcal{R},\mathcal{A}}[1,j]$ after the last request to β in $\mathcal{S}_{\mathcal{R},\mathcal{A}}^{pre}$.*

Let $\mathcal{R}' = \mathcal{R}^{pre}\overline{\mathcal{R}^{suf}}$ denote the sequence $\mathcal{R}^{\leq j,\mathcal{A}}\overline{\mathcal{R}^{>j,\mathcal{A}}}$, and suppose that $\mathcal{R}'$ is not consistent with f. Then $\mathcal{R}^{suf}$ contains a request to β; furthermore, no request to δ in $\mathcal{S}_{\mathcal{R},\mathcal{A}}^{suf}$ ($\mathcal{S}_{\mathcal{R},\mathcal{A}}^{suf} = \mathcal{S}_{\mathcal{R},\mathcal{A}}[j+1, n_{\mathcal{R},\mathcal{A}}]$) occurs earlier than the first request to β in $\mathcal{S}_{\mathcal{R},\mathcal{A}}^{suf}$.

The following lemma guarantees that for any algorithm $\mathcal{A}$ which may make a non-LRU-like eviction at the $(j+1)$-th timestep of some $\mathcal{R} \in \mathcal{I}^f$ (but will make LRU-like evictions for the rest of the timesteps after $j+1$), we can define an algorithm $\mathcal{B}$ that makes the same decisions as $\mathcal{A}$ up until timestep j of any sequence in $\mathcal{I}^f$, makes an LRU-like decision on the $(j+1)$-th timestep, and is no worse than $\mathcal{A}$ under surjective analysis.

Lemma 7.14 *Let $\mathcal{I}^f$ be* all *inputs consistent with f and let j be an integer. Suppose $\mathcal{A}$ is an algorithm with the property that for every input $\mathcal{R} \in \mathcal{I}^f$, $\mathcal{A}$ is* LRU*-like on timestep $t+1$, for all $t \geq j+1$. Then there exists an algorithm $\mathcal{B}$ with the following properties:*

1. *For every input $\mathcal{R} \in \mathcal{I}^f$, $\mathcal{B}$ makes the same decisions as $\mathcal{A}$ on the first j timesteps while serving $\mathcal{R}$ (i.e., $\mathcal{A}$ and $\mathcal{B}$ make the same eviction decisions for each miss in requests up to and including time t).*

2. *For every input $\mathcal{R} \in \mathcal{I}^f$, $\mathcal{B}$ is* LRU*-like on $\mathcal{R}$ at timestep t.*

3. *$\mathcal{B} \preceq_s^f \mathcal{A}$.*

Proof Sketch. The main insight in this proof is the comparison of inputs with different numbers of page requests but the same cost under two different algorithms. If an algorithm $\mathcal{A}$ makes a non-LRU-like decision at some timestep, we construct a surjection that maps it to a sequence with the same schedule under another algorithm $\mathcal{B}$.

At a high level, we use a "sequence reordering" mapping inspired by Lemma 2 of [16]. Let $\mathcal{B}$ be an algorithm that matches the evictions of $\mathcal{A}$ until time t, when it makes LRU-like evictions. Suppose at time t that $\mathcal{A}$ evicted a page σ_{NLRU} and $\mathcal{B}$ evicted a page σ_{LRU}. We construct $\mathcal{B}$ to evict the same pages as $\mathcal{A}$ on the remainder of the sequence.

We construct a surjective mapping π such that for any request sequence $\mathcal{R}$, $\mathcal{B}(\mathcal{R}) \leq \mathcal{A}(\pi(\mathcal{R}))$. There are two main cases based on the continuation of the input after time t. At a high level, if an input has locality of reference, then there are not many requests to different pages. Now, if possible, we swap $\sigma_{\text{LRU}}, \sigma_{\text{NLRU}}$ in the continuation of the input after time t since these will result in the same cost in the continuation.

Case 1: Swapping $\sigma_{\text{LRU}}, \sigma_{\text{NLRU}}$ in the continuation maintains locality. In this case, $\mathcal{A}(\mathcal{R}) = \mathcal{B}(\pi(\mathcal{R}))$ because the different decisions at time t did not affect the number of misses (and therefore the total time) while serving the rest of the input. Swapping the pages where $\mathcal{A}$, $\mathcal{B}$ differ in the continuation of the mapped-to input results in the same behavior.

Case 2: Swapping $\sigma_{\text{LRU}}, \sigma_{\text{NLRU}}$ in the continuation does not maintain locality. There are a few cases when swapping the two pages would disrupt locality.

- If there was a miss on another page before the first request to σ_{NLRU} in the continuation after time t, both algorithms would incur the same cost since the difference in decision does not affect the number of hits and misses in the rest of the input. In this case, we set $\pi(\mathcal{R}) = \mathcal{R}$, and $\mathcal{B}(\mathcal{R}) = \mathcal{A}(\pi(\mathcal{R}))$.

- If there was not a miss before the first request to σ_{NLRU} after time t, $\mathcal{B}$ hits on the first request to σ_{LRU} in the continuation, and we remove requests in $\pi(\mathcal{R})$ so that the schedule of $\mathcal{B}$ serving $\mathcal{R}$ matches the schedule of $\mathcal{A}$ serving $\pi(\mathcal{R})$. Since the schedules match, $\mathcal{B}(\mathcal{R}) = \mathcal{A}(\pi(\mathcal{R}))$.

- The above two cases cover the entire codomain, but not the domain. For the remaining inputs, we can map them arbitrarily to inputs of higher cost such that there are no more than two inputs in the domain mapped to any input in the codomain. By construction, $\mathcal{B}(\mathcal{R}) < \mathcal{A}(\pi(\mathcal{R}))$. We present an example of generating such a mapping from an input $\mathcal{R}$ under algorithms $\mathcal{A}$ and $\mathcal{B}$ given page σ in Figure 7-3.

□

Given any algorithm $\mathcal{A}$, we repeatedly apply Lemma 7.14 to construct a new algorithm $\mathcal{B}$ which is LRU-like after some timestep t and is no worse than $\mathcal{A}$.

Let $n_{\mathcal{R},\mathcal{A}}$ be the time it takes to serve input $\mathcal{R}$ with $\mathcal{A}$, and let B_t be the class of algorithms that make LRU-like decisions on timesteps $n_{\mathcal{R}} - t$ of every input $\mathcal{R} \in \mathcal{I}^f$.

Lemma 7.15 *For every algorithm $\mathcal{A}$ there exists an algorithm $\mathcal{B}_t \in B_t$ such that $\mathcal{B}_t \preceq_s \mathcal{A}$, and for every input $\mathcal{R} \in \mathcal{I}^f$, $\mathcal{B}_t$ makes the same decisions as $\mathcal{A}$ during the first $n_{\mathcal{R},\mathcal{A}} - t$ timesteps while serving $\mathcal{R}$.*

For every lazy algorithm $\mathcal{A}$, Lemma 7.15 guarantees the existence of an algorithm $\mathcal{B}$ that makes LRU-like decisions on all timesteps for any input in $\mathcal{I}^f$ and is no worse than $\mathcal{A}$. The only algorithm with this property is exactly LRU.

Theorem 7.16 *For any lazy caching algorithm $\mathcal{A}$,* LRU $\preceq_s^f \mathcal{A}$.

We have defined a surjection from LRU to any other algorithm through intermediate algorithms that are progressively "closer to LRU". Therefore, we have shown that LRU is the best lazy algorithm under cyclic analysis via surjective analysis and therefore under cyclic analysis by combining Theorem 7.16 and Lemma 7.6.

Theorem 7.17 *For any lazy caching algorithm $\mathcal{A}$,* LRU $\prec_c^f \mathcal{A}$.

This chapter takes the first steps beyond worst-case analysis for multicore caching with the separation of LRU from all other lazy algorithms on inputs with locality via cyclic analysis. The main insight in the proof is to compare inputs of different lengths (in terms of the number of page requests) but the same schedule with a surjective mapping and then to convert the mapping into a bijection. Although we used it the case of multicore caching, cyclic analysis is a general analysis technique that may be applied to other online problems.

7.6 Related multicore caching models

This chapter reviews alternative models for multicore caching in order to explain why we use the free-interleaving model. Specifically, it discusses a class of models for multicore caching called fixed interleaving and the schedule-explicit model introduced by Hassidim [176]. At a high level, these models assume the order in which the requests are served is decided by the adversary. In practice, however, the schedule of an algorithm is implicitly defined through the eviction strategies [250, 251], so the free-interleaving model studied in this chapter is more practical.

Most of the existing work focuses on minimizing either the makespan of caching strategies or on minimizing the number of misses. In the case of single-core caching, minimizing the makespan and number of misses are equivalent as makespan is simply τ times the number of misses. For multicore caching, however, there is no such direct relationship between makespan and number of misses. In this chapter, we introduce the total time, a cost measure with benefits over both makespan and number of misses while capturing aspects of each.

Feuerstein and Strejilevich de Loma [147, 352] introduced multi-threaded caching as the problem of determining an optimal schedule in terms of the optimal interleaved request sequence from a set of individual request sequences from multiple cores. More precisely, given p request sequences $\mathcal{R}_1, \ldots, \mathcal{R}_p$, they study miss and makespan minimization for a "flattened" interleaving of all $\mathcal{R}_i$'s. Our work focuses on algorithms for page replacement rather than ordering (scheduling) of the input sequences. As mentioned, in practice, the schedule of page requests is embedded in the page-replacement algorithm.

Several previous works [31, 88, 216] studied multicore caching in the ***fixed-interleaving model*** (named by Katti and Ramachandran [216]). This model assumes each core has full knowledge of its future request sequence where the offline algorithm has knowledge of the interleaving of requests. The interleaving of requests among cores is the same for all caching algorithms and potentially adversarial (for competitive analysis). Katti and Ramachandran [216] gave lower bounds and a competitive algorithm for fixed interleaving with cores that have full knowledge of their individual request sequences. In practice, cores do not have any knowledge about future requests, and do not necessarily serve requests at the same rate. Instead, they serve requests at different rates depending on whether they need to fetch pages to the cache.

Hassidim [176] introduced a model for multicore caching before free interleaving which we call the ***schedule-explicit model*** that allows offline algorithms to define an explicit schedule (ordering of requests) for the online algorithm. Given an explicit schedule, the online algorithm serves an interleaved sequence in the same way that a single-core algorithm does. The cost of the algorithm, measured in terms of makespan, is then compared against the cost of an optimal offline algorithm (which potentially serves the input using another schedule).

Both schedule-explicit and free-interleaving models include a fetch delay upon a miss, but schedule-explicit gives offline algorithms more power by allowing them to arbitrarily delay the start of sequences at no cost in terms of the number of misses (Theorem 3.1 of [176]). While schedule-explicit provides useful insight about serving multiple request sequences simultaneously, it leads to overly pessimistic results when minimizing the number of misses as it gives offline algorithms an unfair advantage.

Finally, competitive analysis for distributed systems illustrates the difficulty of multiple independent processes. For example, system nondeterminism in distributed algorithms [22] addresses nondeterminism in the system as well as in the input. Furthermore, a recent work [70] confirms the difficulty that online algorithms face in "scheduling" multiple inputs in the distributed setting.

7.7 Conclusion

This chapter takes the first steps beyond worst-case analysis of multicore caching. In Theorem 7.17, this chapter separated LRU from other algorithms on sequences with locality of reference. More generally, it introduced cyclic analysis and demonstrated its flexibility in the direct comparison of online algorithms. I expect cyclic analysis to be useful in the study of other online problems, and leave such application as future work.

I conclude by explaining why I am optimistic about multicore caching. Multicore caching is an important problem in online algorithms and motivated by computer architectures with hierarchical memory. Practitioners have extensively studied cache-replacement policies for multiple cores. The need for theoretical understanding of multicore caching will only grow as multicore architectures become more prevalent.

Locality-first strategy. The separation of LRU from all other online algorithms in this chapter theoretically grounds the locality-first strategy because it shows that cache-replacement algorithms that take advantage of locality perform better in shared caches with parallelism. These results show that despite the concern that locality and parallelism trade off with each other, algorithms that take advantage of locality perform well even in the face of parallelism.

Chapter 8

Multicore Paging Algorithms Cannot Be Competitive

This chapter sets the stage for the exploration of beyond-worst-case measures that mathematically ground the locality-first strategy for the multicore-paging problem in Chapter 7 by presenting lower bounds for the multicore cache-replacement problem. The main result in this chapter is that all lazy algorithms, a large class of algorithms that includes all currently known practical algorithms, are arbitrarily far from the optimal algorithm in the worst case. In practice, algorithms that take advantage of locality perform better, however, motivating the need to consider algorithm behavior beyond the worst case.

This work was conducted in collaboration with Shahin Kamali [209].

Abstract

This chapter answers an open question about the existence of competitive multicore paging algorithms in the negative. Specifically, it shows that all lazy algorithms, which include all practical algorithms, cannot be competitive against the optimal offline algorithm. These lower bounds demonstrate the limits of competitive analysis for capturing real-world performance differences between cache-replacement algorithms.

8.1 Introduction

As detailed in Chapter 7, caching in multicore architectures has been well-studied in practice, but our theoretical understanding lags behind. Unfortunately, most existing theoretical guarantees on online multicore caching algorithms are negative [251] due to the "scheduling" aspect of multicore caching.

Contributions

This chapter confirms the intuition that multicore caching is much harder than single-core caching and show that all practical ***lazy*** algorithms are equivalently non-competitive[1] against OPT (Corollary 8.2). More precisely, we provide adversarial inputs formed by a total of n requests that show the competitive ratio of any lazy algorithm is $\Omega(n^{1/2}/k)$.

An online algorithm is lazy[2] if it 1) evicts a page only if there is a fault 2) evicts at most one page in case of a fault, 3) for all timesteps, does not evict a page that incurred a hit in that timestep, and 4) evicts a page only if there is no space left in the cache. Algorithms with properties 1-3 (but not necessarily 4) are called "honest" algorithms [251]. Lazy algorithms capture natural properties of online algorithms. For example, if there was a hit on a page σ at some timestep, a lazy algorithm does not evict σ in that same timestep. Additionally, once the cache is full, a lazy algorithm keeps it full. Common caching strategies such as LRU and First-In-First-Out (FIFO) are clearly lazy.

Map. This chapter is organized as follows. It omits the multicore caching problem definition since the preliminaries have been covered in Section 7.2. Section 8.2 presents the main lower bound result, and Section 8.3 provides concluding remarks.

8.2 Non-competitiveness of lazy algorithms

This section shows that no lazy algorithm for multicore caching is competitive in terms of the number of faults. It first shows that no algorithm is competitive for the case of two cores in Theorem 8.1, then extends the argument to an arbitrary number of cores in Corollary 8.2.

Theorem 8.1 *When there are $p = 2$ cores, the competitive ratio of any lazy algorithm $\mathcal{A}$ is $\Omega(n^{1/2}/k)$ in terms of the number of faults.*

Proof Sketch. The adversary constructs an input formed by requesting pages from two disjoint sets of k "red" (r) and k "blue" (b) pages over ϕ rounds ($\phi \approx \sqrt{n}/2$). The requests made by each core in each round forms an "easy phase" followed by a "hard phase". Each phase is formed by exactly ℓ requests ($\ell \approx \sqrt{n}$).

Each phase has a color that is associated with the color of most requested pages in that phase. Easy and hard phases of P_1 are all respectively blue and red, while easy and hard phases of P_2 are all respectively red and blue. With the exception of the first phase of P_1 and the last phase of P_2, any phase in each core gets a "partner" phase of the same color in the other core. Specifically, the ith easy phase of P_1 is partnered with the $(i-1)$th hard phase of P_2, and the ith easy phase of P_2 is partnered with the ith hard phase of P_1.

The adversary defines an input such that there exists an (honest but not lazy) offline algorithm OFF that serves the requests in partner phases at the same time.

[1] A multicore caching algorithm is not competitive if its competitive ratio depends on n, the length of the input.

[2]We adopt the k-server definition of lazy algorithms [257]. Lazy algorithms are often called "demand paging" in the systems literature [318]. Algorithms with properties 1-3 (but not necessarily 4) are called "honest" algorithms [251].

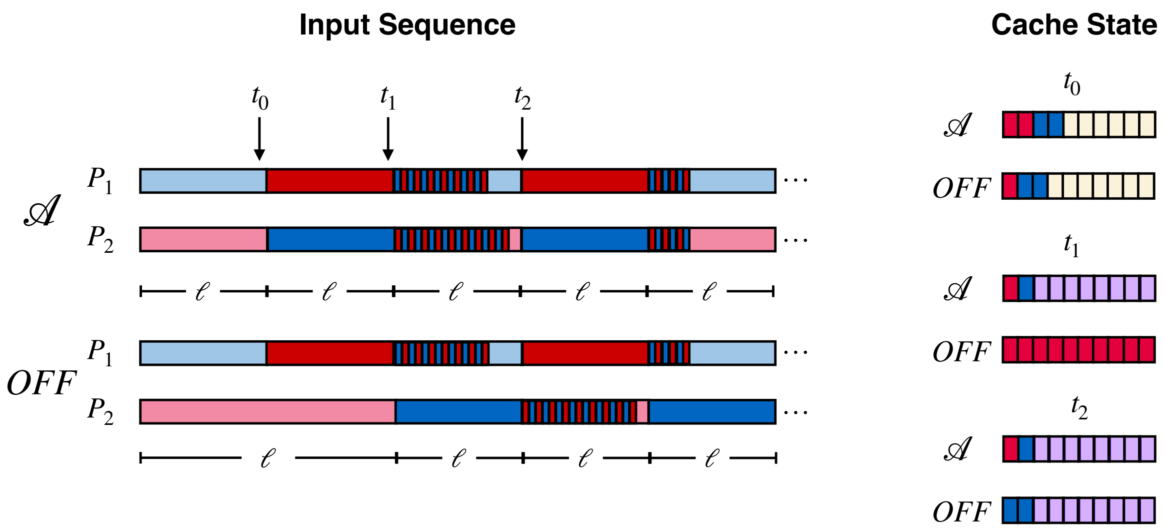


Figure 8-1: An example of the alignment of $\mathcal{A}$ and OFF described in Theorem 8.1 of two cores. For each sequence, easy phases consist of cycles of requests to two distinct pages (stripes of red and blue) followed by requests to a single page (light red and light blue), while hard (dark red and dark blue) phases are adversarial and designed so $\mathcal{A}$ faults on every request.
Left: $\mathcal{A}$ serving all cores in their easy and hard phases at the same time while OFF delays P_2.
Right: Examples of the cache state with $k = 10$ at each of the timesteps marked t_0, t_1, t_2. At t_0, the yellow cells represent empty cells. At t_1 after the first red hard phase, $\mathcal{A}$ also finishes its first blue hard phase while OFF finishes the first blue easy phase. The light purple cells are those that could either be red or blue. At timestep t_2, $\mathcal{A}$ has at least one of each color page in its cache, whereas OFF might only have blue pages.

Figure 8-1 illustrates the alignment of the two cores in $\mathcal{A}$ and OFF as well as the cache state in selected timesteps.

Next, we formalize how the adversary makes requests in each phase. Every phase has exactly ℓ page requests. Easy phases are formed by requests to only two pages. The first phase of P_1 differs from the rest of the easy phases because it does not have a partner and requests two red pages in a loop. All later easy phases are defined based on the decisions of $\mathcal{A}$ during their partner hard phases. Let $Q_{e,i}$ be the ith easy phase of color $c \in \{r, b\}$ and let $q_i^c \leq k$ denote the number of requests to unique pages made in $Q_{e,i}$'s corresponding hard phase $Q_{h,i}$. For all i, the adversary generates $Q_{e,i}$ by repeating requests to two pages of different colors followed by requests to one page of the same color of the phrase. The initial two pages are arbitrarily selected from the set of red/blue pages in the $\mathcal{A}$'s cache just before the start of $Q_{e,i}$. At the beginning of $Q_{e,i}$, the two red/blue pages are requested one after the other for the first q_i^c requests, while the remaining $\ell - q_i^c$ requests are to the single page that has color c. The two pages that form the requests of an easy phase are always in the cache of $\mathcal{A}$ at the beginning of the phase. Moreover, $\mathcal{A}$ serves the easy phases of P_1 at the same time as easy phases of P_2 and incurs all hits for both cores (with the exception of the cold misses in the first easy phase).

Hard phases are made by requesting the pages that are absent from $\mathcal{A}$'s cache.

At the beginning of any hard phase, at least one page of each color is present in the cache of $\mathcal{A}$. Because there are k pages of each color, the adversary generates the hard phase of a given color by always asking for some absent page of that color and $\mathcal{A}$ incurs a fault on every request of a hard phase. The inductive construction ensures the ith request of the two cores are served at the same time. In particular, the hard phases start at the same time and hence there will be congestion in the cache during the hard phases. The number of faults by $\mathcal{A}$ is at least equal to the total length of hard phases. There are $\Theta(\sqrt{n})$ hard phases, each of length $\Theta(\sqrt{n})$, for a total cost of $\mathcal{A}(\mathcal{R}) = \Theta(n)$.

An offline algorithm OFF can schedule the input such that partner easy and hard phases are served at the same time. This is done by “postponing" the first easy red phase (the first phase of P_2) by always evicting the page of the phase that is present in the cache before fetching the other page of the phase. Meanwhile, OFF hits on all requests in the first easy blue phase (the first phase of P_1) except the first two cold misses (it simply brings the two pages into the cache). After the first easy blue phase ends, OFF serves the remainder of the first easy red phase together with its partner, the first red hard phase, in a way that they end at the same time. This scheme also applies for other phases, i.e., any phase starts and ends with its partner. To maintain this alignment, OFF ensures there is a fault in the easy phase for each fault in the hard phase. When a page is requested in the hard phase for the first time, OFF keeps it in the cache until the end of the phase. Therefore, there is a fault in a hard phase only when a page is requested for the first time during the phase. The first q_i^c requests of the partner of the phase (an easy phase) are to two different pages of different colors. Upon a fault in the hard phase, OFF evicts one of these pages. This ensures the next request is a fault in the easy phase. In particular, in the very last fault of the hard phase (after q_i^c requests), OFF evicts the page that has color other than c. Consequently, in the remainder of these two partnered phases, all requested pages are in the cache and OFF hits on all of the remaining requests.

Analysis. The total number of faults by OFF in the two partner phases will be no more than $2k$ (up to k for each). There are $\phi - 1 = \Theta(\sqrt{n})$ pairs of partner phases for a total of $\Theta(k\sqrt{n})$ faults. The first phase of P_1 and the last phase of P_2 have length ℓ and OFF incurs no more than $\ell = \Theta(\sqrt{n})$ faults in them. In conclusion, the total number of faults in $\mathcal{A}$ and OFF are respectively $\Theta(n)$ and $\Theta(k\sqrt{n})$, which gives a competitive ratio of $\Omega(n^{1/2}/k)$ for $\mathcal{A}$. □

For the case of arbitrary $p > 2$, we can partition the cores into two disjoint groups, assign to the two colors, and repeat the input from Theorem 8.1 across cores to get the following corollary.

Corollary 8.2 *For any number of cores, the competitive ratio of any lazy algorithm $\mathcal{A}$ is $\Omega(n^{1/2}/k)$.*

8.3 Conclusion

This chapter showed that no lazy algorithm is competitive because the adversary has the power to artificially delay sequences. The scheduling power of OPT in multicore paging motivates the need for alternative measures of online algorithms such as the one in Chapter 7.

Locality-first strategy. The lower bounds in this chapter inspire the further study in Chapter 7 of inputs with locality of reference. These lower bounds show the equivalence of a large class of online algorithms when considering the entire space of inputs, but the proofs depend on a highly constructed input. In reality, access patterns exhibit naturally-occurring locality, motivating the need for algorithms to take advantage of that locality.

Chapter 9

Closing the Gap Between Cache-Oblivious and Cache-Adaptive Analysis

This chapter mathematically grounds the locality-first strategy via beyond-worst-case analysis of cache-adaptive algorithms. Specifically, it closes the gap between cache-oblivious and cache-adaptive analysis by showing how to make a smoothed analysis of cache-adaptive algorithms via random reshuffling of memory fluctuations due to parallelism. These results validate the locality-first strategy because cache-oblivious algorithms take advantage of temporal locality asymptotically optimally in a fixed-size cache. Therefore, optimizing algorithms for locality is a good strategy for adapting to memory fluctuations.

This work was conducted in collaboration with Michael A. Bender, Rezaul A. Chowdhury, Rathish Das, Rob Johnson, William Kuszmaul, Andrea Lincoln, Quanquan C. Liu, and Jayson Lynch [41].

Abstract

Cache-adaptive analysis was introduced to analyze the performance of an algorithm when the cache (or internal memory) available to the algorithm dynamically changes size. These memory-size fluctuations are, in fact, the common case in multicore machines, where threads share cache and RAM. An algorithm is said to be efficiently cache-adaptive if it achieves optimal utilization of the dynamically changing cache.

Cache-adaptive analysis was inspired by cache-oblivious analysis. Many (or even most) optimal cache-oblivious algorithms have an (a, b, c)-regular recursive structure. Such (a, b, c)-regular algorithms include longest common subsequence, all pairs shortest paths, matrix multiplication, edit distance, Gaussian elimination paradigm, etc. Bender *et al.* [43] showed that some of these optimal cache-oblivious algorithms remain optimal even when cache changes size dynamically, but that in general they can be as much as logarithmic factor away from optimal. However, their analysis depends on constructing a highly structured, worst-case memory profile, or sequences of fluctuations in cache size. These worst-case profiles seem fragile, suggesting that the logarithmic gap may be an artifact of an unrealistically powerful adversary.

This chapter closes the gap between cache-oblivious and cache-adaptive analysis by showing how to make a smoothed analysis of cache-adaptive algorithms via random reshuffling of memory fluctuations. Remarkably, it also shows the limits of several natural forms of smoothing, including random perturbations of the cache size and randomizing the algorithm's starting time. Nonetheless, it shows that if one takes an arbitrary profile and performs a random shuffle on "significant events" occur within the profile, then the shuffled profile becomes optimally cache-adaptive in expectation, even when the initial profile is adversarially constructed.

These results suggest that cache-obliviousness is a solid foundation for achieving cache-adaptivity when the memory profile is not overly tailored to the algorithm structure.

9.1 Introduction

On multi-threaded and multicore systems, the amount of cache available to any single process constantly varies over time as other processes start, stop, and change their demands for cache. On most multicore systems, each core has a private cache and the entire system has a cache shared between cores. A program's fraction of the private cache of a core can change because of time-sharing, and its fraction of the shared cache can change because multiple cores use it simultaneously [94, 122, 123].

Cache-size changes can be substantial. For example, threads in a shared cache frequently exhibit a winner-take-all phenomenon, in which one process grows to monopolize the available cache [132]; researchers have suggested periodically flushing the cache to counteract this effect [374]. With this policy, individual processes would experience cache allocations that slowly grow to the maximum possible size, then abruptly crash down to 0.

Furthermore, even small cache-size changes can have catastrophic effects on the performance of algorithms that are not designed to handle them. When the size of cache shrinks unexpectedly, an algorithm tuned for a fixed-size cache can thrash, causing its performance to drop by orders of magnitude. (And if the cache grows, an algorithm that assumes a fixed cache size can leave performance on the table.)

This is such an important problem that many systems provide mechanisms to manually control the allocation of cache to different processes. For example, Intel's Cache Allocation Technology [281] allows the OS to limit each process's share of the shared processor cache. Linux's cgroups mechanism [267] provides control over each application's use of RAM (which serves as a cache for disk). Although these mechanisms can help avoid thrashing, they require manual tuning and can leave cache underutilized. Furthermore, systems may be forced to always leave some cache unused in order to be able to schedule new jobs as they arrive.

A more flexible approach is to solve this problem in the algorithms themselves. If algorithms could gracefully handle changes in their cache allocation, then the system could always fully utilize the cache. Whenever a new task arrives, the system could reclaim some cache from the running tasks and give it to the new task, without causing catastrophic slowdowns of the older tasks. When a task ends, its memory could be distributed among the other tasks on the system. The OS could also freely redistribute cache among tasks to prioritize important tasks.

Practitioners have proposed many algorithms that heuristically adapt to cache fluctuations [80, 167, 272, 273, 295, 296, 400–402]. However, designing algorithms with guarantees under cache fluctuations is challenging and most of these algorithms have poor worst-case performance.

Theoretical approaches to adaptivity. Barve and Vitter [32, 33] initiated the theoretical analysis of algorithms under cache fluctuations over twenty years ago by generalizing the external-memory/disk-access machine (DAM) model [3] to allow the cache size to change. They gave optimal algorithms under memory fluctuations for sorting, FFT, matrix multiplication, LU decomposition, permutation, and buffer trees. In their model, the cache can change size only periodically and algorithms know the future size of the cache and adapt explicitly to these forecasts.

Writing programs and analyzing algorithms that explicitly adapt to changing memory is complicated because the algorithm needs to pay attention to the changing parameter of cache sizes. Moreover, it's hard to have performance guarantees that apply to all possible ways that memory can change size. Thus, most prior work by practitioners is empirical without guarantees, and even the Barve and Vitter work only has guarantees for a restricted class of memory profiles, which limits its generality.

More recently, Bender *et al.* [43,45] proposed using techniques from cache-oblivious algorithms to solve the adaptivity problem. Since cache-oblivious algorithms are oblivious to the size of the cache, it is compelling that the algorithms should work well when the cache size changes dynamically. They defined the cache-adaptive model, which is akin to the ideal-cache model [153, 154] from cache-oblivious analysis, except that the size of memory can change arbitrarily over time. They showed that many cache-oblivious algorithms remain optimal even when the size of cache changes dynamically. However, they also showed that some important cache-oblivious algorithms are *not* optimal in the cache-adaptive model.

Concretely, they define optimality in terms of how much progress an algorithm makes under a given ***memory profile*** and they also show that only a restricted class of memory profiles need to be considered. A memory profile $m(t)$ is a function specifying the size of memory at each time t. Prior results show that, for cache-oblivious algorithms, and up to a constant factor of resource augmentation, we need only consider ***square profiles***, i.e., memory profiles which can be decomposed into a sequence of boxes $(\square_1, \square_2, \ldots)$, where a box of size x means that memory remains at size x blocks for x time steps. In its strongest form, cache adaptivity requires that for an algorithm $\mathcal{A}$, the total amount of progress that $\mathcal{A}$ makes on a series of boxes $(\square_1.\square_2, \ldots)$ should be within a constant factor of the total potential $\sum_i \rho(|\square_i|)$ of those boxes, where the potential $\rho(|\square_i|)$ of a box $\square_i$ is defined to be the maximum number of operations that $\mathcal{A}$ could possibly perform in $\square_i$, where the max is taken over all possible places that $\square_i$ could occur in the execution of $\mathcal{A}$.[1]

[1]Several variations on this definition have also been used [43, 45] when considering particular problems (e.g., matrix multiplication). For (a, b, c)-regular algorithms, which are the focus of this

Thus, Bender *et al.*'s results show that cache-obliviousness is a powerful technique for achieving adaptivity without the burden of having to explicitly react to cache-size changes. They define optimality in terms of worst-case, adversarial memory profiles, which makes their optimality criteria very strong, but also tough to meet. It's natural to ask how algorithms perform under less adversarial profiles. This is important because for a large class of cache-oblivious algorithms, there exists highly structured and pernicious worst-case profiles on which the algorithms do *not* run optimally.

Cache-oblivious algorithms and (a, b, c)-regularity. One of the fundamental insights in the design of cache-oblivious algorithms is that, by using certain basic recursive structures in the design of an algorithm, one can get optimal cache-obliviousness for free. The algorithms with this recursive structure are known as ***(a, b, c)-regular algorithms***. If an algorithm is (a, b, c)-regular, its I/O-complexity satisfies a recurrence of the form $T(N) = aT(N/b) + \Theta(1 + N^c/B)$, where B is the block size of the cache and $\Theta(1 + N^c/B)$ represents the cost of scanning an array of size N^c.

For the purposes of cache-adaptivity, the only interesting cases are when $a > b$ and $c \leq 1$.[2] When $a > b$, an algorithm's performance is sensitive to the size of the cache, and so adaptivity becomes important.

If cache-oblivious algorithms were always cache-adaptive, then we could view adaptivity as a solved problem. Unfortunately, this is not the case. Bender *et al.* showed that, for $a > b$, (a, b, c)-regular algorithms are adaptive if and only if $c < 1$.

The worst-case gap between obliviousness and adaptivity. When $c = 1$, there can be a logarithmic gap between an algorithm's performance in the ideal cache and cache-adaptive models.[3]

Although many classical cache-oblivious algorithms are (a, b, c)-regular [100, 101, 153, 156, 298, 307, 359], many notable algorithms, including cache-oblivious dynamic programming algorithms [100], naive matrix multiplication [153], sub-cubic matrix multiplications (e.g., Strassen's algorithm [351]), and Gaussian elimination [100], have $a > b$ an $c = 1$ and hence fall into this gap. These algorithms are kernels of many other algorithms, including algorithms for solving linear equations in undirected and directed Laplacian matrices (see e.g., [138, 228, 323]), APSP [326, 333, 406], triangle counting [58], min-weight cycle [384], negative triangle detection and counting [384], and the replacement paths problem [384]. As we shall see in Chapter 10, some algorithms can be rewritten to reduce c, making them adaptive, but the transformation is complex, introduces overhead, and doesn't work for all algorithms.

The goal of this chapter is to show that this gap closes under less stringent notions of optimality.

chapter, the used definitions are equivalent (and thus, as a convention, we use the most general definition).

[2]When $a < b$, the algorithm runs in linear time independent of the cache size, and hence is trivially cache-adaptive. We are not aware of any cache-oblivious algorithms with $c > 1$.

[3](a, b, c)-regular algorithms are cache-adaptive when $a < b$ or $c < 1$. When $a = b$ and $c = 1$, no algorithm can be optimally cache-adaptive because such algorithms are already a $\Theta(\log \frac{M}{B})$ factor away from optimal in the DAM model [119]. This is why two-way merge sort, classic (i.e., not cache-oblivious) FFT, etc. cannot be optimal DAM algorithms.

Beyond the worst-case gap. Previous analysis shows that in the worst case there is a gap between the cache-adaptive and ideal-cache/cache-oblivious models. However, the logarithmic gap may just be an artifact of an unrealistically powerful adversary. The proof depends on exhibiting worst-case memory profiles that force the algorithm to perform poorly. The worst-case profiles mimic the recursive structure of the algorithm and maintain a tight synchronization between the algorithm's execution and the fluctuations in memory size. A concrete example of the worst-case profile for matrix multiplication can be found in Appendix B.1.

The natural question to ask is: what happens to these bad examples when they get tweaked in some way, so that they no longer track the program execution so precisely? Is this gap robust to the inherent randomness that occurs in real systems?

Contributions

The main result in this chapter shows that, given any probability distribution Σ over box-sizes, if each box has size chosen i.i.d. from the distribution Σ, (a, b, c)-regular algorithms achieve optimal performance in the cache-adaptive model, matching their performance in the ideal cache model.

Theorem 9.11 *Consider an (a, b, c)-regular algorithm $\mathcal{A}$, where $a > b$ are constants in $\mathbb{N}$ and $c = 1$. Let Σ be a probability distribution over box sizes, and consider a sequence of boxes $(\square_1, \square_2, \square_3, \ldots)$ drawn independently from the distribution Σ. If all boxes in Σ are sufficiently large in $\Omega(1)$, then $\mathcal{A}$ is cache-adaptive in expectation on the random sequences $(\square_1, \square_2, \ldots)$.*

Proving this requires a number of new combinatorial ideas, an overview of which appear in Section 9.3. Section 9.4 formally proves this positive result.

The proof begins by reinterpreting cache-adaptivity in expectation in terms of the expected stopping time of a certain natural martingale. We then show a relationship between the expected stopping time for a problem and the expected stopping times for the child subproblems. A key obstacle, however, is that the linear scans performed by the algorithm can cause the natural recurrence on stopping times to break. In particular, the recurrence is able to relate the time to complete subproblems (including scans) and the time to complete their parent problems (excluding scans); but is unable to consider the parent problems in their entirety (including scans). We fill in this gap by showing that the total effect of the scans at all levels of the recursion on the expected stopping time is at most a constant factor. By analyzing the aggregate effect of scans across all levels of the recursion, we get around the fact that certain scans at specific levels can have far more impact on the expected stopping time than others.

Robustness to weaker smoothings. We further show that drawing box-sizes independently from one-another is necessary in the sense that several weaker forms of smoothing fail to close the logarithmic gap between the ideal-cache and cache-adaptive models. We show that worst-case profiles are robust to all of the following perturbations: randomly tweaking the size of each box by a constant factor, randomly shifting the start time of the algorithm, and randomly (or even adversarially) altering the recursive structure of the profile. These results are proven in Section 9.5.

These smoothings substantially alter the overall structure of the profile, and eliminate any initial alignment between the program and the structure of the memory profile. Nonetheless, the smoothed profiles remain worst-case in expectation. That is, as long as some recursive structure is maintained within the profile, the algorithm is very likely to gradually synchronize its execution to the profile in deleterious ways. In this sense, even a small amount of global structure between the sizes of consecutive boxes is enough to cause the logarithmic gap.

Map. This chapter is organized as follows. Section 9.2 gives the definitions and conventions used in the rest of the chapter. Section 9.3 explains the intuition for the proofs of the main theorems and sketch the combinatorial ideas behind the proofs. Section 9.4 presents the main positive result. Section 9.5 provides worse-case profiles based on three natural ways to smooth/perturb/randomize the fluctuations in cache sizes. Section 9.6 gives an in-depth examination of previous work, and Section 9.7 gives concluding remarks.

9.2 Preliminaries

This section provides necessary preliminaries that will be used to prove the main results in later sections as well as in Chapter 10. Specifically, this section will introduce the cache-adaptive model that accounts for memory fluctuations and demonstrate how to analyze algorithms in the cache-adaptive model. Finally, it will define the probabilistic measures used in the smoothed analysis in this chapter and define the "No-catch-up lemma," a primitive used throughout this section in the proofs.

Cache-adaptive analysis

The cache-adaptive model. The ***cache-adaptive*** (CA) model [43,45] extends the classical disk access model (DAM) [3] to allow for the cache to change in size in each time step. In the DAM, the machine has a two-level memory hierarchy consisting of a fast ***cache*** (sometimes also referred to as ***memory*** or ***RAM***) of size M words and a slow ***disk***. Data is transferred between cache and disk in blocks of size B. An algorithm's running time is precisely the number of block transfers that it makes. Similarly, each I/O is a unit of time in the CA model.

In the cache-adaptive model, the size of fast memory is a (nonconstant) function $m(t)$ giving the size of memory (in blocks) after the tth I/O. We use $M(t) = B \cdot m(t)$ to represent the size, in words, of memory at time t. We call $m(t)$ and $M(t)$ ***memory profiles*** in blocks and words, respectively. Although the cache-adaptive model allows the size of cache to change arbitrarily from one time-step to the next, prior work showed that we need only consider ***square profiles*** [43]. Throughout this chapter, we use the terms ***box*** and ***square*** interchangeably.

Definition 9.1 (Square Profile [43]) *A memory profile M or m is a **square profile** if there exist boundaries $0 = t_0 < t_1 < \ldots$ such that for all $t \in [t_i, t_{i+1})$, $m(t) = t_{i+1} - t_i$. In other words, a square memory profile is a step function where each step is exactly as long as it is tall. We will use the notation $(\square_1, \square_2, \ldots)$ to refer to a profile in which the i-th square is size $|\square_i|$.*

For convenience, we assume that cache is cleared at the start of each square. The paging results underlying cache-adaptivity [45] explain that this assumption is w.l.o.g. For completeness, we show in Appendix B.7 that in proving either optimality or non-optimality it suffices to consider only these profiles. With this assumption, an algorithm gets to reference exactly X distinct blocks in a square of size X. Any memory profile can be approximated with a square profile up to constant factors [43]. So, notably, any random distribution of generically produced profiles has a corresponding random distribution over square profiles that approximates it.

Recursive algorithms with (a, b, c)-regular structure. This chapter and Chapter 10 focus on algorithms with a common recursive structure.

Definition 9.2 *Let $a, b \in \mathbb{N}$ be constants, $b > 1$ and $c \in [0, 1]$. An **(a, b, c)-regular algorithm** is a recursive algorithm that, when run on a problem of size n blocks (equivalently $N = nB$ words), satisfies the following:*

- *On a problem of size n blocks, the algorithm accesses $\Theta(n)$ distinct blocks.*
- *Until the base case (when $n \in \Theta(1)$), each problem of size b blocks recurses on exactly a subproblems of size n/b.*
- *Within any non-base-case subproblem, the only computation besides the recursion is a **linear scan of size N^c/B**. This linear scan is any sequence of N^c contiguous memory accesses satisfying the property that a cache of a sufficiently large constant size can complete the sequence of accesses in time $O(N^c/B)$. Parts of the scan may be performed before, between, and after recursive calls.*

Remark 9.3 *When referring to the size of a subproblem, box, scan, etc., we use blocks, rather than machine words, as the default unit. We define (a, b, c)-regular algorithms to have a base case of size $O(1)$ blocks. This differs slightly from previous definitions [43, 45] which recurse down to $O(1)$ words.*

Remark 9.4 *The definition of linear scans ensures the following useful property. Consider a linear scan of size N^c/B. Consider any sequence of squares $(\square_1, \square_2, \ldots, \square_j)$, where each $|\square_i|$ is a sufficiently large constant, and where $\sum_{i=1}^{j} |\square_i| = \Omega(N^c/B)$, for a sufficiently large constant in the Ω. Then the sequence of squares can be used to complete the scan in its entirety.*

The following theorem gives a particularly simple rule for when an (a, b, c)-regular algorithm is optimal.

Theorem 9.5 (**(a,b,c)-regular optimality [43], informal**) *Suppose $\mathcal{A}$ is an (a,b,c)-regular algorithm that is optimal in the DAM model. Then $\mathcal{A}$ is optimal in the cache-adaptive model if $c<1$ or if $b>a$ and $a\ge 1$. If $c=1$ and $a\ge b$, then $\mathcal{A}$ is $O(\log_b N)$ away from optimal on an input of size N in the cache-adaptive model. Optimality is defined as in [45], or equivalently as given by the notion of* efficiently cache adaptive*, defined below.*

Chapter goal: closing the logarithmic gap. The above theorem uses a very structured memory profile in the case of $c=1$ and $a\ge b$ to tease out the worst possible performance of $(a,b,1)$-regular algorithms. We explore the smoothing of these profiles when $a>b$ in this chapter. We leave the case of $a=b$ for future work because we prioritize the broader class of algorithms described by $a>b$.

Progress bounds in the cache-adaptive model. When an (a,b,c)-regular algorithm $\mathcal{A}$ is run on a square profile, the ***progress*** of a box is the number of base-case subproblems performed (at least partly) within the box. Define the ***potential*** $\boldsymbol{\rho(|\square|)}$ of a box of size $|\square|$ to be the maximum possible progress that a size $|\square|$ box could ever make starting at any memory access of any execution of $\mathcal{A}$ on a problem of arbitrary size.

Lemma 9.6 *The potential of a box $\square$ for an (a,b,c)-regular algorithm $\mathcal{A}$ is $\rho(|\square|)=\Theta(|\square|^{\log_b a})$.*

Proof. A square $\square$ can complete any subproblem A whose size in blocks is sufficiently small in $\Theta(|\square|)$. This allows $\square$ to complete $\Omega(a^{\log_b |\square|})=\Omega(|\square|^{\log_b a})$ base-case subproblems, which proves that $\rho(|\square|)\ge\Omega(|\square|^{\log_b a})$.

On the other hand, a square $\square$ is unable to complete in full any subproblem A whose size in blocks is sufficiently large in $\Theta(|\square|)$. It follows that $\square$ can complete base cases from at most two such subproblems A (the one that $\mathcal{A}$ begins $\square$ in and the one that $\mathcal{A}$ ends $\square$ in). This limits the number of base cases completed to $\rho(|\square|)\le O(|\square|^{\log_b a})$. $\square$

Intuitively, the potential of a box $\square$ is essentially the same as the number of base-case recursive leaves in a problem of size $|\square|$.

Optimality in the cache-adaptive model. The progress of each square is upper bounded by its potential. An execution of the algorithm $\mathcal{A}$ on a problem of size n blocks and on a square profile M is ***efficiently cache-adaptive*** if the sequence of squares $(\square_1,\square_2,\dots,\square_j)$ used by the algorithm (with the final square rounded down in size to be only the portion of the square actually used) satisfies

$$\sum_{i=1}^{j}\rho(|\square_i|)\le O(n^{\log_b a}). \tag{9.1}$$

The right-hand side of the expression represents the total amount of progress that must be made by any (a,b,c)-regular algorithm on a problem of size n in order to complete. In summary, an execution is efficiently cache-adaptive on the profile if every square in the profile makes progress asymptotically equal to its maximum possible potential.

An algorithm $\mathcal{A}$ (rather than just a single execution) is ***efficiently cache-adaptive*** (or ***cache-adaptive*** for short) if every execution of $\mathcal{A}$ is efficiently cache-adaptive on every infinite square-profile consisting of squares that are all of sizes sufficiently large in $O(1)$.

By Lemma 9.6, Inequality 9.1 can be written as $\sum_{i=1}^{j} |\square_i|^{\log_b a} \le O(n^{\log_b a})$. Since all squares $\square_i$ completed by the algorithm are of size $O(n)$, an equivalent requirement is

$$\sum_{i=1}^{j} \min(n, |\square_i|)^{\log_b a} \le O(n^{\log_b a}). \tag{9.2}$$

This requirement has the advantage that the size of the final square $\square_j$ cannot affect the veracity of the condition. Consequently, when using this version of the condition, we may feel free to *not* round down the size of the final square $\square_j$ in the profile M.

The definition of efficiently cache-adaptive is easily adapted to use an arbitrary progress function and an arbitrary algorithm $\mathcal{A}$ that need not be (a, b, c)-regular.[4]

Beyond-worst-case cache-adaptive analysis

Cache-adaptivity over distributions of profiles. We now define what it means for an algorithm to be cache-adaptive in expectation over a distribution of memory profiles. This definition underlies the smoothed and average-case analyses in this chapter.

Definition 9.7 (Efficiently cache-adaptive in expectation) *Let $\mathcal{M}$ be a distribution over (infinite) square memory profiles. Let M be a square-profile $(\square_1, \square_2, \ldots)$ drawn from the distribution $\mathcal{M}$, and define $\mathcal{S}_n$ to be the number of squares in the profile required by an (a, b, c)-regular $\mathcal{A}$ to complete on any problem of size n. We say that $\mathcal{A}$ is **(efficiently) cache-adaptive in expectation on $\mathcal{M}$** if for all problem sizes n,*

$$\mathbb{E}\left[\sum_{i=1}^{\mathcal{S}_n} \min(n, |\square_i|)^{\log_b a}\right] = O(n^{\log_b a}).$$

The bulk of this chapter is devoted to investigating which memory-profile distributions cause $(a, b, 1)$-regular algorithms to be cache-adaptive in expectation [45].

A useful lemma. We conclude the section by presenting a useful lemma, known as the ***No-catchup Lemma***, that is implicitly present in [45]. The lemma will be used

[4]There is an alternative progress function based on operations. Consider the progress function in which each square makes progress equal to the number of memory accesses it completes. This generalizes our definition to non(a, b, c)-regular algorithms and, for many natural (a, b, c)-regular algorithms, this yields the same definition of cache-adaptivity as the above progress definition. However, the memory-access-based definition of progress can differ from our definition if some large scans are very non-homogeneous, i.e. if they contain portions in which a single small box can complete a large number of memory accesses. We use the sub-problem-based definition so that our results can apply to all (a, b, c)-regular algorithms.

as a primitive throughout the chapter, and for completeness, a full proof is given in Appendix B.5. Intuitively, the No-catchup Lemma states that delaying the start time of an algorithm can never help it finish earlier than it would have without the start time delay.

Lemma 9.8 *Let $\sigma = (r_1, r_2, r_3, \ldots)$ be a sequence of memory references, and let $S = (\square_1, \square_2, \ldots \square_k)$ be a sequence of squares. Suppose that if $\square_1$ starts at r_i, then $\square_k$ finishes at r_j. Then, for all $i' < i$, if $\square_1$ starts at $r_{i'}$, then for some $j' \leq j$, $\square_k$ finishes at $r_{j'}$.*

9.3 Technical overview

This section overviews the main results in this chapter and sketches their proofs. Specifically, it outlines the main smoothing result as well as other negative results from different types of shuffling.

Cache-adaptivity on randomly shuffled profiles

The main technical result of the chapter is that random shuffling of adversarially constructed box-profiles makes (a, b, c)-regular algorithms where $a > b$ and $c = 1$ cache-adaptive in expectation. We use box profiles because previous work [45] showed that only considering box profiles is sufficient for determining cache-adaptivity on all valid profiles in the CA model (for details, see Appendix B.7). In Section 9.4 we prove the following:

Theorem 9.9 (Informal) *Consider an (a, b, c)-regular algorithm $\mathcal{A}$, where $a > b$ ($b > 1$) are constants in $\mathbb{N}$ and $c = 1$. Let Σ be a probability distribution over box sizes, and consider a sequence of boxes $(\square_1, \square_2, \square_3, \ldots)$ with sizes drawn independently from the distribution. Then $\mathcal{A}$ is cache-adaptive (in expectation) on the random sequence of boxes $(\square_1, \square_2, \square_3, \ldots)$.*

For simplicity, we discuss here the case where the block size $B = 1$, and $\mathcal{A}$ has the same values of a, b, c as the not-in-place naïve matrix-multiplication algorithm[5], with $a = 8$ and $b = 4$ and $c = 1$. Moreover, we assume that all box sizes and problem sizes are powers of 4. Additionally, we consider a simplified model of caching: any box of size s that begins in a subproblem of size s or smaller completes to the end of the problem of size s containing it (and goes no further); and any box of size s that begins in the scan of a problem of size greater than s continues either for the rest of the scan or for s accesses in the scan, whichever is shorter. As a final simplification, we assume that each scan in each problem of some size s consists of exactly s memory accesses. In fact, we show in Section 9.4 that these simplifications may be made without loss of generality for arbitrary (a, b, c)-regular algorithms.

Let $\square$ denote a single box drawn from the distribution Σ. The proof of Theorem 9.9 begins by applying the Martingale Optional Stopping Theorem to combinatorially

[5] A full algorithm description can be found in Appendix B.2.

reinterpret what it means for $\mathcal{A}$ to be cache-adaptive in expectation on the random sequence $(\square_1, \square_2, \ldots)$. In particular, if $f(n)$ is the expected number of boxes needed for $\mathcal{A}$ to complete a problem of size n, then cache-adaptivity-in-expectation reduces to:

$$f(n) \leq \frac{O(8^{\log_4 n})}{m_n} = \frac{O(n^{\log_4 8})}{m_n} = \frac{O(n^{3/2})}{m_n}, \tag{9.3}$$

where $m_n = \mathbb{E}\left[\min(n, |\square|)^{3/2}\right]$ is the ***average n-bounded potential*** of a box.

At the heart of the proof of Theorem 9.9 is a somewhat unintuitive combinatorial argument for expressing $f(n)$, the expected number of boxes needed to complete a problem of size n, in terms of $f(n/4)$.

Lemma 9.10 (Stated for the simplified assumptions) *Define* $p = \Pr[|\square| \geq n] \cdot f(n/4)$. *Then, the expected number of squares to complete the subproblems in a problem of size n is exactly* $\sum_{i=1}^{8}(1-p)^{i-1} f(n/4)$, *and the expected number of additional squares needed to complete the final scan is* $(1-\Theta(p)) \cdot \frac{\Theta(n)}{\mathbb{E}[\min(|\square|, n)]}$.

Proof Sketch. When executing a problem of size n, the first subproblem requires $f(n/4)$ boxes to complete, on average. Define q to be the probability the boxes used to complete the first subproblem include a box of size n or larger. Then with probability q, no additional boxes are required[6] to complete the rest of the problem of size n. We will show that $q = p$ later in this proof. Otherwise, an average of $f(n/4)$ additional boxes are needed to complete the next subproblem of size $n/4$. Again, with probability q, one of these boxes completes the rest of the problem in its entirety. Similarly, the probability that the i-th subproblem is completed by a large box from a previous subproblem is $(1-q)^{i-1}$. Thus the expected number of boxes needed to complete all 8 subproblems is

$$\sum_{i=1}^{8}(1-q)^{i-1} f(n/4). \tag{9.4}$$

Remarkably, the probability q can also be expressed in terms of $f(n/4)$. Define S to be the random variable for the number of boxes used to complete the first subproblem of size $n/4$; define $\ell \leq O(n^{3/2})$ to be an upper bound for S; and define X to be the random variable for the number of the boxes in the subsequence $\square_1, \ldots, \square_S$ that are of size n or greater. The expectation of X can be expressed as

$$\mathbb{E}[X] = \sum_{i=1}^{\ell} \Pr[S \geq i] \cdot \Pr[|\square_i| \geq n \mid S \geq i].$$

Since $|\square_i|$ is independent of $|\square_1|, \ldots, |\square_{i-1}|$, we have that $\Pr[|\square_i| \geq n \mid S \geq i] =$

[6]This is due to the aforementioned simplified caching model.

$\Pr[|\Box_i| \geq n]$. Thus

$$\mathbb{E}[X] = \Pr[|\Box| \geq n] \cdot \sum_{i=1}^{\ell} \Pr[S \geq i] = \Pr[|\Box| \geq n] \cdot \mathbb{E}[S] = \Pr[|\Box| \geq n] \cdot f(n/4).$$

Notice, however, that at most one of the boxes $\Box_1, \ldots, \Box_S$ can have size n or larger (since such a box will immediately complete the subproblem). Thus X is an indicator variable, meaning that $q = \Pr[X \geq 1] = \mathbb{E}[X] = \Pr[|\Box| \geq n] \cdot f(n/4) = p$. So $q = p$, as promised. Expanding Equation 9.4, we get that the expected number of boxes needed to complete the 8 subproblems is, as desired, at most

$$\sum_{i=1}^{8} \Big(1 - \Pr[|\Box| \geq n] \cdot f(n/4)\Big)^{i-1} f(n/4) \tag{9.5}$$

Next we consider the boxes needed to complete the final scan. Suppose the scan were to be run on its own. Let K denote the number of boxes needed to complete it, and let $\Box'_1, \ldots, \Box'_K$ denote those squares.

Rather than consider $\mathbb{E}[K]$ directly, we instead consider $\mathbb{E}[K] \cdot \mathbb{E}[\min(|\Box|, n)]$. Through a combinatorial reinterpretation, we have

$$\begin{aligned} E[K] \cdot \mathbb{E}[\min(|\Box|, n)] &= \mathbb{E}[\min(|\Box|, n)] \cdot \sum_{i=1}^{\ell} \Pr[K \geq i] \\ &= \sum_{i=1}^{\ell} \Pr[K \geq i] \cdot \mathbb{E}[\min(|\Box'_i|, n) \mid K \geq i] = \mathbb{E}\left[\sum_{i=1}^{K} \min(|\Box'_i|, n)\right]. \end{aligned}$$

The quantity in the final expectation has the remarkable property that it is *deterministically* between n and $2n - 1$. Thus the same can be said for its expectation, implying that $\mathbb{E}[K] \cdot \mathbb{E}[\min(|\Box|, n)] = \Theta(n)$.

Recall that K is the expected number of boxes to complete the scan on its own. In our problem, the scan is at the end of a problem, and thus with probability $1 - (1 - p)^8 = \Theta(p)$, the scan is completed by a large box from one of the preceding subproblems. Hence the expected number of additional boxes to complete the scan is $(1 - \Theta(p)) \cdot \frac{\Theta(n)}{\mathbb{E}[\min(|\Box|, n)]}$.

□

Lemma 9.10 suggests a natural inductive approach to proving Theorem 9.9. Rather than explicitly showing that $f(n) \leq \frac{O(n^{3/2})}{m_n}$, one could instead prove the result by induction, arguing for each n that

$$\frac{f(n)}{f(n/4)} \leq \frac{n^{3/2}/m_n}{(n/4)^{3/2}/m_{n/4}} = 8 \cdot \frac{m_{n/4}}{m_n}. \tag{9.6}$$

One can construct example box-size distributions Σ showing that the Equation (9.6) does not always hold, however. In particular, the scan at the end of a subproblem of size n can make $f(n)$ sufficiently larger than $f(n/4)$ that Equation (9.6) is violated. To get around this problem, one could attempt to instead prove that

$$\frac{f'(n)}{f(n/4)} \leq 8 \cdot \frac{m_{n/4}}{m_n}, \tag{9.7}$$

where $f'(n)$ is the expected number of boxes needed to complete a problem of size n, without performing the final scan at the end. Unlike Equation (9.6), Equation (9.7) does not inductively imply a bound of the form $f(n) \leq \frac{O(n^{3/2})}{m_n}$, which is necessary for cache-adaptivity in expectation. If one additionally proves that

$$\prod_{4^k \leq n} \frac{f(4^k)}{f'(4^k)} \leq O(1), \tag{9.8}$$

then Equation (9.8) could be used to "fill in the holes in the induction" in order to complete a proof of cache-adaptivity. Equation (9.8) is somewhat unintuitive in the sense that individual terms in the product can actually be as large as $1 + \Omega(1)$. The inequality states that, even though the scans in an individual subproblem size could have a significant impact on $f(n)$, the aggregate effect over all sizes is no more than constant.

To make this semi-inductive proof structure feasible, one additional insight is necessary. Rather than proving Equation (9.7) for all values n, one can instead restrict oneself only to values n such that

$$f(n) \geq C \cdot \frac{n^{3/2}}{m_n}, \tag{9.9}$$

where C is an arbitrarily large constant of our choice. In particular, if n_0 is the largest power of 4 less than our problem-size such that $f(n_0) < C \cdot \frac{n^{3/2}}{m_n}$, then we can use cache-adaptivity within problems of size n_0 as a base case, and then prove Equation (9.7) only for problem-sizes between n_0 and n. Similarly, we can restrict the product in Equation (9.8) to ignore problem sizes of size smaller than n_0.

When Equation (9.9) holds, Equation (9.7) can be interpreted as a negative feedback loop, saying that as we look at problem sizes $n = 1, 4, 16, \ldots$, whenever $f(n)$ becomes large enough to be on the cusp of violating cache-adaptivity, there exists downward pressure (in the form of Equation (9.7)) that prevents it from continuing to grow in an unmitigated fashion.

The full proof of Theorem 9.9 takes precisely the structure outlined above. At its core are the combinatorial arguments used in Lemma 9.10, which allow us to recast $f(n)$ and $f'(n)$ in terms of $f(n/4)$ and $f'(n/4)$. When applied in the correct manner, these arguments can be used to show Equation (9.7) (assuming Equation (9.9)) with only a few additional ideas. The proof of Equation (9.8) ends up being somewhat more sophisticated, using combinatorial ideas from Lemma 9.10 in order to expand each of the terms $f(4^k)/f'(4^k)$, and then relying on a series of subtle cancellation arguments in order to bound the end product by a constant.

Robustness of worst-case profiles

Section 9.5 considers three natural forms of smoothing on worst-case profiles. Remarkably, the worst-case nature of the profiles persists in all three cases. The canonical worst-case profile is highly structured, giving the algorithm a larger cache precisely when the algorithm does not need it. It is tempting to conclude that bad profiles must remain closely synchronized with the progression of the algorithm. By exploiting self-symmetry within worst-case profiles as well as the power of the No-catchup Lemma, the results in Section 9.5 establish that this is not the case. The No-Catchup Lemma, in particular, allows us to capture the idea of an algorithm resynchronizing with a profile, even after the profile has been perturbed.

We begin Section 9.5 by defining a canonical (a, b, c)-regular algorithm $\mathcal{A}_n$ on problems of size n, and a corresponding worst-case profile $M_{a,b}$. The profile $M_{a,b}$ completes each scan of size k in $\mathcal{A}$ in a single box of size k, thereby ensuring that every box makes its minimum possible progress. The profile $M_{a,b}$ is the ***limit profile*** of the sequence of profiles $M_{a,b}(n)$ for $n = 1, b, b^2, \ldots$, constructed recursively by defining $M_{a,b}(n)$ by concatenating together a copies of $M_{a,b}(n/b)$ and then placing a box of size n at the end. The algorithm $\mathcal{A}_n$ requires the entirety of the profile $M_{a,b}(n)$ to complete. One can check inductively that $M_{a,b}(n)$ has total potential $n^{\log_b a} \cdot \log n$, thereby making $M_{a,b}$ a worst-case profile.

We present examples of smoothing a worst-case profile in Figure B-1.

Box-size perturbations. Consider any probability distribution $\mathcal{P}$ over $[0, t]$ for $t \leq \sqrt{n}$ such that for X drawn at random from $\mathcal{P}$, $\mathbb{E}[X] = \Theta(t)$. Let $X_1, X_2, \ldots$ be drawn iid from $\mathcal{P}$ and define $\mathcal{M}$ to be the distribution over box profiles obtained by replacing each box $\square_i$ in $\mathcal{M}$ with a box of size $|\square_i| \cdot X_i$. In Section 9.5.1, we show that the highly perturbed box profiles in $\mathcal{M}$ still remain worst-case in expectation.

The proof takes two parts. We begin by defining T to be the smallest power of b greater than t, and considering the profile $T \cdot M_{a,b}$ obtained by multiplying each box's size by T. Exploiting self-symmetry in the definition of $M_{a,b}$, we are able to reinterpret $T \cdot M_{a,b}$ as the profile $M_{a,b}$ in which all boxes of size smaller than T have been *removed*. Recall that $M_{a,b}(n)$ denotes the prefix of $M_{a,b}$ on which $\mathcal{A}_n$ completes. Using the fact that $T \leq \sqrt{n}$, we prove that the boxes of size smaller than T contain at most a constant fraction of the potential in the prefix $M_{a,b}(n)$. On the other hand, by iterative applications of the No-Catchup Lemma, the removal of the boxes cannot facilitate $\mathcal{A}$ to finish earlier in the profile. Combining these facts, we establish that $T \cdot M_{a,b}$ remains worst-case.

To obtain an element of $\mathcal{M}$ from $T \cdot M_{a,b}$, one simply multiplies the size of each box $\square_i$ in $T \cdot M_{a,b}$ by X_i/T, where T is drawn from the distribution $\mathcal{P}$. Using that $\mathbb{E}[X_i/T] = \Theta(1)$ and that $n^{\log_b a}$ is a convex function, Jensen's inequality tells us that the expected potential of the new box of size $\frac{|\square_i| \cdot X_i}{T}$ is at least a constant fraction of the original potential. Since the perturbations preserve the expected potentials of the boxes in $T \cdot M_{a,b}$ up to a constant factor, we can prove that the resulting profile is worst-case in expectation by demonstrating that the perturbations do not result in $\mathcal{A}_n$ finishing earlier in $T \cdot M_{a,b}$ then it would have otherwise. Since each perturbation can only reduce the size of a box in $T \cdot M_{a,b}$, this can be shown by iterative applications of the No-Catchup Lemma.

Start-time perturbations. Section 9.5.2 analyzes what happens if the memory profile $M_{a,b}(n)$ is cyclic-shifted by a random amount. This corresponds to executing $\mathcal{A}_{a,b}(n)$ starting at a random start-time in the cyclic version of $M_{a,b}(n)$. Again, the resulting distribution of profiles remains worst-case in expectation.

The key insight in the proof is that $M_{a,b}(n)$ can be expressed as the concatenation of two profiles $A = (\square_1, \ldots, \square_x)$ and $B = (\square'_1, \ldots, \square'_y)$ such that

$$\sum_{i=1}^{x} |\square_i| \geq \Omega\left(\sum_{i=1}^{y} |\square'_i|\right), \tag{9.10}$$

$$\sum_{i=1}^{x} |\square_i|^{\log_b a} \leq O\left(\sum_{i=1}^{y} |\square'_i|^{\log_b a}\right). \tag{9.11}$$

Equation (9.10) establishes that with at least constant probability, a random selected start-time in the profile $M_{a,b}(n)$ falls in the prefix A. By a slight variant on the No-Catchup Lemma, if $\mathcal{A}$ is executed starting at that random start-time, it is guaranteed to use all of the boxes in the suffix B. By Equation (9.11), however, these boxes contain a constant fraction of the potential from the original worst-case profile $M_{a,b}(n)$. Thus, with constant probability the algorithm $\mathcal{A}$ runs at a random start-time that results in a profile that is still worst-case. This ensures that the randomly shifted profile will be worst-case in expectation.

Box-order perturbations. The bad profile, $M_{a,b}$, is constructed recursively by making a copies of $M_{a,b}(n/b)$ followed by a box of size n. The box comes at the end, intuitively, because all $(a, b, 1)$-regular algorithms with upfront scans in each subproblem can converted to an equivalent $(a, b, 1)$-regular algorithm, where the scans in all subproblems are at the end, preceded by a single linear scan[7].

Section 9.5.3 considers a relaxation of the construction of $M_{a,b}$. When constructing $M_{a,b}(n)$ recursively, rather than always placing a box of size n after the *final* instance of $M_{a,b}(n/b)$, we instead allow ourselves to place the box of size n after *any* of the a recursive instances of $M_{a,b}(n/b)$ (each of which may no longer be identical to the others due to the non-determinedness of the new recursive construction).

Although at first glance moving the largest box in the profile seems to closely resemble the random shuffling considered in Section 9.4, we prove that the resulting distribution over box profiles again remains worst-case in expectation. In fact, we can prove a stronger statement: for memory profile M drawn from the resulting distribution $\mathcal{M}$ of box profiles, M is a worst-case profile with probability *one*.

To prove this, we begin by constructing what we call a ***universal worst-case profile*** $U_{a,b}$. The prefixes $U_{a,b}(n)$ of the profile $U_{a,b}$ are recursively constructed in the same manner as $M_{a,b}$, except with the following twist: rather than concatenating together a copies of $U_{a,b}(n/b)$ with a single box of size n at the end, we instead concatenate together a copies of $U_{a,b}(n/b)$ with a box of size n at the end of each

[7]The details can be found in Lemma B.3 in Appendix B.6.

copy. Exploiting self-symmetry in the construction of $M_{a,b}$, we show that $U_{a,b}$ is also a worst-case profile.

Each box profile in the distribution $\mathcal{M}$ can be obtained from $U_{a,b}$ by removing a $\frac{a-1}{a}$ fraction of the boxes from $U_{a,b}$. By iterative applications of the No-Catchup Lemma, such removals cannot facilitate the algorithm $\mathcal{A}_{a,b}$ to finish earlier in the profile than it would have otherwise. On the other hand, because an $\frac{1}{a}$-fraction of the boxes of each size remain after the removals, the total potential in each prefix $U_{a,b}(n)$ of $U_{a,b}$ is affected only by a constant factor by the removals. Thus all box profiles from the distribution $\mathcal{M}$ is guaranteed to still be worst-case.

9.4 Cache-adaptivity of randomly shuffled profiles

Consider an (a, b, c)-regular algorithm, where $a > b$ are constants in $\mathbb{N}$ and $0 < c \leq 1$. Let Σ be a probability distribution over box sizes, and consider a sequence of boxes $(\square_1, \square_2, \square_3, \ldots)$ with sizes drawn independently from the distribution Σ. (Note that each $|\square_i|$ is a random variable.) This section proves that for any such algorithm, and for any distribution of box sizes Σ, the algorithm will be cache-adaptive (in expectation) on the random sequence of boxes $(\square_1, \square_2, \ldots)$.

Theorem 9.11 *Consider an (a, b, c)-regular algorithm $\mathcal{A}$, where $a > b$ are constants in $\mathbb{N}$ and $c = 1$. Let Σ be a probability distribution over box sizes, and consider a sequence of boxes $(\square_1, \square_2, \square_3, \ldots)$ drawn independently from the distribution Σ. If all boxes in Σ are sufficiently large in $\Omega(1)$, then $\mathcal{A}$ is cache-adaptive in expectation on the random sequences $(\square_1, \square_2, \ldots)$.*

Remark 9.12 *Theorem 9.11 does not hold in the case of $a = b$. Consider, for example, the case of $a = b = 2$ and $c = 1$ (e.g., mergesort) with block size $B = 1$. Moreover, suppose each scan in each problem of size s occurs at the end of the problem, and consists of exactly s distinct block accesses, one after another. Finally, suppose Σ contains only one box-size $\sqrt{n}$.*

Then the potential of a box is $O(\sqrt{n})$, since each subproblem of size $\Theta(\sqrt{n})$ contains $\Theta(\sqrt{n})$ recursive leaves. In order for cache-adaptivity to be achieved, it follows that $\mathcal{S}_n$ (which is deterministically determined since there is only one box-size) must be $O(\sqrt{n})$, that way $\mathcal{S}_n \cdot \Theta(\sqrt{n})$ will be within a constant factor of the total progress n to be made for the full problem. However, every scan of size $s \geq 2\sqrt{n}$ requires $\Omega(s/\sqrt{n})$ boxes devoted entirely to that scan in order to be completed. Since the sum of the sizes of the scans at each level of recursion is n, it follows that the sum of the sizes of all scans of size at least $2\sqrt{n}$ is $\Theta(n \log n)$. Hence the total number of boxes required is at least

$$\mathcal{S}_n \geq \Omega(n \log n) \cdot \frac{1}{\sqrt{n}} = \Omega(\sqrt{n} \log n),$$

meaning the algorithm is not cache-adaptive in expectation.

Before continuing, we introduce some notation. Throughout the section, let $\square$ denote a single box whose size is drawn from the distribution Σ. As a convention, we will use $\mathcal{S}_n$ to denote the index of the final box $\square_{\mathcal{S}_n}$ required for the algorithm to complete on an input of size n (in blocks). (If different inputs to problems of size n require different numbers of boxes, then we define $\mathcal{S}_n$ to be the maximum over all possible inputs.) The quantity $\mathcal{S}_n$ is sometimes also referred to as the ***stopping time***.

For each problem-size n, define the ***work function*** $W_n = n^{\log_b a}$, the number of base-case recursive leaves in a problem of size n. We define the ***n-bounded potential*** $m_n(s)$ of a box of size s to be $\min(W_n, W_s)$, corresponding (up to a constant factor) with the size of the largest subproblem that the box can complete within a problem of size n. We will also sometimes use m_n to denote $\mathbb{E}[m_n(|\square|)]$.

With this notation, cache-adaptivity in expectation reduces to the statement

$$\mathbb{E}\left[\sum_{i=1}^{\mathcal{S}_n} m_n(|\square_i|)\right] \leq O(W_n).$$

The remainder of the section is devoted to proving Theorem 9.11. Next, this section reduces the proof of Theorem 9.11 to a simpler, more specialized version of the theorem. This section concludes by proving the specialized version of the theorem.

A simplified problem

In this section, we present a series of simplifications to the problem of proving Theorem 9.11. Each simplification builds on the previous ones.

Recall that the algorithm $\mathcal{A}$ is cache-adaptive in expectation if, for all problem sizes n,

$$\mathbb{E}\left[\sum_{i=1}^{\mathcal{S}_n} m_n(|\square_i|)\right] \leq O(W_n). \tag{9.12}$$

We begin with a lemma that provides a simpler formulation of the quantity on the left side of Equation (9.12).

Lemma 9.13 *For any box-size distribution Σ, and any (a, b, c)-regular algorithm $\mathcal{A}$,*

$$\mathbb{E}\left[\sum_{i=1}^{\mathcal{S}_n} m_n(|\square_i|)\right] = \mathbb{E}[\mathcal{S}_n] \cdot m_n.$$

The lemma follows immediately from a specialized variant of the Martingale Optional Stopping Theorem [381]. We include the proofs of Lemma 9.13 and Theorem 9.14 in Appendix B.4.

Theorem 9.14 (Martingale Optional Stopping Theorem [381]) *Let $X_1, X_2, \ldots$ be iid random variables, and let γ be a function such that $\gamma(X_i)$ has finite mean μ. Consider an arbitrary process that runs in steps, and at each step i is given the value of X_i. Suppose that the process terminates after no more than C*

steps for some value C. Let S be the random variable denoting the number of steps that the process runs. Then,

$$\mathbb{E}\left[\sum_{i=1}^{S} \gamma(X_i)\right] = \mathbb{E}[S] \cdot \mu.$$

This brings us to our first simplification:

Simplification 9.15 *To prove cache-adaptivity on a problem of size n, it suffices to show that*

$$\mathbb{E}[\mathcal{S}_n] \leq O\left(\frac{W_n}{m_n}\right).$$

Our second simplification, which is a consequence of Lemma B.3 in Appendix B.6, has to do with the structure of scans within subproblems:

Simplification 9.16 *We may restrict ourselves to (a, b, c)-regular algorithms in which scans occur exclusively at the end of subproblems (rather than before or between recursions), with the exception of the largest subproblem, which may also perform scan work at the beginning of the algorithm.*

We will refer to the scan work at the beginning of the algorithm, due to the largest problem, as the ***upfront scan***. As a convention, when we refer to the ***scan*** in a given subproblem, we will by default always be referring to the scan at the end of the subproblem, and not including the upfront scan, even if the subproblem we are discussing is the full problem on which the algorithm is running.

The next observation has to do with what we call the hard-stopping rule, which can be applied to any (a, b, c)-regular algorithm $\mathcal{A}$ satisfying the property from Simplification 9.16 (that scans appear at the ends of subproblems, with the exception of an upfront scan). Let $q \in O(1)$ be the smallest positive power of b (depending on the(a, b, c)-regular algorithm $\mathcal{A}$) such that any subproblem of any size s (ignoring any upfront scan) accesses at most $q \cdot s$ distinct blocks, such that the scan at the end of any subproblem of any size s can be completed by any sequence of boxes (with sizes sufficiently large in $\Omega(1)$) whose sizes sum to at least $q \cdot s$, and such that the upfront scan in any problem of any size n can also be completed by any such sequence of boxes. Notice that this implies that any box of size qs or larger can complete any subproblem of size s (ignoring the upfront scan). The ***hard-stopping rule*** is a modification to the manner in which the algorithm $\mathcal{A}$ is executed on a problem of size n:

- The upfront scan is simulated as being of length exactly $q \cdot n$. The upfront scan is complete after l boxes for the smallest l such that $\sum_{i=1}^{l} |\square_i| \geq q \cdot n$. The l-th box is not allowed to continue past the upfront scan (i.e., the box is not allowed to perform any additional memory accesses after the scan), and the $(l + 1)$-th box begins immediately after the upfront scan.

- Once the upfront scan is complete, whenever a box of some size t is generated within a subproblem of size $\frac{t}{q}$ or smaller, the box continues to the end of the largest subproblem of size $\frac{t}{q}$ or smaller that contains the box. The box stops at the end of that subproblem (i.e., the next subproblem begins using the next box).

- Finally, scans at the end of subproblems of each size s are simulated to always be of length $q \cdot s$. In particular, if the scan is not completed by some box of size qs or larger (possibly generated within the scan), then the scan requires a sequence of boxes whose sizes sum to $q \cdot s$ or larger in order for the scan to complete. The final of these boxes stops at the end of the scan, and the next box generated then starts at the beginning of whatever follows the scan.

As permitted by Theorem 9.11, we will assume that boxes have sizes large enough in $\Omega(1)$ that whenever their sizes sum to $q \cdot s$, they can complete any scan in a problem of size s (or an upfront scan in a problem of size s). We will also assume, in general, that every box in the distribution Σ has size large enough to complete any base-case problem while following the hard-stopping rule. (That is, if the largest size a base-case problem can be is n_0, then every box has size at least $q \cdot n_0$.). Thus every time a box is drawn, its behavior is guaranteed to be covered by one of the cases described above. Note that these assumptions on box sizes, which we will call the ***base-box-size assumption***, is permitted by the statement of Theorem 9.11, which allows us to require box sizes be at least arbitrarily large constants.

The hard-stopping rule has a number of appealing properties. The number of squares needed to complete a given algorithm on a problem of size n becomes a function of n only (and is unaffected by the algorithm input). Moreover, boxes always either complete some subproblem in its entirety, or finish in the same scan in which they began.

Note that the hard-stopping rule, in general, only inhibits the work completed in a box. In particular, any box of size s is always capable of completing any subproblem of size s/q or smaller, and any sequence of boxes (with sizes sufficiently large in $\Omega(1)$) whose sizes sum to $q \cdot s$ or larger are always capable to completing a scan of size s. Thus the hard-stopping rule is a modification of the execution of the algorithm such that boxes are sometimes asked to complete before they would normally need to.

Intuitively, when an execution follows the hard-stopping rule, the number of boxes for the algorithm to complete cannot decrease. This can be formalized by repeated applications of the No-Catchup Lemma (Lemma B.2). In particular, each time that a box terminates early, rather than continuing, the No-Catchup Lemma tells us that the number of additional boxes needed to complete the problem (without following the hard-stopping rule) cannot decrease.

We use $\mathcal{S}_n^h$ as the random variable denoting the number of boxes needed to complete a problem of size n, given that the execution is following the hard-stopping rule. The key observation is that $\mathcal{S}_n^h \geq \mathcal{S}_n$. This brings us to our fifth simplification:

Simplification 9.17 *In order to prove that $\mathbb{E}[\mathcal{S}_n] \leq O(W_n/m_n)$, it suffices to prove that $\mathbb{E}[\mathcal{S}_n^h] \leq O(W_n/m_n)$, while making the base-box-size assumption.*

In our next simplification, we exploit the hard-stopping rule in order to remove upfront scans from the problem. In particular, consider an (a, b, c)-regular algorithm $\mathcal{A}$ on a problem of size n with scans only at the ends of subproblems, and one upfront scan at the beginning of the full problem. (Moreover, suppose $\mathcal{A}$ satisfies the base-box-size assumption when executed with the hard-stopping rule.) When bounding $\mathbb{E}[\mathcal{S}_n]$, we may assume without loss of generality that, in $\mathcal{A}$, the final scan at the end of the full problem contains as a suffix a copy of the upfront scan (since appending the upfront scan to the final scan can only increase $\mathbb{E}[\mathcal{S}_n]$). Moreover, rather then bounding $\mathbb{E}[\mathcal{S}_n]$, Simplification 9.17 allows us to instead focus on bounding $\mathbb{E}[\mathcal{S}_n^h]$ for $\mathcal{A}$. Let $\mathcal{B}$ be the same algorithm except with the scan at the front of $\mathcal{A}$ removed. (Notice that the value q used for $\mathcal{B}$ when following the hard-stopping rule will be the same as the value used for $\mathcal{A}$.) Any sequence of boxes which would complete B while following the hard-stopping rule would also complete the scan at the upfront of $\mathcal{A}$. (Indeed, even the boxes that complete the final scan in $\mathcal{B}$ suffice to complete the upfront scan of $\mathcal{A}$.) Thus if $\mathcal{S}_n^h$ denotes the number of boxes to complete $\mathcal{B}$ while following the hard-stopping rule, then the expected number of boxes to complete the upfront scan in $\mathcal{A}$ (while following the hard-stopping rule) is at most $\mathbb{E}[\mathcal{S}_n^h]$. The expected total number of boxes to complete $\mathcal{A}$ (while following the hard stopping rule) is therefore at most $2\mathbb{E}[\mathcal{S}_n^h]$. This brings us to our next simplification, which extends Simplification 9.17:

Simplification 9.18 *In order to prove that $\mathbb{E}[\mathcal{S}_n] \leq O(W_n/m_n)$ (i.e., prove Theorem 9.11), it suffices to consider only algorithms in which all scans (including those in the largest subproblem) occur exclusively at the end of their respective subproblems, and to then prove that $\mathbb{E}[\mathcal{S}_n^h] \leq O(W_n/m_n)$, while making the base-box-size assumption.*

Our next simplification is that we may restrict ourselves to problem sizes n which are powers of b. In particular, for any algorithm $\mathcal{A}$ run on a problem of size n that is not a power of b, we can define $r < b$ such that $n \cdot r$ is a power of b, and then define $\mathcal{B}$ to be the same algorithm except with the problem size of $\mathcal{B}$ formally defined to be r times the corresponding problem size in $\mathcal{A}$.[8] The value of $\mathcal{S}_n^h$ for $\mathcal{A}$ is the same as the values of $\mathcal{S}_{nr}^h$ for $\mathcal{B}$, and the values of m_n and W_n for $\mathcal{A}$ are within a constant factor of the values of m_{nr}, and W_{nr} for $\mathcal{B}$. Thus if we prove that $\mathbb{E}[\mathcal{S}_{nr}^h] \leq O(W_{nr}/m_{nr})$ for $\mathcal{B}$, then we will have proven that $\mathbb{E}[\mathcal{S}_n^h] \leq O(W_n/m_n)$ for $\mathcal{A}$.

Simplification 9.19 *We may further assume without loss of generality that all problem sizes are powers of b.*

It will be convenient for the value q used in the hard-stopping rule to always be $q = 1$. Next we show that this may be assumed without loss of generality. Consider an (a, b, c)-regular algorithm $\mathcal{A}$, with scans only at the ends of subproblems, which

[8]Note that $\mathcal{B}$'s base case for recursion happens on problems r times as large as in $\mathcal{A}$. This is acceptable since $r \leq O(1)$. Also note that if a distribution Σ of box-sizes for $\mathcal{A}$ satisfies the base-box-size assumption, then so will the same distribution for $\mathcal{B}$, since the base-case problems in both algorithms use the same block-access patterns.

is executed on a problem of size n using the hard-stopping rule with some value $q = t$ and using some sequence of box-sizes $(S_1, S_2, \ldots)$ satisfying the base-box-size assumption.

Then consider an (a, b, c)-regular algorithm $\mathcal{A}'$ which is executed on a problem of size nt using the hard-stopping rule with $q = 1$, and with base-case problem-size one. (Note that because q is always a power of b, multiplying n by q does not change whether n is a power of b.) The key insight is that, due to the hard-stopping rule, a given box will interact with problems of size s in the execution of $\mathcal{A}$, in precisely the same way that the same box interacts with problems of size $s \cdot t$ in the the execution of $\mathcal{A}'$. Hence the number of boxes used to complete the second execution will be precisely equal to the number used to complete the first.

Now let Σ be a distribution of box sizes satisfying the base-box-size assumption for $\mathcal{A}$. Suppose that $\mathcal{S}^h_{nt} \leq O(W_{nt}/m_{nt})$ for algorithm $\mathcal{A}'$ on distribution Σ. Since $t \in O(1)$, and since $\mathcal{S}^h_{nt}$ for $\mathcal{A}'$ equals $\mathcal{S}^h_n$ for $\mathcal{A}$, it follows that $\mathcal{S}^h_n \leq O(W_n/m_n)$ for algorithm $\mathcal{A}$ on distribution Σ. Hence we have the following simplification:

Simplification 9.20 *We may further assume without loss of generality that $q = 1$. Additionally, we may assume that the base-case problem size is 1.*

For the rest of the section, since $q = 1$, we will adapt the convention of saying that the ***size of a scan*** is simply the same as the size of the subproblem it is in (i.e., the size of a scan in a subproblem of size l is simply l).

Simplifications 9.19 and 9.20 combine to give us one final simplification: that without loss of generality, all box sizes are powers of b. In particular, since the problem sizes are powers of b, and since $q = 1$, the hard-stopping rule does not distinguish between boxes of different sizes in the range $[b^r, b^{r+1})$. Rounding each box size down to the nearest power of b changes m_n by at most a constant factor and does not change $\mathbb{E}[\mathcal{S}^h_n]$ or W_n. Thus we get our final simplification:

Simplification 9.21 *We may further assume without loss of generality that all box sizes are powers of b.*

In light of Simplifications 9.20 and 9.21, the hard-stopping rule has a very simple interpretation: any box of size s which is started in a problem of size s or smaller continues to the end of whatever problem of size s it's in; and a box of size s started in the scan of size $l > s$ finishes within the same scan, and completes the scan if and only if the sum of the sizes of the boxes started within the scan sum to l or greater.

Combining these simplifications, we get that in order to prove Theorem 9.11, it suffices to prove the following specialized version:

Theorem 9.22 *Let $a > b$ be constants in $\mathbb{N}$ and consider $c \in [0, 1]$. Consider an (a, b, c)-regular algorithm $\mathcal{A}$, in which all scans occur exclusively at the ends of subproblems, and suppose $\mathcal{A}$ is executed with the hard-stopping rule using $q = 1$. Then for any problem size n that is a power of b, and for any box-size distribution Σ in which all boxes have sizes that are powers of b,*

$$\mathbb{E}[\mathcal{S}^h_n] \leq O(W_n/m_n).$$

The remainder of the section is devoted to proving Theorem 9.22.

Proof of Theorem 9.22

We begin by introducing some additional notation. Let $f(n)$ denote $\mathbb{E}[\mathcal{S}_n^h]$, the expected number of boxes needed to complete a problem of size n (while following the hard-stopping rule). Note that $f(n)$ is well defined since the number of boxes needed to complete a problem while following the hard-stopping rule is a function of n only. In order to prove cache-adaptivity for problems of size n, we wish to prove that

$$f(n) \leq O(W_n/m_n). \tag{9.13}$$

(Recall that W_n is the work function, satisfying $W_n = n^{\log_b a}$, and m_n is the expectation of the n-bounded potential function, satisfying $m_n = \mathbb{E}[\min(W_n, W_{|\square|})]$, where $\square$ is a box of size drawn from the distribution Σ.)

Intuitively, Equation (9.13) says that the ***typical progress*** of a box is m_n, allowing for roughly W_n/m_n boxes to complete the problem, on average. Formally, we say that the ***typical progress*** of a box is given by

$$\phi(n) = W_n/f(n),$$

the amount of work in a problem of size n divided by the average number of boxes needed. In order to prove cache-adaptivity in expectation, we therefore wish to show that

$$\phi(n) \geq \Omega(m_n),$$

for all problem sizes n.

A natural approach might be to prove this by induction on n. (Recall that the eligible problem sizes n are the powers of b.) In particular, it would suffice to prove the relationship

$$\frac{\phi(n)}{\phi(n/b)} \geq \frac{m_n}{m_{n/b}}. \tag{9.14}$$

Unfortunately, Equation (9.14) does not always hold, as shown in the following example.

Example 9.23 *Suppose that the box sizes $|\square|$ take some value l deterministically. Then the expected number of boxes $f(l)$ to complete a problem of size l will be one. But the expected number of boxes $f(b \cdot l)$ to complete a problem of size $b \cdot l$ will be $a + b$. In particular, a boxes are used to solve the a subproblems, and b boxes are used for the scan. This means that $\phi(l \cdot b)/\phi(l)$ will be*

$$\frac{\phi(l \cdot b)}{\phi(l)} = \frac{W_{lb}/f(lb)}{W_l/f(l)} = \frac{W_{lb}}{W_l} \cdot \frac{f(l)}{f(lb)} = a \cdot \frac{1}{a+b} < 1.$$

Note that in the final inequality we use the useful fact that $\frac{W_{lb}}{W_l} = \frac{(lb)^{\log_b a}}{l^{\log_b a}} = a$.

On the other hand, m_l and m_{lb} will both be W_l (since all boxes are size l), meaning that $\frac{m_{lb}}{m_l} = 1$, and thereby violating Equation (9.14). The reason for the violation is intuitively that the long scan in the problem of size $b \cdot l$ causes the typical progress of a box to shrink (when compared to a problem of size b), while the average bounded potential is left unchanged.

The next lemma shows that Example 9.23 is extremal in the sense that one will always have $\frac{\phi(l \cdot b)}{\phi(l)} \geq \frac{a}{a+b}$.

Lemma 9.24 *For all problem sizes l, $\frac{f(l \cdot b)}{f(l)} \leq a + b$. Consequently, $\frac{\phi(l \cdot b)}{\phi(l)} = \frac{W_{lb}/f(lb)}{W_l/f(l)} \geq \frac{a}{a+b}$.*

Proof. Note that the expected number of boxes needed to complete a subproblems of size l is at most $a \cdot f(l)$. Moreover, a scan of size $b \cdot l$ can require at most b times as many boxes to complete (in expectation) as does a scan of size l. Since a subproblem of size l contains a scan of size l, it follows that a scan of size $b \cdot l$ requires at most $b \cdot f(l)$ boxes to complete (in expectation).[9] In total, we get that $f(l \cdot b) \leq af(l) + bf(l)$, as desired. □

To resolve the issue in Example 9.23, one can separate the analysis of each scan from the analysis of the subproblems preceding the scan. Define $f'(n)$ to be the expected number of boxes needed to complete a problems of size n/b one after another. (i.e., $f'(n)$ is the expected number of boxes to complete a problem of size n, while ignoring the final scan.) Similarly, define $\phi'(n) = W_n/f'(n)$. Rather than proving Equation (9.14), which Example 9.23 proves false, one could instead attempt to show that

$$\frac{\phi'(n)}{\phi(n/b)} \geq \frac{m_n}{m_{n/b}}. \tag{9.15}$$

Then, rather than individually considering $\phi(n)/\phi'(n)$ for each problem size n, one could instead bound the total impact of scans across all problem sizes on the typical progress function, proving that

$$\prod_{b^t \leq n} \phi(b^t)/\phi'(b^t) \geq \Omega(1). \tag{9.16}$$

In particular, the goal would be to prove that, although the scans at any particular level of recursion can bring down the typical progress of a box by as much as a constant factor, in aggregate the scans across all levels of recursion bring the typical progress of a box down by no more than a constant factor.

Combined, Equation (9.15) and Equation (9.16) would suffice for proving cache adaptivity in expectation. (In particular, when combined, they imply that $\phi(n) \geq \Omega(m_n)$. The approach we take differs from this in one additional important way. Rather than proving Equation (9.15) for all n (that are powers of b), we instead

[9]Here we are implicitly using the fact that for any sequence of boxes, the hard-stopping rule allows to complete a problem of size l, the hard-stopping rule would have also allowed them to complete a single scan of size l.

restrict ourselves to values of n for which $\phi(n/b)$ is on the cusp of being too small for cache adaptivity (meaning $\phi(n/b)/m_{n/b}$ is a sufficiently small constant). For these values, Equation (9.15) can be viewed as a sort of negative feedback loop, showing that whenever $\phi(n)/m_n$ starts to become small, there is upward pressure so that $\phi'(n \cdot b)/m_{n \cdot b}$ does not become even smaller. Formally, we will prove the following propositions, the proofs of which we will present in the coming subsections.

Proposition 9.25 *Consider a problem size $l > b$, and suppose l/b satisfies $\phi(l/b) < \frac{m_{l/b}}{4a}$. Then,*

$$\frac{\phi'(l)}{\phi(l/b)} \geq \frac{m_l}{m_{l/b}}.$$

Proposition 9.26 *Consider the largest problem size $l \leq n$ for which $\phi(l) \geq \frac{m_l}{4a}$. Then,*

$$\prod_{l < b^t \leq n} \phi(b^t)/\phi'(b^t) \geq \Omega(1).$$

In fact, we will not need Proposition 9.26 in its full strength. Rather, we will use the nearly equivalent fact that

$$\prod_{b \cdot l < b^t \leq n} \phi(b^t)/\phi'(b^t) \geq \Omega(1),$$

which is implied by Proposition 9.26 and the observation that $\phi(b \cdot l)/\phi'(b \cdot l) \leq 1$. Assuming the propositions, Theorem 9.22 can be proven as follows.
Proof. Suppose we wish to prove cache adaptivity on problems of size n. Consider the largest subproblem size l for which $\phi(l) \geq \frac{m_l}{4a}$. (Note that $l = 1$ necessarily satisfies this property since $\phi(1) = W_1/f(1) = 1$ and $\frac{m_1}{4a} = \frac{1}{4a}$.) If $l = n$, then $\phi(n) \geq \Omega(m_n)$, which proves cache adaptivity. On the other hand, if $l < n$, then we can express $\phi(n)$ as

$$\phi(n) = \phi(b \cdot l) \cdot \prod_{b \cdot l < b^t \leq n} \frac{\phi'(b^t)}{\phi(b^{t-1})} \cdot \frac{\phi(b^t)}{\phi'(b^t)}.$$

By Proposition 9.25 (which holds for all problem sizes greater than $b \cdot l$) and Proposition 9.26, this becomes

$$\begin{aligned}\phi(n) &\geq \Omega\left(\phi(b \cdot l) \cdot \prod_{b \cdot l < b^t \leq n} \frac{m_{b^t}}{m_{b^{t-1}}}\right) \\ &= \Omega\left(\phi(b \cdot l) \cdot \frac{m_n}{m_{b \cdot l}}\right).\end{aligned}$$

Recall that our goal is to establish that the typical progress $\phi(n)$ is at least $\Omega(m_n)$, the average n-bounded potential of a box. To complete the proof, it therefore suffices

Name	Symbol	Expansion
work function	W_n	$n^{\log_a b}$
n-bounded potential	m_n	$\mathbb{E}[\min(W_n, W_{\|\square\|})]$
expected stopping time	$f(n)$	–
expected stopping time ignoring final scan	$f'(n)$	–
typical progress	$\phi(n)$	$W_n/f(n)$
typical progress ignoring final scan	$\phi'(n)$	$W_n/f'(n)$

Figure 9-1: A reference table of the functions used to analyze cache-adaptivity in expectation.

to show that $\phi(b \cdot l) \geq \Omega(m_{bl})$. Recall that, by the definition of l, $\phi(l) = \frac{W_l}{f(l)} \geq \frac{m_l}{4a}$, meaning that $f(l) \leq 4a \cdot W_l/m_l$. Since $W_{bl} = a \cdot W_l$, and since $m_{bl} \leq a \cdot m_l$ (in particular, $\min(W_{|\square|}, W_l) \leq a \cdot \min(W_{|\square|}, W_{bl})$), it follows that $f(l) \leq 4a \cdot W_{lb}/m_{lb}$. By Lemma 9.24, we then get that $f(l \cdot b) \leq (a+b) \cdot 4a \cdot W_{bl}/m_{bl}$, and thus $\phi(bl) \geq \frac{m_{bl}}{4a \cdot (a+b)} \geq \Omega(m_{bl})$, as desired. □

The remainder of the section is devoted to proving Propositions 9.25 and 9.26. For reference by the reader, Figure 9-1 gives a table of the functions defined in this section that continue to be used in the proofs of the propositions.

Proof of Proposition 9.25

We begin by comparing the average l-bounded potential m_l to the average l/b-bounded potential $m_{l/b}$, assuming $\phi(l/b) < \frac{m_{l/b}}{4a}$.

Lemma 9.27 *Suppose l/b satisfies $\phi(l/b) < \frac{m_{l/b}}{4a}$. Then,*

$$\frac{m_l}{m_{l/b}} < 1 + \frac{\Pr[|\square| \geq l] \cdot f(l/b)}{4}.$$

Proof. The only box sizes s for which the l-bounded potential differs from the l/b-bounded potential are the sizes $s \geq l$. In particular,

$$m_l - m_{l/b} = \Pr[|\square| \geq l] \cdot (W_l - W_{l/b}) \leq \Pr[|\square| \geq l] \cdot W_l.$$

Hence

$$\frac{m_l - m_{l/b}}{m_{l/b}} \leq \frac{\Pr[|\square| \geq l] \cdot W_l}{m_{l/b}}.$$

Because $\phi(l/b) = W_{l/b}/f(l/b) < \frac{m_{l/b}}{4a}$, it follows that

$$\begin{aligned}\frac{m_l - m_{l/b}}{m_{l/b}} &< \frac{\Pr[|\square| \geq l] \cdot W_l}{4aW_{l/b}/f(l/b)} \\ &= \frac{\Pr[|\square| \geq l] \cdot aW_{l/b}}{4aW_{l/b}/f(l/b)} \\ &= \frac{\Pr[|\square| \geq l] \cdot f(l/b)}{4}.\end{aligned}$$

Adding one to both sides,

$$\frac{m_l}{m_{l/b}} < 1 + \frac{\Pr[|\square| \geq l] \cdot f(l/b)}{4}.$$

□

Next we compare the typical progress $\phi'(l)$ of a box in a problem of size l (ignoring the scan) to the typical progress $\phi(l/b)$ of a box in a problem of size l/b. Note that

$$\begin{aligned}\phi'(l)/\phi(l/b) &= \frac{W_l/f'(l)}{W_{l/b}/f(l/b)} \\ &= \frac{af(l/b)}{f'(l)}.\end{aligned}$$

Thus in order to prove a lower bound for $\phi'(l)/\phi(l/b)$ (i.e., to show that the typical progress has increased between problem sizes), it suffices to compare $f(l/b)$ and $f'(l)$.

Lemma 9.28 *Consider any problem size $l \geq b$. Then,*

$$f'(l) \leq a \cdot f(l/b) \cdot (1 - f(l/b) \cdot \Pr[|\square| \geq l]/2).$$

Recall that $f'(l)$ is the expected number of boxes needed to complete a consecutive problems of size l/b, while $f(l/b)$ is the expected number of boxes needed to complete a single such problem.

Proof. Consider what happens when the algorithm $\mathcal{A}$ is run on a problem of size l/b (while following the hard-stopping rule as in Theorem 9.22). Define p to be the probability that a box of size l or greater is generated during the problem. By the Martingale Optional Stopping Theorem (Theorem 9.14), the expected number of such boxes generated is $f(l/b) \cdot \Pr[|\square| \geq l]$. (In this application of Theorem 9.14, the variables X_i are defined to be $X_i = |\square_i|$, and $\gamma(X_i)$ is the indicator variable $\mathbb{I}(|\square_i| \geq l)$.) Note that no more than one such box can be generated, however, since any box of size l or greater will complete the problem of size l/b. Thus the probability

p of such a box being generated is equal to the expected number of such boxes, and

$$p = f(l/b) \cdot \Pr[|\square| \geq l].$$

With this in mind, we consider $f'(l)$, the expected number of boxes needed to complete a problems of size l/b. The first of the a subproblems will require $f(l/b)$ boxes to complete, in expectation. With probability p, one of these boxes will be of size at least l, and will thus complete the entire computation. Otherwise, the second of the a subproblems will then require $f(l/b)$ boxes to complete, in expectation. Again, with probability p, one of these boxes will be of size at least l, and thus complete the entire computation. Continuing like this, the probability that the i-th subproblem is handled by a large box from a previous subproblem is $1-(1-p)^{i-1}$, and thus the expected number of additional boxes needed to handle the i-th subproblem is $(1-p)^{i-1} \cdot f(l/b)$. Summing over the subproblems,

$$\begin{aligned} f'(l) &= \sum_{i=1}^{a} (1-p)^{i-1} \cdot f(l/b) \\ &\leq f(l/b) + \sum_{i=2}^{a} (1-p)^{i-1} \cdot f(l/b) \\ &\leq a \cdot f(l/b) \cdot (1 - p/2). \end{aligned}$$

Expanding p,

$$f'(l) \leq a \cdot f(l/b) \cdot (1 - f(l/b) \cdot \Pr[|\square| \geq l]/2),$$

as desired. □

Combining Lemmas 9.27 and 9.28, we can now complete the proof of Proposition 9.25.

PROOF. Recall that $\frac{\phi'(l)}{\phi(l/b)}$ expands to

$$\frac{\phi'(l)}{\phi(l/b)} = \frac{af(l/b)}{f'(l)}.$$

By Lemma 9.28, it follows that

$$\frac{\phi'(l)}{\phi(l/b)} \geq \frac{1}{(1 - f(l/b) \cdot \Pr[|\square| \geq l]/2)} \geq 1 + f(l/b) \cdot \Pr[|\square| \geq l]/2.$$

Comparing this to Lemma 9.27, we see that

$$\frac{\phi'(l)}{\phi(l/b)} \geq \frac{m_l}{m_{l/b}},$$

as desired. □

Proof of Proposition 9.26

Proposition 9.26 is essentially about bounding the impact of scans on typical progresses. We begin by considering the number of boxes needed to perform a scan.

Define $Q(b^t)$ to be the expected number of boxes needed to run a scan of size b^t on its own (while following the hard-stopping rule). A box of a given size s could potentially make progress as much as $\min(b^t, s)$ through the scan. Thus one might intuitively expect $Q(b^t)$ to be roughly $\frac{b^t}{\mathbb{E}[\min(b^t, |\square|)]}$. The following lemma proves this up to a constant factor.

Lemma 9.29

$$\frac{b^t}{\mathbb{E}[\min(b^t, |\square|)]} \leq Q(b^t) < \frac{2b^t}{\mathbb{E}[\min(b^t, |\square|)]}.$$

Proof. Consider the sequence of box sizes $A_1, A_2, \ldots, A_S$ needed to complete the scan. (Here S is a random variable.) For the box A_S to complete the scan, the progress across the entire sequence must be

$$\sum_{i=1}^{S} \min(b^t, |A_i|) \geq b^t.$$

Moreover, since each box A_i except the final box A_S makes progress exactly $|A_i| = \min(b^t, |A_i|)$ in the scan,

$$\sum_{i=1}^{S} \min(b^t, |A_i|) \leq b^t + \min(b^t, |A_S|) < 2b^t.$$

Since the quantity $\sum_{i=1}^{s} \min(b^t, |A_i|)$ is deterministically between b^t and $2b^t - 1$, its expectation must also be between b^t and $2b^t - 1$, satisfying

$$b^t \leq \mathbb{E}\left[\sum_{i=1}^{S} \min(b^t, |A_i|)\right] < 2b^t.$$

On the other hand, the Martingale Optional Stopping Theorem (Theorem 9.14) tells us that

$$\mathbb{E}\left[\sum_{i=1}^{S} \min(b^t, |A_i|)\right] = \mathbb{E}[S] \cdot \mathbb{E}[\min(b^t, |\square|)] = Q(b^t) \cdot \mathbb{E}[\min(b^t, |\square|)].$$

Hence,

$$\frac{b^t}{\mathbb{E}[\min(b^t, |\square|)]} \leq Q(b^t) < \frac{2b^t}{\mathbb{E}[\min(b^t, |\square|)]},$$

as desired. $\square$

We now complete the proof of Proposition 9.26.

Proof. In order to prove that

$$\prod_{l<b^t\leq n} \phi(b^t)/\phi'(b^t) \geq \Omega(1),$$

it suffices to prove that

$$\prod_{l<b^t\leq n} f(b^t)/f'(b^t) \leq O(1), \tag{9.17}$$

since $\phi(b^t)/\phi'(b^t) = f'(b^t)/f(b^t)$. We can restate Equation (9.17) as

$$\prod_{l<b^t\leq n} \left(1 + \frac{f(b^t) - f'(b^t)}{f'(b^t)}\right) \leq O(1).$$

Consider the scan at the end of a problem of size b^t. Let p the probability that a box of size at least b^t is generated during one of the a subproblems of size b^{t-1}. Then with probability p, the box generated during the a subproblems will complete the scan. On the other hand, with probability $(1-p)$, the scan will require additional boxes. It follows by Lemma 9.29 that

$$f(b^t) - f'(b^t) = (1-p) \cdot Q(b^t) < (1-p) \cdot \frac{2b^t}{\mathbb{E}[\min(b^t, |\square|)]}.$$

To prove the proposition, it therefore suffices to show that

$$\prod_{l<b^t\leq n} \left(1 + (1-p) \cdot \frac{2b^t}{\mathbb{E}[\min(b^t, |\square|)] \cdot f'(b^t)}\right) \leq O(1).$$

Equivalently, we wish to prove that

$$\sum_{l<b^t\leq n} \ln\left(1 + (1-p) \cdot \frac{2b^t}{\mathbb{E}[\min(b^t, |\square|)] \cdot f'(b^t)}\right) \leq O(1).$$

Since $\ln(1+x) \leq x$ for all $x \geq 0$, we may instead prove that

$$\sum_{l<b^t\leq n} \left((1-p) \cdot \frac{2b^t}{\mathbb{E}[\min(b^t, |\square|)] \cdot f'(b^t)}\right) \leq O(1). \tag{9.18}$$

Let use take a moment to solve for p. Recall that within a problem of size b^t, p is the probability that a box of size at least b^t is generated during any one of the a subproblems of size b^{t-1}. By the Martingale Optional Stopping Theorem (Theorem 9.14), the expected number of such boxes generated is given by $\Pr[|\square| \geq b^t] \cdot f'(b^t)$. Moreover, at most one such box can be generated (since it will then complete the problem of size b^t). Hence the number of such boxes is an indicator variable, and the probability p of such a box being generated is also $\Pr[|\square| \geq b^t] \cdot f'(b^t)$. Expanding p in Equation (9.18), the sum we wish to bound becomes

$$\sum_{l<b^t\leq n}\left((1-\Pr[|\square|\geq b^t]\cdot f'(b^t))\frac{2b^t}{\mathbb{E}[\min(b^t,|\square|)]\cdot f'(b^t)}\right)$$
$$=\sum_{l<b^t\leq n}\left(\frac{2b^t}{\mathbb{E}[\min(b^t,|\square|)]\cdot f'(b^t)}-\frac{2b^t\cdot\Pr[|\square|\geq b^t]}{\mathbb{E}[\min(b^t,|\square|)]}\right).$$

Next we focus on $f'(b^t)$. By the fact that $b^t > l$ and by the definition of l in the statement of the proposition, we know that $\phi(b^t)\leq\frac{m_{b^t}}{4a}$. Expanding $\phi(b^t)$, this means that $W_{b^t}/f(b^t)\leq\frac{m_{b^t}}{4a}$, and thus that

$$f(b^t)\geq\frac{4a\cdot W_{b^t}}{m_{b^t}}=\frac{4a^{t+1}}{m_{b^t}}. \tag{9.19}$$

In order to transform this into a statement about $f'(b^t)$, notice that $f'(b^t)\geq f(b^t)/2$. In particular, $f'(b^t)$ counts the number of boxes needed to complete a subproblems of size b^{t-1}. This includes a scans of size b^{t-1}. The expected number of boxes needed to complete these scans alone is at least the number of boxes needed to complete a scan of size $a\cdot b^{t-1}>b^t$. Since the hard-stopping rule guarantees that any sequence of boxes which completes the a subproblems could have also completed the a scans alone (without the additional portions of the subproblems), it follows that a subproblems of size b^{t-1} require, in expectation, at least as many boxes as a scan of length b^t, meaning that $f'(b^t)\geq f(b^t)-f'(b^t)$, and thus that $f'(b^t)\geq f(b^t)/2$. Combining this with Equation (9.19), we get that

$$f'(b^t)\geq\frac{2a^{t+1}}{m_{b^t}}.$$

Plugging this into our sum, it suffices to prove the bound

$$\sum_{l<b^t\leq n}\left(\frac{b^t\cdot m_{b^t}}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}-\frac{2b^t\cdot\Pr[|\square|\geq b^t]}{\mathbb{E}[\min(b^t,|\square|)]}\right)\leq O(1).$$

Define $r_{b^u}=\Pr[|\square|=b^u]$ to be the probability of a box taking size b^u. Our sum expands to

$$\begin{aligned}
&\sum_{l<b^t\le n}\left(\frac{b^t\cdot\left(\sum_{u<t} r_{b^u}W_{b^u}+\Pr[|\square|\ge b^t]\cdot W_{b^t}\right)}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}-\frac{2b^t\cdot\Pr[|\square|\ge b^t]}{\mathbb{E}[\min(b^t,|\square|)]}\right)\\
&=\sum_{l<b^t\le n}\left(\frac{b^t\cdot\left(\sum_{u<t} r_{b^u}a^u+\Pr[|\square|\ge b^t]\cdot a^t\right)}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}-\frac{2b^t\cdot\Pr[|\square|\ge b^t]}{\mathbb{E}[\min(b^t,|\square|)]}\right)\\
&=\sum_{l<b^t\le n}\left(\frac{b^t\cdot\sum_{u<t} r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}+\frac{b^t\cdot\Pr[|\square|\ge b^t]}{\mathbb{E}[\min(b^t,|\square|)]\cdot a}-\frac{2b^t\cdot\Pr[|\square|\ge b^t]}{\mathbb{E}[\min(b^t,|\square|)]}\right).
\end{aligned}$$

Using half of the third sum to dominate the second sum, this is at most

$$\sum_{l<b^t\le n}\left(\frac{b^t\cdot\sum_{u<t} r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}-\frac{b^t\cdot\Pr[|\square|\ge b^t]}{\mathbb{E}[\min(b^t,|\square|)]}\right). \tag{9.20}$$

Focusing on the positive summands,

$$\begin{aligned}
&\sum_{l<b^t\le n}\frac{b^t\cdot\sum_{u<t} r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}\\
&=\sum_{b^u<n}\ \sum_{b^u,l<b^t\le n}\frac{b^t\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}\\
&=\sum_{b^u\le l}\ \sum_{l<b^t\le n}\frac{b^t\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}+\sum_{l<b^u\le n}\ \sum_{b^u<b^t\le n}\frac{b^t\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}\\
&\le\sum_{b^u\le l}\ \sum_{l<b^t\le n}\frac{b^t\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}+\sum_{l<b^u\le n}\ \sum_{b^u<b^t\le n}\frac{b^t\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^u,|\square|)]\cdot a^{t+1}}\\
&=\sum_{b^u\le l}\ \sum_{l<b^t\le n}\frac{b^t\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}+\sum_{l<b^u\le n}\frac{b^u\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^u,|\square|)]\cdot a^{u+1}}\cdot\sum_{b^u<b^t\le n}(b/a)^{t-u}\\
&<\sum_{b^u\le l}\ \sum_{l<b^t\le n}\frac{b^t\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}+\sum_{l<b^u\le n}\frac{b^u\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^u,|\square|)]\cdot a^{u+1}}\cdot\sum_{s\ge 0}((a-1)/a)^s\\
&=\sum_{b^u\le l}\ \sum_{l<b^t\le n}\frac{b^t\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}+\sum_{l<b^u\le n}\frac{b^u\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^u,|\square|)]\cdot a^{u+1}}\cdot a\\
&=\sum_{b^u\le l}\ \sum_{l<b^t\le n}\frac{b^t\cdot r_{b^u}a^u}{\mathbb{E}[\min(b^t,|\square|)]\cdot a^{t+1}}+\sum_{l<b^u\le n}\frac{b^u\cdot r_{b^u}}{\mathbb{E}[\min(b^u,|\square|)]}.
\end{aligned}$$

Adding back in the negative terms from Equation (9.20), we get that Equation (9.20) is at most

$$\sum_{b^u \leq l} \sum_{l < b^t \leq n} \frac{b^t \cdot r_{b^u} a^u}{\mathbb{E}[\min(b^t, |\square|)] \cdot a^{t+1}} + \sum_{l < b^u \leq n} \frac{b^u \cdot r_{b^u}}{\mathbb{E}[\min(b^u, |\square|)]} - \sum_{l < b^t \leq n} \frac{b^t \cdot \Pr[|\square| \geq b^t]}{\mathbb{E}[\min(b^t, |\square|)]}.$$

Since $\Pr[|\square| \geq b^t] \geq r_{b^t}$, the third sum dominates the second sum, and thus we are left with at most

$$\sum_{b^u \leq l} \sum_{l < b^t \leq n} \frac{b^t \cdot r_{b^u} a^u}{\mathbb{E}[\min(b^t, |\square|)] \cdot a^{t+1}}.$$

Since $u < t$, we have $r_{b^u} b^u \leq \mathbb{E}[\min(b^t, |\square|)]$, and thus our expression is at most

$$\begin{aligned}
&\sum_{b^u \leq l} \sum_{l < b^t \leq n} \frac{b^{t-u} \cdot a^u}{a^{t+1}} \\
&\leq \sum_{b^u \leq l} \frac{1}{a} \cdot \left(\frac{b}{a}\right)^{-u} \cdot \sum_{l < b^t} \left(\frac{b}{a}\right)^{t}.
\end{aligned}$$

If we define k such that $l = b^k$, then the sum can be rewritten as

$$\begin{aligned}
&\sum_{u \leq k} \frac{1}{a} \cdot \left(\frac{b}{a}\right)^{k-u} \cdot \sum_{t > 0} \left(\frac{b}{a}\right)^{t} \\
&< \sum_{u \leq k} \frac{1}{a} \cdot \left(\frac{b}{a}\right)^{k-u} \cdot \sum_{t \geq 0} \left(\frac{a-1}{a}\right)^{t} \\
&= \sum_{u \leq k} \frac{1}{a} \cdot \left(\frac{b}{a}\right)^{k-u} \cdot a \\
&= \sum_{u \leq k} \left(\frac{b}{a}\right)^{k-u} = O(1),
\end{aligned}$$

which completes the proof. □

9.5 Robustness of worst-case profiles

This section considers three ways to smooth worst-case profiles: box-size perturbations, start-time perturbations, and box-order perturbations. In all three cases the smoothed profiles will remain examples of profiles on which any $(a, b, 1)$-regular algorithm (where $a > b$), $\mathcal{A}_{a,b}$, is not adaptive. We show that the canonical worst-case profiles are robust to these perturbations and not brittle, by exploiting self-symmetry within worst-case profiles and the power of the No-catchup Lemma. This is surprising because the canonical worst-case profile seems fragile—it gives $\mathcal{A}_{a,b}$ memory only when the algorithm can't use it, and gives as much memory as possible at those times.

Throughout the section, we examine a specific (a, b, c)-regular algorithm with constants $a, b \in \mathbb{N}$ satisfying $a > b$ and $c = 1$. We define the ***canonical $(a, b, 1)$-regular algorithm*** $\mathcal{A}_{a,b}(n)$ on problem-sizes n that are powers of b as follows: If $n > 1$, then the algorithm $\mathcal{A}_{a,b}(n)$ first recursively performs a subproblems of size n/b. Then (regardless of whether $n > 1$), the algorithm $\mathcal{A}_{a,b}(n)$ accesses each of the blocks $1, 2, \ldots, n$, one after another.

For n a power of b, the ***canonical worst-case profile*** $M_{a,b}(n)$ is the profile constructed so that each scan in the algorithm $\mathcal{A}_{a,b}$ will be covered by a box exactly the size of that scan (i.e., the number of block-accesses in the scan). In particular, $M_{a,b}(n)$ consists of a single box of size 1 when $n = 1$, and otherwise recursively consists of a instances of $M_{a,b}(n/b)$ followed by a box of size n. We define $M_{a,b}$ to be the infinite profile containing $M_{a,b}(n)$ as a prefix for each n that is a power of b.

We say that a box profile M is ***worst-case*** for the algorithm $\mathcal{A}_{a,b}(n)$ if the sequence of boxes used to complete $\mathcal{A}_{a,b}(n)$, $M = \square_1, \ldots, \square_k$, satisfies

$$\sum_{i=1}^{k} \min(n, |\square_i|)^{\log_b a} \geq \Omega(\log n) \cdot n^{\log_b a}.$$

In particular, such a profile M ensures that the algorithm $\mathcal{A}$ is at least an $\Omega(\log n)$ factor off from cache-adaptive, which by the results of [43] make M a worst-case profile (up to a constant factor). Similarly, we say that a probability distribution $\mathcal{M}$ on box profiles is ***worst-case (in expectation)*** for the algorithm $\mathcal{A}_{a,b}(n)$ if for M randomly selected from $\mathcal{M}$, the sequence of boxes $(\square_1, \ldots, \square_k)$ used to complete M (note that now k is a random variable) satisfies

$$\mathbb{E}\left[\sum_{i=1}^{k} \min(n, |\square_i|^{\log_b a})\right] = \Omega(\log n) \cdot n^{\log_b a}.$$

When all the box-sizes considered are trivially of size n or smaller, as will often be the case in this section, we omit the minimum in the above sum.

Bender *et al.* [43] showed that $M_{a,b}$ is a worst-case profile for $\mathcal{A}_{a,b}(n)$ for all n. Notice, in particular, that an execution of $\mathcal{A}_{a,b}(n)$ on $M_{a,b}$ will use exactly the boxes in the prefix $M_{a,b}(n)$ of $M_{a,b}$, with each scan using a box of precisely the same size as the scan; and by induction on n, the sum of $|\square|^{\log_b a}$ over the boxes in $M_{a,b}(n)$ is $\log_b n \cdot n^{\log_b a}$.

In this section, we consider the robustness of the worst-case memory profile $M_{a,b}$ to three types of smoothing:

- **Box-size perturbations:** In Section 9.5.1 we consider what happens if each box in $M_{a,b}$ has its size randomly perturbed (i.e., multiplied by a value in $[0, 1]$ drawn from a distribution $\mathcal{P}$ that has constant expectation). We show that the resulting distribution $\mathcal{M}$ remains worst-case in expectation.

- **Start-time perturbations:** In Section 9.5.2, we consider what happens if the memory profile $M_{a,b}(n)$ is cyclic-shifted by a random quantity. Shifting the memory profile is equivalent to executing algorithm $\mathcal{A}_{a,b}(n)$ starting at a random start-time in the cyclic version of $M_{a,b}(n)$. Again, the resulting distribution of profiles remains worst-case in expectation.

- **Box-order perturbations:** In Section 9.5.3, we consider a relaxation of the construction of $M_{a,b}(n)$ in which rather than always placing a box of size n after the *final* instance of $M_{a,b}(n/b)$, we instead allow ourselves to place the box of size n after *any* of the a recursive instances of $M_{a,b}(n/b)$. We prove that the resulting distribution over box sizes again remains worst-case in expectation. In fact, for a box profile M drawn from the distribution at random, we find that M remains worst-case with probability *one*. This is a large contrast to random shuffling considered in Section 9.4, where the random shuffle causes the algorithms to be adaptive in expectation.

9.5.1 Box-size perturbations

In this section, we consider the distribution over box profiles generated by randomly perturbing the sizes of boxes within the profile $M_{a,b}$ and show that algorithm $\mathcal{A}_{a,b}$ is not cache-adaptive on the modified profile (for all problem sizes n).

Theorem 9.30 *Let n be the size of an input to the algorithm $\mathcal{A}_{a,b}$, $t \in [1, \sqrt{n}]$ be an arbitrary value, and let $\mathcal{P}$ be a probability distribution on $[0, t]$. Suppose that the expected value of a random variable drawn from $\mathcal{P}$ is at least $\Omega(t)$.*

Let $X_1, X_2, \ldots$ be iid random variables drawn from $\mathcal{P}$. Let $M'_{a,b}$ be constructed by replacing each box $\square_i$ in $M_{a,b}$ with a box of size $X_i \cdot |\square_i|$. Then $M'_{a,b}$ is worst-case for $\mathcal{A}_{a,b}(n)$ in expectation.

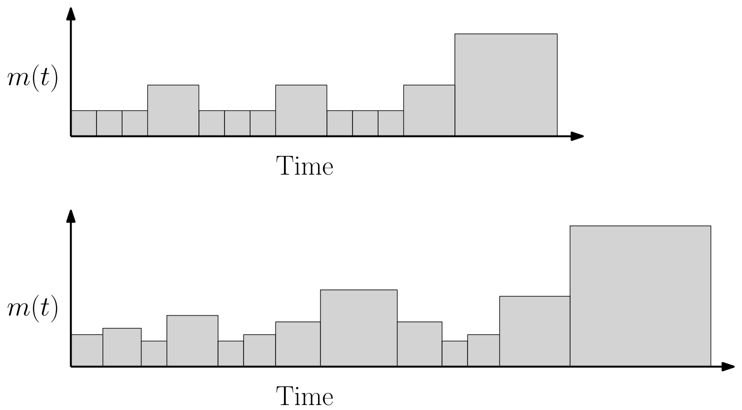


Figure 9-2: An example of randomly perturbing box sizes. The top profile is $M_{a,b}$ when $a = 3$ and $b = 2$. The bottom profile is an example of a random multiplication of the box sizes in $M_{a,b}$.

We begin by presenting two lemmas about algorithm performance on modified box sizes.

Lemma 9.31 *Let $M = (\square_1, \square_2, \ldots)$ be an arbitrary box profile. Suppose that when $\mathcal{A}_{a,b}(n)$ is executed on M, it completes at box $\square_k$. Now define $M' = (\square'_1, \square'_2, \ldots)$*

to be a box profile such that $|\square'_i| \leq |\square_i|$ for all i. (For convenience, we will even allow $|\square'_i| = 0$.) Then when $\mathcal{A}_{a,b}(n)$ is executed on M', it completes at some box $\square'_{k'}$ satisfying $k' \geq k$.

Proof. This follows by repeated applications of the No-catchup Lemma (Lemma B.2). Suppose that M and M' differ only in their i-th box, with $|\square'_i| < |\square_i|$. Then the first $(i-1)$ boxes of each of M and M' will finish at the same point within $\mathcal{A}_{a,b}(n)$ (i.e., will finish after the same block access). The i-th box of M' will then finish at the same point or earlier within $\mathcal{A}_{a,b}(n)$ than does the i-th box of M. By the No-catchup Lemma, it follows that the k-th box of M' will also finish at the same point within $\mathcal{A}_{a,b}(n)$ or earlier than does the k-th box of M. Since $\mathcal{A}_{a,b}(n)$ requires k boxes to complete on profile M, $\mathcal{A}_{a,b}(n)$ will also require at least k boxes to complete on profile M'.

The above reasoning assumes that M and M' differ in only a single box. By k repeated applications of the argument, we may instead allow M and M' to differ in all of their first k boxes. This implies the full lemma. $\square$

Lemma 9.32 *Let $\alpha \cdot M_{a,b}$ denote the memory profile obtained by multiplying the size of each box in $M_{a,b}$ by α. If $\alpha \leq \sqrt{n}$ is a power of b, then $\alpha \cdot M_{a,b}$ is still a worst-case profile for $\mathcal{A}_{a,b}(n)$.*

Proof. The proof of the lemma takes advantage of the self-symmetry implicitly present within $M_{a,b}$. In particular, notice that $\alpha \cdot M_{a,b}$ can be obtained from $M_{a,b}$ by *removing* every box in $M_{a,b}$ of size smaller than α.

Define $M'_{a,b}(n)$ to be the profile obtained by removing every box from $M_{a,b}(n)$ of size smaller than α. By Lemma 9.31, the algorithm $\mathcal{A}_{a,b}(n)$ will require (at least) all of the boxes in $M'_{a,b}(n)$ to complete, since it requires all of the boxes in $M_{a,b}(n)$ to complete.

Since $\alpha \cdot M_{a,b}$ contains $M'_{a,b}(n)$ as a prefix, in order to prove that $\alpha \cdot M_{a,b}$ is a worst-case profile, it suffices to show that the sum of $|\square|^{\log_b a}$ over the boxes in $M'_{a,b}(n)$ is $\Omega(\log n \cdot n^{\log_b a})$. That is, if $M'_{a,b}(n) = (\square_1, \ldots, \square_k)$, then we wish to show that

$$\sum_{i=1}^{k} |\square_i|^{\log_b a} = \Omega(\log n \cdot n^{\log_b a}).$$

Notice that for each box-size n/b^j such that $b^j \leq \sqrt{n}$, the profile $M'_{a,b}(n)$ contains a^j instances of a box of size n/b^j. (In particular, the recursive construction of $M_{a,b}$

includes a^j subproblems of size n/b^j.) Thus

$$\begin{aligned}
&\sum_{i=1}^{k} |\square_i|^{\log_b a} \\
&\geq \sum_{b^j=1}^{b^j=\sqrt{n}} a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b a} \\
&= \sum_{b^j=1}^{b^j=\sqrt{n}} a^j \cdot \frac{n^{\log_b a}}{a^j} \\
&= \Theta(\log n) \cdot n^{\log_b a},
\end{aligned}$$

as desired. □

Combining Lemmas 9.31 and 9.32, we can now complete the proof of Theorem 9.30.

Proof. Let T be the smallest power of b greater than t. By Lemma 9.32, the box profile $T \cdot M_{a,b}$ is worst-case for $\mathcal{A}_{a,b}(n)$.

Suppose that $\mathcal{A}_{a,b}(n)$ uses k boxes to complete on $T \cdot M_{a,b}$. Since $T \cdot M_{a,b}$ has the property that its i-th box is of size at least as large as the i-th box of $M'_{a,b}$, Lemma 9.31 tells us that $\mathcal{A}_{a,b}$ also requires at least k boxes to complete on $M'_{a,b}$.[10] Since $T \cdot M_{a,b}$ is a worst-case profile, in order to complete the proof, it therefore suffices to show that

$$\mathbb{E}\left[\sum_{i=1}^{k} (X_i \cdot |\square_i|)^{\log_b a}\right] \geq \Omega\left(\sum_{i=1}^{k} (T \cdot |\square_i|)^{\log_b a}\right).$$

By linearity of expectation, it suffices to prove that $\mathbb{E}[X_i^{\log_b a}] \geq T^{\log_b a}$. Since the function $f(x) = x^{\log_b a}$ is convex (because $a > b$), Jensen's inequality tells us that

$$\mathbb{E}[X_i^{\log_b a}] \geq \mathbb{E}[X_i]^{\log_b a} \geq \Omega(T^{\log_b a}),$$

as desired. □

9.5.2 Start-time perturbations

In this section, we consider another natural form of smoothing, in which the algorithm $\mathcal{A}_{a,b}(n)$ begins at a random start-time within a cyclic version of the profile $M_{a,b}(n)$. Random start times simulate jobs starting at arbitrary times while a system is running.

Define the profile $M^{\circ}_{a,b}(n) = M_{a,b}(n) \circ M_{a,b}(n) \circ M_{a,b}(n) \circ \cdots$ to be the infinite profile consisting of duplicates of the profile $M_{a,b}(n)$. We will use $(\square_1, \square_2, \ldots)$ to denote the boxes in $M^{\circ}_{a,b}(n)$.

[10]Note that if a box in $M_{a,b}$ has its size multiplied by zero, resulting in an empty box in $M'_{a,b}$, we still consider that box when talking about the i-th box of $M'_{a,b}$.

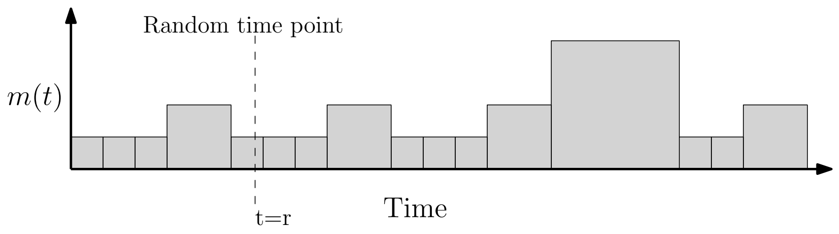


Figure 9-3: An example of picking a random start time with the $M^{\circ}_{a,b}$ profile where $a = 3$ and $b = 2$. If we start the algorithm at time $t = r$ instead of $t = 0$ we have created a cyclic shift.

We will now generate truncated profiles simulating random start times by removing boxes from the beginning of $M^{\circ}_{a,b}(n)$. Let k be the number of boxes in $M_{a,b}(n)$ and $t = \sum_{i=1}^{k} |\square_i|$ denote the sum of the sizes of the boxes in $M_{a,b}(n)$. Define a distribution $\mathcal{M}$ over infinite profiles such that $M \in \mathcal{M}$ is constructed as follows: first select a random $r \in \{0, 1, \ldots, t-1\}$; then identify the first box $\square_j$ such that $\sum_{i=1}^{j} |\square_i| \geq r$; finally, construct M by removing each of the boxes $\square_1, \ldots, \square_{j-1}$, and replacing the box $\square_j$ with a box of size $\left(\sum_{i=1}^{j} |\square_i|\right) - r$.

The purpose of this section is to prove the following theorem:

Theorem 9.33 *Suppose $n > 1$ is a power of b. The distribution $\mathcal{M}$, in which a random start-time is selected within the cyclic profile $M^{\circ}_{a,b}(n)$, is worst-case in expectation for $\mathcal{A}_{a,b}(n)$.*

Proof. Recall that the profile $M_{a,b}(n)$ can be expressed as a copies of the profile $M_{a,b}(n/b)$, along with a box of size n. Let A denote the prefix of $M_{a,b}(n)$ consisting of the first $a-1$ copies of the profile $M_{a,b}(n/b)$, and let B denote the final copy of the profile $M_{a,b}(n/b)$ along with the box of size n.

Let $(\square_1, \ldots, \square_x)$ denote the boxes in A and $(\square'_1, \ldots, \square'_y)$ denote the boxes in B. We claim that

$$\sum_{i=1}^{x} |\square_i| \geq \Omega\left(\sum_{i=1}^{y} |\square'_i|\right), \tag{9.21}$$

and that

$$\sum_{i=1}^{x} |\square_i|^{\log_b a} \leq O\left(\sum_{i=1}^{y} |\square'_i|^{\log_b a}\right). \tag{9.22}$$

Before proving Equation (9.21) and Equation (9.22), we first use them to complete the proof of the theorem. When constructing a random profile M in $\mathcal{M}$, Equation (9.21) tells us that with probability $\Omega(1)$, r will satisfy $r \leq \sum_{i=1}^{x} |\square_i|$. When this occurs, the profile M can be obtained from the profile $M^{\circ}_{a,b}(n)$ by eliminating and shrinking boxes in the subsequence $(\square_1, \ldots, \square_x)$ and not modifying any other boxes. If we identify each of the boxes $\square_{x+1}, \square_{x+2}, \ldots$ in $M^{\circ}_{a,b}(n)$ with their counterparts in M, then by Lemma 9.31, when $\mathcal{A}_{a,b}(n)$ is executed on M it will still use all of the boxes $\square_{x+1}, \ldots, \square_{x+y}$. Recall that $M_{a,b}(n)$ has the property that

$$\sum_{i=1}^{x+y} |\square_i|^{\log_b a} = \log n \cdot n^{\log_b a}.$$

By Equation (9.22), it follows that

$$\sum_{i=x+1}^{x+y} |\square_i|^{\log_b a} \geq \Omega\left(\log n \cdot n^{\log_b a}\right). \tag{9.23}$$

Thus when we condition on $r \leq \sum_{i=1}^{x} |\square_i|$, the box-profile M is guaranteed to be worst-case. Since this occurs with probability $\Omega(1)$, it follows that $\mathcal{M}$ is worst-case in expectation.

It remains to prove Equation (9.21) and Equation (9.22). Since A consists of $a-1$ copies of $M_{a,b}(n/b)$ and B contains a single copy of $M_{a,b}(n/b)$ followed by a box of size n, it follows that

$$n + \sum_{i=1}^{x} |\square_i| \geq \left(\sum_{i=1}^{y} |\square'_i|\right), \tag{9.24}$$

and that

$$\sum_{i=1}^{x} |\square_i|^{\log_b a} \leq (a-1) \cdot \sum_{i=1}^{y} |\square'_i|^{\log_b a}. \tag{9.25}$$

Since $a \geq 2$, Equation (9.25) implies Equation (9.22), as desired. Since the box sequence $A = (\square_1, \ldots, \square_x)$ contains an instance of $M_{a,b}(n/b)$, it must contain at least one box of size n/b. Thus

$$(b+1) \cdot \sum_{i=1}^{x} |\square_i| \geq n + \sum_{i=1}^{x} |\square_i|.$$

Combining this with Equation (9.24), we get Equation (9.21), as desired. $\square$

9.5.3 Box-order perturbations

In this section, we consider smoothing via shuffling positions of boxes within the profile $M_{a,b}$. In particular, for n a power of b, we define $\mathcal{T}_{a,b}(n)$ to be the set of profiles constructed as follows: When $n = 1$, $\mathcal{T}_{a,b}(n)$ contains a single profile consisting of a box of size one. When $n > 1$, $\mathcal{T}_{a,b}(n)$ consists of all profiles M that can be constructed by selecting sub-profiles $X_1, \ldots, X_a \in \mathcal{T}_{a,b}(n/b)$, inserting a box of size n after one of the profiles X_i, and then concatenating the sub-profiles together. That is,

$$M = X_1 \circ X_2 \circ \cdots \circ X_i \circ \square \circ X_{i+1} \circ \cdots \circ X_a,$$

for some $i \in \{1, \ldots, a\}$, and where $\square$ is a box of size n. We depict an example of a re-ordered profile, M, in Figure 9-4.

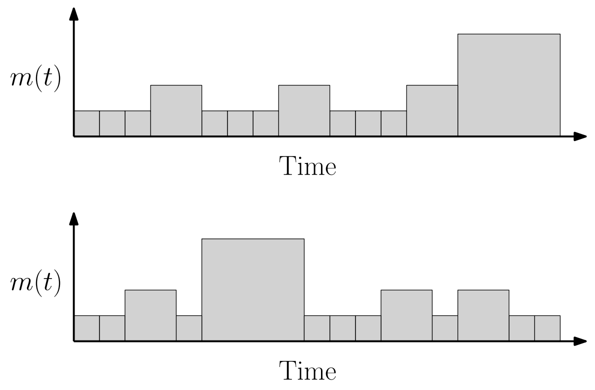


Figure 9-4: An example of re-ordering an $M_{a,b}$ profile. In this case $a = 3$ and $b = 2$. The top profile is $M_{a,b}$, the bottom profile is M.

The profiles M in the set $\mathcal{T}_{a,b}(n)$ can be thought of as relaxations of the profile $M_{a,b}(n)$. In particular, they are the profiles obtained by allowing the recursive construction of $M_{a,b}(n)$ to, at each step in the recursion, place a scan after an arbitrary subprofile rather than always after the final subprofile.

We define the set $\mathcal{T}_{a,b}$ of infinite box-profiles to contain all profiles M that for all n (that are powers of b) contain as a prefix an element of $\mathcal{T}_{a,b}(n)$.

The main purpose of this section is to prove that all profiles $M \in \mathcal{T}_{a,b}$ are worst-case profiles for the algorithm $\mathcal{A}_{a,b}$. That, is, the recursive construction of the worst-case profile $M_{a,b}$ is robust to random shuffling of large boxes within the recursive structure.

Theorem 9.34 *All profiles $M \in \mathcal{T}_{a,b}$ are worst-case profiles for the algorithm $\mathcal{A}_{a,b}$.*

Proof. For n a power of b, define the ***Universal Worst-Case Profile*** $U_{a,b}(n)$ as follows. When $n = 1$, $U_{a,b}(n)$ consists of a single box of size 1 repeated a times. When $n > 1$, we construct $U_{a,b}(n)$ by concatenating together a copies of $U_{a,b}(n/b)$, and inserting a box of size n after each of them. That is,

$$U_{a,b}(n) = (U_{a,b}(n/b) \cdot \Box)^a,$$

where $\Box$ is a box of size n, and the multiplication operator is defined to perform concatenation.

Define the infinite-box profile $U_{a,b}$ to be the unique infinite-box profile that contains each $U_{a,b}(n)$ as a prefix. We claim that $U_{a,b}$ is a worst-case profile for $\mathcal{A}_{a,b}$. In fact, a stronger statement is true: The profile $U_{a,b}(n)$ is *the same* as the profile $M_{a,b}(n \cdot b)$, except without the final box of size $n \cdot b$ that appears at the end of the latter. This statement follows immediately by induction on $\log_b n$. It follows that $U_{a,b} = M_{a,b}$, and that $U_{a,b}$ is a worst-case profile.

We now demonstrate how to construct each profile $M \in \mathcal{T}_{a,b}(n)$ by removing boxes from $U_{a,b}(n)$. In particular, we claim that each profile $M \in \mathcal{T}_{a,b}(n)$ can be obtained from the universal worst-case profile $U_{a,b}(n)$ by removing exactly an $\frac{a-1}{a}$ fraction of the boxes in each size-class from $U_{a,b}(n)$. When $n = 1$, this is immediate, since $U_{a,b}(n)$ consists of a boxes of size one, and the only profile in $\mathcal{T}_{a,b}(1)$ consists of a single box of size one. When $n > 1$ is a power of b, the claim follows by induction, using as an inductive hypothesis that $U_{a,b}(n/b)$ can be transformed into any element of $\mathcal{T}_{a,b}(n/b)$ by removing a $\frac{a-1}{a}$ fraction of the boxes in each size-class. Recall, in particular, that each profile $M \in \mathcal{T}_{a,b}(n)$ is obtained by selecting a profiles $X_1, \ldots, X_a \in \mathcal{T}_{a,b}(n/b)$, and concatenating them together with a single box of size n after one of them. The universal profile $U_{a,b}(n)$, on the other hand, is obtained by pasting together a copies of $U_{a,b}(n/b)$ with a box of size n after each of them. By removing all but of one of the boxes of size n from $U_{a,b}(n)$ (which corresponds with removing a $\frac{a-1}{a}$ fraction of the boxes of size n), and then applying the inductive hypothesis to each of the copies of $U_{a,b}(n/b)$ in order to transform it into X_i, it follows that each $M \in \mathcal{T}_{a,b}(n)$ can be constructed by removing from $U_{a,b}(n)$ some choice of $\frac{a-1}{a}$ fraction of the boxes in each size-class.

Since $U_{a,b}(n)$ is a prefix of $M_{a,b}(n \cdot b)$, when $\mathcal{A}_{a,b}(n \cdot b)$ is executed on profile $U_{a,b}$, it must use all the boxes in $U_{a,b}(n)$. For each box-size s, let t_s denote the number of boxes of size s in $U_{a,b}(n)$. By the claim in the preceding paragraph, and by Lemma 9.31, when $\mathcal{A}_{a,b}(n \cdot b)$ is executed on any element $M \in \mathcal{T}_{a,b}$, it must use at least t_s/a boxes of each size s. If $(\square'_1, \square'_2, \ldots)$ denotes the profile M, and k is the number of boxes that $\mathcal{A}_{a,b}(n \cdot b)$ uses when executed on M, it follows that

$$\sum_{i=1}^{k} |\square'_i|^{\log_b a} = \sum_s \frac{t_s}{a} \cdot s^{\log_b a} \geq \Omega\left(\sum_s t_s \cdot s^{\log_b a}\right). \tag{9.26}$$

Since $U_{a,b}(n)$ is the same as $M_{a,b}(n \cdot b)$, except with the final box of size n removed, $|\square|^{\log_b a}$ over the boxes in $U_{a,b}(n)$ is $\Omega(\log n \cdot n^{\log_b a})$. Thus the right-hand side of Equation (9.26) is $\Omega(\log n \cdot n^{\log_b a})$. It follows that the profile $M \in \mathcal{T}_{a,b}$ is a worst-case profile, as desired. $\square$

9.6 Related work

This section reviews related work in the area of modeling real-world performance of algorithms under memory fluctuations. In order to apply our algorithms to real-world systems, it is important to find the right model in which the theoretical efficiency of our algorithms closely matches their practical efficiency.

The ***disk access model (DAM)*** was formulated [3, 186] to account for multi-level memory hierarchies (present in real systems) where the size of memory available for computation and the speed of computation differs in each level. The DAM [3] models a 2-level memory hierarchy with a large (infinite sized) but slow disk, and a small (bounded by M) but fast cache. The drawback of DAM is that efficient algorithms developed in this model require knowledge of cache size. The ***ideal-cache model*** [153, 307] was proposed to counteract this drawback by building an automatic paging algorithm into the model and providing *no knowledge* of the cache size to algorithms. Thus, ***cache-oblivious algorithms*** [116, 225] are independent of the memory parameter and can be applied to complex multi-level architectures where the size of each memory-level is unknown. There exists a plethora of previous work on the performance analysis and implementations of cache-oblivious algorithms (on single-core and multicore machines) [46, 50, 63, 78, 99, 102, 106, 152, 153, 230, 393, 394]. Among other limits [40, 77], one critical limit of the cache-oblivious model is that it does not account for *changing cache-size*. In fact, as we shall see in Chapter 10, preliminary experimental studies have shown that two cache-oblivious algorithms (with the same I/O-complexity) might in fact perform vastly differently under a changing cache.

Changing cache size can stem from a variety of reasons. For example, shared caches in a multicore environment may allocate different portions of the cache to different processes at any time (and this allocation could be independent of the memory needed by each process). There has been substantial work on paging in shared-cache environments. For example, Peserico formulated alternative models for page replacement [302] provided a fluctuating cache. However, Peserico's page-replacement model differs from the cache-adaptive model because in his model, the cache-size changes at specific locations in the page-request sequence as opposed to being temporally related to each individual I/O. Other page replacement policies have applied to multicore shared-cache environments [216] where several processes share the same cache [32, 176, 194, 250, 251, 398] leading to situations where page sizes can vary [251] and where an application can adjust the cache size itself [194, 398].

Theoretical [32,33] and empirical studies [295,401,402] have been done in the past to study partial aspects of adaptivity to memory fluctuations [80, 167, 272, 273, 296, 400, 402]. Barve and Vitter [32, 33] were the first to generalize the DAM model to account for changing cache size. In their model, they provide optimal algorithms for sorting, matrix multiplication, LU decomposition, FFT, and permutation but stops just short of a generalized technique for finding algorithms that are optimal under memory fluctuations [32, 33]. In their model, the cache is guaranteed to stay at size M for M/B I/Os. In this way, their model is very similar to our notion of square profiles.

The cache-adaptive model [45] introduced the notion of a ***memory profile***. The memory profile provides the cache size at each time step (defined as an I/O-operation), and at each time step the cache can increase by 1 block or decrease by an arbitrary amount. Bender *et al.* [43] went on to show that *any* optimal (in the DAM) (a, b, c)-regular algorithm where $a > b$ and $c < 1$ is ***cache-adaptive*** or *optimal* under this model. However, disappointingly, they showed that (a, b, c)-regular algorithms where $c = 1$ can be up to a log-factor away from optimal [43]. This leads to the the question of whether non-adaptive (a, b, c)-regular algorithms can be turned into cache-adaptive algorithms via some procedure. Chapter 10 takes the first step in this direction by introducing a ***scan-hiding*** procedure for turning certain non-adaptive (a, b, c)-regular algorithms into cache-adaptive ones. Although scan-hiding takes polynomial time, it introduces too much overhead and also does not apply to all (a, b, c)-regular algorithms where $a > b$ and $c = 1$.

This chapter takes another important step in this direction by showing that (a, b, c)-regular algorithms where $a > b$ and $c = 1$ *are cache-adaptive in expectation.* Whereas previous work analyzed all algorithms in the *worst-case*, we believe that this is, in fact, unnecessary and does not accurately depict real-world architectures. We introduce the notion of average-case cache-adaptivity in what we hope to be a more accurate picture of shared-cache multicore systems.

9.7 Conclusion

This chapter presents the first beyond-worst-case analysis of (a, b, c)-regular cache-adaptive algorithms. The main positive result in this chapter gives hope for cache-adaptivity: even though the worst-case profile from previous work [43, 45] is robust under random perturbations and shuffling, many (a, b, c)-regular algorithms become cache-adaptive in expectation on profiles generated from any distribution. Notably, to our knowledge, all currently known sub-cubic matrix multiplication algorithms (such as Strassen's [351], Vassilevska Williams' [383], Coppersmith-Winograd's [107], and Le Gall's [158]) were a logarithmic factor away from adaptive under worst-case analysis, but are adaptive in expectation on random profiles via smoothed analysis. Our results provide guidance for analyzing cache-adaptive algorithms on profiles beyond the adversarially constructed worst-case profile.

Cache fluctuations are a fact of life on modern hardware, but many open questions remain. In this chapter, we randomized memory profiles for deterministic (a, b, c)-regular algorithms. Could randomized algorithms also overcome worst-case profiles and result in cache-adaptivity? On the empirical side, which patterns of memory fluctuations occur in the real world? Further exploration of beyond-worst-case analysis may help model practical memory patterns more accurately.

Locality-first strategy. The smoothed analysis in this chapter theoretically grounds the locality-first strategy by closing the gap between cache-oblivious and cache-adaptive algorithms. This chapter shows that cache-oblivious algorithms that achieve optimal use of locality in a fixed-size cache also are optimal when the available cache size fluctuates in shared memory when the fluctuations are not highly tailored to the algorithm. These results validate the real-world benefits of the locality-first strategy of focusing on locality for overall performance even in the face of parallelism.

Chapter 10

Cache-Adaptive Exploration: Experimental Results and Scan-Hiding for Adaptivity

This chapter presents “scan-hiding,” a technique for converting non-cache-adaptive algorithms to cache-adaptive ones by applying the locality-first strategy to enhance temporal locality. This chapter demonstrates how to create algorithms that adapt to cache fluctuations in the worst case via algorithmic transformations. It also includes an experimental investigation of cache-adaptive algorithms that suggests that the empirical advantage of cache-adaptive algorithms extends to more practical situations than just the worst case. These results support the locality-first strategy for algorithm development both theoretically and practically.

This work was conducted in collaboration with Andrea Lincoln, Jayson Lynch, and Quanquan C. Liu [246]. We would like to thank Michael Bender and Rob Johnson for their writing input and technical comments. Thanks to Rishab Nithyanand for contributions to the experimental section. We would like to thank Rezaul Chowdhury, Rathish Das, and Erik D. Demaine for helpful discussions.

Abstract

Despite the increasing popularity of shared-cache systems, the theoretical behavior of most algorithms in the face of memory fluctuations is not yet well understood. There is a gap between our knowledge about how algorithms perform in a fixed-size (static) cache versus a dynamic cache where the amount of memory available to a program fluctuates.

Cache-adaptive analysis is a method of analyzing how well algorithms use a dynamic cache. Bender *et al.* showed that optimal cache-adaptivity does not follow from cache-optimality in a static cache. Specifically, they proved that some cache-optimal algorithms in a static cache are suboptimal when subject to certain memory profiles (patterns of memory fluctuations). For example, the canonical cache-oblivious divide-and-conquer formulation of Strassen’s algorithm for matrix multiplication is suboptimal in the cache-adaptive model because it does a linear scan to add submatrices together.

This chapter introduces "scan-hiding," the first technique for converting a class of non-cache-adaptive algorithms with linear scans to optimally cache-adaptive variants. It provides a concrete example of scan-hiding on Strassen's algorithm, a subcubic algorithm for matrix multiplication that involves linear scans at each level of its recursive structure. All currently known subcubic algorithms for matrix multiplication include linear scans, however, so our technique applies to a large class of algorithms.

This chapter also experimentally evaluates different algorithms in the face of memory fluctuations to explore how theoretical analysis of cache-adaptivity manifests in practice. These findings suggest that memory fluctuations affect algorithms with the same theoretical cache performance in a static cache differently. For example, the optimally cache-adaptive naive matrix multiplication algorithm achieved fewer relative faults than the non-adaptive variant in the face of changing memory size. These experiments suggest that the performance advantage of cache-adaptive algorithms extends to more practical situations beyond the carefully-crafted memory specifications in proofs of worst-case behavior.

10.1 Introduction

As detailed in Chapter 9, the amount of memory available to a single process in a shared cache may vary dynamically over time as multiple processors compete for space. These memory fluctuations have an effect on both theoretical and practical performance. Significant effort has been devoted to practical algorithms adapt to changing memory size in a shared cache, but these algorithms are susceptible to worst-case fluctuations. Theoretical guarantees on ***cache-adaptive algorithms*** that handle memory fluctuations gracefully analyze algorithm performance in the worst case and show that optimal cache-adaptivity does not necessarily follow from optimality in a fixed-size cache [43, 45].

Specifically, Bender *et al.* [43] gave a framework for designing and analyzing cache-adaptive algorithms in the worst case. Specifically, they completely characterize when a linear-space-complexity Master-method-style or mutually recursive linear-space-complexity Akra-Bazzi-style algorithm is optimal in the cache-adaptive model. For example, the in-place recursive naive[1] cache-oblivious matrix multiplication algorithm is optimally cache-adaptive, while the naive cache-oblivious matrix multiplication that does the additions upfront (and not in-place) is not optimally cache-adaptive. They provide a toolkit for the analysis and design of cache-oblivious algorithms in certain recursive forms and show how to determine if an algorithm in a certain recursive form is optimally cache-adaptive and if not, to determine how far it is from optimal.

The main contribution of Bender *et al.*'s study of cache-adaptive analysis [43] is an algorithmic toolkit for recursive algorithms in specific forms. At a high level, cache-oblivious algorithms that have long ($\omega(1)$ block transfers) scans[2] (such as the

[1]This chapter uses "naive" matrix multiplication to refer to the $O(n^3)$ work algorithm for matrix multiplication.

[2]That is, the recurrence for their cache complexity has the form $T(n) = aT(n/b) + \Omega(n/B)$ where B is the cache line size in words.

not-in-place n^3 matrix multiplication algorithm) in addition to their recursive calls are not immediately cache-adaptive. However, there exists an in-place, optimally cache-adaptive version of naive matrix multiplication.

Although these results take important steps in cache-adaptive analysis, open questions remain regarding the limits of this theoretical framework. Is there a way to transform other algorithms that do $\omega(n/B)$ block transfers at each recursive call (where B is the cache line size in words), such as Strassen's algorithm [351], into optimally cache-adaptive algorithms? Furthermore, Bender *et al.* [45] gave a worst-case analysis in which the non-adaptive naive matrix multiplication is a $\Theta(\lg N)$ factor off from optimal. Does the predicted slow down manifest in reality?

Contributions

This chapter takes the first steps towards answering these questions in two ways.

First, it introduces a new technique called ***scan-hiding*** for making a certain class of non-cache-adaptive (a, b, c)-regular algorithms adaptive. As a case study, this chapter uses scan-hiding to construct a cache-adaptive version of Strassen's algorithm for matrix multiplication. Strassen's algorithm involves linear scans in its recurrence, which makes the algorithm as described non-adaptive via a theorem from Bender *et al.* [43]. Scan-hiding is the first method in the cache-adaptive setting to transform non-adaptive algorithms into adaptive algorithms.

Next, this chapter empirically evaluates the performance of various algorithms for matrix multiplication and sorting when subject to memory fluctuations and finds that algorithms that are "more adaptive" (i.e. closer to optimal cache-adaptivity) are more robust under memory changes. Moreover, the tested algorithms exhibit performance differences even when memory sizes do not change adversarially.

Map. This chapter is organized as follows. It omits preliminaries about cache-adaptive algorithms and analysis because they have been covered in Section 9.2. Section 10.2 presents the general scan-hiding technique for converting a class of non-adaptive but "scan-hideable" algorithms into adaptive ones. To concretize the scan-hiding technique, Sections 10.3 and 10.4 apply the technique to Strassen's subcubic matrix multiplication algorithm as a case study. Section 10.3 describes the Strassen algorithm and shows it is non-adaptive. Section 10.4 applies scan-hiding to Strassen's algorithm and shows the resulting algorithm is adaptive. Section 10.5 empirically evaluates several algorithms subjected to memory fluctuations.Finally, Section 10.6 provides concluding remarks.

10.2 Generalized scan-hiding

This section presents a generalized framework for converting non-adaptive algorithms into adaptive algorithms via "scan-hiding." It will briefly sketch the scan-hiding procedure. Section 10.4 concretizes the technique using Strassen's algorithm as a case study. Next, this section identifies a class of "scan-hideable algorithms" that the technique applies to. Finally, it shows that applying the scan-hiding technique generates an optimally progressing cache-adaptive algorithm as defined in Chapter 9. This generalized scan-hiding procedure can be applied to Master-method-style recursive algorithms that contain "independent" linear scans in each level of the recursion.

At a high level, scan-hiding breaks up long (up to linear) scans at each level of a recursive algorithm and distributes the pieces evenly throughout the algorithm's "recursion tree." An algorithm's ***recursion tree*** is the tree created from a recursive algorithm $\mathcal{A}$ such that each node of the tree contains all subproblems defined by that node.

First, we will specify the class of "scan-hideable" algorithms that scan-hiding applies to.

Definition 10.1 (Scan-hideable algorithms) *Let $\mathcal{A}$ be an (a, b, c)-regular algorithm with input size n^c. If $\mathcal{A}$ is non-adaptive, it is* ***scan-hideable*** *if it has the following characteristics:*

- *$\mathcal{A}$ has a runtime that can be computed as a function that follows the Master Theorem style equations of the form $T(n) = aT(n/b) + O(n^c)$ in the DAM model where $\log_b(a) > c$ for some constants $a > 0$, $b \geq 1$, and $c \geq 1$.*

- *In terms of I/Os, the base case of $\mathcal{A}$ is $T(M) = \frac{M}{B}$ where M is the cache size.*

- *At each level of the recursion tree, a linear scan is performed with a more "work-consuming" subproblem. Let the* ***work*** *of $\mathcal{A}$ be the amount of computation in words performed in a square profile of size m by m by some subproblem of $\mathcal{A}$. A subprocess is more* ***work-consuming*** *if it uses more work in a square profile of size m by m. For example, a naive matrix multiplication subproblem is more work-consuming than a scan since it uses $(mB)^{\log_2 3}$ work as opposed to a scan which uses mB work.*

- *Each of the more work-consuming subproblems in each node of the recursion tree only depends on the the results of the scans performed in the subtrees to the left of the path from the current node to the root.*

- *Each node's scans depend on the result of the subprocesses of the ancestors (including the parent) of the current node in $\mathcal{A}$'s recursion tree.*

Next, we will show that even if a scan-hideable algorithm has an upfront scan that cannot be hidden, it can still become cache-adaptive. The scan-hiding technique involves hiding all scans "inside" the recursive structure in subcalls. If an algorithm (e.g. Strassen) requires an initial linear scan for even the first subcall, it cannot hide the first scan in recursive subcalls.

Therefore, we show that an algorithm $\mathcal{A}$ is optimally progressing even if $\mathcal{A}$ has an initial scan of length $O(n^c)$. We will be using $\mathcal{A}_{\mathsf{scan\_hiding}}$ as the name for the algorithm using this scan-hiding technique.

Lemma 10.2 *Let $\mathcal{A}$ be a scan-hideable (a, b, c)-regular algorithm and $\mathcal{A}_{\mathsf{scan\_hiding}}$ be the resulting algorithm after applying scan-hiding to $\mathcal{A}$. Additionally, let us assign* *potential in integer units to accesses, much as we do for work. If the following are true:*

- *The optimal $\mathcal{A}$ algorithm in the DAM model (i.e. ignoring wasted time due to scans) takes total work $n^{\lg_b(a)}$ and respects the progress bound $\rho(m(t)) = d_0(m(t)B)^{\log_b(a)/c}$ where d_0 is a constant greater than 0. Let m be a profile that starts at time step 0 and ends at time step T where the optimal $\mathcal{A}$ algorithm completes.*

- *$\mathcal{A}_{\mathsf{scan\_hiding}}$ is an algorithm which computes the solution to the problem that $\mathcal{A}$ solves and has total work $d_1 n^{\lg_b(a)}$ and has total potential $d_2 n^{\lg_b(a)}$ and completes $c_3(mB)^{\log_b(a)/c}$ work and potential in any m by m square profile where d_1, d_2 and d_3 are all constants greater than 0 and where $mB < n^c$.*

- *Finally, $\mathcal{A}_{\mathsf{scan\_hiding}}$ must also have the property that if the total work plus potential completed is $(d_1 + d_2)n^{\lg_b(a)}$, $\mathcal{A}_{\mathsf{scan\_hiding}}$ is guaranteed to have finished its last access.*

Then $\mathcal{A}_{\mathsf{scan\_hiding}}$ is cache-adaptive.

Proof. Let $m'(t)$ be the inner square profile of $m(t)$. When $n^{\mathcal{C}} < m'(t)B$ the entirety of $\mathcal{A}$ completes and $\mathcal{A}_{\mathsf{scan\_hiding}}$ will complete given a constant factor expansion.

The optimal $\mathcal{A}$ algorithm in the worst case makes a constant factor $d_4 \geq 1$ less progress on the inner profile for some constant d_4.

With time augmentation $\frac{1}{d_3 \cdot d_4}$, $\mathcal{A}_{\mathsf{scan\_hiding}}$ completes as much progress on this square as A did in the associated part of the profile, so over the entire profile $\mathcal{A}_{\mathsf{scan\_hiding}}$ completes at least $n^{\lg_b(a)}$ work and potential. With time augmentation $\frac{d_1 \cdot d_2}{d_3 \cdot d_4}$, $\mathcal{A}_{\mathsf{scan\_hiding}}$ completes at least $(d_1 + d_2)n^{\lg_b(a)}$ work and potential.

Thus $\mathcal{A}_{\mathsf{scan\_hiding}}$ must have completed. □

Finally, we prove that algorithm $\mathcal{A}_{\mathsf{scan\_hiding}}$ is optimally progressing. Specifically, we show that any scan-hideable algorithm $\mathcal{A}$ can be converted into an optimally-progressing version $\mathcal{A}_{\mathsf{scan\_hiding}}$ via scan-hiding.

Theorem 10.3 *Let $\mathcal{A}$ be a scan-hideable algorithm with running time of the form $T(n) = aT(n/b) + O(n^c)$ and $\mathcal{A}_{\mathsf{scan\_hiding}}$ be the version of $\mathcal{A}$ with scan-hiding applied. $\mathcal{A}_{\mathsf{scan\_hiding}}$ is optimally progressing with progress bound $\rho(m(t)) = (m(t)B)^{\frac{\log_b(a)}{c}}$. To manage all the pointers we also require $m(t) \geq \log_b n$ for all t.*

Proof. For this proof, we charge all writeouts to read-ins; therefore, we do not specifically argue the cache-adaptivity of the writeouts of $\mathcal{A}_{\mathsf{scan\_hiding}}$. We will proceed with a potential argument.

Since the total amount of required amount of work by $\mathcal{A}$ is $O\left(n^{\log_b(a)}\right)$, an optimally progressing version of $\mathcal{A}$ performs $\Omega\left((mB)^{\frac{\log_b(a)}{c}}\right)$ work for each inner square with side length m in the inner square profile of the usable profile.

We first show that the amount of total potential during upfront scan does not exceed $O\left(n^{\log_b a}\right)$. Then, by Lemma 10.2, we know that $\mathcal{A}_{\mathsf{scan\_hiding}}$ is still optimally progressing. We assign $O(n^{\log_b(a)-c})$ potential to each of the $O(n^c)$ scans; thus for each scan we complete $\Omega\left(n^{\log_b(a)-c}mB\right) = \Omega\left((mB)^{\frac{\log_b(a)}{c}}\right)$ progress, as long as $m < n^c$.

Scan-hiding intersperses the scans with the more work-consuming processes associated with the other parts of the algorithm $\mathcal{A}$, resulting in $\Omega\left((mB)^{\frac{\log_b a}{c}}\right)$ work to be done at each level of the recursion in any m by m square where at least half of the cache misses are used to perform this work.

Let us consider a recursive call solving a subproblem of size s. The scan in this subproblem has length at most $O(s+\log_b(n))$ with an additive $O(\log_b n)$ if the pointers are passed around naively. The condition that $m(t) \geq \log_b n$ allows for naive splits of the sizes of scans to be acceptable by making the additive factor $\log_b(n)$ at most a factor of 2 of the size of the solved problem.

Thus, algorithm $\mathcal{A}$ is optimally progressing. $\square$

Since scan-hiding amortizes the work of part of scan against each leaf node of the recursion tree, each leaf node must be sufficiently large to hide part of a scan. Therefore, these analyses assume that $m(t) \geq \log_b n$. Given a specific problem, one can usually find a way to split the scans such that this requirement is unnecessary. General scan-hiding uses this minimum cache size to make passing pointers to scans easy and inexpensive, however.

As an immediate consequence of Theorem 10.3 above, we get the following corollary.

Corollary 10.4 *Given a scan-hideable algorithm $\mathcal{A}$ with running time of the form $T(N) = aT(N/b) + O(N)$, $\mathcal{A}_{\mathsf{scan\_hiding}}$ is cache-adaptive. If a node's subprocesses depend on the scans of the nodes in the left subtree, then we also require $m(t) \geq \log n$.*

Scan-hiding directly broadens Theorem 7.3 in [43] to show cache-adaptivity for a specific subclass of Master-method-style problems when $\log_b(a) > c$.

10.3 Strassen's algorithm

This section provides necessary background to understand the Strassen scan-hiding case study in Section 10.4. First, it motivates the study of Strassen's algorithm for matrix multiplication in this chapter with a discussion of practical usage of Strassen's algorithm. Next, it reviews the details of Strassen's algorithm and show that a straightforward implementation with linear scans is not optimally progressing.

Practical usage of Strassen's algorithm. Currently the most efficient matrix multiplication algorithms in practice for very large matrices use the Strassen algorithm [160, 200]. For example, the GNU Multi-Precision Library uses Strassen to perform matrix multiplication for decimal digits in the range $10,000$ to $40,000$, and the Java uses its Strassen implementation for above $74,000$ decimal digits.

Algorithm definition. Algorithm 10.5 provides the equations and pseudocode for Strassen's matrix multiplication algorithm. Recall that the recurrence for the runtime for Strassen's algorithm for multiplying two $n \times n$ matrices is $T(n) = 7T(n/2)+O(n^2)$ (in the RAM model).

Algorithm 10.5 STRASSEN(X, Y, Z):
Let X, Y be input matrices and Z be the output matrix. We define the matrix quadrants as follows for X (quadrants for Y and Z are defined in the same way):

$$X = \begin{bmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{bmatrix} .$$

Strassen's algorithm recursively computes 7 intermediate matrix products with 10 linear scans:

$$\begin{aligned}
S_1 &= (X_{1,1} + X_{2,2}) \cdot (Y_{1,1} + Y_{2,2}) \\
S_2 &= (X_{2,1} + X_{2,2}) \cdot Y_{1,1} \\
S_3 &= X_{1,1} \cdot (Y_{1,2} - Y_{2,2}) \\
S_4 &= X_{2,2} \cdot (Y_{2,1} - Y_{1,1}) \\
S_5 &= (X_{1,1} + X_{1,2}) \cdot Y_{2,2} \\
S_6 &= (X_{2,1} - X_{1,1}) \cdot (Y_{1,1} + Y_{1,2}) \\
S_7 &= (X_{1,2} - X_{2,2}) \cdot (Y_{2,1} + Y_{2,2}) \ .
\end{aligned}$$

The quadrants of the resulting Z matrix can be computed in terms of $S_1, \dots, S_7$ as follows:

$$\begin{aligned}
Z_{1,1} &= S_1 + S_4 - S_5 + S_7 \\
Z_{1,2} &= S_3 + S_5 \\
Z_{2,1} &= S_2 + S_4 \\
Z_{2,2} &= S_1 - S_2 + S_3 + S_6 \ .
\end{aligned}$$

`NaiveStrassen` is not optimally progressing. We will show that a straightforward implementation of Strassen's algorithm, called `NaiveStrassen`, is not optimally progressing and therefore not cache-adaptive.

Definition 10.6 `NaiveStrassen` *computes the* 10 *matrix sums needed to produce the input for its* 7 *recursive calls, makes its* 7 *recursive calls each of which return an associated output matrix, then does the necessary* 8 *matrix sums to produce its output. This algorithm results in a recurrence of the form $T(n) = 7T(n/2) + O(n^2)$, or in terms of its input size $N = n^2$ we have the recurrence $T(N) = 7T(N/4) + O(N)$.*

Lemma 10.7 `NaiveStrassen` *is* not *optimally progressing with respect to the progress bound $\mathrm{P}_{\mathcal{P}}(M) = M^{\lg(7)/2}$, even if $\forall t, M(t) > B \lg(n)$.*

PROOF. By Theorem 7.3 from [43], if an algorithm with linear space complexity has a recurrence of the form $T(N) = aT(N/b) + O(N^c)$ with a tall cache assumption that $M(t) > \lg(n)B$ then the algorithm is a $\lg(N/(B\lg(n)))$ factor[3] away from being optimally progressing [43]. $\square$

10.4 Scan-hiding and Strassen's algorithm

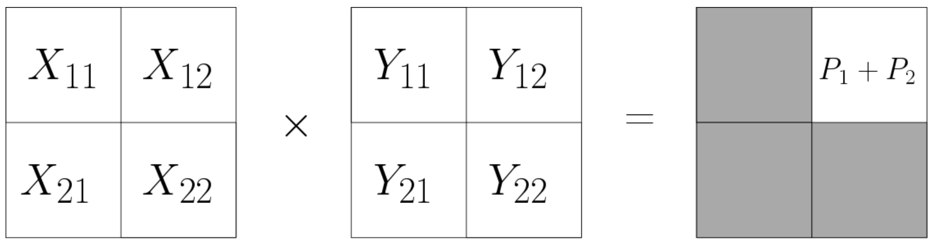
This section concretizes the scan-hiding technique using Strassen's algorithm as a case study. First, this section defines the `AdaptiveStrassen` algorithm that applies scan-hiding to Strassen's algorithm. Appendix B.8 contains omitted pseudocode for the algorithm and its subroutines. Finally, this section shows that the `AdaptiveStrassen` algorithm is cache adaptive.

`AdaptiveStrassen` *definition*

Figure 10-1 illustrates a small example of computing the upper right 2×2 result submatrix of a 4×4 Strassen call.

Figure 10-2 shows a part of the recursion tree of `AdaptiveStrassen`, a modified version of Strassen via scan-hiding. At a high level, the multiplications in the leaves are spread evenly around additions, or broken-up scans. This even mix is what "homogenizes" the program, allowing for all squares in a memory profile to make within a constant factor of optimal progress. `AdaptiveStrassen` first sets up the pre- and post-scans as input and output to the entire algorithm. It then completes the remaining work of the algorithm in recursive calls. We provide pseudocode for `AdaptiveStrassen` in Figure B-6, which calls `AdaptiveStrassenRecurse` (Figure B-7) for its recursive subcalls. Each leaf of `AdaptiveStrassenRecurse` takes the scans it must compute as well as the elements it must multiply. Additionally, we define a subroutine `ReturnSplitScans`(scans,B) that takes an array of pointers to scans as input and splits the scans into even portions, but of size no shorter than B. When the total size of scans is $< 7B$, some children will be handed empty lists.

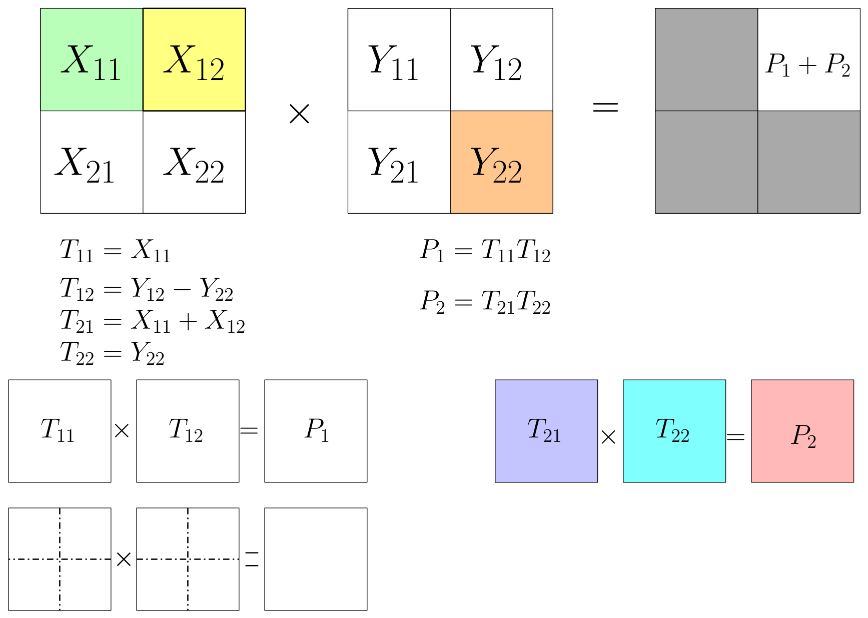
[3]This is only a constant if the full input size fits in $O(B\lg(n))$ words, which is a very small input size.



$T_{11} = X_{11}$
$T_{12} = Y_{12} - Y_{22}$
$T_{21} = X_{11} + X_{12}$
$T_{22} = Y_{22}$

$P_1 = T_{11}T_{12}$
$P_2 = T_{21}T_{22}$

Figure 10-1: Computing the upper right (2×2) output matrix of a 4×4 Strassen call. Before we can compute result submatrix, we have to compute P_1 and P_2. Before computing P_1 and P_2, however, we have to compute T_{12} and T_{21} which are sums of input submatrices. We will discuss how and when these are pre-computed given that P_1 and P_2 are the first two recursive calls made.

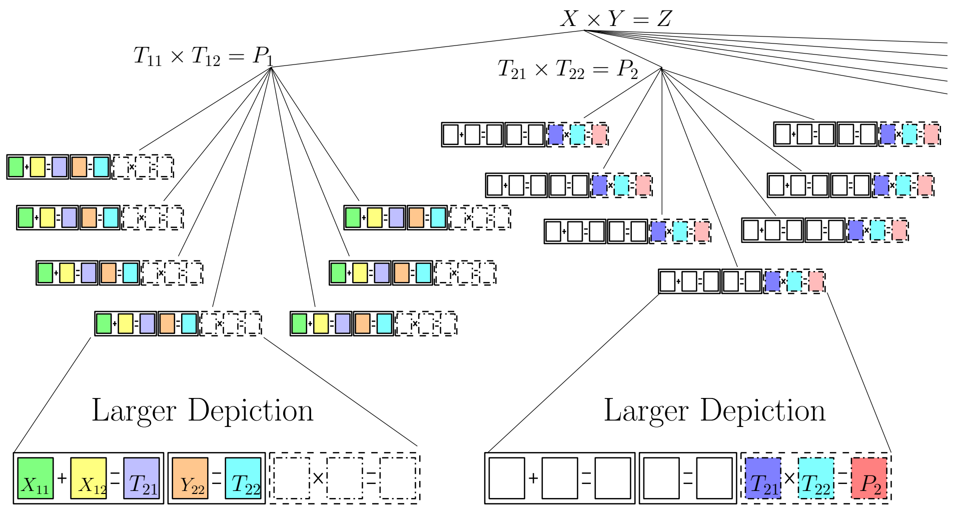


X_{11} X_{12} X_{21} X_{22} $\times$ Y_{11} Y_{12} Y_{21} Y_{22} $=$ $P_1 + P_2$
$T_{11} = X_{11}$
$T_{12} = Y_{12} - Y_{22}$
$T_{21} = X_{11} + X_{12}$
$T_{22} = Y_{22}$
$P_1 = T_{11}T_{12}$
$P_2 = T_{21}T_{22}$
$T_{11} \times T_{12} = P_1$
$T_{21} \times T_{22} = P_2$

In the intial scan T_{11} and T_{12} will be pre-computed.

Seven Mulitplications for P_1 (dashed squares). Interspersed are additions (solid lines).

The matricies T_{21} and T_{22} must be computed before the mulitplication of P_1 finishes.



$X \times Y = Z$
$T_{11} \times T_{12} = P_1$
$T_{21} \times T_{22} = P_2$
Larger Depiction
Larger Depiction
X_{11} + X_{12} = T_{21} Y_{22} = T_{22} × =
+ = = T_{21} × T_{22} = P_2

Figure 10-2: The pre-computation scan of size $O(n^2)$ would in this case pre-compute T_{11} and T_{21}. Then, all multiplications can be done. Assume that the smallest size of subproblem (3 small boxes) fit in memory. Then we show how the (dotted line boxes not filled in) multiplications needed for P_1 can be inter-spersed with the (complete line and colored in) additions or scans needed to pre-compute T_{21} and T_{22}. Note that T_{21} and T_{22} will be done computing before we try to compute the multiplication of P_2. Thus, we can repeat the process of multiplies interspersed with pre-computation during the multiplication for P_2. The additions or scans during P_2 will be for the inputs to the next multiplication, P_3 (not listed here). The multiplications in P_2 are computed based on the pre-computed matrices T_{21} and T_{22} (dotted line boxes filled in).

`AdaptiveStrassen` *is optimally progressing*

Although Section 10.2 showed that `AdaptiveStrassen` is optimally progressing with the general scan-hiding framework, the following proofs provide details of how scan-hiding generates optimally progressing algorithms.

We first describe an assignment of work such that the total work done over the execution is $O(n^{log_2(7)})$ and that on any $m(t) \times m(t)$ block of execution, $\Omega((mB)^{\lg(7)/2})$ work is done. An optimal algorithm OPT for Strassen has $\Theta(n^{\lg(7)})$ accesses and has a progress function of $\rho(m(t)) = (m(t)B)^{\lg(7)/2}$.

We show adaptivity via a potential argument. At a high level, we assign "extra" work to each square of the square profile in which `AdaptiveStrassen` is not performing within a constant factor of OPT. The initial scans in `AdaptiveStrassen` may not be optimally progressing, but we describe an assignment of work and potential such that the overall algorithm is optimally progressing.

For example, suppose at time t_1 of an algorithm $\mathcal{A}$ can perform w_1 work on a square of size m_1 and at time t_2, $\mathcal{A}$ can perform w_2 work on a square of size m_1 where $w_1 < w_2$, then we can assign $w_2 - w_1$ *potential* to $\mathcal{A}$ at t_1.

If an $\mathcal{A}$ performs within a constant factor of OPT in terms of amount of *work and potential*, then it is cache-adaptive. Throughout most of the algorithm, we assign one unit of progress to each memory access an algorithm makes. We reassign progress in our potential argument and show that `AdaptiveStrassen` makes steady progress throughout its execution. Specifically, we assigned progress to the initial and end scans because they are "harder" and are not optimally progressing. We will refer to our reassigned progress as work.

Lemma 10.8 *Let the input matrices to an instance of Strassen have size $n \times n$. If the following are true:*

- *The optimal Strassen algorithm takes time $n^{\lg(7)}$ in the RAM model.*
- *The optimal Strassen algorithm respects the progress function $\rho_{\mathcal{P}}(\square_m(t)) = c_0\,(m(t))^{\lg(7)/2}$ where c_0 is a constant such that $c_0 > 0$. Let $M(t)$ be a profile that starts at time step 0 and ends at time step T when the optimal Strassen algorithm completes.*
- *$\mathcal{A}$ is an algorithm which computes matrix multiplication and has total work $c_1 n^{\lg(7)}$, total potential $c_2 n^{\lg(7)}$, and completes $c_3(m(t)B)^{\lg(7)/2}$ work+potential in any $m(t) \times m(t)$ square of a profile where c_1, c_2 and c_3 are all constants such that $c_1, c_2, c_3 > 0$ and where $mB < n^2$.*
- *If the total progress plus potential completed is $(c_1 + c_2)n^{\lg(7)}$ during $\mathcal{A}$'s execution, $\mathcal{A}$ is guaranteed to have finished its last access.*

Then $\mathcal{A}$ is cache adaptive.

Proof. Let work be progress plus potential.

Let M be a profile and M' be the inner square profile of M such that for all t, $m(t)B < n^2$. If $m(t)B \geq n^2$, the entire problem can fit into cache and $\mathcal{A}$ will complete given a constant factor expansion.

The optimal Strassen algorithm makes at most a constant factor $c_4 > 0$ less progress on the inner profile M' than it did on the original profile M.

$\mathcal{A}$ makes at least as much progress on each square of M with time augmentation $1/(c_3 \cdot c_4)$ as OPT does in the non-augmented corresponding square. Therefore, A completes at least $n^{\lg(7)}$ work and potential over the entire profile. A completes at least $(c_1 + c_2)n^{\lg(7)}$ work and potential with time augmentation $(c_1 \cdot c_2)/(c_3 \cdot c_4)$.

Thus $\mathcal{A}$ must have completed. □

We show the recursive part of `AdaptiveStrassen`, called `AdaptiveStrassenRecurse`, is optimally progressing when $m(t) > \lg(n)$. This is not a surprise given that, if N is the initial input size, it has a recurrence of the form $T(n) = 7T(n/2) + O(\min\{\lg(N), n^2\})$.

Lemma 10.9 `AdaptiveStrassenRecurse` *is optimally progressing if*

1. *$\forall t$, $m(t) > \lg(n)$ and*
2. `AdaptiveStrassenRecurse` *is aware of the size of the cache line size B with respect to the progress bound $\rho(m(t)) = (m(t)B)^{\lg(7)/2}$.*

PROOF. We will show that in any square of size $m(t) \times m(t)$, `AdaptiveStrassenRecurse` does $\Omega((mB)^{\lg(7)/2})$ work, as long as $mB < n^2$.

We will assign work to the multiplications at the leaves of the recursion. That is, we count each multiplication operation as making progress in terms of work. The total number of such multiplications is $O(n^{\lg(7)})$.

We will show that `AdaptiveStrassenRecurse` incurs only a constant factor more misses than classic Strassen. Suppose that we reached a level of the recursion where the side length of the matrices $x < \sqrt{mB/10}$, i.e. the problem at this level of the recursion fits in memory.

The length of the list of scans is $\min(\lg(n), x/B)$ words. Therefore, reading in the list incurs at most $\min(\lg(n)/B, x/B^2)$ cache misses.

The total size of $P[s][i]$, the additional scans that each leaf needs to do, for all $s \in \{x_1 = 0, x_2 = 1, y_1 = 2, y_2 = 3, z_1 = 4, z_2 = 5\}$ for $i < \lg(x)$ is $< 5x^2$, so all of these smaller scans require no extra cache misses. Additionally, the multiplications only require reading in the size of the problem once the problem fits in cache.

We will now show tht the cache misses due to extra scans passed down to each elaf from interleaved scans is at most the size of the cache. The total number of cache misses from scans is bounded by the size of the scans assigned to each child node. Additionally, `AdaptiveStrassenRecurse` may incur one cache miss from each of the levels of P. That is, the number of faults due to interleaving scans is at most

$$\lg(n) + (1/B) \sum_{i=0}^{\lg(n)-\lg(x)} x^2 4^i/(7^i) < 4/3x^2/B + \lg(n).$$

Therefore, the number of cache misses incurred by a problem of size x is at most $(4/3+5)x^2/B+\lg(n)$ when $x<\sqrt{mB/10}$. A full call to `AdaptiveStrassenRecurse` (x, level) does $x^{\lg(7)}$ work.

Suppose we reached an $m(t)\times m(t)$ square in which at least half of the square requires cache misses for `AdaptiveStrassenRecurse`. We can compute at least 1 call to `AdaptiveStrassenRecurse` $(x, \mathit{level}, \mathit{input}, \mathit{scans})$ where $(mB-\lg(n))/(6\times 2\times 2)<x<(mB-\lg(n))/(6\times 2)$ in that square. Thus, if $mB>4\lg(n)B$ then `AdaptiveStrassenRecurse` completes at least $(mB)^{\lg(7)/2}/24=\Omega((mB)^{\lg(7)/2})$ work.

Note that every $m(t)\times m(t)$ square in the profile must either be at least half scans or at least half calls to `AdaptiveStrassenRecurse`. Therefore, `AdaptiveStrassenRecurse` is optimally progressing in every square. □

Next, we show that the linear scans at the beginning and end of `AdaptiveStrassen` do not preclude adaptivity via a potential argument.

Lemma 10.10 *Let M be a square profile. For all t,* `AdaptiveStrassen` *completes at least $\Omega((m(t)B)^{\lg(7)/2-1}m(t)B)$ work plus potential on each $m(t)\times m(t)$ square.*

Furthermore, if `AdaptiveStrassen` *completed all the work plus potential as per an intial assignment of work and potential, then it must have completed its last access.*

PROOF. First we will consider the time that initial scans and end scans take, or the work in `AdaptiveStrassen` excluding the work of `AdaptiveStrassenRecurse`.

Since the pre and post-scans require allocatin sextra space, we first compute how long these allocations take. The size of the array P of scans is

$$\sum_{i=0}^{\lg(n)}\sum_{j=0}^{\lg(n)-i}\frac{n^2}{4^i4^j}<\sum_{i=0}^{\lg(n)}\frac{4n^2}{3\cdot 4^i}<\frac{16n^2}{9}.$$

Therefore, the allocation must do a scan of length $16/9n^2$. We start with a pre-scan of length $\sum_{i=0}^{\lg(n)} n^2/(4^i)<4/3n^2$ and end with a post-scan of the same length.

We will assign progress to these scans such that the total potential is $O(n^{\lg(7)})$. We assign $n^{\lg(7)-2}$ potential to each operation of the pre and post-scans. The total potential assigned to the scans is $n^2n^{\lg(7)-2}=O(n^{\lg(7)})$. Thus, in an $m(t)\times m(t)$ square we complete $\Omega(n^{\lg(7)-2}m(t)B)$ progress. Note that $n^{\lg(7)-2}mB=\Omega((mB)^{\lg(7)/2-1}mB)$ as long as $mB<n^2$. If $mB>n^2$, `AdaptiveStrassen` could just have completed all of its work in one square with augmentation. □

Finally we want to show that scan-hiding does not introduce asymptotic computational overhead.

Lemma 10.11 `AdaptiveStrassen` *takes $O(n^{\lg(7)})$ time in the word-RAM model.*

PROOF. The running time for the pre and post-scans is $O(n^2)$.

Let N be the initial input size. The recurrence for the runtime of `AdaptiveStrassenRecurse` is $T(n)=7T(n/2)+\min(\lg(N),n^2)$. Therefore, $T(n)=7T(n/2)+O(n^2)=O(n^{\lg(7)})$. □

10.5 Experimental study

This section empirically compares the performance of different algorithms under randomized memory fluctuations and finds that cache-adaptive (or nearly cache-adaptive) algorithms incur fewer faults with random memory fluctuations than algorithms that are farther from optimally progressing, lending empirical support to the cache-adaptive model. First, this section compares algorithms for cubic matrix multiplication. It then compares various external-memory sorting algorithms.

Each point on the graphs in Figures 10-3 and 10-4 represents the ratio of the average number of faults (or runtime) during the changing memory profile to the average number of faults (or runtime) without the modified adversarial profile.

System. We ran experiments on a node with and tested their behavior on a node with a two core Intel® Xeon™ CPU E5-2666 v3 at 2.90GHz. Each core has 32KB of L1 cache and 256 KB of L2 cache. Each socket has 25 Megabytes (MB) of shared L3 cache.

Naive matrix multiplication

We compare the faults and runtime of `MM-Scan` and `MM-Inplace` as described in [43] in the face of memory fluctuations. `MM-Inplace` is the in-place divide-and-conquer naive multiplication algorithm, while `MM-Scan` is not in place and does a scan at the end of each recursive call. `MM-Inplace` is cache adaptive while `MM-Scan` is not. The worst-case profile as described by Bender *et al.* [45] took too long to complete on any reasonably large input size for `MM-Scan`.

We measured the faults and runtime of both algorithms under a fixed cache size and under a modified version of the adversarial memory profile for naive matrix multiplication. Figure 10-3 shows the runtime and faults of both algorithms under a changing cache normalized against the runtime and faults of both algorithms under a fixed cache, respectively.

Figure 10-3 shows that the relative number of faults that `MM-Scan` incurs during the random profile is higher than the corresponding relative number of faults due to `MM-Inplace` on a random profile drawn from the same distribution. As Bender *et al.* [43] shows, `MM-Scan` is a $\Theta(\lg N)$ factor from optimally progressing on a worst-case profile while `MM-Inplace` is optimally progressing on all profiles.

The relative faults incurred by `MM-Scan` grows at a non-constant rate with respect to the problem size. In contrast, the performance of `MM-Inplace` decays gracefully with respect to the problem size. The large measured difference between `MM-Scan` and `MM-Inplace` may be due to the overhead of repopulating the cache after a flush incurred by `MM-Scan`.

Sorting

We compared the cache performance of different sorting algorithms from the standard template library following STL for XXL datasets (`STXXL`) [120] with three different sorting algorithms in Figure 10-4. In order to measure performance with memory changes, we first chose an initial memory size M and ran each algorithm while chang-

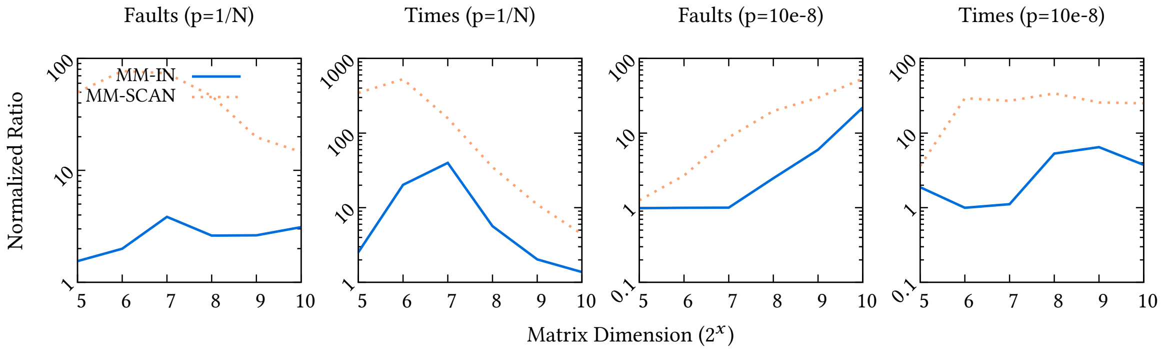


Figure 10-3: An empirical comparison of faults and runtime of `MM-Scan` and `MM-Inplace` under memory fluctuations. Each plot shows the normalized faults or runtime under a randomized version of the worst-case profile.
The first two plots show the faults and runtime during a random profile where the memory drops with probability $p = 1/N$ at the beginning of each recursive call.
Similarly, in the last two plots, we drop the memory with probability $p = 5 \times 10^{-8}$ at the beginning of each recursive call. Recall that the theoretical worst-case profile drops the memory at the beginning of each recursive call.

ing the memory size in the range $[100MB, 2\text{M}]$ every second. We used Linux cgroups to control the memory available to each algorithm.

The three sorting algorithms from `STXXL` are as follows.

1. `std::sort` from the `C++` standard library (`libstdc++`), which implements introspective sort (introsort), a hybrid sorting algorithm which uses quick sort until a maximum recursion depth, at which point it switches to heap sort [245, 278].

2. `stxxl::sort` from `STXXL`. The library implements an asynchronous variant on standard k-way merge sort as described in [121].

3. Cache-oblivious funnel sort implemented in [287].

The sorting algorithms have different structures, so we measured the performance of each algorithm on profiles independent of algorithm structure. The performance of the "more adaptive" sorting algorithms is therefore not a result of friendlier profiles but because the profiles are independent of the algorithm structure. In practice, profiles are often not tied to algorithm structure (e.g. fluctuations based on other parallel computations), so it is meaningful to compare the algorithms over randomized profiles.

Sorting algorithms that have better cache-adaptive guarantees incurred relatively fewer faults during a random profile. Specifically, `std::sort` incurred relatively more faults than both funnel sort and `stxxl::sort`, Funnel sort and `stxxl::sort` are closer to adaptivity than `std::sort`, so they incur fewer faults when the size of memory changes. A possible contributing factor to the difference between the observed adaptivity of the experiments is that funnel sort at `stxxl::sort` are engineered for external-memory computations, while `std::sort` is not.

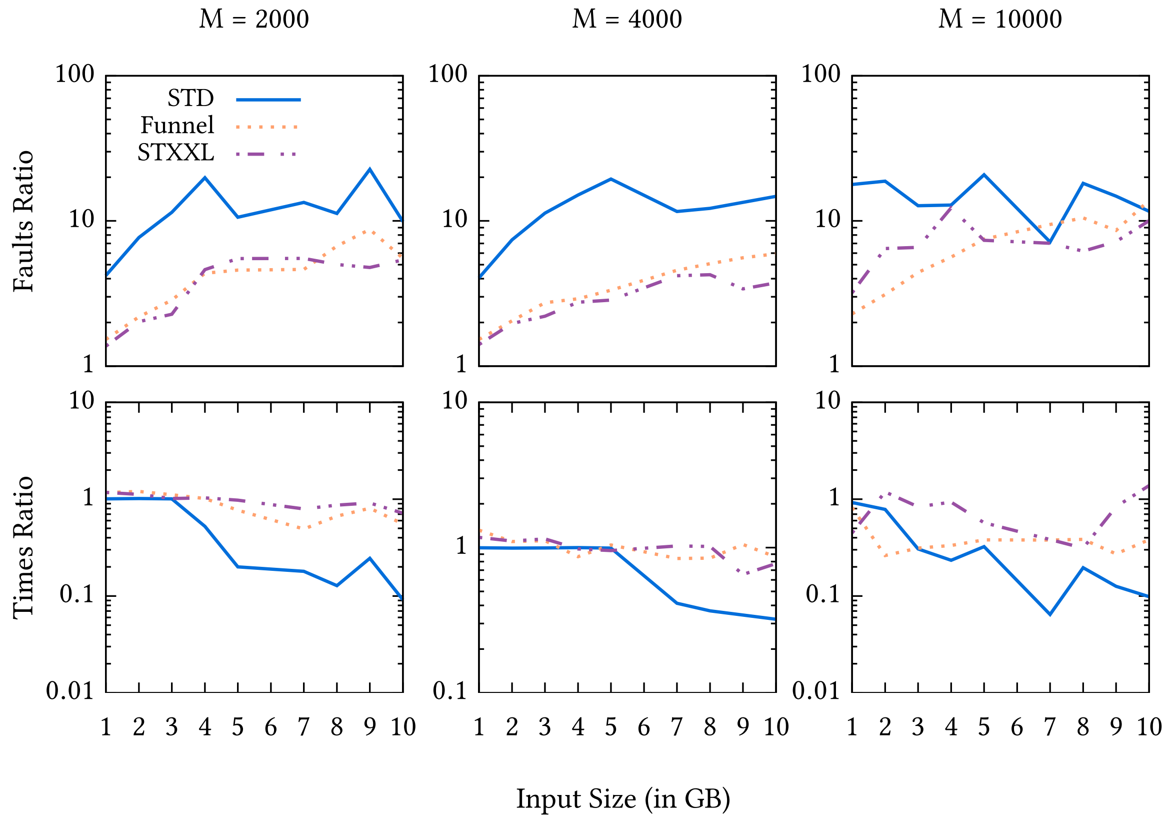


Figure 10-4: Each point on the plot represents the ratio of the faults incurred during a random profile to the faults incurred on the same input with a fixed, unchanging profile. In the third and fourth plots, we show the faults incurred during different sorting algorithms on changing memory profiles. Each of these two plots represents a different starting memory M in Megabytes. The random profile changes the memory to anywhere in the range [100, M] Megabytes each second. We normalize the faults incurred during the random profile on a certain input against the faults incurred when the available memory is fixed at M at the beginning of execution.

10.6 Conclusion

This chapter presents scan-hiding, the first constructive method for converting non-adaptive recursive divide-and-conquer algorithms with scans into adaptive recursive algorithms through a new scan-hiding technique. For example, Strassen's algorithm for matrix multiplication is not immediately adaptive because of the scan at the beginning and end of each recursive call. Our construction applies to Strassen, Coppersmith-Winograd, Vassilevska Williams and Legall's ($o(n^3)$) matrix multiplication algorithms [107, 158, 351, 383]. Scan-hiding applies to all matrix multiplication algorithms which achieve their bound by bounding the matrix multiplication tensor, which include all currently known subcubic matrix multiplication algorithms.

Furthermore, the experiments in this chapter suggest that the cache-adaptive model captures real-world performance trends. The adaptive naive matrix multiply performed significantly better even under variants of the theoretical worst-case profile. These results suggest that performance differences due to cache adaptivity are not restricted to a theoretical, pathological case.

This chapter and prior work raise both theoretical and experimental questions. For example, scan-hiding applies to many recursive algorithms but may not work for others with a superlinear step in the beginning. One example is cache-oblivious 3SUM which begins by constructing a size $n \lg n$ data-structure in $sort(n) \lg n$ time and in each recursive step scans through $n'M/\lg M$ elements where n' is the size of the problem in the recursion [28]. Additionally, algorithms that start with long scans do not immediately admit cache-adaptive algorithms. Notably, Karstadt and Schwartz [215] gave an algorithm for matrix multiplication that saves a constant factor of 5/6 but starts with a scan of length $O(n^2 \lg(n))$. Is there a variant of scan-hiding for even longer scans? Are there techniques to port other algorithms and data structures to the cache-adaptive model?

Scan-hiding is the first instance of a general transformation on algorithms to generate adaptive algorithms. Our findings suggest that other classes of algorithms that are not initially adaptive may become adaptive through techniques such as scan-hiding. Future research includes building a full framework for cache-adaptive algorithmic transformations.

Prior work gave constructions of worst-case profiles and showed that non-adaptive algorithms are not optimal on such profiles. Measuring natural memory fluctuations on real systems may give us insight into the real-world performance impact of cache-adaptivity.

Locality-first strategy. This chapter introduced scan-hiding, an application of the locality-first strategy to cache-adaptive algorithms that converts non-adaptive to adaptive algorithms by improving temporal locality. Specifically, scan-hiding spreads out long scans into the leaves of an algorithm's recursion tree, circumventing the worst-case memory profile of high-memory periods during scans. These results validate the locality-first strategy of improving locality first as a method of creating efficient algorithms even in the presence of parallelism.

We conclude by explaining why we are optimistic about cache adaptivity. Classical external memory and cache-oblivious algorithms are well-studied and motivated by modern computer architectures with hierarchical memory. Practitioners have empirically studied performance in the face of memory fluctuations for years and have developed heuristics and experimentally fast algorithms for major operations such as database sorts and joins. However, the need for theoretical guarantees of cache adaptivity will only grow with the rise of multicore architectures and shared-memory programs.

Chapter 11

Work-Efficient Parallel Algorithms for Accurate Floating-Point Prefix Sums

This chapter presents CAST_BLK and PAIR_BLK, two fast-and-accurate algorithms for parallel prefix sums that apply the locality-first strategy to take advantage of spatial locality for performance. The prefix-sums problem exhibits minimal temporal locality, so the focus is on spatial locality. Since the input is laid out as an array, spatial locality is relatively easy to achieve, so the focus of this chapter is not explicitly about locality. As we shall see, in practice, fast prefix sums algorithms optimize for locality first by trading off some parallelism to take advantage of spatial locality. The new algorithms presented in this section achieve comparable or better performance than the state-of-the-art prefix-sums implementation with much higher accuracy.

This work was conducted in collaboration with Sean Fraser and Charles E. Leiserson [151]. Thanks Guy Blelloch of Carnegie Mellon, Julian Shun of MIT, and Yan Gu of UC Riverside for providing technical benchmarks used in this work. Additional thanks to Alexandros-Stavros Iliopoulos of MIT for helpful discussions and feedback. The MIT Supercloud provided an invaluable environment for our experiments.

Abstract

Existing work-efficient parallel algorithms for floating-point prefix sums exhibit either good performance or good numerical accuracy, but not both. Consequently, prefix-sum algorithms cannot easily be used in scientific-computing applications that require both high performance and accuracy. We have designed and implemented two new algorithms, called CAST_BLK and PAIR_BLK, whose accuracy is significantly higher than that of the high-performing prefix-sum algorithm from the Problem Based Benchmark Suite, while running with comparable performance on modern multicore machines. Specifically, the root mean squared error of the PBBS code on a large array of uniformly distributed 64-bit floating-point numbers is 8 times higher than that of CAST_BLK and 5.8 times higher than that of PAIR_BLK. These two codes employ the PBBS three-stage strategy for performance, but they are designed to achieve high accuracy, both theoretically and in practice. A vectorization enhancement to these two scalar codes trades off a small amount of accuracy to match or outperform the PBBS code while still maintaining lower error.

11.1 Introduction

The ***prefix sum*** (also known as ***scan*** [61]) is a fundamental algorithmic building block for parallel computing, and consequently, it is often targeted for efficient implementation [61, 187]. This chapter studies floating-point prefix sums, which underlie applications in scientific computing including summed-area table generation [178] and the fast multipole method [118]. For many floating-point calculations, numerical accuracy is as important, or often more important, than absolute performance. In the summed-area table problem, for example, practitioners sacrifice performance for accuracy [405]. Although floating-point prefix sums require both accuracy and high performance [182], traditional summation methods are usually optimized for performance. At the other extreme, compensated-summation algorithms significantly reduce round-off error by accounting for its propagation, but they tend to be unreasonably computationally expensive [59, 183, 202]. This chapter presents algorithms for computing prefix sums of floating-point values that offer both accuracy and performance.

The ***prefix-sums operation*** computes the "running sum" of an array of n numbers.

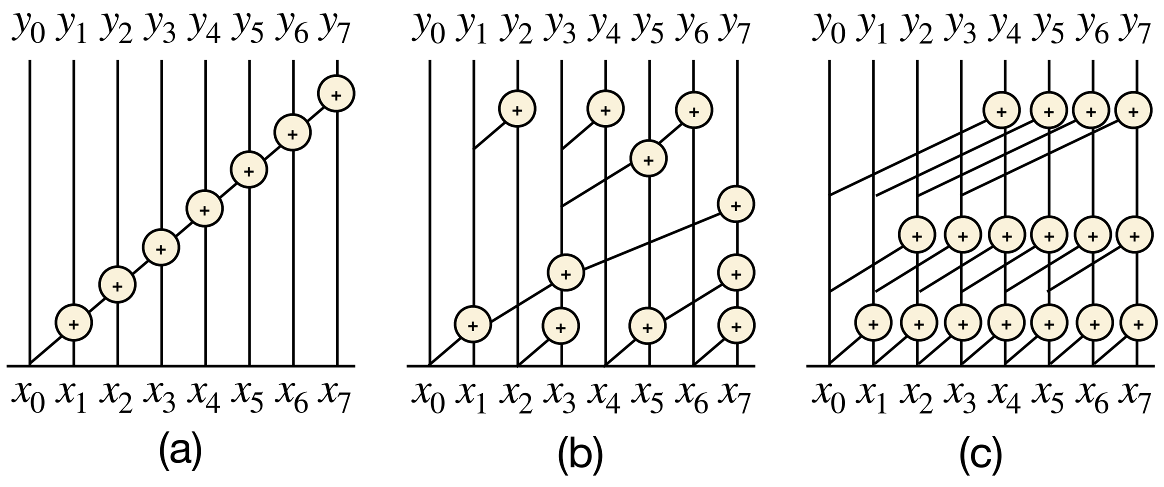
Definition 11.1 (Prefix-sums operation) *The prefix-sums operation takes an array* $x = [x_0, x_1, \ldots, x_{n-1}]$ *of* n *elements and returns the "running sum"* $y = [y_0, y_1, \ldots, y_{n-1}]$, *where*

$$y_k = \begin{cases} x_0 & \text{if } k = 0, \\ x_k + y_{k-1} & \text{if } k \geq 1 \ . \end{cases} \tag{11.1}$$

Although our codes handle arbitrary n, to simplify our analysis, we shall generally assume that n is an exact power of 2.

Three fundamental prefix-sum algorithms, illustrated in Figure 11-1, have appeared in the literature. The naive **FWD_SCAN** algorithm directly implements the recursion in (12.1) and is illustrated in Figure 11-1(a). Although FWD_SCAN is serial and has low accuracy, it runs fast in practice, because it performs only $n - 1$ floating-point additions, the minimum possible, and it takes advantage of architectural features, such as prefetching [365]. In contrast, the canonical ***pairwise*** prefix sum, shown in Figure 11-1(b), which we will call **PAIR_SCAN**, is parallelizable and achieves better accuracy, but it requires $2n - \lg n - 2$ additions, a constant-factor more overhead [61]. Moreover, its structure matches modern architectural features less well. Finally, the Kogge-Stone algorithm [221], shown in Figure 11-1(c), which we will call **KS_SCAN** (also described by Hillis and Steele [184]), achieves even higher accuracy than pairwise ordering requiring $n \lg n - n + 1 = \Theta(n \lg n)$ additions.

Prefix sums are so ubiquitous that they have been included as primitives in some languages such as C++ [109], and more recently have been considered as a primitive for GPU computations in CUDA [174]. The fastest prefix sum on a CPU for large inputs is implemented in the Problem-Based Benchmark Suite (PBBS) Library [336]. The scan in PBBS, which we will call **FWD_BLK** due to its structure, achieves good per-



y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
(a)
(b)
(c)

Figure 11-1: The canonical prefix-sum algorithms: **(a)** `FWD_SCAN`, **(b)** `PAIR_SCAN`, and **(c)** `KS_SCAN`. Each circle with a plus sign represents an addition operation taking as inputs two values below and outputting their sum above.

formance but was not optimized for accuracy. The performance of the compensated-summation algorithm, which we call **`COMP_SCAN`**, is sufficiently slow that it is rarely used in practice, even though it has great accuracy (although `COMP_SCAN` is a useful benchmark for accuracy). This chapter introduces prefix-sum algorithms with comparable performance to `FWD_BLK` but with significantly better accuracy, although generally not attaining the levels of `COMP_SCAN`.

Analysis strategy

We shall analyze prefix-sum algorithms using the ***work-span model*** [108, Chapter 27] for performance and the "sum-depth" which provides a useful proxy for accuracy. The ***work*** is the total time to execute the entire algorithm on a given input on one processor. We say that a parallel algorithm is (asymptotically) ***work-efficient*** if its work is within a constant factor of the work of the best serial algorithm for the problem. The ***span***[1] is the longest serial chain of dependencies in the computation (or the runtime on an ideal computer with no scheduling overhead and an infinite number of processors). The ***parallelism*** of an algorithm on a given input is the work divided by the span. Given a summation algorithm (e.g. reduction, prefix sum), the ***sum-depth*** is the longest chain of additions along any path from the inputs to the output(s). The worst-case backward error bound of a sum calculation is proportional to its sum-depth [59, 150, 182].

We can compare the three algorithms in terms of work, span, parallelism, and sum-depth in a task-parallel model, such as that which Cilk [191] provides. We generally analyze work, span, and parallelism asymptotically, because constant factors in these measures are often dominated by machine overheads. We express the sum-depth exactly, however, because accuracy is not influenced by machine performance. `FWD_SCAN` requires $\Theta(n)$ work, $\Theta(n)$ span, $\Theta(1)$ parallelism, and $n-1$ sum-depth.

[1]Sometimes called **critical-path length** or **computational depth**.

PAIR_SCAN can be implemented by a divide-and-conquer strategy involving $\Theta(n)$ work, $\Theta(\lg n)$ span, $\Theta(n/\lg n)$ parallelism, and $2\lg n - 2$ sum-depth (assuming, as we have mentioned, that n is an exact power of 2). Thus, it is work-efficient, as it is within a factor of 2 of the best-possible implementation. KS_SCAN requires $\Theta(n \lg n)$ work, $\Theta(lg^2 n)$ span, $\Theta(n/\lg n)$ parallelism, and $\lg n$ sum-depth. The reason that the span of KS_SCAN is $\Theta(\lg^2 n)$ rather than $\Theta(\lg n)$ is that its implementation involves $\Theta(\lg n)$ nested parallel loops over n iterations, and in the Cilk model, each parallel loop has span $\Theta(\lg n)$, resulting in a total span of $\Theta(\lg^2 n)$. The PAIR_SCAN algorithm strikes a good balance: it is work-efficient (as opposed to KS_SCAN) and achieves low sum-depth (as opposed to FWD_SCAN).

When it comes to engineering a good parallel algorithm for prefix sum, constants matter. The parallel PAIR_SCAN algorithm, which has much better sum-depth (and hence accuracy) than FWD_SCAN, performs only double the number of floating-point additions and it can perform many of those operations in parallel. But a naive implementation of PAIR_SCAN is slower than FWD_SCAN in practice, because there are many other considerations, such as coping with limited memory bandwidth and processor-pipeline overheads. The PBBS implementation of FWD_BLK manages to overcome the performance limitations of the serial FWD_SCAN algorithm, and its sum-depth is a bit better than FWD_SCAN's, but it was not designed to minimize numerical round-off, making it unsuitable for use in numerical codes that require high accuracy.

Contributions

The main contributions in this chapter are two new algorithm implementations for floating-point prefix sum, called CAST_BLK and PAIR_BLK. These two algorithms achieve performance by adopting PBBS's three-stage blocked strategy, but within the stages, they are designed to be much more accurate, both in theory and in practice. Both CAST_BLK and PAIR_BLK are theoretically work-efficient and have small sum-depth. In practice, they both run fast on a modern multicore computer and exhibit high accuracy, achieving a good balance between the two concerns.

Figure 11-2 summarizes the accuracy and performance of the two algorithms. As shown in the figure, CAST_BLK and PAIR_BLK dominate FWD_BLK on medium-sized inputs. On large inputs (Figure 11-2(c)), FWD_BLK exhibits the best performance, but CAST_BLK and PAIR_BLK perform competitively and are much more accurate.

To be specific, our contributions are as follows:

- The design and Cilk [191] implementation of two low-sum-depth, high-performance algorithms for prefix sums, called CAST_BLK and PAIR_BLK.

- An experimental study of CAST_BLK and PAIR_BLK and five other prefix-sum algorithms that demonstrates that high performance and numerical accuracy can be achieved simultaneously.

- A vectorization enhancement to CAST_BLK, called CAST_BLK_SIMD, and a corresponding vectorization enhancement to PAIR_BLK, called PAIR_BLK_SIMD, which

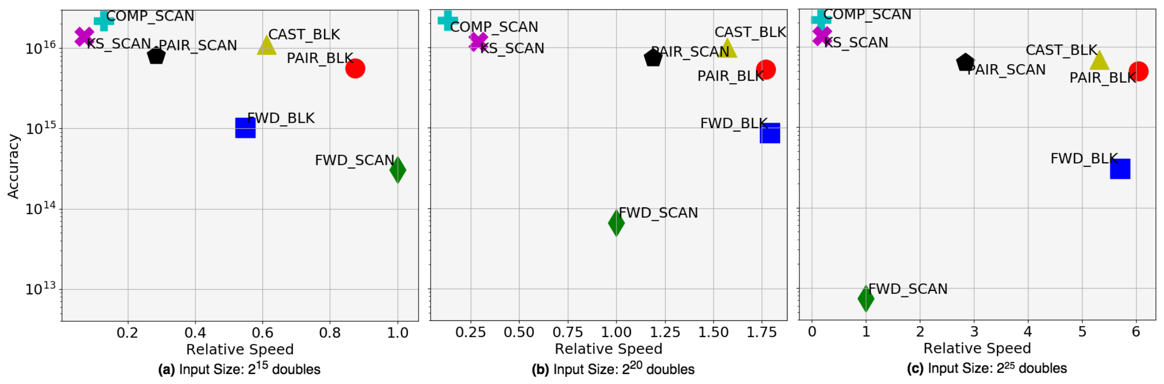


Figure 11-2: A comparison of the numerical accuracy and performance of `PAIR_BLK` and `CAST_BLK` with five other prefix-sum algorithm implementations. All algorithms were run on three different input sizes of 64-bit floating-point values (doubles) uniformly distributed on the interval $[0, 1]$ using the multicore computer described in Section 12.6. The horizontal axis in each graph shows the ratio of the running time of each algorithm to the naive `FWD_SCAN` algorithm (right is better). The vertical axis shows the reciprocal root mean square relative error of the output (up is better).

trades off a small amount of accuracy for improvements in performance, especially for small input sizes.

Map. The rest of the chapter is organized as follows. Section 11.2 provides a taxonomy of building blocks for prefix sum algorithms that we will use to exactly specify the more complicated optimized prefix sums in this chapter. Section 11.3 describes and analyzes `CAST_BLK` and `PAIR_BLK`. Section 11.4 presents an experimental evaluation of prefix-sum algorithms. Section 11.5 describes how to further optimize the two prefix-sum algorithms with vectorization. Finally, Section 11.6 provides concluding remarks.

11.2 Characterizing prefix-sum algorithms

This section defines building blocks for prefix sums in order to organize and specify more complicated algorithms in later sections. The building blocks are composed of the summation ***kernel*** (either a scan or reduce) and the ***ordering*** that it follows. As described in Section 11.1, scan computations can have forward, pairwise, or Kogge-Stone orderings. Reductions can also have a forward or pairwise ordering.

We will use `S` and `R` to denote scans and reductions, respectively, and prepend them with `f`, `p`, or `k` for forward, pairwise, or Kogge-Stone, respectively, to specify an ordering. For example, the naive forward scan `FWD_SCAN` is exactly the building block `fS`.

Prefix sums in stages

More complex blocked algorithms such as `FWD_BLK` may compose these primitives sequentially in ***stages*** by dividing the input into blocks and running kernels on each block in parallel. A blocked scan may run a different summation algorithm in each stage, or even a broadcast (denoted by C). Blocking coarsens parallel implementations by processing the blocks in parallel but doing the work of each block in serial. Furthermore, blocking decreases the sum-depth by decreasing the length of the longest chain of additions.

We use the building blocks to specify stages of algorithms by listing the primitive in each stage. For example, FWD_BLK divides the input into blocks and executes in three stages. In the first stage, it runs a forward reduce on each block. In the second stage, it runs FWD_SCAN on the results of the first stage. In the third stage, it runs FWD_SCAN to propagate the results of the the second stage to each block. Therefore, FWD_BLK is exactly specified with the building blocks fRfSfS.

11.3 Low sum-depth prefix sums

This section describes CAST_BLK and PAIR_BLK, two new blocked prefix-sum algorithms optimized for low sum-depth as well as for performance. We illustrate the difference between CAST_BLK and PAIR_BLK in Figure 11-3, specify them according to the building blocks in Section 11.2 and summarize the theoretical bounds on all discussed algorithms in Table 11.1.

Reducing sum-depth via broadcast

The first algorithm, which we will call CAST_BLK, reduces the sum-depth by replacing one of the summation stages in FWD_BLK with a broadcast. Specifically, it replaces the reduction in stage 1 and the forward scan in stage 2 with the PAIR_SCAN subroutine. In order to compute the prefix sum, CAST_BLK only needs to broadcast the end of each block to every entry in the next block. In our implementation of CAST_BLK, we replace stage 1 recursively with a second level of blocking and run CAST_BLK again, which reduces the sum-depth and does not affect the asymptotic work and span.

Analysis. CAST_BLK is work-efficient and achieves lower span and sum-depth than FWD_BLK. Given block sizes B, B' for the first and second level of blocking (respectively), CAST_BLK has $\Theta(\lg n)$ span and $2\lg n - 4$ sum-depth. We omit the proofs of the theoretical bounds for space, but they are all generated by aggregating the bounds on the building blocks from Table 11.1.

Pairwise summation

The next algorithm, which we will call PAIR_BLK, replaces the forward summation subroutines in FWD_BLK with low sum-depth prefix sums. PAIR_BLK also divides the input into blocks of size B and proceeds in stages. Specifically, it runs a pairwise reduction in the first stage and PAIR_SCAN in the second stage. The last stage runs CAST_BLK on blocks of size $B' < B$.

The PAIR_BLK algorithm can be parallelized block-wise in the same way as FWD_BLK.

Table 11.1: Prefix-sum algorithms, their descriptions according to the taxonomy in Section 11.2, and their theoretical work, span, and sum-depth on inputs of size $n \geq 4$. For blocked algorithms, we denote the block size at the first level of blocking with B, where $B, n/B \geq 4$.

Algorithm	*Description*	*Source*	*Work*	*Span*	*Parallelism*	*Sum-Depth*
`FWD_SCAN`	`fS`	[182]	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$n-1$
`PAIR_SCAN`	`pS`	[61]	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(n/\lg n)$	$2\lg n - 2$
`KS_SCAN`	`kS`	[221]	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$	$\Theta(n/\lg n)$	$\lg n$
`FWD_BLK`	`fRfSfS`	[336]	$\Theta(n)$	$\Theta(B + n/B)$	$\Theta(B + n/B)$	$2B + n/B + 1$
`CAST_BLK`	`(pSpSC)pSC`	[this work]	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(n/\lg n)$	$2\lg n - 4$
`PAIR_BLK`	`pRpS(pSpSC)`	[this work]	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(n/\lg n)$	$2\lg n + \lg B - 5$

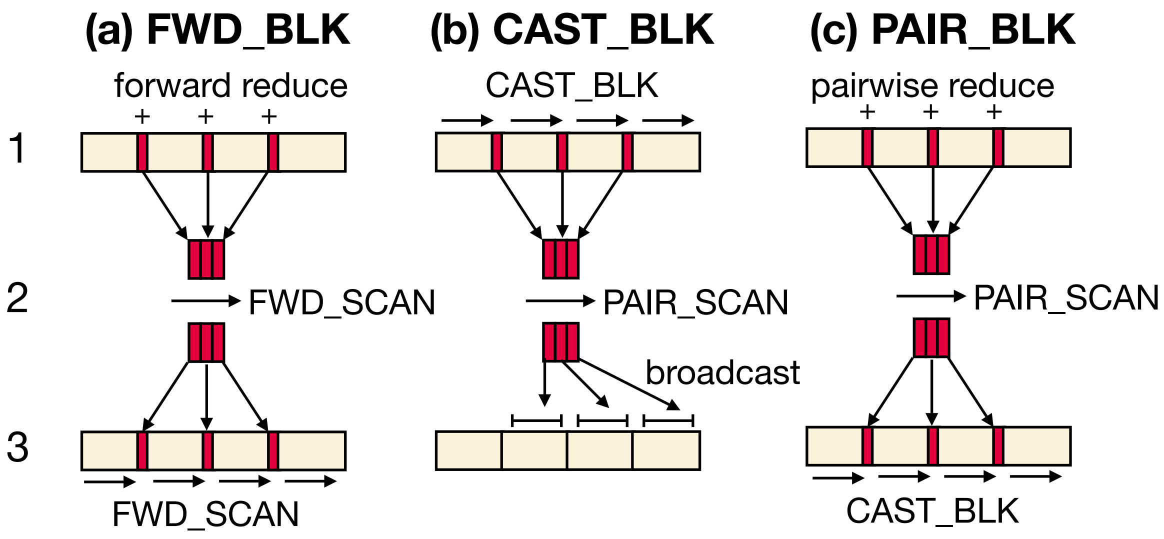


Figure 11-3: Blocked prefix-sum algorithms in stages.

Analysis. `PAIR_BLK` is work-efficient and achieves lower sum-depth than `FWD_BLK`. Given a first-level block size B, `PAIR_BLK` has $\Theta(\lg n)$ span and $2\lg n + \lg B - 5$ sum-depth.

11.4 Evaluation

This section presents an experimental evaluation of prefix sum algorithms on a CPU in terms of both performance and accuracy. As we will see, `CAST_BLK` and `PAIR_BLK` achieve competitive performance with `FWD_BLK` but are up to an order of magnitude more accurate.

Experimental setup

We used a general-purpose multicore from MIT Supercloud [274] with 20 physical cores (with 2-way hyperthreading) and 2 Intel Xeon Gold 6248 @ 2.50GHz processors.

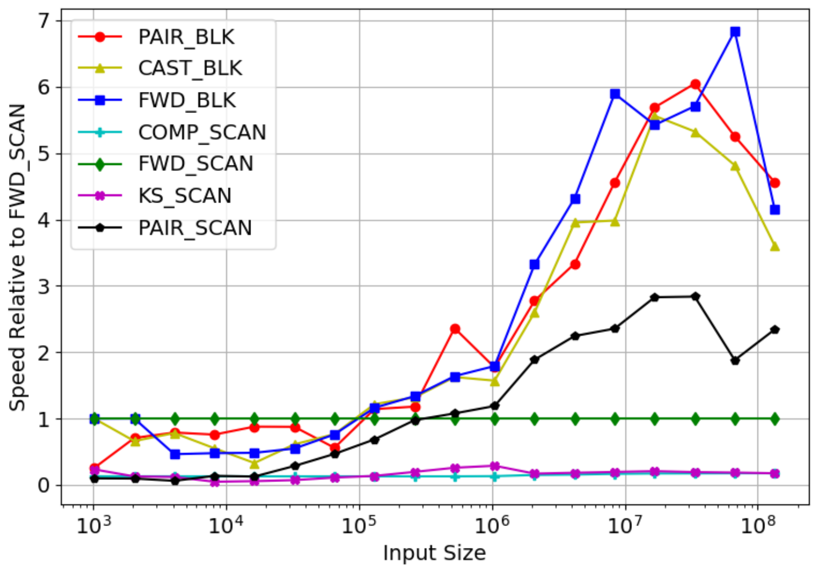


Figure 11-4: A comparison of the performance of `PAIR_BLK` and `CAST_BLK` with five other prefix-sum algorithms implementations on uniformly distributed doubles on the interval $[0, 1]$. On this plot, up is better.

We implemented all algorithms in C++ using Cilk [191] for fork-join parallelism. We used the Tapir/LLVM [321, 322] branch of the LLVM [234, 235] compiler (version 8) with the `-O3` and `-march=native` flags.

Our data set consists of `IEEE754` double-precision[2] 64-bit floats randomly generated with the Mersenne Twister 19937 generator [261].

For the blocked algorithms, we set $B = 1024$ to match `FWD_BLK` for the fairest comparison and set $B' = 16$, although different block sizes may result in lower sum-depths or better performance in practice.

Performance

Figure 11-4 shows the speedup[3] obtained for the different algorithms over serial `FWD_SCAN`. For small inputs, `PAIR_BLK`, `CAST_BLK`, and `FWD_BLK` exhibit similar performance. Since `FWD_BLK` is optimized for larger inputs[4] where memory bandwidth is the bottleneck, it performs up to 1.4× better than `PAIR_BLK` and `CAST_BLK`.

As shown in Figure 11-4, the speedup for all parallel prefix sum algorithms is relatively small compared to the number of physical cores. This limited scalability is due to the memory bandwidth because the actual computation involved in a scan (one addition per element) is small compared to the cost of data movement. Therefore, prefix sum algorithms are often memory-bound on CPUs and can experience performance variability due to data transfer on large inputs.

[2]The results are the same for single-precision floats given no overflow.
[3]We measured runtime as the median of 7 trials.
[4]In these experiments, about 4 million doubles fit in cache.

Accuracy

We measured the numerical error of the prefix sum algorithms on doubles under distributions from Higham's methodology [182]. Specifically, we drew numbers according to Unif$(0, 1)$ (the uniform distribution between 0 and 1), Exp(1) (the exponential distribution with $\lambda = 1$), and Norm$(0, 1)$ (the standard normal distribution).

Since worst-case floating-point rounding error bounds tend to be pessimistic, we follow the methodology described by Higham [182]. We experimentally evaluate the accuracy of summations as follows:

- We use higher-precision floating point values[5] [290] as a reference point to compare relative error.
- We draw random inputs from uniform, exponential and normal distributions.
- We use the compensated summation algorithm[6] COMP_SCAN [202] as an accuracy benchmark.
- We quantify error as the root mean square relative error.

In floating-point arithmetic, the summation ordering determines the computed sum. For all $k = 0, 1, \ldots, n-1$, let S_k be the ***real value*** of the scan at index k ($S_k = \sum_{i=0}^{k} x_i$), and let $\hat{S}_k$ be the computed sum. The ***relative error*** of $\hat{S}_k$ is defined as $E_k = \hat{S}_k - S_k$. Given n summation results $\hat{S}_0, \ldots, \hat{S}_{n-1}$ and real values $S_0, \ldots, S_{n-1}$, the ***root mean square relative error*** is as follows:

$$\text{RMSE} = \left(\frac{1}{n} \sum_{k=0}^{n-1} E_k^2 \right)^{1/2} .$$

We measure error on the different distributions as the RMSE. The machine epsilon ($\varepsilon = 2.22 \times 10^{-16}$ for doubles) is an upper bound on the relative error of any single summation due to rounding [183].

Discussion

As shown in Figure 11-5, both the CAST_BLK and PAIR_BLK algorithm exhibit up to $10\times$ more error than compensated summation. Although the compensated summation algorithm has the highest accuracy, it is at about $20\times$ slower than CAST_BLK and PAIR_BLK.

Overall, CAST_BLK and PAIR_BLK are much more accurate than forward summation-based algorithms such as FWD_BLK and FWD_SCAN. The CAST_BLK algorithm achieves up to $8\times$ less error than FWD_BLK and up to $103\times$ less error than FWD_SCAN on large inputs. Similarly, PAIR_BLK achieves up to $5.8\times$ less error than FWD_BLK and up to $76\times$ less error than FWD_SCAN. Therefore, CAST_BLK and PAIR_BLK attain much better accuracy with comparable performance to FWD_BLK.

[5] We used 100-digit precision floating-point values via Boost.

[6] Compensated summation is sometimes called Kahan summation.

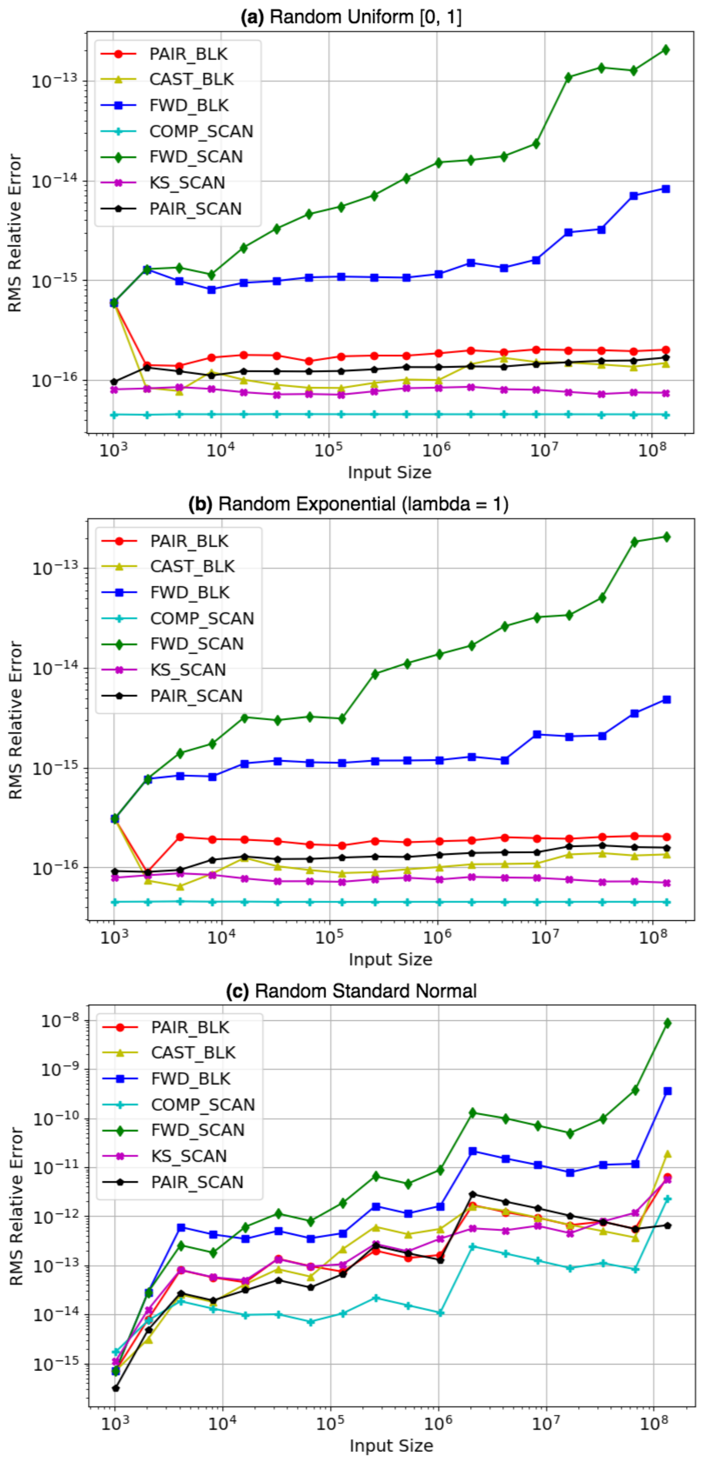


Figure 11-5: A comparison of the numerical error of PAIR_BLK and CAST_BLK with five other prefix-sum algorithms. On this plot, down is better.

11.5 Vectorizing prefix sums

This section describes a vectorized forward scan algorithm called SCAN_SIMD. We evaluate SCAN_SIMD as a subroutine in blocked scan algorithms and show that it strictly improves FWD_BLK. In pairwise blocked algorithms, vectorization trades off accuracy for improved performance.

The vectorized prefix-sum subroutine SCAN_SIMD divides the input array into chunks of size vector width V (e.g. 256 bits in Intel AVX2 [236]), performs a vectorized version of KS_SCAN on each chunk, and processes the chunks serially from left to right. Although KS_SCAN is not work-efficient, it is well-suited to SIMD operations because it has high data-level parallelism. Figure 11-6 contains an example of SCAN_SIMD on one vector. In general, given a vector width V, SCAN_SIMD requires $2\log V + 4$ vector operations to compute a scan on one block, while the scalar FWD_SCAN requires $3V$ scalar operations [150].

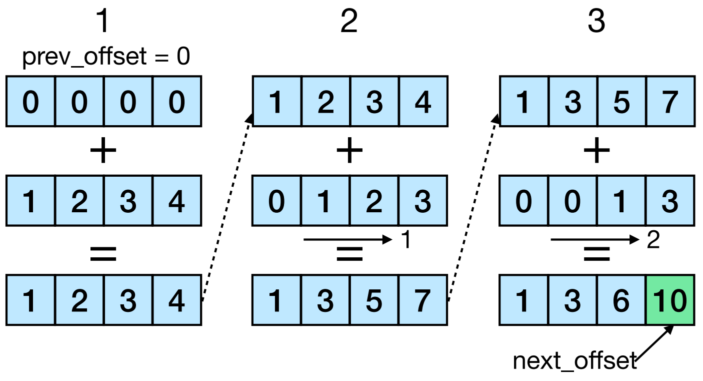


Figure 11-6: An example of SCAN_SIMD on one vector ($V = 4$). A solid arrow means a vector shift by the number next to it, additions are vector adds, and a dotted arrow denotes use of an output at a previous step as input to the next step.

Evaluation

We implemented SCAN_SIMD with Intel Intrinsics [192] and use it as a subroutine in FWD_BLK, CAST_BLK, and PAIR_BLK. We call the resulting algorithms FWD_BLK_SIMD, CAST_BLK_SIMD, and PAIR_BLK_SIMD, respectively. All experiments were run on the same setup from Section 12.6.

As shown in Figure 11-7, SCAN_SIMD and FWD_BLK_SIMD strictly dominate their scalar counterparts FWD_SCAN and FWD_BLK in both performance and accuracy because SCAN_SIMD improves the throughput and lowers the sum-depth over FWD_SCAN. Specifically, SCAN_SIMD is up to 2.2× faster and up to 2.5× more accurate than FWD_SCAN. Furthermore, FWD_BLK_SIMD is up to 2× faster when inputs fit in cache and comparable on larger inputs while achieving 2× less error than FWD_BLK.

Algorithm description

Vectorizing scans in CAST_BLK and PAIR_BLK trades off accuracy for performance. CAST_BLK_SIMD and PAIR_BLK_SIMD are up to 2× faster than FWD_BLK when inputs fit

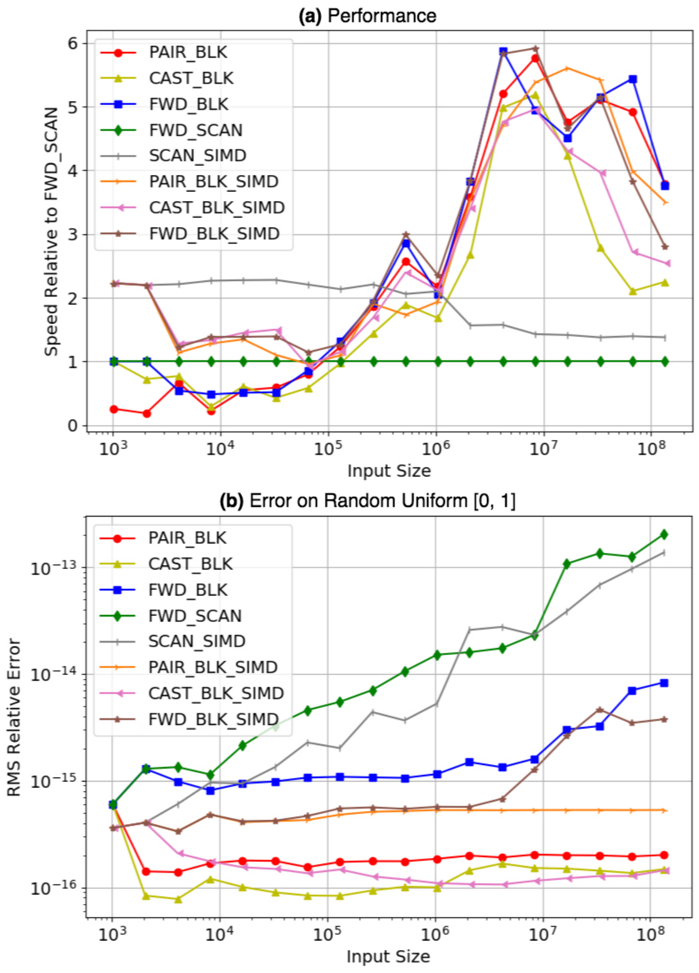


Figure 11-7: A comparison of the performance and error between `FWD_SCAN`, `CAST_BLK`, `PAIR_BLK`, and `FWD_BLK` and their vectorized counterparts.

in the cache, and they are competitive with `FWD_BLK` when the inputs are large. Finally, `CAST_BLK_SIMD` and `PAIR_BLK_SIMD` are about 10× and about 2× more accurate than `FWD_BLK`, respectively.

11.6 Conclusion

In scientific computing, floating-point prefix sums require both high accuracy and performance. This chapter introduced two new algorithms, `CAST_BLK` and `PAIR_BLK`, which achieve competitive performance and much better accuracy than the state-of-the-art CPU parallel scan. Furthermore, we showed that augmenting parallel-prefix sums with vectorization improves performance. Since many applications are implemented on CPUs, a faster and more accurate prefix-sum library for general-purpose multicores has the potential to speed up a wide variety of programs while providing numerical precision. We conclude with an avenue for future research and a brief discussion of the role of GPUs in computing scans.

A standard practice for enhancing the precision of dot products and other computations that involve summing a large number of floating-point values is to maintain the internal sums with extended precision. The two fast-and-accurate algorithms we have studied, `CAST_BLK` and `PAIR_BLK`, would seem to fare differently if intermediate values can be kept with extended precision. The `CAST_BLK` algorithm would require the extended precision values resulting from the first stage to be maintained until they can be used in the third stage, whereas the `PAIR_BLK` algorithm would require only the intermediate stage to manage extended precision. Consequently, for situations where extended precision is available, we believe that `PAIR_BLK` would likely show a performance advantage over `CAST_BLK`, but we leave this study to future research.

What role might GPUs play in fast-and-accurate scans? After all, GPUs provide considerably more floating-point capability than does a typical CPU. Unfortunately, for general-purpose computations, transferring data from a multicore to an attached GPU accelerator is so slow that a computation such as a floating-point scan cannot avail itself of the faster computational capability. GPUs can effectively perform scans within a GPU computation (for example, NVIDIA provides such a library [174]). But since they are unsuitable for performing scans as a subroutine within a general-purpose program, multicores need their own fast-and-accurate parallel algorithms, such as `CAST_BLK` and `PAIR_BLK`.

Locality-first strategy. The fast-and-accurate parallel prefix sums algorithms in this chapter employ the locality-first strategy for prefix sums to take advantage of spatial locality. The blocking techniques for in `CAST_BLK` and `PAIR_BLK` exploit spatial locality by processing the input in chunks rather than at the granularity of individual elements. Although blocking trades off with parallelism, there is ample parallelism in the problem so taking advantage of locality improves overall performance.

Chapter 12

Multidimensional Included and Excluded Sums

This chapter applies the locality-first strategy to create the bidirectional box-sum (BDBS) algorithm and box-complement algorithms, two efficient algorithms for the "included-sums" and "excluded-sums" problems, by reducing the work while maintaining spatial locality. The first step in the locality-first is to understand locality in the problem: there is not much temporal locality in these problems, but there is spatial locality. Since spatial locality is relatively easy to achieve in these problems, locality is not the main focus of the chapter, but the algorithms in this chapter take advantage of spatial locality by storing and operating on contiguous data. Additionally, the algorithms in this chapter improve overall asymptotic efficiency by reducing the work required to solve the problems.

This work was conducted in collaboration with Sean Fraser and Charles E. Leiserson [389, 390].

Abstract

This chapter presents algorithms for the ***included-sums*** and ***excluded-sums*** problems used by scientific computing applications such as the fast multipole method. These problems are defined in terms of a d-dimensional array of N elements and a binary associative operator $\oplus$ on the elements. The included-sum problem requires that the elements within overlapping boxes cornered at each element within the array be reduced using $\oplus$. The excluded-sum problem reduces the elements outside each box. The ***weak*** versions of these problems assume that the operator $\oplus$ has an inverse $\ominus$, whereas the ***strong*** versions do not require this assumption. In addition to studying existing algorithms to solve these problems, this chapter introduces three new algorithms.

The ***bidirectional box-sum (BDBS)*** algorithm solves the strong included-sums problem in $\Theta(dN)$ time, asymptotically beating the classical ***summed-area table (SAT)*** algorithm, which runs in $\Theta(2^d N)$ and which only solves the weak version of the problem. Empirically, the BDBS algorithm outperforms the SAT algorithm in higher dimensions by up to 17.1×.

The ***box-complement*** algorithm can solve the strong excluded-sums problem

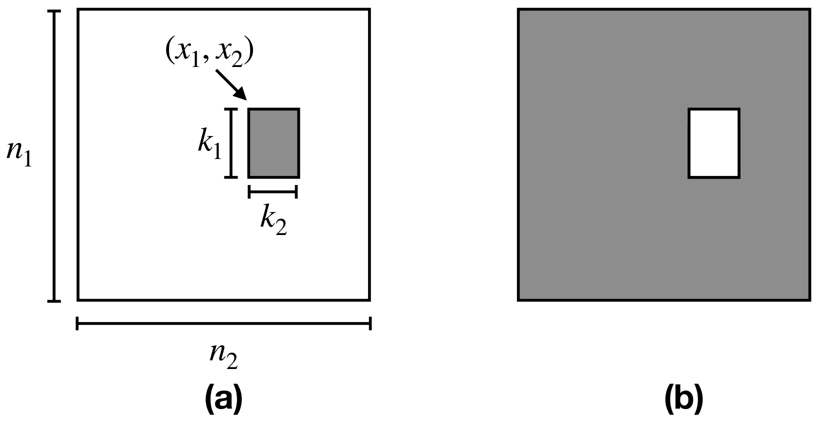


Figure 12-1: An illustration of included and excluded sums in 2 dimensions on an $n_1 \times n_2$ matrix using a (k_1, k_2)-box. **(a)** For a coordinate (x_1, x_2) of the matrix, the included-sums problem requires all the points in the $k_1 \times k_2$ box cornered at (x_1, x_2), shown as a grey rectangle, to be reduced using a binary associative operator $\oplus$. The included-sums problem requires that this reduction be performed at *every* coordinate of the matrix, not just at a single coordinate as is shown in the figure. **(b)** A similar illustration for excluded sums, which reduces the points outside the box.

in $\Theta(dN)$ time, asymptotically beating the state-of-the-art ***corners*** algorithm by Demaine *et al.*, which runs in $\Omega(2^d N)$ time. In 3 dimensions the box-complement algorithm empirically outperforms the corners algorithm by about 1.4× given similar amounts of space.

The weak excluded-sums problem can be solved in $\Theta(dN)$ time by the ***bidirectional box-sum complement (BDBSC)*** algorithm, which is a trivial extension of the BDBS algorithm. Given an operator inverse $\ominus$, BDBSC can beat box-complement by up to a factor of 4.

12.1 Introduction

Many scientific computing applications require reducing many (potentially overlapping) regions of a tensor, or multidimensional array, to a single value for each region quickly and accurately. For example, the integral-image problem (or summed-area table) [74, 110] preprocesses an image to answer queries for the sum of elements in arbitrary rectangular subregions of a matrix in constant time. The integral image has applications in real-time image processing and filtering [178]. The fast multipole method (FMM) is a widely used numerical approximation for the calculation of long-ranged forces in various N-particle simulations [36, 170]. The essence of the FMM is a reduction of a neighboring subregion's elements, excluding elements too close, using a multipole expansion to allow for fewer pairwise calculations [105, 111]. Specifically, the multipole-to-local expansion in the FMM adds relevant expansions outside some close neighborhood but inside some larger bounding region for each element [36, 356]. High-dimensional applications include the FMM for particle simulations in 3D space [96, 171] and direct summation problems in higher dimensions [259].

These problems give rise to the excluded-sums problem [118], which underlies applications that require reducing regions of a tensor to a single value using a binary associative operator. For example, the excluded-sums problem corresponds to the translation of the local expansion coefficients within each box in the FMM [170]. The problems are called "sums" for ease of presentation, but the general problem statements (and therefore algorithms to solve the problems) apply to any context involving a ***monoid*** $(S, \oplus, e)$, where S is a set of values, $\oplus$ is a binary associative operator defined on S, and $e \in S$ is the identity for $\oplus$.

Although the excluded-sums problem is particularly challenging and meaningful for multidimensional tensors, let us start by considering the problem in only 2 dimensions. And, to understand the excluded-sums problem, it helps to understand the included-sums problem as well. Figure 12-1 illustrates included and excluded sums in 2 dimensions, and Figure 12-2 provides examples using ordinary addition as the $\oplus$ operator. We have an $n_1 \times n_2$ matrix $\mathcal{A}$ of elements over a monoid $(S, \oplus, e)$. We also are given a "box size" $\mathbf{k} = (k_1, k_2)$ such that $k_1 \leq n_1$ and $k_2 \leq n_2$. The ***included sum*** at a coordinate (x_1, x_2), as shown in Figure 12-1(a), involves ***reducing*** — accumulating using $\oplus$ — all the elements of $\mathcal{A}$ inside the $\mathbf{k}$-box ***cornered*** at (x_1, x_2), that is,

$$\bigoplus_{y_1=x_1}^{x_1+k_1-1} \bigoplus_{y_2=x_2}^{x_2+k_2-1} \mathcal{A}[y_1, y_2] ,$$

where if a coordinate goes out of range, we assume that its value is the identity e. The ***included-sums problem*** computes the included sum for all coordinates of $\mathcal{A}$, which can be straightforwardly accomplished with four nested loops in $\Theta(n_1 n_2 k_1 k_2)$ time. Similarly, the ***excluded sum*** at a coordinate, as shown in Figure 12-1(b), reduces all the elements of $\mathcal{A}$ outside the $\mathbf{k}$-box cornered at (x_1, x_2). The ***excluded-sums problem*** computes the excluded sum for all coordinates of $\mathcal{A}$, which can be straightforwardly accomplished in $\Theta(n_1 n_2 (n_1 - k_1)(n_2 - k_2))$ time. We shall see much better algorithms for both problems.

Excluded sums and operator inverse

One way to solve the excluded-sums problem is to solve the included-sums problem and then use the inverse $\ominus$ of the $\oplus$ operator to "subtract" out the results from the reduction of the entire tensor. This approach fails for operators without an inverse, however, such as the maximum operator max. As another example, the FMM involves solving the excluded-sums problem over a domain of functions which cannot be "subtracted," because the functions exhibit singularities [118]. Even for simpler domains, using the inverse (if it exists) may have unintended consequences. For example, subtracting finite-precision floating-point values can suffer from catastrophic cancellation [118, 405] and high round-off error [182]. Some contexts may permit the use of an inverse, but others may not.

Consequently, we refine the included- and excluded-sums problems into ***weak*** and ***strong*** versions. The weak version requires an operator inverse, while the strong version does not. Any algorithm for the included-sums problem trivially solves the weak excluded-sums problem, and any algorithm for the strong excluded-sums problem trivially solves the weak excluded-sums problem. This chapter presents efficient algorithms for both the weak and strong excluded-sums problems.

Summed-area table for weak excluded sums

The ***summed-area table (SAT)*** algorithm uses the classical summed-area table method [74, 110, 360] to solve the weak included-sums problem on a d-dimensional tensor $\mathcal{A}$ having N elements in $O(2^d N)$ time. This algorithm precomputes prefix sums along each dimension of $\mathcal{A}$ and uses inclusion-exclusion to "add" and "subtract" prefixes to find the included sum for arbitrary boxes. The SAT algorithm cannot be used to solve the strong included-sums problem, however, because it requires an operator inverse. The summed-area table algorithm can easily be extended to an algorithm for weak excluded-sums by totaling the entire tensor and subtracting the solution to weak included sums. We will call this algorithm the ***SAT complement (SATC) algorithm***.

$$\left(\begin{array}{ccccc}
1&3&6&2&5\\
3&9&1&1&2\\ \cline{3-4}
5&1&\multicolumn{1}{|c}{5}&\multicolumn{1}{c|}{3}&2\\
4&3&\multicolumn{1}{|c}{2}&\multicolumn{1}{c|}{0}&9\\ \cline{3-4}
6&2&1&7&8
\end{array}\right)
\qquad
\begin{pmatrix}
16&19&10&10&7\\
18&16&10&8&4\\
13&11&10&14&11\\
15&8&10&24&17\\
8&3&8&15&8
\end{pmatrix}
\qquad
\begin{pmatrix}
75&72&81&81&84\\
73&75&81&83&87\\
78&80&81&77&80\\
76&83&81&67&74\\
83&78&83&76&83
\end{pmatrix}$$

(a) **(b)** **(c)**

Figure 12-2: Examples of the included- and excluded-sums problems on an input matrix in 2 dimensions with box size $(3, 3)$ using the max operator. **(a)** The input matrix. The square shows the box cornered at $(3, 3)$. **(b)** The solution for the included-sums problem with the $+$ operator. The highlighted square contains the included sum for the box in (a). The included-sums problem requires computing the included sum for every element in the input matrix. **(c)** A similar example for excluded sums. The highlighted square contains the excluded sum for the box in (a).

Corners algorithm for strong excluded sums

The naive algorithm for strong excluded sums that just sums up the area of interest for each element runs in $O(N^2)$ time in the worst case, because it wastes work by recomputing reductions for overlapping regions. To avoid recomputing sums, Demaine *et al.* [118] introduced an algorithm that solve the strong excluded-sums problem in arbitrary dimensions, which we will call the ***corners algorithm***.

At a high level, the corners algorithm partitions the excluded region for each box into 2^d disjoint regions that each share a distinct vertex of the box, while collectively filling the entire tensor, excluding the box. The algorithm heavily depends on prefix and suffix sums to compute the reduction of elements in each of the disjoint regions.

Since the original article that proposed the corners algorithm does not include a formal analysis of its runtime or space usage in arbitrary dimensions, we present one

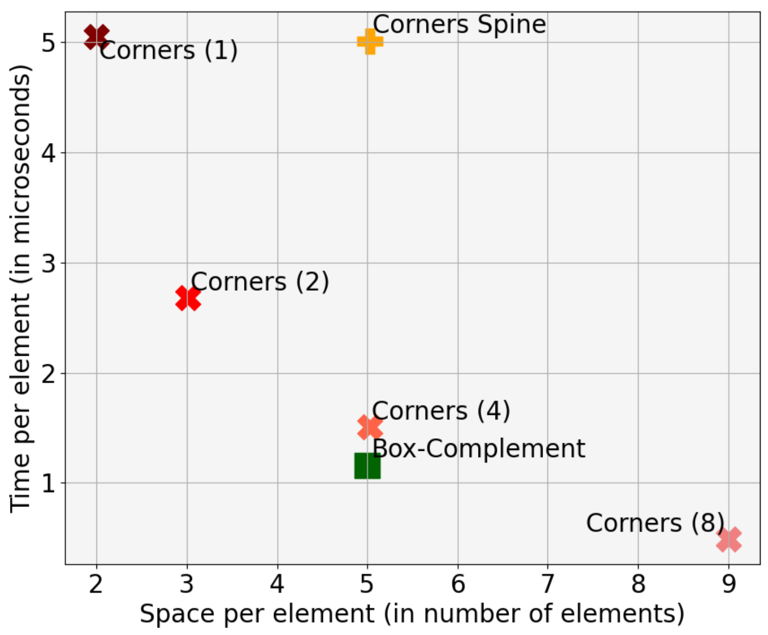


Figure 12-3: Space and time per element of the corners and box-complement algorithms in 3 dimensions. We use `Corners(c)` and `Corners Spine` to denote variants of the corners algorithm with extra space. We set the number of elements $N = 681472 = 88^3$ and the box lengths $k_1 = k_2 = k_3 = 4$ (for $K = 64$).

Algorithm	*Source*	*Time*	*Space*	*Included or Excluded?*	*Strong or Weak?*
Naive included sum	[This work]	$\Theta(KN)$	$\Theta(N)$	Included	Strong
Naive included sum complement	[This work]	$\Theta(KN)$	$\Theta(N)$	Excluded	Weak
Naive excluded sums	[This work]	$\Theta(N^2)$	$\Theta(N)$	Excluded	Strong
Summed-area table (SAT)	[110, 360]	$\Theta(2^d N)$	$\Theta(N)$	Included	Weak
Summed-area table complement (SATC)	[110, 360]	$\Theta(2^d N)$	$\Theta(N)$	Excluded	Weak
Corners(c)	[118]	$\Theta((d + 1/c)2^d N)$	$\Theta(cN)$	Excluded	Strong
Corners Spine(c)	[118]	$\Theta((2^c + 2^d(d - c) + 2^d)N)$	$\Theta(cN)$	Excluded	Strong
Bidirectional box sum (BDBS)	[This work]	$\Theta(dN)$	$\Theta(N)$	Included	Strong
Bidirectional box sum complement (BDBSC)	[This work]	$\Theta(dN)$	$\Theta(N)$	Excluded	Weak
Box-complement	[This work]	$\Theta(dN)$	$\Theta(N)$	Excluded	Strong

Table 12.1: A summary of all algorithms for excluded sums in this chapter. All algorithms take as input a d-dimensional tensor of N elements. We include the runtime, space usage, whether an algorithm solves the included- or excluded-sums problem, and whether it solves the strong or weak version of the problem. We use K to denote the volume of the box (in the runtime of the naive algorithm). The corners algorithm takes a parameter c of extra space that it uses to improve its runtime.

in Appendix C.1. Given a d-dimensional tensor of N elements, the corners algorithm takes $\Omega(2^d N)$ time to compute the excluded sum in the best case because there are 2^d corners and each one requires $\Omega(N)$ time to add its contribution to each excluded box. As we'll see, the bound is tight: given $\Theta(dN)$ space, the corners algorithm takes $\Theta(2^d N)$ time. With $\Theta(N)$ space, the corners algorithm takes $\Theta(d2^d N)$ time.

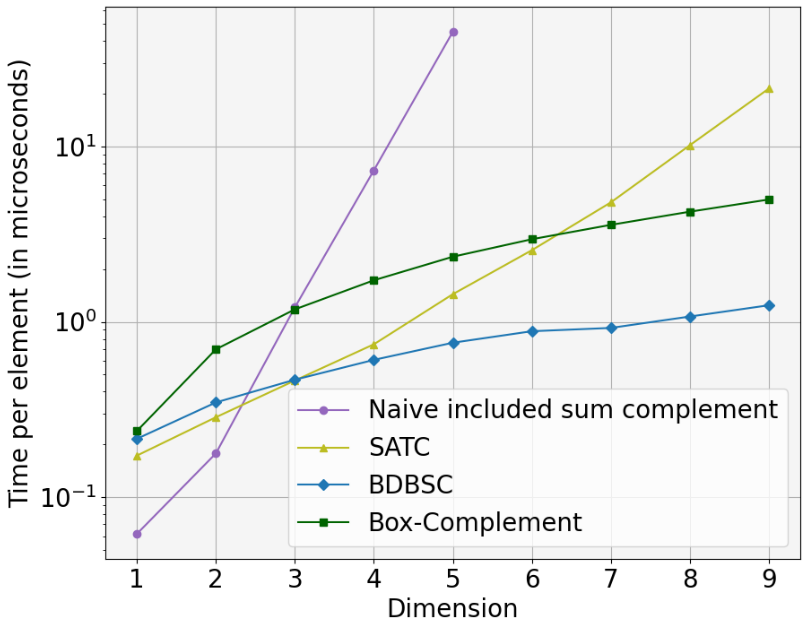


Figure 12-4: Time per element of algorithms for excluded sums in arbitrary dimensions. The number of elements N of the tensor in each dimension was in the range $[2097152, 134217728]$ (selected to be a exact power of the number of dimensions). For each number of dimensions d, we set the box volume $K = 8^d$.

Contributions

This chapter presents algorithms for included and strong excluded sums in arbitrary dimensions that improve the runtime from exponential to linear in the number of dimensions. For strong included sums, we introduce the ***bidirectional box-sum*** (BDBS) algorithm that uses prefix and suffix sums to compute the included sum efficiently. The BDBS algorithm can be easily extended into an algorithm for weak excluded sums, which we will call the ***bidirectional box-sum complement*** (BDBSC) algorithm. For strong excluded sums, the main insight in this chapter is the formulation of the excluded sums in terms of the "box complement" on which the ***box-complement*** algorithm is based. Table 12.1 summarizes all algorithms considered in this chapter.

Figure 12-3 illustrates the performance and space usage of the box-complement algorithm and variants of the 3D corners algorithm. Since the paper that introduced the corners algorithm stopped short of a general construction in higher dimensions, the 3D case is the highest dimensionality for which we have implementations of the box-complement and corners algorithm. The 3D case is of interest because applications such as the FMM often present in three dimensions [96, 171]. We find that the box-complement algorithm outperforms the corners algorithm by about 1.4× when given similar amounts of space, though the corners algorithm with twice the space as box-complement is 2× faster. The box-complement algorithm uses a fixed (constant) factor of extra space, while the corners algorithm can use a variable amount of space. We found that the performance of the corners algorithm depends heavily on its space usage. We use `Corners(c)` to denote the implementation of the corners algorithm that uses a factor of c in space to store leaves in the computation tree and gather the results into the output. Furthermore, we also explored a variant of the corners algorithm in Appendix C.1, called `Corners Spine`, which uses extra space to store the spine of the computation tree and asymptotically reduce the runtime.

Figure 12-4 demonstrates how algorithms for weak excluded sums scale with dimension. We omit the corners algorithm because the original paper stopped short of a construction of how to find the corners in higher dimensions. We also omit an evaluation of included-sums algorithms because the relative performance of all algorithms would be the same. The naive and summed-area table perform well in lower dimensions but exhibit crossover points (at 3 and 6 dimensions, respectively) because their runtimes grow exponentially with dimension. In contrast, the BDBS and box-complement algorithms scale linearly in the number of dimensions and outperform the summed-area table method by at least 1.3× after 6 dimensions. The BDBS algorithm demonstrates the advantage of solving the weak problem, if you can, because it is always faster than the box-complement algorithm, which doesn't exploit an operator inverse. Both algorithms introduced in this chapter outperform existing methods in higher dimensions, however.

To be specific, our contributions are as follows:

- the bidirectional box-sum (BDBS) algorithm for strong included sums;
- the bidirectional box-sum complement (BDBSC) algorithm for weak excluded sums;
- the box-complement algorithm for strong excluded sums;
- theorems showing that, for a d-dimensional tensor of size N, these algorithms all run in $\Theta(dN)$ time and $\Theta(N)$ space;
- implementations of these algorithms in `C++`; and
- empirical evaluations showing that the box-complement algorithm outperforms the corners algorithm in 3D given similar space and that both the BDBSC algorithm and box-complement algorithm outperform the SATC algorithm in higher dimensions.

Map. The rest of this chapter is organized as follows. Section 12.2 provides necessary preliminaries and notation to understand the algorithms and proofs. Section 12.3 presents an efficient algorithm to solve the included-sums problem, which will be used as a key subroutine in the box-complement algorithm. Section 12.4 formulates the excluded sum as the "box-complement," and Section 12.5 describes and analyzes the resulting box-complement algorithm. Section 12.6 presents an empirical evaluation of algorithms for excluded sums. Finally, we provide concluding remarks in Section 12.7.

12.2 Preliminaries

This section reviews tensor preliminaries used to describe algorithms in later sections. It also formalizes the included- and excluded-sums problems in terms of tensor notation. Finally, it describes the prefix- and suffix-sums primitive underlying the main algorithms in this chapter.

Tensor preliminaries

We first introduce the coordinate and tensor notation we use to explain our algorithms and why they work. At a high level, tensors are d-dimensional arrays of elements over some monoid $(S, \oplus, e)$. In this chapter, tensors are represented by capital script letters (e.g., $\mathcal{A}$) and vectors are represented by lowercase boldface letters (e.g., $\mathbf{a}$).

We shall use the following terminology. A d-dimensional ***coordinate domain*** U is the cross product $U = U_1 \times U_2 \times \ldots \times U_d$, where $U_i = \{1, 2, \ldots, n_i\}$ for $n_i \geq 1$. The ***size*** of U is $n_1 n_2 \cdots n_d$. Given a coordinate domain U and a monoid $(S, \oplus, e)$ as defined in Section 12.1, a ***tensor*** $\mathcal{A}$ can be viewed for our purposes as a mapping $\mathcal{A} : U \to S$. That is, a tensor maps a coordinate $\mathbf{x} \in U$ to an ***element*** $\mathcal{A}[\mathbf{x}] \in S$. The ***size*** of a tensor is the size of its coordinate domain. We omit the coordinate domain U and monoid $(S, \oplus, e)$ when they are clear from context.

We use Python-like ***colon notation*** $x : x'$, where $x \leq x'$, to denote the half-open interval $[x, x')$ of coordinates along a particular dimension. If $x : x'$ would extend outside of $[1, n]$, where n is the maximum coordinate, it denotes only the coordinates actually in the interval, that is, the interval $\max\{1, x\} : \min\{n + 1, x'\}$. If the lower bound is missing, as in $: x'$, we interpret the interval as $1 : x'$, and similarly, if the upper bound is missing, as in $x :$, it denotes the interval $[x, n]$. If both bounds are missing, as in $:$, we interpret the interval as the whole coordinate range $[1, n]$.

We can use colon notation when indexing a tensor $\mathcal{A}$ to define ***subtensors***, or ***boxes***. For example, $\mathcal{A}[3:5, 4:6]$ denotes the elements of $\mathcal{A}$ at coordinates $(3, 4), (3, 5), (4, 4), (4, 5)$. For full generality, a box B ***cornered*** at coordinates $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ and $\mathbf{x}' = (x'_1, x'_2, \ldots, x'_d)$, where $x_i < x'_i$ for all $i = 1, 2, \ldots, d$, is the box $(x_1 : x'_1, x_2 : x'_2, \ldots, x_d : x'_d)$. Given a ***box size*** $\mathbf{k} = (k_1, \ldots, k_d)$, a **k*-box cornered*** at coordinate $\mathbf{x}$ is the box cornered at $\mathbf{x}$ and $\mathbf{x}' = (x_1 + k_1, x_2 + k_2, \ldots, x_d + k_d)$. A ***(tensor) row*** is a box with a single value in each coordinate position in the colon notation, except for one position, which includes that entire dimension. For example, if $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ is a coordinate of a tensor $\mathcal{A}$, then $\mathcal{A}[x_1, x_2, \ldots, x_{i-1}, :, x_{i+1}, x_{i+2}, \ldots, x_d]$ denotes a row along dimension i.

The colon notation can be combined with the reduction operator $\oplus$ to indicate the reduction of all elements in a subtensor:

$$\bigoplus \mathcal{A}[x_1 : x'_1, x_2 : x'_2, \ldots, x_d : x'_d] = \bigoplus_{y_1 \in [x_1, x'_1)} \bigoplus_{y_2 \in [x_2, x'_2)} \cdots \bigoplus_{y_d \in [x_d, x'_d)} \mathcal{A}[y_1, y_2, \ldots, y_d] \,.$$

Problem definitions

We can now formalize the included- and excluded-sums problems from Section 12.1.

Definition 12.1 (Included and Excluded Sums) *An algorithm for the* ***included-sums problem*** *takes as input a d-dimensional tensor $\mathcal{A} : U \to S$ with size N and a box size $\mathbf{k} = (k_1, k_2, \ldots, k_d)$. It produces a new tensor $\mathcal{A}' : U \to S$ such that every output element $\mathcal{A}'[\mathbf{x}]$ holds the reduction under $\oplus$ of elements within the $\mathbf{k}$-box of $\mathcal{A}$ cornered at $\mathbf{x}$. An algorithm for the* ***excluded-sums problem*** *is defined similarly, except that the reduction is of elements outside the $\mathbf{k}$-box cornered at $\mathbf{x}$.*

In other words, an included-sums algorithm computes, for all $\mathbf{x} = (x_1, x_2, \ldots, x_d) \in U$, the value $\mathcal{A}'[\mathbf{x}] = \bigoplus \mathcal{A}[x_1 : x_1 + k_1, x_2 : x_2 + k_2, \ldots, x_d : x_d + k_d]$. It's messier to write the output of an excluded-sums problem using colon notation, but fortunately, our proofs do not rely on it.

As noted in Section 12.1, there are weak and strong versions of both problems which allow and do not allow an operator inverse, respectively.

Prefix and suffix sums

The ***prefix-sums operation*** [61] takes an array $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ of n elements and returns the "running sum" $\mathbf{b} = (b_1, b_2, \ldots, b_n)$, where

$$b_k = \begin{cases} a_1 & \text{if } k = 1, \\ a_k \oplus b_{k-1} & \text{if } k > 1 \ . \end{cases} \tag{12.1}$$

Let Prefix denote the algorithm that directly implements the recursion in Equation 12.1. Given an array $\mathbf{a}$ and indices $\mathit{start} \leq \mathit{end}$, the function $\textsc{Prefix}(\mathbf{a}, \mathit{start}, \mathit{end})$ computes the prefix sum in the range $[\mathit{start}, \mathit{end}]$ of $\mathbf{a}$ in $O(\mathit{end} - \mathit{start})$ time. Similarly, the ***suffix-sums operation*** is the reverse of the prefix sum and computes the sum right-to-left rather than left-to-right. Let $\textsc{Suffix}(\mathbf{a}, \mathit{start}, \mathit{end})$ be the corresponding algorithm for suffix sums.

12.3 Included Sums

This section presents the ***bidirectional box-sum algorithm (BDBS) algorithm*** to compute the included sum along an arbitrary dimension, which is used as a main subroutine in the box-complement algorithm for excluded sums. As a warm-up, we will first describe how to solve the included-sums problem in one dimension and extend the technique to higher dimensions. We include the one-dimensional case for clarity, but the main focus of this chapter is the multidimensional case.

We will sketch the subroutines for higher dimensions in this section. Appendix C.2 includes all the pseudocode and omitted proofs for BDBS in 1D. This section sketches the key subroutines in higher dimensions and omits their formalization because they straightforwardly extend the computation from 1 dimension.

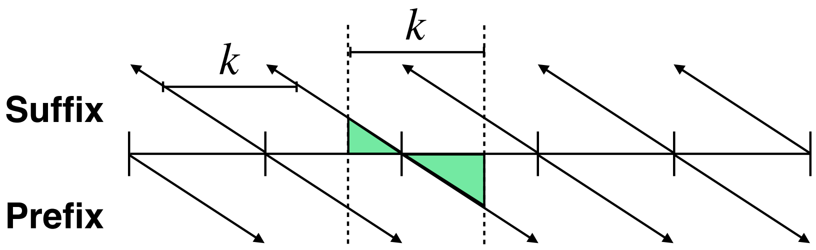


Figure 12-5: An illustration of the computation in the bidirectional box-sum algorithm. The arrows represent prefix and suffix sums in runs of size k, and the shaded region represents the prefix and suffix components of the region of size k outlined by the dotted lines.

Position	1	2	3	4	5	6	7	8
A	2	5	3	1	6	3	9	0
A_p	2	7	10	11	6	9	18	18
A_s	11	9	4	1	18	12	9	0
A'	11	15	13	19	18	12	9	0

Figure 12-6: An example of computing the 1D included sum using the bidirectional box-sum algorithm, where $N = 8$ and $k = 4$. The input array is A, the k-wise prefix and suffix sums are stored in A_p and A_s, respectively, and the output is in A'.

Included sums in 1D

Before investigating the included sums in higher dimensions, let us first turn our attention to the 1D case for ease of understanding. This section presents an algorithm BDBS-1D which takes as input a list A of length N and a (scalar) box size[1] k and outputs a list A' of corresponding included sums. At a high level, the BDBS-1D algorithm generates two intermediate lists A_p and A_s, each of length N, and performs N/k prefix and suffix sums of length k on each intermediate list. By construction, for $x = 1, 2, \ldots, N$, we have $A_p[x] = A[k\lfloor x/k\rfloor : x+1]$, and $A_s[x] = A[x : k\lceil (x+1)/k\rceil]$.

Finally, BDBS-1D uses A_p and A_s to compute the included sum of size k for each coordinate in one pass. Figure 12-5 illustrates the ranged prefix and suffix sums in BDBS-1D, and Figure 12-6 presents a concrete example of the computation.

[1]For simplicity in the algorithm descriptions and pseudocode, we assume that $n_i \bmod k_i = 0$ for all dimensions $i = 1, 2, \ldots, d$. In implementations, the input can either be padded with the identity to make this assumption hold, or it can add in extra code to deal with unaligned boxes.

BDBS-1D solves the included-sums problem on an array of size N in $\Theta(N)$ time and $\Theta(N)$ space. First, it uses two temporary arrays to compute the prefix and suffix as illustrated in Figure 12-5 in $\Theta(N)$ time. It then makes one more pass through the data to compute the included sum, requiring $\Theta(N)$ time. Figure C-3 in Appendix C.2 contains the full pseudocode for BDBS-1D.

Generalizing to arbitrary dimensions

The main focus of this work is multidimensional included and excluded sums. Computing the included sum along an arbitrary dimension is almost exactly the same as computing it along 1 dimension in terms of the underlying ranged prefix and suffix sums. We sketch an algorithm BDBS that generalizes BDBS-1D to arbitrary dimensions.

Let $\mathcal{A}$ be a d-dimensional tensor with N elements and let $\mathbf{k}$ be a box size. The BDBS algorithm computes the included sum along dimensions $i = 1, 2, \ldots, d$ in turn. After performing the included-sum computation along dimensions $1, 2, \ldots, i$, every coordinate in the output $\mathcal{A}_i$ contains the included sum in each dimension up to i:

$$\begin{aligned} &\mathcal{A}_i[x_1, x_2, \ldots, x_d] = \\ &\qquad \bigoplus \mathcal{A}[\underbrace{x_1 : x_1 + k_2, \ldots, x_i : x_i + k_i}_{i}, \underbrace{x_{i+2}, \ldots, x_d}_{d-i}]. \end{aligned}$$

Overall, BDBS computes the full included sum of a tensor with N elements in $\Theta(dN)$ time and $\Theta(N)$ space by performing the included sum along each dimension in turn.

Although we cannot directly use BDBS to solve the strong excluded-sums problem, the next sections demonstrate how to use the BDBS technique as a key subroutine in the box-complement algorithm for strong excluded sums.

12.4 Excluded sums and the box complement

The main insight in this section is the formulation of the excluded sum as the recursive "box complement". This section shows how to partition the excluded region into $2d$ non-overlapping parts in d dimensions. This decomposition of the excluded region underlies the box-complement for strong excluded sums in the next section.

First, let's see how the formulation of the "box complement" relates to the excluded sum. At a high level, given a box B, a coordinate $\mathbf{x}$ is in the "i-complement" of B if and only if $\mathbf{x}$ is "out of range" in some dimension $j \leq i$, and "in the range" for all dimensions greater than i.

Definition 12.2 (Box complement) *Given a d-dimensional coordinate domain U and a dimension $i \in \{1, 2, \ldots, d\}$, the **$i$-complement** of a box B cornered at coordi-*

nates $\mathbf{x} = (x_1, \ldots, x_d)$ *and* $\mathbf{x}' = (x'_1, \ldots, x'_d)$ *is the set*

$$\begin{aligned} C_i(B) = \{(y_1, \ldots, y_d) \in U :\ & \textit{there exists } j \in [1, i] \\ & \textit{such that } y_j \notin [x_j, x'_j),\ \textit{and}\ \textit{for all } m \in [i+1, d], \\ & y_m \in [x_m, x'_m)\}. \end{aligned}$$

Given a box B, the reduction of all elements at coordinates in $C_d(B)$ is exactly the excluded sum with respect to B. The box complement recursively partitions an excluded region into disjoint sets of coordinates.

Theorem 12.3 (Recursive box-complement) *Let B be a box cornered at coordinates $\mathbf{x} = (x_1, \ldots, x_d)$ and $\mathbf{x}' = (x'_1, \ldots, x'_d)$ in some coordinate domain U. The i-complement of B can be expressed recursively in terms of the $(i-1)$-complement of B as follows:*

$$\begin{aligned} C_i(B) = (\underbrace{:, \ldots, :, :x_i}_{i}, \underbrace{x_{i+1} : x'_{i+1}, \ldots, x_d : x'_d}_{d-i}) \cup \\ (\underbrace{:, \ldots, :}_{i-1}, x'_i :, \underbrace{x_{i+1} : x'_{i+1}, \ldots, x_d : x'_d}_{d-i}) \cup C_{i-1}(B), \end{aligned}$$

where $C_0(B) = \emptyset$.

Proof. For simplicity of notation, let $\mathrm{RHS}_i(B)$ be the right-hand side of the equation in the statement of Theorem 12.3. Let $\mathbf{y} = (y_1, \ldots, y_d)$ be a coordinate. In order to show the equality, we will show that $\mathbf{y} \in C_i(B)$ if and only if $\mathbf{y} \in \mathrm{RHS}_i(B)$.
Forward direction: $\mathbf{y} \in C_i(B) \to \mathbf{y} \in \mathrm{RHS}_i(B)$.
We proceed by case analysis when $\mathbf{y} \in C_i(B)$. Let $j \le i$ be the highest dimension at which $\mathbf{y}$ is "out of range," or where $y_j < x_j$ or $y_j \ge x'_j$.

Case 1: $j = i$.
Definition 12.2 and $j = i$ imply that either $y_i < x_i$ or $y_i \ge x'_i$, and $x_m \le y_m \le x'_m$ for all $m > i$. By definition, $y_i < x_i$ implies $\mathbf{y} \in \big(:, \ldots, :, :x_i, x_{i+1} : x'_{i+1}, \ldots, x_d : x'_d\big)$. Similarly, $y_i \ge x'_i$ implies $\mathbf{y} \in \big(:, \ldots, :, x'_i :, x_{i+1} : x'_{i+1}, \ldots, x_d : x'_d\big)$. These are exactly the first two terms in $\mathrm{RHS}_i(B)$.

Case 2: $j < i$.
Definition 12.2 and $j < i$ imply that $\mathbf{y} \in C_{i-1}(B)$.

Backward direction: $\mathbf{y} \in \mathrm{RHS}_i(B) \to \mathbf{y} \in C_i(B)$.
We again proceed by case analysis.

Case 1: $\mathbf{y} \in \big(:, \ldots, :, :x_i, x_{i+1} : x'_{i+1}, \ldots, x_d : x'_d\big)$ or
$\mathbf{y} \in \big(:, \ldots, :, x'_i :, x_{i+1} : x'_{i+1}, \ldots, x_d : x'_d\big)$.
Definition 12.2 implies $\mathbf{y} \in C_i(B)$ because there exists some $j \le i$ (in this case, $j = i$) such that $y_j < x_j$ and $x_m \le y_m < x'_m$ for all $m > i$.

Case 2: $\mathbf{y} \in C_{i-1}(B)$.
Definition 12.2 implies that there exists j in the range $1 \le j \le i-1$ such that $y_j < x_j$

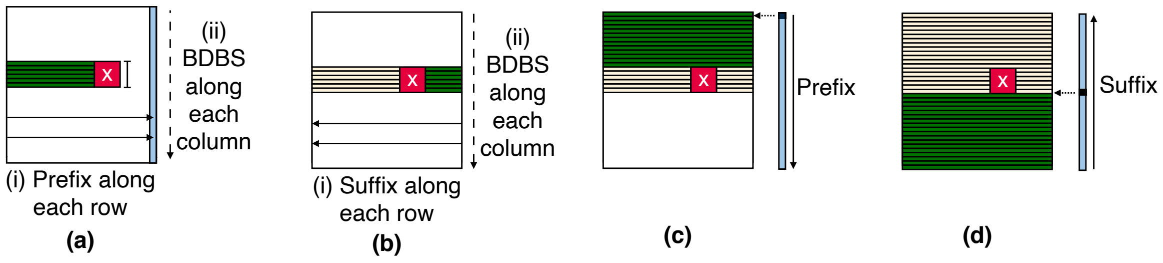

(ii) BDBS along each column
(i) Prefix along each row
(a)
(ii) BDBS along each column
(i) Suffix along each row
(b)
Prefix
(c)
Suffix
(d)
x

Figure 12-7: Steps for computing the excluded sum in 2 dimensions with included sums on prefix and suffix sums. The steps are labeled in the order they are computed. The 1-complement **(a)** prefix and **(b)** suffix steps perform a prefix and suffix along dimension 1 and an included sum along dimension 2. The numbers in **(a)**,**(b)** represent the order of subroutines in those steps. The 2-complement **(c)** prefix and **(d)** suffix steps perform a prefix and suffix sum on the reduced array, denoted by the blue rectangle, from step **(a)**. The red box denotes the excluded region, and solid lines with arrows denote prefix or suffix sums along a row or column. The long dashed line represents the included sum along each column.

or $y_j \geq x'_j$ and that for all $m \geq i$, we have $x_m \leq y_m < x'_m$. Therefore, $\mathbf{y} \in C_{i-1}(B)$ implies $\mathbf{y} \in C_i(B)$ since there exists some $j \leq i$ (in this case, $j < i$) where $y_j < x_j$ or $y_j \geq x'_j$ and $x_m \leq y_m < x'_m$ for all $m > 1$.

Therefore, $C_i(B)$ can be recursively expressed as $\text{RHS}_i(B)$. □

In general, unrolling the recursion in Theorem 12.3 yields $2d$ disjoint partitions that exactly comprise the excluded sum with respect to a box.

Corollary 12.4 (Excluded-sum components) *The excluded sum can be represented as the union of* $2d$ *disjoint sets of coordinates as follows:*

$$C_d(B) = \bigcup_{i=1}^{d} \Bigg((\underbrace{:, \ldots, :}_{i-1}, :x_i, \underbrace{x_{i+1} : x'_{i+1}, \ldots, x_d : x'_d}_{d-i}) \cup (\underbrace{:, \ldots, :}_{i-1}, x_i + k_i :, \underbrace{x_{i+1} : x'_{i+1}, \ldots, x_d : x'_d}_{d-i}) \Bigg) .$$

We use the box-complement formulation in the next section to efficiently compute the excluded sums on a tensor by reducing in disjoint regions of the tensor.

12.5 Box-complement algorithm

This section describes and analyzes the box-complement algorithm for strong excluded sums, which efficiently implements the dimension reduction in Section 12.4. The box-complement algorithm relies heavily on prefix, suffix, and included sums as described in Sections 12.2 and 12.3.

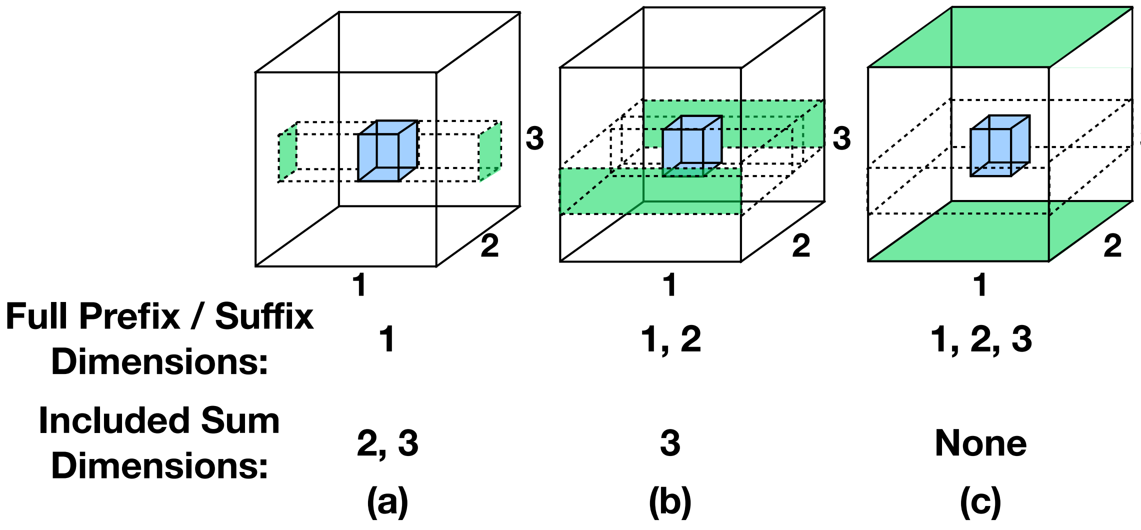


Figure 12-8: An example of the recursive box-complement in 3 dimensions with dimensions labeled $1, 2, 3$. The subfigures **(a)**, **(b)**, and **(c)** illustrate the 1-, 2-, and 3-complement, respectively. The blue region represents the coordinates inside the box, and the regions outlined by dotted lines represent the partitions defined by Corollary 12.4. For each partition, the face against the edge of the tensor is highlighted in green.

Given a d-dimensional tensor $\mathcal{A}$ of size N and a box size $\mathbf{k}$, the box-complement algorithm solves the excluded-sums problem with respect to $\mathbf{k}$ for coordinates in $\mathcal{A}$ in $\Theta(dN)$ time and $\Theta(N)$ space. Appendix C.3 contains all omitted pseudocode and proofs for the serial box-complement algorithm.

Algorithm sketch

At a high level, the box-complement algorithm proceeds by dimension reduction. That is, the algorithm takes d dimension-reduction steps, where each step adds two of the components from Corollary 12.4 to each element in the output tensor. In the ith dimension-reduction step, the box-complement algorithm computes the i-complement of B (Definition 12.2) for all coordinates in the tensor by performing a prefix and suffix sum along the ith dimension and then performing the BDBS technique along the remaining $d - i$ dimensions. After the ith dimension-reduction step, the box-complement algorithm operates on a tensor of $d-i$ dimensions because i dimensions have been reduced so far via prefix sums. Figure 12-7 presents an example of the dimension reduction in 2 dimensions, and Figure 12-8 illustrates the recursive box-complement in 3 dimensions.

```
BOX-COMPLEMENT($\mathcal{A}, \mathbf{k}$)
// Input: Tensor $\mathcal{A}$ with $d$-dimensions, box size $\mathbf{k}$
// Output: Tensor $\mathcal{A}'$ with size and dimensions
// matching $\mathcal{A}$ containing the excluded sum.
init $\mathcal{A}'$ with the same size as $\mathcal{A}$
$\mathcal{A}_p = \mathcal{A}; \mathcal{A}_s = \mathcal{A}$
// Current dimension-reduction step
for $i = 1$ to $d$
// Saved from previous dimension-reduction step.
    $\mathcal{A}_p = \mathcal{A}$ reduced up to dimension $i - 1$
    $\mathcal{A}_s = \mathcal{A}_p$    // Save input to suffix step
    // PREFIX STEP
    // Reduced up to $i$ dimensions.
    PREFIX-ALONG-DIM along dimension $i$ on $\mathcal{A}_p$.
    $\mathcal{A} = \mathcal{A}_p$    // Save for next round
    // Do included sum on dimensions $[i + 1, d]$.
    for $j = i + 1$ to $d$
        // $\mathcal{A}_p$ reduced up to $i$ dimensions
        BDBS-ALONG-DIM on dimension $j$ in $\mathcal{A}_p$
    // Add into result
    ADD-CONTRIBUTION from $\mathcal{A}_p$ into $\mathcal{A}'$

    // SUFFIX STEP
    // Do suffix sum along dimension $i$
    SUFFIX-ALONG-DIM along dimension $i$ in $\mathcal{A}_s$
    // Do included sum on dimensions $[i + 1, d]$
    for $j = i + 1$ to $d$
        // $\mathcal{A}_s$ reduced up to $i$ dimensions
        BDBS-ALONG-DIM on dimension $j$ in $\mathcal{A}_s$
    // Add into result
    ADD-CONTRIBUTION from $\mathcal{A}_s$ into $\mathcal{A}'$
return $\mathcal{A}'$
```

Figure 12-9: Pseudocode for the box-complement algorithm. For ease of presentation, we omit the exact parameters to the subroutines and describe their function in the algorithm. The pseudocode with parameters can be found in Figure C-6.

Prefix and suffix sums

In the ith dimension reduction step, the box-complement algorithm uses prefix and suffix sums along the ith dimension to reduce the elements "out of range" along the ith dimension in the i-complement. That is, given a tensor $\mathcal{A}$ of size $N = n_1 \cdot n_2 \cdots n_d$ and a number $i < d$ of dimensions reduced so far, we define a subroutine Prefix-Along-Dim$(\mathcal{A}, i)$ that fixes the first i dimensions at $n_1, \ldots, n_i$ (respectively), and then computes the prefix sum along dimension $i+1$ for all remaining rows in dimensions $i+2, \ldots, d$. The pseudocode for Prefix-Along-Dim$(\mathcal{A}, i)$ can be found in Figure C-4 in Appendix C.3, and the proof that it incurs $O\left(\prod_{j=i+1}^{d} n_j\right)$ time can be found in Appendix C.3.

The subroutine Prefix-Along-Dim computes the reduction of elements "out of range" along dimension i. That is, after Prefix-Along-Dim$(\mathcal{A}, i)$, for each coordinate $x_{i+1} = 1, 2, \ldots, n_{i+1}$ along dimension $i+1$, every coordinate in the (dimension-reduced) output $\mathcal{A}'$ contains the prefix up to that coordinate in dimension $i+1$:

$$\mathcal{A}'[\underbrace{n_1, \ldots, n_i}_{i}, x_{i+1}, \underbrace{x_{i+2}, \ldots, x_d}_{d-i-1}] =$$
$$\bigoplus \mathcal{A}[\underbrace{n_1, \ldots, n_i}_{i}, :x_{i+1}+1, \underbrace{x_{i+2}, \ldots, x_d}_{d-i-1}].$$

Since the similar subroutine Suffix-Along-Dim has almost exactly the same analysis and structure, we omit its discussion.

Included sums

In the ith dimension reduction step, the box-complement algorithm uses the BDBS technique along the ith dimension to reduce the elements "in range" along the ith dimension in the i-complement. That is, given a tensor $\mathcal{A}$ of size $N = n_1 \cdot n_2 \cdots n_d$ and a number $i < d$ of dimensions reduced so far, we define a subroutine BDBS-Along-Dim.

BDBS-Along-Dim computes the included sum for each row along a specified dimension after dimension reduction. Let $\mathcal{A}$ be a d-dimensional tensor, $\mathbf{k}$ be a box size, i be the number of reduced dimensions so far, and j be the dimension to compute the included sum along such that $j > i$. BDBS-Along-Dim$(\mathcal{A}, \mathbf{k}, i, j)$ computes the included sum along the jth dimension for all rows $(n_1, \ldots, n_i, :, \ldots, :)$. That is, for each coordinate $\mathbf{x} = (n_1, \ldots, n_i, x_{i+1}, \ldots, x_d)$, the output tensor $\mathcal{A}'$ contains the included sum along dimension j:

$$\mathcal{A}'[\mathbf{x}] \quad = \quad \bigoplus \mathcal{A}[\underbrace{n_1, \ldots, n_i}_{i}, \underbrace{x_{i+1}, \ldots, x_j}_{j-i}, x_{j+1} : x_{j+1} \quad + \quad k_{j+1}, \underbrace{x_{j+2}, \ldots, x_d}_{d-j-1}].$$

BDBS-Along-Dim$(\mathcal{A}, \mathbf{k}, i, j)$ takes $\Theta\left(\prod_{\ell=i+1}^{d} n_\ell\right)$ time because it iterates over $\left(\prod_{\ell=i+1}^{d} n_\ell\right)/n_{j+1}$ rows and runs in $\Theta(n_{j+1})$ time per row. It takes $\Theta(N)$ space using the same technique as BDBS-1D.

Adding in the contribution

Each dimension-reduction step must add its respective contribution to each element in the output. Given an input tensor $\mathcal{A}$ and output tensor $\mathcal{A}'$, both of size N, the function ADD-CONTRIBUTION takes $\Theta(N)$ time to add in the contribution with a pass through the tensors. The full pseudocode can be found in Figure C-5 in Appendix C.3.

Putting it all together

Finally, we will see how to use the previously defined subroutines to describe and analyze the box-complement algorithm for excluded sums. Figure 12-9 presents pseudocode for the box-complement algorithm. Each dimension-reduction step has a corresponding prefix and suffix step to add in the two components in the recursive box-complement. Given an input tensor $\mathcal{A}$ of size N, the box-complement algorithm takes $\Theta(N)$ space because all of its subroutines use at most a constant number of temporaries of size N, as seen in Figure 12-9.

Given a tensor $\mathcal{A}$ as input, the box-complement algorithm solves the excluded-sums problem by computing the recursive box-complement components from Corollary 12.4. By construction, for dimension $i \in [1, d]$, the prefix-sum part of the ith dimension-reduction step outputs a tensor $\mathcal{A}_p$ such that for all coordinates $\mathbf{x} = (x_1, \ldots, x_d)$, we have

$$\mathcal{A}_p[x_1, \ldots, x_d] = \bigoplus \mathcal{A}[\underbrace{:, \ldots, :}_{i}, :x_{i+1}, \\ \underbrace{x_{i+2} : x_{i+2} + k_{i+2}, \ldots, x_d : x_d + k_d}_{d-i-1}].$$

Similarly, the suffix-sum step constructs a tensor $\mathcal{A}_s$ such that for all $\mathbf{x}$,

$$\mathcal{A}_s[x_1, \ldots, x_d] = \bigoplus \mathcal{A}[\underbrace{:, \ldots, :}_{i}, x_{i+1} + k_{i+1} :, \\ \underbrace{x_{i+2} : x_{i+2} + k_{i+2}, \ldots, x_d : x_d + k_d}_{d-i-1}].$$

We can now analyze the performance of the box-complement algorithm.

Theorem 12.5 (Time of box-complement) *Given a d-dimensional tensor $\mathcal{A}$ of size $N = n_1 \cdot n_2 \cdot \ldots \cdot n_d$,* BOX-COMPLEMENT *solves the excluded-sums problem in $\Theta(dN)$ time.*

PROOF. We analyze the prefix step (since the suffix step is symmetric, it has the same running time). Let $i \in \{1, \ldots, d\}$ denote a dimension.

The ith dimension reduction step in BOX-COMPLEMENT involves 1 prefix step and $(d - i)$ included sum calls, which each have $O\left(\prod_{j=i}^{d} n_j\right)$ time. Furthermore, adding in the contribution at each dimension-reduction step takes $\Theta(N)$ time. The total time over d steps is therefore $\Theta\left(\sum_{i=1}^{d}\left((d-i+1)\prod_{j=i}^{d}n_j + N\right)\right)$. Adding in the contribution is clearly $\Theta(dN)$ in total.

Next, we bound the runtime of the prefix and included sums. In each dimension-reduction step, reducing the number of dimensions of interest exponentially decreases the size of the considered tensor. That is, dimension reduction exponentially reduces the size of the input: $\prod_{j=i}^{d} n_j \leq N/2^{i-1}$. The total time required to compute the box-complement components is therefore

$$\begin{aligned}\sum_{i=1}^{d}(d-i+1)\prod_{j=i}^{d}n_j &\leq \sum_{i=1}^{d}(d-i+1)\frac{N}{2^{i-1}} \\ &\leq 2(d+2^{-d}-1)N = \Theta(dN).\end{aligned}$$

Therefore, the total time of Box-Complement is $\Theta(dN)$. □

12.6 Experimental evaluation

This section presents an empirical evaluation of strong and weak excluded-sums algorithms. In 3 dimensions, we compare strong excluded-sums algorithms: specifically, we evaluate the box-complement algorithm and variants of the corners algorithm and find that the box-complement outperforms the corners algorithm given similar space. Furthermore, we compare weak excluded-sums algorithms in higher dimensions. Lastly, to simulate a more expensive operator than numeric addition when reducing, we compare the box-complement algorithm and variants of the corners algorithm using an artificial slowdown.

Experimental setup. We implemented all algorithms in `C++`. We used the Tapir/LLVM [322] branch of the LLVM [234, 235] compiler (version 9) with the `-O3` and `-march=native` and `-flto` flags.

All experiments were run on a 8-core 2-way hyper-threaded Intel Xeon CPU E5-2666 v3 @ 2.90GHz with 30GB of memory from AWS [10]. For each test, we took the median of 3 trials.

To gather empirical data about space usage, we interposed `malloc` and `free`. The theoretical space usage of the different algorithms can be found in Table 12.1.

Strong excluded sums in 3D

Figure 12-3 summarizes the results of our evaluation of the box-complement and corners algorithm in 3 dimensions with a box length of $k_1 = k_2 = k_3 = 4$ (for a total box volume of $K = 64$) and number of elements $N = 681472$. We tested with varying N but found that the time and space per element were flat (full results in Appendix C.4). We found that the box-complement algorithm outperforms the corners algorithm by about 1.4× when given similar amounts of space, though the corners algorithm with 2× the space as the box-complement algorithm was 2× faster.

We explored two different methods of using extra space in the corners algorithm based on the computation tree of prefixes and suffixes: (1) storing the spine of the computation tree to asymptotically reduce the running time, and (2) storing the leaves of the computation tree to reduce passes through the output. Although storing the leaves does not asymptotically affect the behavior of the corners algorithm, we found that reducing the number of passes through the output has significant effects on empirical performance. Storing the spine did not improve performance, because the runtime is dominated by the number of passes through the output.

Excluded sums with different operators

Most of our experiments used numeric addition for the $\oplus$ operator. Because some applications, such as FMM, involve much more costly $\oplus$ operators, we studied how the excluded-sum algorithms scale with the cost of $\oplus$. To do so, we added a tunable slowdown to the invocation of $\oplus$ in the algorithms. Specifically, they call an unoptimized implementation of the standard recursive Fibonacci computation. By varying the argument to the Fibonacci function, we can simulate $\oplus$ operators that take different amounts of time.

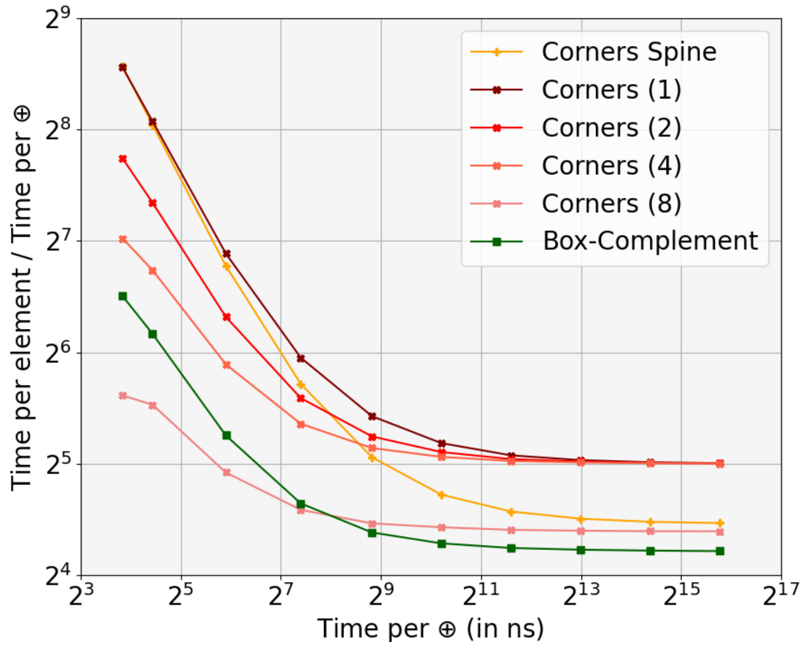


Figure 12-10: The scalability of excluded-sum algorithms as a function of the cost of operator $\oplus$ on a 3D domain of $N = 4096$ elements. The horizontal axis is the time in nanoseconds to execute $\oplus$. The vertical axis represents the time per element of the given algorithm divided by the time for $\oplus$. We inflated the time of $\oplus$ using increasingly large arguments to the standard recursive implementation of a Fibonacci computation.

Figure 12-10 summarizes our findings. We ran the algorithms on a 3D domain of $N = 4096$ elements. (Although this domain may seem small, Appendix C.4 shows that the results are relatively insensitive to domain size.) For inexpensive $\oplus$ operators, the box-complement algorithm is the second fastest, but as the cost of $\oplus$ increases, the box-complement algorithm dominates. The reason for this outcome is that box-complement performs approximately 12 $\oplus$ operations per element in 3D, whereas the most efficient corners algorithm performs about 22 $\oplus$ operations. As $\oplus$ becomes more costly, the time spent executing $\oplus$ dominates the other bookkeeping overhead.

Weak excluded sums in higher dimensions

Figure 12-4 presents the results of our evaluation of weak excluded-sum algorithms in higher dimensions. For all dimensions $i = 1, 2, \ldots, d$, we set the box length $k_i = 8$ and chose a number of elements N to be a perfect power of dimension i. Table 12.1 presents the asymptotic runtime of the different excluded-sum algorithms.

The weak naive algorithm for excluded sums with nested loops outperforms all of the other algorithms up to 2 dimensions because its runtime is dependent on the box volume, which is low in smaller dimensions. Since its runtime grows exponentially with the box length, however, we limited it to 5 dimensions.

The summed-area table algorithm outperforms the BDBS and box-complement algorithms up to 6 dimensions, but its runtime scales exponentially in the number of dimensions.

Finally, the BDBS and box-complement algorithms scale linearly in the number of dimensions and outperform both naive and summed-area table methods in higher dimensions. Specifically, the box-complement algorithm outperforms the summed-area table algorithm by between 1.3× and 4× after 6 dimensions. The BDBS algorithm demonstrates an advantage to having an inverse: it outperforms the box-complement algorithm by 1.1× to 4×. Therefore, the BDBS algorithm dominates the box-complement algorithm for weak excluded sums.

12.7 Conclusion

This chapter introduced the box-complement algorithm for the excluded-sums problem, which improves the running time of the state-of-the-art corners algorithm from $\Omega(2^d N)$ to $\Theta(dN)$ time. The space usage of the box-complement algorithm is independent of the number of dimensions, while the corners algorithm may use space dependent on the number of dimensions to achieve its running-time lower bound.

The three new algorithms from this chapter parallelize straightforwardly. In the work/span model [108], all three algorithms are work-efficient, achieving $\Theta(dN)$ work. The BDBS and BDBSC algorithms achieve $\Theta(d \log N)$ span, and the box-complement algorithm achieves $\Theta(d^2 \log N)$ span.

Locality-first strategy. The BDBS and box-complement algorithms in this chapter apply the locality-first strategy by first understanding opportunities for locality in the problem, taking advantage of that locality, and then improving the overall efficiency of algorithms for the included-sums and excluded-sums problems. Since the overall work complexity of both are algorithms is $\Theta(dN)$ for a d-dimensional tensor of size N, there is not much opportunity for temporal locality, or speedups in M, because the work is close to the input size. Therefore, the focus of this work is on finding faster algorithms for the problems.

Chapter 13

Conclusion

This chapter provides concluding remarks and a discussion of the applicability of the locality-first strategy. Section 13.1 summarizes the main artifacts in this thesis in the context of the broader locality-first strategy. Section 13.2 takes a step back from the principal artifacts to discuss the wider applicability of the locality-first strategy for multicore optimization. It describes the breadth of the strategy by identifying other instances of the strategy in the literature. It then discusses how to apply the locality-first strategy depending on the problem.

13.1 Thesis summary

This thesis contends that algorithm developers and performance engineers should use a locality-first strategy as a roadmap to create theoretically and practically efficient parallel algorithms for multicores. To demonstrate the potential of optimizing for locality first, this thesis studies a variety of problems that exhibit different types of locality and parallelism. The principal artifacts demonstrate how to use locality-first algorithm engineering to solve large problems on just a single multicore.

Although multicores have become widely accessible in the past decade and are now available on-demand with just a credit card, their full potential has not yet been realized due to the difficulty of writing efficient codes that take advantage of the underlying hardware. Specifically, the main features of shared-memory multicores are 1) the multiple cores for parallelism and 2) the memory system for locality. Many large problems from applications such as social networks and scientific computing can be solved on a single general-purpose shared-memory multicore by taking advantage of these hardware features.

Enhancing locality by changing the data layout

Chapters 3-6 enhance spatial locality by changing the underlying data layout before introducing parallelism.

Chapters 3-5 apply the locality-first strategy to problems with low temporal and spatial locality and demonstrate that optimizing for locality, even at the cost of some parallelism, can improve overall performance because there is ample parallelism in the common case. To do so, they introduce algorithms and data structures that apply the locality-first strategy by first taking advantage of spatial locality in sparse problems and then parallelizing the computations. Specifically, Chapters 3 and 4 introduce cache-friendly data structures for sparse graph problems that improve upon the state of the art by exploiting spatial locality without sacrificing parallelism. Furthermore, Chapter 5 introduces an efficient algorithm for estimating the fill, which is a crucial step in tailoring blocked representations to the patterns of locality in individual tensors without incurring too much overhead. These artifacts apply the locality-first strategy to algorithm and data structure design to enhance spatial locality first for problems with low spatial and temporal locality.

Furthermore, Chapter 6 takes the first step towards an efficient parallel data structure via the locality-first strategy by presenting a serial cache-optimized skip list data structure. By exploiting locality first, a future parallel version of the cache-friendly skip list can achieve closer to peak performance. These artifacts apply the locality-first strategy to a variety of problems to improve theoretical and practical performance by focusing on spatial locality as a first step before parallelism.

Exploiting locality without changing the data layout

Chapters 7-12 take advantage of locality by changing algorithm access patterns.

Chapters 7-10 mathematically ground the locality-first strategy by demonstrating that algorithms that are designed to fully exploit temporal locality perform well despite potential disruptions to cache-friendly access patterns due to parallelism. They provide support for the locality-first strategy via beyond-worst-case analysis of algorithms in shared memory. Chapters 7 and 8 study cache-replacement algorithms in shared memory and provide theoretical evidence for the established superiority of Least-Recently-Used in practice. Furthermore, Chapters 9 and 10 study algorithms in shared-memory and mathematically close the gap between cache-oblivious and cache-adaptive algorithms, validating the cache-friendliness of the divide-and-conquer algorithm design pattern in shared memory. These artifacts address a potential concern with the locality-first strategy that locality and parallelism trade off by demonstrating that algorithms that take advantage of locality are mathematically good even in shared memory.

Chapters 11 and 12 apply the locality-first strategy to demonstrate how to understand and exploit locality in problems with spatial locality but not much temporal locality. They first understand locality in the prefix-sums, included-sums, and excluded-sums problems by realizing that there is easily-available spatial locality and a lack of temporal locality. Therefore, these artifacts optimize for other measures, such as accuracy and total work.

The artifacts developed in this thesis provide mathematical and practical ***frameworks*** for creating and analyzing fast parallel algorithms for shared-memory multicores. For example, Chapters 3 and 4 introduce frameworks for efficiently processing dynamic graphs based on locality-optimized data structures. Additionally, Chapter 7 introduces a mathematical framework called cyclic analysis for beyond-worst-case analysis of general online algorithms.

13.2 Applicability of the locality-first strategy

This section demonstrates the general applicability of the locality-first strategy. First, it provides examples from the literature outside of the artifacts in this thesis that illustrate the potential for locality-first algorithm design in the theory and practice of parallel algorithms. Finally, it describes guidelines for using the locality-first strategy based on the type of locality that a problem exhibits.

Other examples of locality-first from the literature

Applications of the locality-first strategy appear throughout the literature in the context of developing efficient parallel codes for multicores. I highlight a few examples:

- Past work showed that when optimizing loops for locality and parallelism simultaneously, performance engineers should optimize first for locality and then introduce parallelism while maintaining memory access order [217].

- The Ligra+ graph-processing framework demonstrates that improving spatial locality via compression can overcome traditional tradeoffs between parallelization and cache-friendliness [337]. This work shows that compression improves overall performance of graph computations with parallelization despite initially posing challenges to parallelization due to serialization during the decompression phase.

- The improved performance of the parallel Greedy graph-coloring algorithm compared to the other parallel ordering heuristics stems from improved spatial locality from processing vertices in order [204].

- Attempts to reduce contention by disrupting spatial locality in parallel breadth-first search yielded overall slower performance, demonstrating the importance of maintaining locality even at the cost of some parallel scalability [239, 319].

- The Quilter algorithm for high-throughput image alignment in connectomics achieves speedup by first exploiting temporal locality by caching previously-used data before parallelization [203, 206, 377].

I also point out a few examples of the importance of optimizing for locality outside of direct parallel algorithm design. These examples differ from the previous ones because they focus on theoretical algorithm and data structure design.

- Graveyard hashing, a variant of linear probing, avoids primary clustering in hash tables while maintaining cache-friendliness [53]. These results suggest that in many important applications such as hash tables, data representations should favor locality despite traditional tradeoffs.

- Traditionally, graph algorithms have resisted significant speedups in the DAM model due to a lack of spatial and temporal locality [119]. The negative triangle problem illustrates the potential to circumvent these issues via the locality-first approach in this domain [292]. By viewing the data as an array and the problem as a repeated scan of lists, past work [292] was able to break through the barrier of speedups just in B and achieve speedup in terms of M.

Guidelines for using the locality-first strategy

The first step in the locality-first strategy is to understand the types of locality present in a problem. As shown in Figure 1-1, there are four main quadrants depending on whether a problem exhibits spatial and/or temporal locality. Once a performance engineer has identified which quadrant their problem falls into, they may apply different techniques depending on the type of problem they have.

The first quadrant contains problems with low spatial and low temporal locality, such as sparse problems. Since there is not much temporal locality in this case, it is important to lay out the data in a cache-friendly way to take full advantage of spatial locality. This thesis illustrates how to design parallel cache-friendly data structures and representations for a variety of sparse problems in Chapters 3-5.

The second quadrant contains problems with high spatial and low temporal locality, such as the prefix-sums problem. In these situations, the data is already laid out in a cache-friendly way, so classical techniques such as blocking capture the available spatial locality. Therefore, as discussed in Chapters 11, optimizations may focus exclusively on parallelization or on measures beyond runtime such as accuracy.

The third quadrant contains problems with low spatial and high temporal locality, such as satisfiability solvers [162]. Although this thesis does not address these problems explicitly in the artifacts, previous work on optimizing these applications has demonstrated the potential for a locality-first approach by developing local search algorithms for improved temporal locality [159].

Finally, the fourth quadrant contains problems with high spatial and high temporal locality, such as dense matrix multiplication. Since the data is already laid out in a cache-friendly way for spatial locality, optimizing for this case involves taking advantage of temporal locality. For example, Chapters 7 and 9 theoretically ground classical techniques for exploiting temporal locality such as Least-Recently-Used and divide-and-conquer in the presence of parallelism.

In summary, this thesis has shown that a locality-first strategy for algorithm design can overcome tradeoffs between cache-friendliness and parallelism and create theoretically and practically efficient multicore algorithms. As discussed in Chapter 1, optimizing algorithms for locality and parallelism individually is challenging, and even more challenging when the two are combined because they trade off with each other. The locality-first strategy is a principled method for creating efficient multicore algorithms that can scale to ever-growing problems.

Appendix A

Packed Compressed Sparse Row

This appendix presents Packed Compressed Sparse Row (PCSR), a serial dynamic graph storage format that optimizes for locality-first by storing data contiguously using a Packed Memory Array (PMA) data structure. PCSR is the first step towards the more complex parallel dynamic graph storage formats in Chapters 3 and 4. It optimizes for spatial locality by storing data contiguously while still supporting efficient updates.

This work was conducted in collaboration with Brian Wheatman [378]. We would like to thank Julian Shun for helpful comments.

Abstract

Perhaps the most popular sparse graph storage format is Compressed Sparse Row (CSR). CSR excels at storing graphs compactly with minimal overhead, allowing for fast traversals, lookups, and basic graph computations such as PageRank. Since elements in CSR format are packed together, additions and deletions often require time linear in the size of the graph.

This appendix introduces a new dynamic sparse graph representation called *Packed Compressed Sparse Row* (PCSR), based on an array-based dynamic data structure called the Packed Memory Array. PCSR is similar to CSR, but leaves spaces between elements, allowing for asymptotically faster insertions and deletions in exchange for a constant factor slowdown in traversals and a constant factor increase in space overhead.

The contributions of this chapter are twofold. First, it describes PCSR and review the theoretical guarantees for update, insert, and search for PCSR. We also implemented PCSR as well as other basic graph storage formats and report our findings on a variety of benchmarks. This chapter shows that PCSR supports inserts orders of magnitude faster than CSR and is only a factor of two slower on graph traversals. These results suggest that PCSR is a lightweight dynamic graph representation that supports fast inserts and competitive searches.

The remainder of this appendix is organized as follows. Section A.1 reviews fundamental graph storage formats. Section A.2 introduces Packed Compressed Sparse

	Adjacency Matrix	AL	BAL	CSR	**PCSR** (amortized)
Storage cost / scanning whole graph	$O(n^2/B)$	$O(n+m)$	$O((m+n)/B)$	$O((m+n)/B)$	$O((m+n)/B)$
Add new edge	$O(1)$	$O(1)$	$O(1)$	$O((m+n)/B)$	$O(\lg^2(m+n)/B)$
Update or delete edge from vertex v	$O(1)$	$O(deg(v))$	$O(deg(v)/B)$	$O((m+n)/B)$	$O(\lg^2(m+n)/B)$
Add node	$O(n^2/B)$	$O(1)$	$O(1)$	$O(1)^*$	$O(\lg^2(m+n)/B)$
Finding all neighbors of a vertex v	$O(n/B)$	$O(deg(v))$	$O(deg(v)/B)$	$O(deg(v)/B)$	$O(deg(v)/B)$
Finding if w is a neighbor of v	$O(1)$	$O(deg(v))$	$O(deg(v)/B)$	$O(\lg_B(deg(v))$	$O(\lg_B(deg(v)))$
Sparse matrix-vector multiplication	$O(n^2/B)$	$O(n/B+m+n)$	$O((m+n)/B)$	$O((m+n)/B)$	$O((m+n)/B)$

Table A.1: Cache behavior of various sparse graph and matrix operations. $n = |V|$, $m = |E|$. The table lists various graph representations and the algorithmic runtime of common graph operations in the external memory model by Aggarwal and Vitter, [3] where B is the cache line (or disk block) size. The RAM model (without cache analysis) is the special case where B or $\lg(B)$ is 1. We analyze PCSR in the right-most column.

We use a `C++` vector for our implementation of CSR, so we do not need to rebuild the vertex list every time we add a vertex.

Row. Section A.3 empirically evaluates PCSR and the formats introduced in Section A.1.

A.1 Graph storage formats

This section describe the following graph storage formats: adjacency matrix, adjacency list, blocked adjacency list, and CSR. We detail their respective space/time tradeoffs in Table A.1. For a graph $G = (V, E)$, we denote the number of nodes by $n = |V|$ and number of edges by $m = |E|$.

Adjacency matrix

An ***adjacency matrix*** is the most basic graph storage format. It stores an $n \times n$ matrix for a graph of n nodes. The entry at $[u, v]$ corresponds to the value of the edge (u, v) (or has 0 if the edge does not exist). It excels in storing dense graphs because it does not store any pointers and therefore minimizes overhead if the graph is almost fully connected.

The adjacency matrix wastes space when the graph is sparse because it requires n^2 space. Furthermore, adding nodes requires rebuilding the entire data structure. Finally, sparse graph traversals on adjacency matrices require iterating over the entire matrix of size n^2. Since the number of edges is $m \ll n^2$ for many sparse graphs, a graph traversal using an adjacency matrix is not work efficient.

Adjacency list

Another common sparse graph storage format is the ***adjacency list*** (AL). Adjacency lists keep an array of nodes where each entry stores a pointer to a linked list of edges. The pointer at index u in the node list points to a linked list where each element v in the linked list is an outgoing edge (u, v).

Adjacency lists support fast inserts but have high space overhead and slow searches because the edges are unsorted. Adjacency lists also exhibit poor cache behavior because they lack locality. A variant of adjacency lists called ***blocked adjacency lists*** (BAL) uses blocks to store edges. Blocked adjacency lists exhibit faster traversals because of improved locality but require extra space for extremely sparse graphs. Blandford, Blelloch, and Kash [60] introduced a dynamic graph data structure based on BAL with many constant-factor improvements but stop short of giving theoretical guarantees. For simplicity, we compare PCSR with standard adjacency lists of various block sizes.

Figure A-1 shows an example of a graph stored in adjacency list format.

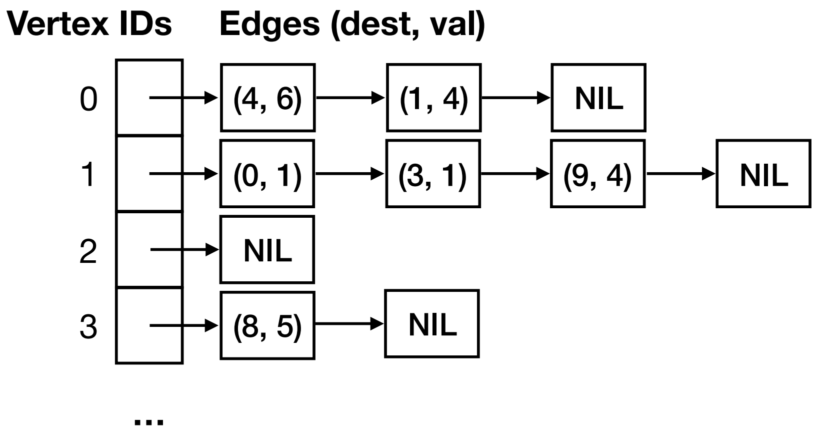


Figure A-1: An example of a graph stored in an adjacency list. Each entry in the nodes array points to a linked list of edges. The vertex ID in the nodes array implicitly stores the source. For weighted graphs, we store a tuple of destination vertex and edge value for each edge.

Compressed Sparse Row

Compressed sparse row (CSR) is a popular format for storing sparse graphs and matrices. It efficiently packs all the entries together in arrays, allowing for quick traversals of the data structure.

CSR uses three arrays to store a sparse graph: a node array, an edge array, and a values array[1]. Each entry in the node array contains the starting index in the edge array where the edges from that node are stored in sorted order by destination. The edge array stores the destination vertices of each edge. CSR stores a graph $G = (V, E)$ in size $O(|V| + |E|)$, but needs to be rebuilt upon any changes. Figure A-2 contains an example of a graph stored in CSR format.

[1]The values array is not needed in the unweighted case.

Inserting an edge into a graph in CSR format takes time linear in the size of the graph in the worst case. To insert an edge (u, v) into a graph in CSR format, we first must search all edges with source vertex u to find the edge with the smallest destination larger than v. Then we insert (u, v) into the edge list and slide all elements after it over by one. We then increment the elements in the node array for all vertices greater than u by one. The entire edge array may need to be resized and copied into a larger block of memory if there are too many elements in the structure.

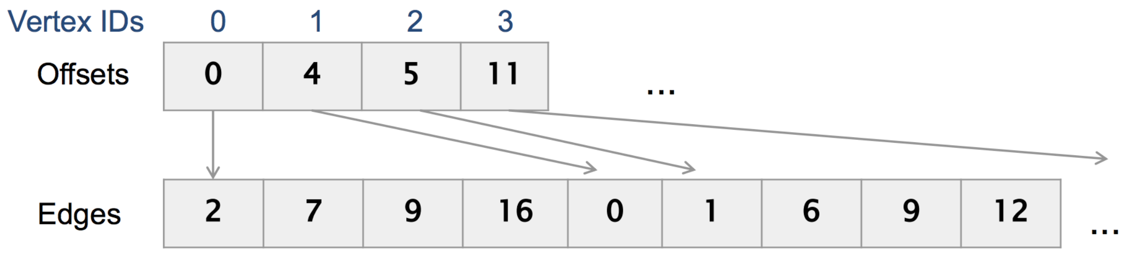


Figure A-2: An example of an unweighted graph stored in compressed sparse row. The values stored in the edges array represent the destination. The vertex ID in the offset array implicitly stores the source. For weighted graphs, there is an additional values array.

Pinar and Heath [304] introduced a variant of CSR called Blocked Compressed Sparse Row (BCSR), where the locations of nonzero blocks are stored in CSR format. This chapter focuses on unblocked CSR for simplicity.

A.2 Packed Compressed Sparse Row

This section introduces Packed Compressed Sparse Row, a graph storage format based on the Packed Memory Array data structure. It first reviews the structure and theoretical properties of the packed memory array (PMA) [52]. The PMA maintains edges in sorted order and leaves space between elements to support fast inserts and deletes. Next, it shows how to use a PMA as a graph storage format in PCSR. Finally, it describes how to implement graph operations in PCSR.

Packed Memory Array

The PMA stores N items in an ordered list of size $O(N)$ and supports inserts and deletes in $O(\lg^2(N))$.

At a high level, the PMA avoids changing the entire data structure after each insert or delete by maintaining spaces between elements and rebalancing when there are too many or too few elements in a range. It maintains spaces of size $O(1)$ between groups of elements of size $O(1)$ to enable easy insertions and deletions. The PMA maintains these spaces by rebalancing the structure and redistributing the elements whenever a section of the data structure becomes too sparse or too dense. Specifically, if the size of the data structure at some time t is n_t, the PMA keeps an implicit tree with $n_t/\lg(n_t)$ leaves of the data structure where each leaf has $\lg(n_t)$ slots. The density of a node is determined by the number of elements in it divided by the total number of slots in the node. If the density is too low or too high, the PMA rebalances the elements across a child or parent, respectively. If the entire array becomes too dense or too sparse, we resize the entire data structure.

Figure A-3 shows an example of the implicit binary tree on the PMA intervals. If an interval becomes too dense, we walk up the tree and redistribute when we find an interval that is appropriately dense.

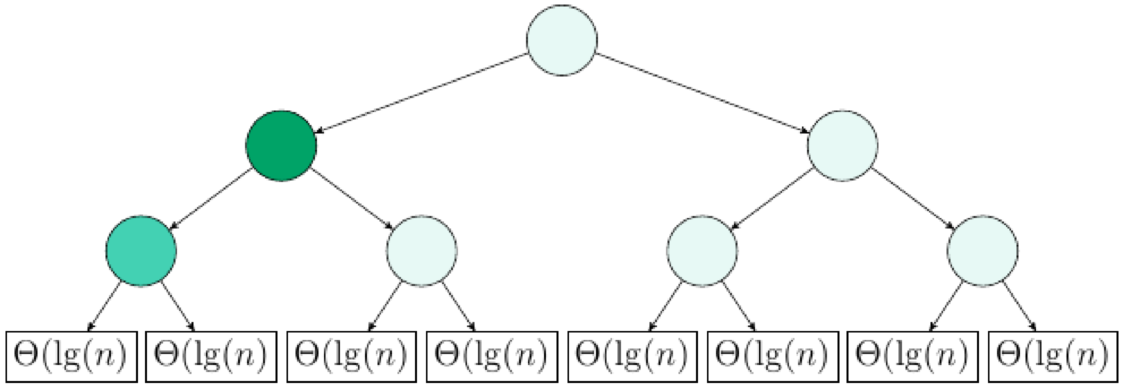


Figure A-3: An example of the implicit binary tree on the PMA intervals. If we insert a new element in a leaf and the corresponding interval becomes too dense (shown in light grey), we walk up the tree until we find an interval with a density in the allowed range (shown in dark grey). In the worst case, we walk up to the root and do a rebalance of the entire PMA. This figure is from [117].

PCSR structure

PCSR uses the same vertex and edge lists as CSR but uses a PMA instead of an array for the edge list. Adding both edges and nodes to the graph requires updates to both the vertex and edge lists. We use a `c++` vector with doubling and halving for the node list. Each element in the node list stores start and end pointers into the edge list for its range. Each nonempty entry in the edge list contains the destination vertex and the edge value. Each node's range in the edge list has a corresponding sentinel entry in the edge list which points back to the source in the node list for updating the node pointers.

We present an example of a graph stored in PCSR format in Figure A-4.

The size of the node list is $O(n)$ since it stores two pointers for each node. The size of the edge PMA is $O(n+m)$ since it stores an entry for each edge and node. The size of an PMA is $O(N)$ where N is the number of elements in the PMA. Therefore, the total space usage of PCSR is $O(n+m)$, the same as standard CSR.

Operations

Adding a node. We add nodes by extending the length of the node array by one with a pointer to the end of the edge structure. We then add the sentinel edge into the edge structure.

Adding an element to the end of the node structure is $O(1)$ and adding an element to the edge structure is $O(\lg^2(n+m))$, so the overall time is $O(\lg^2(n+m))$.

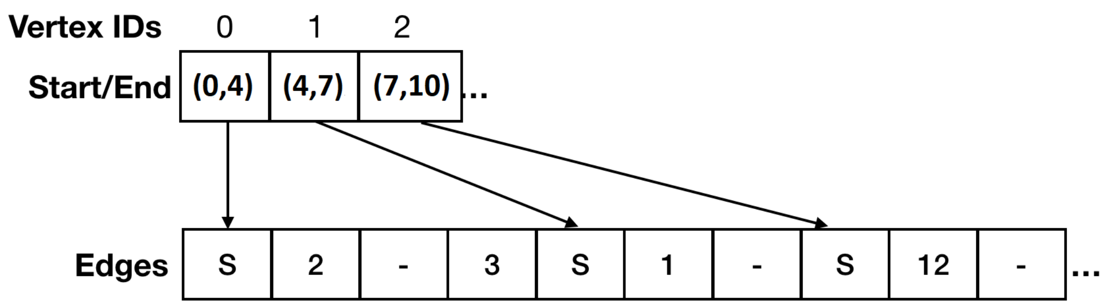


Figure A-4: An example of a graph stored in PCSR. S denotes the sentinels. The ranges (start, end) in the vertex array denote the start and end of the corresponding edges in the edge array.

Adding an edge. Adding an edge first requires finding the node in the node array, then requires a binary search on the relevant section of the edge array to insert the edge in sorted order indexed by its destination. If a rebalance is triggered, we check every moved edge to see if it is a sentinel. If so, we update the node array with its new location.

Finding the location in the node structure is $O(1)$, binary searching the relevant section of the edge array is $O(\lg(deg(v)))$, and inserting is $O(\lg^2(n+m))$, giving us $O(\lg^2(n+m))$ for the overall time.

Removing an edge. Removing an edge is symmetric to adding an edge. We find the edge with binary search and then remove it from the PMA and rebalance if necessary. Therefore, the runtime is the same as adding an edge: $(O(\lg^2(n+m)))$.

Removing a node. First, we set the start and end pointers into the edge array to null. We can also keep track of the number of removed nodes and rebuild the entire structure when the number of non-removed nodes equals the number of removed nodes. Nodes can only be removed after all of their edges have been removed[2]. We need to both mark the node in the node structure and remove the sentinels from the edge structure. This takes time $O(\lg^2(n+m))$.

To maintain the node list with $O(n)$ entries we can simply rebuild the structure every time the number of removed nodes exceeds half the number of nodes before node deletions.

We have not implemented removing edges and nodes, but their asymptotic performance is symmetric to adding edges and nodes.

A.3 Results

We evaluated PCSR against CSR, adjacency list (AL), and blocked adjacency lists (BAL). We do not compare to the adjacency matrix due to its inability to scale to large graphs. We will evaluate the structures on their performance and their space usage. We focus on the sparse case since the adjacency matrix outperforms all other

[2]It would be possible to implement a faster bulk edge removal by deleting all the edges at once and not doing rebalances until the end.

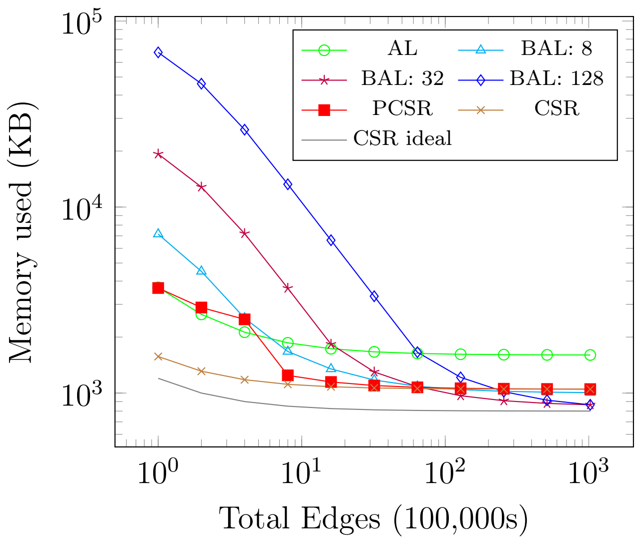


Figure A-5: Size per 100,000 edges of each data structure with 100,000 nodes and a variable number of edges. The x-axis represents the number of edges, while the y-axis represents the size per 100,000 elements.

graph representations if the graph is dense. We randomly generated variable numbers of edges in a graph with a constant number of nodes for our tests.

System. We ran our experiments on an AWS instance with 18 cores, with hyper-threading, and 2.9GHz clock speed. The machine had 64GB of RAM, 32K of L1 cache, 256K of L2 cache, and 25600K of L3 cache. Programs were written in c++ and compiled with GCC 4.8.5 with -O3. All programs were run sequentially.

Memory footprint

We measured the memory footprint of each data structure for a fixed number of nodes and variable number of edges. Figure A-5 shows the relative growth of the memory footprint of each graph representation.

The BALs use much more size than necessary when the average degree is small because most of the space in the blocks is empty.

The `c++` vector for the edge list in CSR doubled the speed of inserts since our implementation of CSR (on average) only needs to copy half of the elements and not all of them on each insert. Therefore, we also compare to the ideal CSR size without extra space.

We found that there is about a factor of 2 between the size of an ideal CSR (without extra padding) and the worst AL and that PCSR only has a space overhead of between 20% and 30%.

Inserts

We benchmarked the time to insert unique edges on all of the data structures. We generated edges uniformly at random without replacement. Figure A-6 shows the time to insert $100,000$ edges with a fixed number of nodes and a variable number of edges. AL-based representations supported fast inserts, while CSR was the slowest.

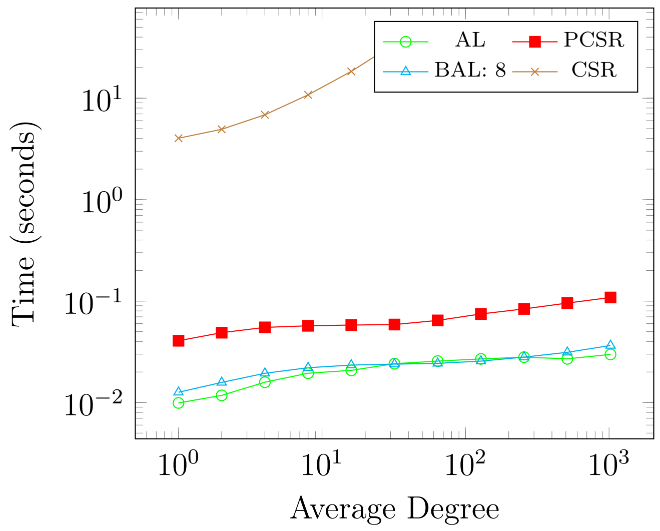


Figure A-6: Time to insert 100,000 edges with a fixed number of nodes. We used 100,000 nodes and added a variable number of edges.

CSR starts about 3 orders of magnitude slower than all other representations and also scales much worse. Therefore, we are unable to run it for large numbers of edges. We find that in practice that PCSR is about 3-4 times slower than AL based representations.

Updates

We benchmarked update operations on all of the data structures. We generated edges uniformly at random with replacement. Figure A-6 shows the time to insert edges that potentially exist in the edge list with a fixed number of nodes. The difference between update and insert is that update requires a search beforehand to check if the edge is already in the structure. We again show the time for updating (or inserting) $100,000$ edges.

PCSR outperformed all other structures when the average degree grew to reasonable sizes, as expected from Table A.1. Once again, CSR is several orders of magnitude worse and is too slow to complete on reasonable input sizes. Additionally, the AL-based representations take linear time to search and take much longer than the $O(\lg^2(n))$ search time of PCSR. While the high search cost in AL-based representations can be somewhat offset by increasing the size of the block, larger block sizes increase the size of the AL and slow insertions.

Sparse matrix-vector multiplication

Figure A-8a shows the time to perform a sparse matrix-vector multiplication using the different structures with 100,000 nodes and a variable number of edges.

Although the asymptotic complexity for SpMV is the same for all of the structures, the AL-based structures can suffer from poor cache behavior. Increasing the block size in BALs can improve cache performance. PCSR avoids the problem of cache locality because it stores all of its edges in a single array. SpMV takes longer in AL than PCSR because the PCSR has better cache behavior. SpMV in PCSR is within

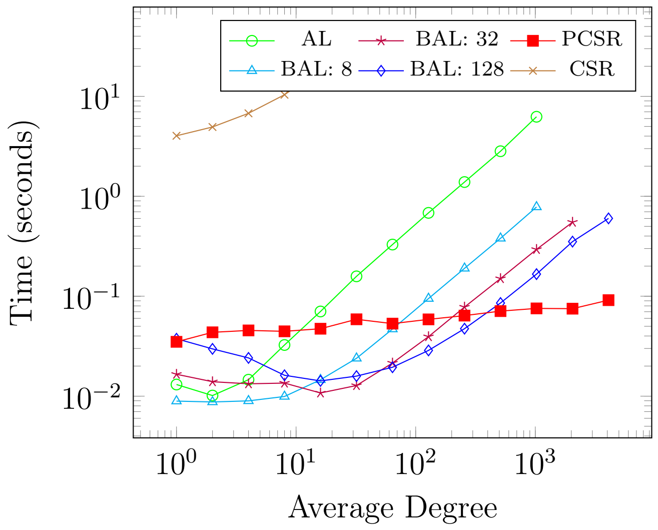


Figure A-7: Time to insert or update 100,000 edges with a fixed number of nodes. We used 100,000 nodes and added a variable number of edges.

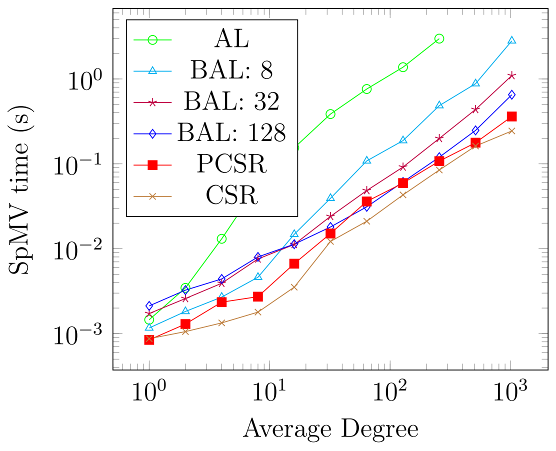
a factor of 2 and often within 20% of SpMV in CSR.

PageRank and BFS

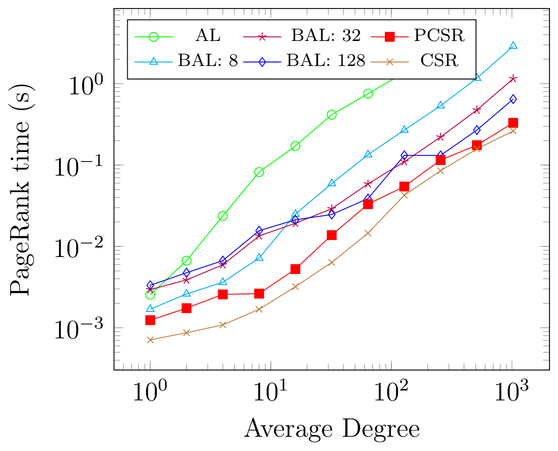
Figure A-8b shows the time to perform an iteration of PageRank using the different structures with 100,000 nodes and a variable number of edges.

Figure A-8c shows the time to compute the distance to each node from a randomly chosen source node using each of the different structures with 100,000 nodes and a variable number of edges.

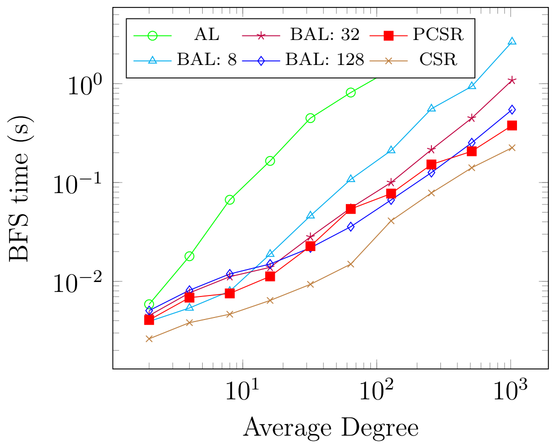
The time to perform a BFS and an iteration of PageRank scales with the number of edges in the graph in all representations.



(a) Sparse matrix-vector multiplication.



(b) PageRank.



(c) BFS.

Figure A-8: Time with 100,000 nodes and a variable number of edges. The x-axis represents the number of edges, while the y-axis represents the time

Real-world graphs

We also tested on three social network graphs of varying sizes from the Stanford Large Network Dataset Collection and report our results in Table A.2. They were Slashdot, with 77,360 nodes and 905,468 edges, Pokec with 1,632,803 nodes and 30,622,564 edges, and LiveJournal with 4,847,571 nodes and 68,993,773 edges.

For adding and updating edges, we added $1,000$ random edges chosen without replacement with the same distribution as the edges in the original graph.

We found that PCSR was about a factor of 2 slower than CSR on graph computations but had much faster updates. The AL-based representations had similar size to PCSR and were between 2 to 10 times slower on graph computations but about 4 times faster in adding edges.

Graph Format	*AL*	*BAL 8*	*BAL 32*	*BAL 128*	*CSR*
Slashdot					
Size	0.88	0.82	1.47	4.71	0.41
SpMV	10.87	1.29	1.45	1.32	0.39
BFS	8.86	1.20	1.42	1.17	0.47
PageRank	13.38	1.64	1.85	1.72	0.36
Adding edges	0.25	0.25	0.25	0.25	525.00
Updating edges	10.50	1.25	1.00	0.75	508.75
Pokec					
Size	0.93	0.71	0.98	2.75	0.45
SpMV	15.95	2.43	1.21	1.17	0.51
BFS	7.25	1.64	1.02	1.00	0.48
PageRank	11.77	3.04	1.78	1.72	0.54
Adding edges	0.25	0.50	0.25	0.25	31628.50
Updating edges	9.17	2.50	0.83	0.67	21005.83
Livejournal					
Size	1.05	0.87	1.36	4.00	0.49
SpMV	20.40	2.77	2.20	2.10	0.59
BFS	9.55	2.30	1.34	1.40	0.53
PageRank	16.20	5.36	2.40	2.73	0.54
Adding edges	0.25	0.25	0.25	0.50	70787.00
Updating edges	13.17	4.00	1.50	1.17	46835.00

Table A.2: Real-world graphs. We tested on Slashdot, pokec, and Livejournal. All times are normalized against PCSR.

Appendix B

Cache Adaptivity

B.1 What bad memory profiles look like

We begin by explaining how an (a, b, c)-regular algorithm can fail to be adaptive in the worst case, and why there is reason to hope that the worst cases are brittle.

`MM-Scan`: a canonical non-adaptive algorithm. Consider a divide-and-conquer matrix-multiplication algorithm `MM-Scan` that computes eight subresults and then merges them together using a linear scan (see pseudocode in Appendix B.2). `MM-Scan` is an $(8, 4, 1)$-regular cache-oblivious algorithm and its recurrence relation is $T(N) = 8T(N/4) + \Theta(N/B)$. Its I/O complexity is $O(N^{3/2}/\sqrt{M}B)$, which is optimal for an algorithm that performs all the elementary multiplications of a naïve nested-loop matrix multiply [153, 154].

However, since `MM-Scan` has $c = 1$ in its recurrence, it is not adaptive: there are bad memory profiles that cause it to run slowly despite giving `MM-Scan` ample aggregate resources[1]. This section gives intuition for what these bad profiles look like.

A worst-case profile for `MM-Scan`. Here's how to make a bad profile for `MM-Scan` [43]. The intuition is to give the algorithm lots of memory when it cannot benefit from it, i.e., when it is doing scans, and give it a paucity of memory when it could most use it, i.e., during subproblems.

Concretely, during a scan of size N, which takes N/B I/Os, set the memory to the fixed size N/B. Repeat recursively. Thus, a bad profile for `MM-Scan` on a problem of size N consists of eight recursive bad profiles for $N/4$ followed by a "square" of size N/B I/Os by N/B blocks of cache; see Figure B-1.[2] This recursion continues down to squares of size $\Theta(B)$ blocks[3]. `MM-Scan`'s I/O cost with this worst-case profile is exactly the same as if the memory stayed constant at its smallest possible value. `MM-Scan` can perform exactly one multiply of $\Theta(\sqrt{N} \times \sqrt{N})$ matrices on this profile. `MM-Inplace`, on the other hand, can perform $\Omega(\log \frac{N}{B})$ multiplies on this profile [43]. This proves that `MM-Scan` is not optimal in the cache-adaptive model.

This worst-case profile exactly tracks the execution of `MM-Scan`. From the perspective of the algorithm, the memory profile *does the wrong thing at every time step*; whenever `MM-Scan` cannot use more memory, it gets the maximum amount possible, and whenever it can use more memory, that memory gets taken away. This bad example for matrix multiplication generalizes to any $(a, b, 1)$-regular algorithm. When $c < 1$, this construction is ineffective—the scans are simply too small to waste a significant amount of resources.

The `MM-Scan` example reveals a fascinating combinatorial aspect of divide-and-

[1]There is an alternate form of the algorithm, `MM-Inplace`, that immediately adds the results of elementary multiplications into the output matrix as they are computed. Since it needs no linear scan to merge results from sub-problems, it is an $(8, 4, 0)$-regular algorithm. Consequently, its I/O complexity in the DAM model is also $O(N^{3/2}/\sqrt{M}B)$, but it *is* optimally cache-adaptive.

[2]In the cache-adaptive model, it's enough to analyze cache-oblivious algorithms only on ***square profiles***, defined as follows [43]. Whenever the RAM size changes to have the capacity for x blocks, it stays constant x I/Os before it can change again. Chapter 9 focuses exclusively on cache-oblivious algorithms, so we use square profiles throughout.

[3]We stop at $\Theta(B)$ blocks due to the ***tall-cache requirement*** of `MM-Scan` [153].

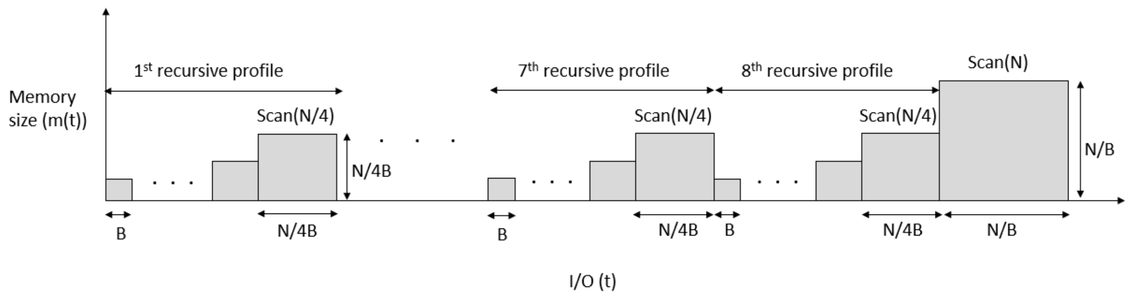


Figure B-1: A bad profile for MM-SCAN as defined recursively.

conquer algorithms. At some points of the execution, the I/O performance is sensitive to the size of memory and sometimes it is almost entirely insensitive. These changes in memory sensitivity make cache-adaptive analysis nontrivial.

B.2 Pseudocode for `MM-Scan`

```
MM-Scan($n, A, B$)
if $N = 1$
    return $A \times B$
else
    $X_{TL} = $ MM-Scan($n/2, A_{TL}, B_{TL}$)
    $X_{TR} = $ MM-Scan($n/2, A_{TL}, B_{TR}$)
    $X_{BL} = $ MM-Scan($n/2, A_{BL}, B_{TL}$)
    $X_{BR} = $ MM-Scan($n/2, A_{BL}, B_{TR}$)
    $Y_{TL} = $ MM-Scan($n/2, A_{TR}, B_{BL}$)
    $Y_{TR} = $ MM-Scan($n/2, A_{TR}, B_{BR}$)
    $Y_{BL} = $ MM-Scan($n/2, A_{BR}, B_{BL}$)
    $Y_{BR} = $ MM-Scan($n/2, A_{BR}, B_{BR}$)
    $C = X + Y$ ▷ Linear scan
    return $C$
```

Figure B-2: Cache-oblivious matrix multiply of two $n \times n$ matrices (each of size $N = n^2$) with $\Theta(1 + N/B)$ linear scan [153]. In this pseudocode, A_{BR} refers to the Bottom Right quadrant of a matrix, A_{TL} the Top Left, etc.

B.3 Additional figures

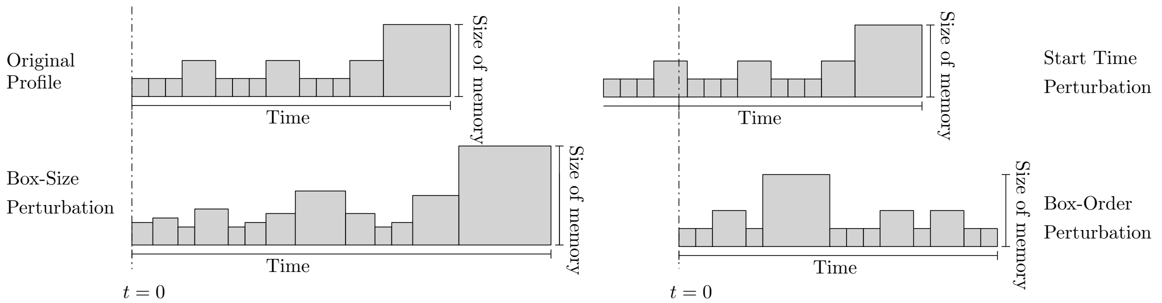


Figure B-3: Four box profiles providing an example of the three smoothing transformations explored in Section 9.5. The first profile is an example of an original profile. The next three profiles represent respectively: *box-size Perturbations*, *start time perturbations* and, *box-order perturbations* of the Original Profile.

B.4 Proof of Theorem 9.14 and Lemma 9.13

Theorem 9.14 (Martingale Optional Stopping Theorem [381]) *Let $X_1, X_2, \ldots$ be iid random variables, and let γ be a function such that $\gamma(X_i)$ has finite mean μ. Consider an arbitrary process that runs in steps, and at each step i is given the value of X_i. Suppose that the process terminates after no more than C steps for some value C. Let S be the random variable denoting the number of steps that the process runs. Then,*

$$\mathbb{E}\left[\sum_{i=1}^{S} \gamma(X_i)\right] = \mathbb{E}[S] \cdot \mu.$$

Proof. Expanding $\mathbb{E}[\sum_{i=1}^{S} \gamma(X_i)]$ gives

$$\mathbb{E}\left[\sum_{i=1}^{S} \gamma(X_i)\right] = \sum_{i=1}^{C} \Pr[S \geq i] \cdot \mathbb{E}[\gamma(X_i) \mid S \geq i].$$

The key observation is that $\mathbb{E}[\gamma(X_i) \mid S \geq i] = \mu$, since the decision of whether $S \geq i$ is a function only of $X_1, \ldots, X_{i-1}$, and is therefore independent of X_i. Thus

$$\mathbb{E}\left[\sum_{i=1}^{S} \gamma(X_i)\right] = \mu \cdot \sum_{i=1}^{C} \Pr[S \geq i] = \mu \cdot \mathbb{E}[S].$$

□

Lemma B.1 *For any box-size distribution Σ, and any (a, b, c)-regular algorithm $\mathcal{A}$,*

$$\mathbb{E}\left[\sum_{i=1}^{\mathcal{S}_n} m_n(|\square_i|)\right] = \mathbb{E}[\mathcal{S}_n] \cdot m_n.$$

Proof. Consider the random process that selects boxes $\square_1, \ldots, \square_{\mathcal{S}_n}$, each of size independently drawn from a distribution Σ, until the algorithm $\mathcal{A}$ is able to use the box to complete on *any* problem of size n. Then since $\mathcal{S}_n$ is bounded above by a function of n (i.e., $\mathcal{S}_n \leq O(n^{\log_b a})$), Theorem 9.14 tells us that

$$\mathbb{E}\left[\sum_{i=1}^{\mathcal{S}_n} m_n(|\square_i|)\right] = \mathbb{E}[\mathcal{S}_n] \cdot m_n.$$

□

B.5 Proof of the No-catch-up Lemma

Lemma B.2 *Let $\sigma = (r_1, r_2, r_3, \ldots)$ be a sequence of memory references, and let $S = (\square_1, \square_2, \ldots \square_k)$ be a sequence of squares. Suppose that if $\square_1$ starts at r_i, then $\square_k$ finishes at r_j. Then, for all $i' < i$, if $\square_1$ starts at $r_{i'}$, then for some $j' \leq j$, $\square_k$ finishes at $r_{j'}$.*

Proof. We prove this via induction on k, the number of squares in the sequence. We first prove the base case of $k = 1$.

In the base case when $k = 1$, if $\mathcal{A}$ starts $\square_1$ at access r_i and finishes $\square_1$ at access r_j, then the number of *distinct* blocks in the sequence $r_i, \ldots, r_{j+1}$ is $|\square_1| + 1$. If $\mathcal{A}$ instead starts $\square_1$ on access $r_{i'}$ for some $i' < i$, then the number of distinct blocks in the sequence $r_{i'}, \ldots, r_i, \ldots, r_{j+1}$ will also be at least $|\square_1| + 1$. Thus, $\square_1$ cannot now finish at any $r_{j'}$ satisfying $j' \geq j + 1$.

For our inductive step, we assume that the lemma holds for all $k \leq l$. We now prove the lemma holds for $k = l + 1$. Given $\square_1, \ldots, \square_l$ and a starting point r_i, let r_q be the access at which $\square_l$ finishes when $\mathcal{A}$ starts $\square_1$ at access r_i. By our inductive hypothesis, if $\mathcal{A}$ starts $\square_1$ at access $r_{i'}$ where $i' < i$, then $\mathcal{A}$ must finish $\square_l$ at access $r_{q'}$ where $q' \leq q$. Applying our proof of the base case (when $k = 1$), the memory access r_j at which $\square_{l+1}$ will finish if it starts at r_{q+1}, must come at or after the memory access $r_{j'}$ at which $\square_{l+1}$ will finish if it starts at $r_{q'+1}$. This completes the proof of the theorem. $\square$

B.6 Standardizing $(a, b, c = 1)$-regular algorithms

Lemma B.3 *Let $\mathcal{A}$ be an $(a, b, c = 1)$-regular algorithm, where $b < a \in O(1)$. Then there is an $(a, b, c = 1)$-regular algorithm $\mathcal{A}'$ which has the same access pattern as $\mathcal{A}$ but which can be written as (1) a single scan consisting of at most $O(n)$ block accesses followed by (2) an $(a, b, c = 1)$-regular algorithm $\mathcal{B}$ in which in which each subproblem has its scan entirely at the end of the subproblem (rather than between or before sub-calls to smaller subproblems).*

The proof of Lemma B.3 uses a variant of the scan-hiding technique from Chapter 10.
Proof. For each subproblem in $\mathcal{A}$, we break the scan into $a+1$ ***scan pieces***, where the first scan piece is the portion of the scan that occurs before any recursion, the second scan piece is the portion of the scan that occurs between the first and second recursive subcall, and so on.

Consider an execution of $\mathcal{A}$ on an input of size n. Call a non-base-case subproblem S in $\mathcal{A}$ a ***prefix subproblem*** if S is either the entire problem of size n, or is the subproblem resulting from the first recursive call of another prefix subproblem. Call a scan piece a ***prefix scan piece*** if it appears at the beginning of a prefix subproblem, before any recursive calls are made within the prefix subproblem.

Notice that in the execution of $\mathcal{A}$, the prefix scan pieces are performed before any other part of the computation. We define $\mathcal{A}'$ to begin by performing the prefix scan pieces together as a single large scan. For each subproblem size, there can be at most one prefix subproblem of that size. Thus the sum of the sizes of the prefix scan pieces is at most

$$O\left(\sum_{i=1}^{\log_b n} b^i\right) \leq O(n).$$

Moreover, since there are only $O(\log n)$ such pieces, their concatenation will still satisfy the property that a sufficiently large cache of constant size can complete them in $O(n)$ accesses.

The algorithm $\mathcal{A}'$ must then perform the portions of $\mathcal{A}$ that are not prefix scan pieces. Next we reinterpret these portions as an (a, b, c)-regular algorithm $\mathcal{B}$ in which scans occur only at the ends of subproblems. In a subproblem S of $\mathcal{A}$, we call a scan piece ***unassigned*** if it is not a prefix scan piece and occurs before the final recursive subcall in the subproblem.

For each unassigned scan piece in $\mathcal{A}$, we "assign ownership" for the scan piece to whichever subproblem finishes the latest out of the subproblems that finish before the scan piece. (Note that such a subproblem will exist because the scan-piece is not a prefix scan piece.) We then define $\mathcal{B}$ to be the algorithm with the same access pattern as $\mathcal{A}$ (without the prefix scan pieces), except that in the execution of $\mathcal{B}$ each subproblem (including base-case subproblems) includes any later scan pieces to which the subproblem has been assigned ownership. (Note, in particular, that each subproblem in $\mathcal{A}$ appears immediately before all scan-pieces to which it has been assigned ownership, with no other memory accesses in-between.)

Let us consider the sum of the sizes of the scan-pieces assigned to any given subproblem S of some size m. Note that any subproblem S' of size greater than $b \cdot m$ cannot assign ownership of any of its scan pieces to S; in particular, the subproblem of size $b \cdot m$ containing S must complete before any scan pieces in any such subproblem S' can occur, thereby preventing the scan pieces from being assigned to S. Moreover, for each sub-problem-size $k \leq b \cdot m$ there can be at most one subproblem of size k that assigns ownership of any of its scan pieces to S. Thus the total combined size of the scan pieces assigned to S can be at most

$$O\left(\sum_{i=1}^{\log_b(b \cdot m)} b^i\right) \leq O(m).$$

This ensures that the scans in each subproblem of size m in $\mathcal{B}$ access $O(m)$ distinct blocks, and more importantly, can be completed by a constant-size cache in time $O(m)$, meaning that algorithm $\mathcal{B}$ is, in fact, an (a, b, c)-regular algorithm. Since all of the scans in $\mathcal{B}$ occur only at the ends of subproblems, the proof is complete. $\square$

B.7 Triangle profiles

Previous work showed that only considering square profiles is sufficient [45] for determining cache-adaptivity of an algorithm. In this section, we show the same result but with right triangles. A *triangle profile* can be described as a square profile where the cache is cleared between each square, hence producing a "triangular" profile composed of many adjacent right triangles.

Throughout this section, we use $\mathcal{A}$ to refer to some particular (a, b, c)-regular algorithm where $a, b \in \mathbb{N}$ and a, b, c are constants. Recall Lemma 9.6 relating potential to (a, b, c)-regular algorithms; since the algorithm $\mathcal{A}$ in question is (a, b, c)-regular, we let $\rho(|\square_i|) = \Theta(|\square_i|^{\log_b a})$. Finally, we let $W_n = \Theta(n^{\log_b a})$ be the total amount of progress $\mathcal{A}$ *must make* on a problem of size n in order to complete. Throughout this section, we treat both W_n and $\rho(|\square_i|)$ as fixed polynomials in n and $|\square_i|$, respectively, provided constants a, b, and c.

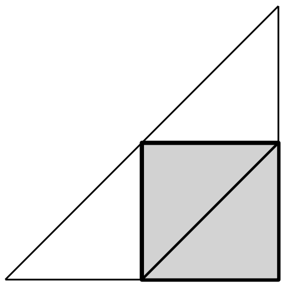


Figure B-4: A triangle contained in a box and a triangle containing a box. The box is shaded in light gray.

Intuitively, we show that triangle profiles are sufficient in proving the optimality (or non-optimality) of algorithms by noting that a box of X cache lines that lasts for X time steps fits under a triangle that starts with 0 cache lines, ends at $2X$ cache lines, and lasts for $2X$ time steps[4]. This logic accounts for the outer triangle in Fig. B-4. We also note that a triangle of height and width X fits inside a box of size X, accounting for the inner triangle in Fig. B-4. This intuition tells us that any square profile can be upper and lower bounded by a triangle profile up to a factor of 2 (or 1/2).

We now formalize this intuition. First, take a square profile and consider two related triangular profiles, the lower triangular profile and the upper triangular profile.

Definition B.4 *A triangle, $\triangle$, of size X lasts for X IOs and on the i^{th} IO the size of the cache is i cache lines.*

The size of a triangle is represented as $|\triangle| = X$.

Definition B.5 *A triangular profile, $M(t)$, is formed by a sequence of k triangles $\triangle_1 \circ \triangle_2 \circ \ldots \circ \triangle_k$ of sizes $\triangle_1 = X_1$.*

Definition B.6 *Given a square profile $M(t) = \square_1 \circ \ldots \circ \square_k$ we will define the lower and upper triangular profiles.*

[4]Recall that *time steps* are counted in terms of block read-ins from disk to cache.

The lower triangular profile of $M(t)$ is $M_{LT}(t)$ where $\triangle_1 \circ \triangle_2 \circ \ldots \circ \triangle_k$ and $|\triangle_i| = |\Box_i|$.

The upper triangular profile of $M(t)$ is $M_{UT}(t)$ where $\triangle'_1 \circ \triangle'_2 \circ \ldots \circ \triangle'_k$ and $|\triangle'_i| = 2|\Box_i|$.

We give an example of an upper and lower triangular profile in Figure B-5. We will similarly need to bound a triangular profile by square profiles.

Definition B.7 *Given a triangular profile $T(t)$ $\triangle_1 \circ \triangle_2 \circ \ldots \circ \triangle_k$ we will define the lower square profile.*

The upper square profile of $T(t)$ is $T_{US}(t)$ where $\Box_1 \circ \Box_2 \circ \ldots \circ \Box_k$ and $|\Box_i| = |\triangle_i|$.

The lower square profile of $T(t)$ is $T_{LS}(t)$ where $\Box'_1 \circ \Box'_2 \circ \ldots \circ \Box'_k$ and $|\Box'_i| = |\triangle_i|/2$.

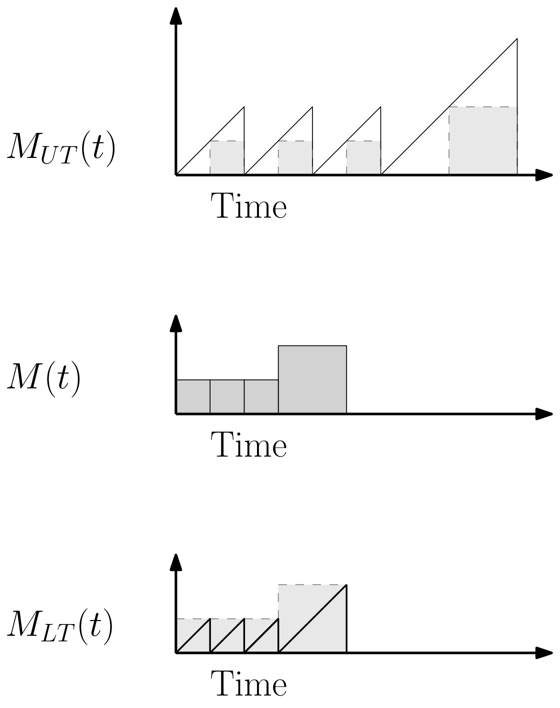


Figure B-5: An example of a lower triangular profile and an upper triangular profile. The boxes from the profile $M(t)$ are presented with dashed lines on the corresponding upper and lower triangular profiles.

Definition B.8 *Let $\rho(\triangle)$ be the maximum possible progress that $\mathcal{A}$ can make on a triangle of size $|\triangle|$.*

Lemma B.9 *Consider $|\triangle_1| = x$, $|\triangle_2| = 2x$ and $|\Box_1| = x$.*

Then the progress for our :

$$\rho(\triangle_1) = \Theta\left(x^{\log_b(a)}\right) \tag{B.1}$$

$$\rho(\triangle_2) = \Theta\left(x^{\log_b(a)}\right) \tag{B.2}$$

$$\rho(\Box_1) = \Theta\left(x^{\log_b(a)}\right) \tag{B.3}$$

PROOF. Note that we can solve a problem of size x using $\angle_1$. Thus, we can make $x^{\log_b(a)}$ progress. By Lemma 9.6 we have that $\rho(\square_1) = \Theta\left(x^{\log_b(a)}\right)$. A triangle of size x fits inside a square of size x and by memory monotonicity we have that $\rho(\angle_1) = O\left(x^{\log_b(a)}\right)$. Thus, $\rho(\angle_1) = \theta\left(x^{\log_b(a)}\right)$.

By this we have that $\rho(\angle_2) = \theta\left((2x)^{\log_b(a)}\right)$. Simplified $\rho(\angle_2) = \theta\left(x^{\log_b(a)}\right)$. $\square$

Corollary B.10 *Let $M(t)$ be a square profile. Then the progress for our (a, b, c)-regular algorithm is:*

$$\rho(M_{LT}(t)) = \Theta\left(\rho(M(t))\right)$$

and

$$\rho(M_{UT}(t)) = \Theta\left(\rho(M(t))\right).$$

PROOF. $\rho(M(t))$ is the sum of the progress in the squares that make up the profile. $\rho(M_{UT}(t))$ and $\rho(M_{LT}(t))$ are the sums of the progress in the triangles that make up the profile. For any particular square, $|\square_i| = x$, in $M(t)$ there is a one to one correspondence with a triangle $|\angle_i| = x$ in profile $M_{LT}(t)$ and a triangle $|\angle'_i| = 2x$ in profile $M_{UT}(t)$. By Lemma B.9 these all have the same progress up to constant factors. Thus, the sum of the progress of the boxes and triangles that make up these profiles will be within constant factors of each other. $\square$

Lemma B.11 *If $\mathcal{A}$ completes on $M(t)$ then it completes on $M_{UT}(t)$.*
If $\mathcal{A}$ doesn't complete on $M(t)$ then it also doesn't complete on $M_{LT}(t)$.

PROOF. **For the first statement:** We prove this by induction. Let $\mathcal{A}$ be the sequence of accesses $a_1, a_2, \ldots, a_y$. Let $M(t) = \square_1 \circ \ldots \circ \square_k$. Let $M_{UT}(t)$ where $\angle'_1 \circ \angle'_2 \circ \ldots \circ \angle'_k$ and $|\angle'_i| = 2|\square_i|$.

Assume $\mathcal{A}$ would complete access a_j by the end of $\square_i$ and $\mathcal{A}$ would complete access a_ℓ by the end of $\angle'_i$. Further assume $i \leq \ell$. Let $a_{j'}$ be the access that $\mathcal{A}$ finishes by the end of $\square_{i+1}$. Let $a_{\ell'}$ be the access that $\mathcal{A}$ finishes by the end of $\angle'_{i+1}$. $\square_{i+1}$ starts with at most $|\square_{i+1}|$ cache lines in memory and can bring in at most $|\square_{i+1}|$ new cache lines. $\angle'_{i+1}$ starts with zero cache lines in memory, and can bring in $2|\square_{i+1}|$ cache lines into memory. If $j' > \ell'$ then during $\square_{i+1}$ $\mathcal{A}$ completes more accesses in x cache misses with at most x cache lines in memory at the start. However, $\angle'_{i+1}$ can do any computation over $2x$ IOs, thus $\mathcal{A}$ run on $\angle'_{i+1}$ can compute everything that $\mathcal{A}$ run on $\square_{i+1}$ computes. This is a contradiction. Thus, if $j \leq \ell$ then $j' \leq \ell'$.

Base case: Before the start of the first boxes $j = \ell = 0$. Thus, by the end of $\square_1$ if $\mathcal{A}$ reaches a_j and $\mathcal{A}$ gets to access a_ℓ by the end of $\angle'_1$ then $j \leq \ell$.

For the second statement: Proof by contradiction, if $\mathcal{A}$ completes on $M_{LT}(t)$ then by memory monotonicity it must also complete on $M(t)$. However, by assumption it does not, thus $\mathcal{A}$ does not complete on $M_{LT}(t)$. $\square$

Lemma B.12 *If $\mathcal{A}$ is adaptive on square profile $M(t)$ then it is adaptive on $M_{UT}(t)$ as well.*

PROOF. If $\mathcal{A}$ completes on $M(t)$ then it completes on $M_{UT}(t)$ and $\rho(M_{UT}(t)) = \Theta(\rho(M(t)))$.

Thus, if $\mathcal{A}$ is adaptive on $M(t)$ it continues to be non-adaptive. $\square$

Lemma B.13 *If $\mathcal{A}$ is adaptive on triangular profile $M_{LT}(t)$ then it is adaptive on $M(t)$ as well.*

PROOF. If $\mathcal{A}$ completes on $M_{LT}(t)$ then it completes on $M(t)$ and $\rho(M(t)) = \Theta(\rho(M_{LT}(t)))$.

Thus, if $\mathcal{A}$ is adaptive on $M_{LT}(t)$ it continues to be non-adaptive. $\square$

The following theorem will allow us to use triangular profiles when proving non-adaptivity and non-adaptivity in expectation.

Theorem B.14 *Let $\square_n$ be a box of size n.*

If $\mathcal{A}$ is non-adaptive on triangular profile $T(t)$ then it is also non-adaptive on the square profile $T_{LS}(t)$.

PROOF. If $\mathcal{A}$ is non adaptive on $T(t)$ then

$$\rho\left(T(t)\right) = \omega\left(W\right).$$

Furthermore, $T(t)$ is the upper triangular profile of $T_{LS}(t)$ and thus

$$\rho\left(T_{LS}(t)\right) = \omega\left(W\right).$$

Let us define i and j such that $\mathcal{A}$ completes on the i^{th} triangle of $T(t)$ and $\mathcal{A}$ completes on the j^{th} square of $T_{LS}(t)$. Then, because $T(t)$ is the upper triangular profile of $T_{LS}(t)$ we can use Lemma B.11 to say that the finishing point of $\mathcal{A}$ is later in $T_{LS}(t)$ than in $T(t)$. So, we have that $j \geq i$.

When $\mathcal{A}$ completes it makes W_n progress, but the available potential progress in the boxes of $T_{LS}(t)$ that it uses will be $\omega\left(W_n\right)$. The algorithm will continue to be non-adaptive on this related profile.

A distribution over triangular profiles can be connected to a distribution over square profiles by converting each triangular profile into $M_{LT}(t)$, a square profile. $\square$

The following theorem will allow us to use triangular profiles when proving results about algorithms being adaptive in expectation.

Theorem B.15 *Let D be a distribution over triangular profiles $T(t)$.*

If $\mathcal{A}$ is adaptive in expectation over a distributions D $\mathcal{A}$ is adaptive over a distribution D', where D' is formed by taking every triangular profile $T(t) \in D$ and replacing it with $T_{US}(t)$.

PROOF. Note that the lower triangular profile of $T_{US}(t)$ is $T(t)$. So, we can apply Theorem B.14 to $T(t)$ and $T_{US}(t)$. Every adaptive profile $T(t)$ corresponds to an adaptive $T_{US}(t)$. Additionally, every non-adaptive profile $T(t)$ corresponds to an non-adaptive $T_{US}(t)$. However, in addition to this, the optimal progress over every profile $T(t)$ and $T_{US}(t)$ are the same up to constant factors. Thus, given an algorithm run on $T(t)$ and the same algorithm run on $T_{US}(t)$ the contribution to the expected optimal progress is within constant factors. $\square$

B.8 Pseudocode for **`AdaptiveStrassen`**

```
ADAPTIVESTRASSEN($pX, pY, pZ, n$)
// We define the global variables
$start\_n = n$
$numLeaves = 0$
// We use the length of a cache line, as mentioned.
$B$ = length of a cache line
// We initialize the sum arrays for input and outputs.
// We need a place to store are pre-computed and post-computed scans.
for $s \in \{x_1 = 0, x_2 = 1, y_1 = 2, y_2 = 3, z_1 = 4, z_2 = 5\}$
    Set $P[s]$ = pointer array of length $\lg(n) + 1$
    for $i \in [0, \lg(n)]$
        $P[s][i]$ = pointer array of length $i + 1$
        for $j \in [0, i - 1]$
            $P[s][i][j]$ =matrix of size $2^j$ by $2^j$
Set $IsActive$ = is a pointer array of length $\lg(n)$
Set $IsPrecompute$ = is a pointer array of length $\lg(n)$
for $i \in [0, \lg(n)]$
    $IsActive[i] = [P[x_1][i], P[y_1][i], P[z_1][i]]$
    $IsPrecompute[i] = [P[x_2][i], P[y_2][i], P[z_2][i]]$
// Now initialize $P[x_1][\lg(n)]$ and $P[y_1][\lg(n)]$

// to be the input matrices for the spines
DOPRECURSESPINE(IsActive)
// We will let the input and scans be represented by pointers for two places
// to read and one to write.
// Next, we will give the length of the input.
// Finally, we will also for convenience use a last entry to mark $\times, +, -, copy$
$input = [P[x_1][\lg(n)], P[y_1][\lg(n)], P[z_1][\lg(n)], n, \times]$
$scans = []$
// Make the recursive call to Strassen where we hide scans
ADAPTIVESTRASSENRECURSE($n, \lg(n), input, scans$)
// Output from multiplications need to be summed together
DOPOSTPROCESSSPINE(IsActive, IsPrecompute)
```

Figure B-6: Pseudocode for the top-level ADAPTIVESTRASSEN routine.

```
AdaptiveStrassenRecurse($n, level, input, scans$)
if $n > 1$
    AdaptiveStrassenRecurseSplit($n, level, input, scans$)
elseif $n \leq 1$
    // Do the multiplication you were asked to do
    $input[0] * input[1] = input[2]$
    // Do the additions you were asked to do by your parent
    for scan in scans
        // Read all the information about your scan out
        $in1 = scan[0]$
        $in2 = scan[1]$
        $out = scan[2]$
        $length = scan[3]$
        $op = scan[4]$
        for $i \in [0, length - 1]$
            // Do the requested operation to the input and write to output
            $out[i] = op(in1[i], in2[i])$
```

Figure B-7: Pseudocode for the top-level AdaptiveStrassenRecurse routine.

```
AdaptiveStrassenRecurseSplit($n, level, input, scans$)
 1  // Now we want to split the scan into seven parts
    // Specifically come up with 7 lists each having about 1/7 of the
    // total scan work
 2  Let $childScan$ = ReturnSplitScans(scans, $B$)
 3  // Also each child needs to do the scans for its sibling
    // These scans represent additions needed to produce matrices for the input
    // to Strassen.
 4  $outX = IsPrecompute[level - 1][0]$
 5  $outY = IsPrecompute[level - 1][1]$
 6  $outZ = IsPrecompute[level - 1][2]$
 7  $inX = input[0]$
 8  $inY = input[1]$
 9  $inZ = input[2]$
10  // Each input has to copy or add matrices
11  preScan(inX, inY, outX, outY, n)
12  // We also need the child nodes to write the outputs of the previous
    // multiplications to the output for the parent multiplication
    // following Strassen's equation.
13  postScan(childScan, inZ, outZ)
14  // Here we have 7 calls to do the recursive tasks
15  for $i \in [0, 6]$
16       // Make the call to each child
17       $\ell = level - 1$
18       $si = childScan[i]$
19       $inChild = [outX, outY, outZ, \times]$
20       AdaptiveStrassenRecurse(n/2, $\ell$, inChild, $si$)
21       // Switch the active and preComputation pointers so my sibling
22       // can use the information
23       $active = IsActive[level]$
24       $IsActive[level] = IsPrecompute[level]$
25       $IsPrecompute[level] = IsActive[level]$
```

Figure B-8: Pseudocode for the strassenAdaptiveRecursiveSplit subroutine.

```
preScan(inX, inY, outX, outY, n)
// Now we want to split the input scan into seven parts.
$childScan[0].append([inX[0], inX[3n/4], outX, +])$
$childScan[0].append([inY[0], inY[3n/4], outY, +])$
$childScan[1].append([inX[2n/4], inX[3n/4], outX, +])$
$childScan[1].append([inY[0], inY[0], outY, copy])$
$childScan[2].append([inX[3n/4], inX[3n/4], outX, copy])$
$childScan[2].append([inY[2n/4], inY[0], outY, -])$
$childScan[3].append([inX[0], inX[1n/4], outX, +])$
$childScan[3].append([inY[3n/4], inY[3n/4], outY, copy])$
$childScan[4].append([inX[3n/4], inX[0], outX, -])$
$childScan[4].append([inY[0], inY[2n/4], outY, +])$
$childScan[5].append([inX[1n/4], inX[3n/4], outX, -])$
$childScan[5].append([inY[2n/4], inY[3n/4], outY, +])$
```

Figure B-9: Pseudocode for the preScan subroutine.

```
postScan(inZ, outZ)
$childScan[1].append([outZ, inZ[0], inZ[0], +])$
$childScan[1].append([outZ, inZ[3], inZ[3], +])$
$childScan[2].append([outZ, inZ[3], inZ[3], 1])$
$childScan[3].append([outZ, inZ[1], inZ[1], +])$
$childScan[3].append([outZ, inZ[3], inZ[3], +])$
$childScan[4].append([outZ, inZ[0], inZ[0], +])$
$childScan[4].append([outZ, inZ[2], inZ[2], +])$
$childScan[5].append([outZ, inZ[0], inZ[0], -])$
$childScan[5].append([outZ, inZ[1], inZ[1], +])$
$childScan[6].append([outZ, inZ[3], inZ[3], +])$
$childScan[7].append([outZ, inZ[0], inZ[0], +])$
```

Figure B-10: Pseudocode for the postScan subroutine.

Appendix C

Included and Excluded Sums

C.1 Analysis of corners algorithm

This section presents an analysis of the time and space usage of the ***corners algorithm*** [118] for the excluded-sums problem. The original article that proposed the corners algorithm did not include an analysis of its performance. As we will see, the runtime of the corners algorithm is a function of the space it is allowed.

Algorithm description. Given a d-dimensional tensor $\mathcal{A}$ of size N and a box B, the corners algorithm partitions the excluded region $C_d(B)$ into 2^d disjoint regions corresponding to the corners of the box. Each excluded sum is the sum of the reductions of each of the corresponding 2^d regions. The corners algorithm computes the reduction of each partition with a combination of prefix and suffix sums over the entire tensor and saves work by reusing prefixes and suffixes in overlapping regions. Figure C-1 illustrates an example of the corners algorithm.

We can represent each length-d combination of prefixes and suffixes as a length-d binary string where a 0 or 1 in the i-th position corresponds to a prefix or suffix (resp.) at depth i. As illustrated in Figure C-2, the corners algorithm defines a computation tree where each node represents a combination of prefixes and suffixes, and each edge from depth $i-1$ to i represents a full prefix or suffix along dimension i. The total height of this computation tree is d, so there are 2^d leaves.

Analysis. The most naive implementation of the corners algorithm that computes every root-to-leaf path without reusing computation between paths takes $\Theta(N)$ space, but $\Theta(dN)$ time per leaf, for total time $\Theta(d2^dN)$. We will see how to use extra space to reuse computation between paths and reduce the total time.

Theorem C.1 (Time / space tradeoff) *Given a multiplicative space allowance c such that $1 \leq c \leq d$, the corners algorithm solves the excluded-sums problem in $\Theta((2^c + 2^d(d-c) + 2^d)N)$ time if it is allowed $\Theta(cN)$ space.*

PROOF. The corners algorithm must traverse the entire computation tree in order to compute all of the leaves. If it follows a depth-first traversal of the tree, one possible use of the extra $\Theta(cN)$ allowed space is to keep the intermediate combination of

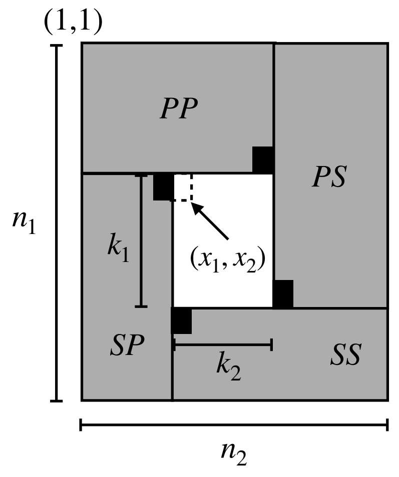


Figure C-1: An example of the corners algorithm in 2 dimensions on an $n_1 \times n_2$ matrix using a (k_1, k_2)-box cornered at (x_1, x_2). The grey regions represent excluded regions computed via prefix and suffix sums, and the black boxes correspond to the corner of each region with the relevant contribution. The labels PP, PS, SP, SS represent the combination of prefixes and suffixes corresponding to each vertex.

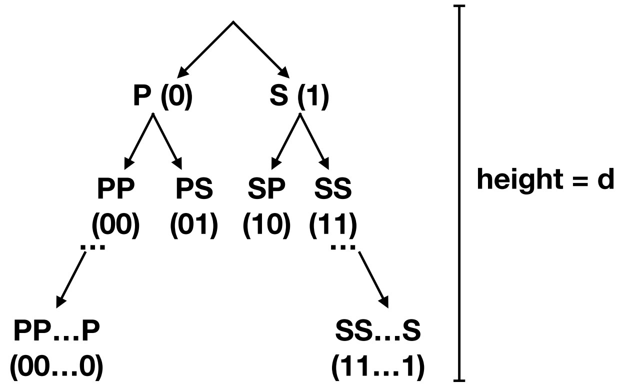


Figure C-2: The dependency tree of computations in the corners algorithm. P and S represent full-tensor prefix and suffix sums, respectively. Each leaf is a string of length d that denotes a series of prefix and suffix sums along the entire tensor.

prefix and suffices at the first c internal nodes along the current root-to-leaf path in the traversal. We will analyze this scheme in terms of 1) the amount of time that each leaf requires independently, and 2) the total shared work between leaves. The total time of the algorithm is the sum of these two components.

Independent work: For each leaf, if the first c prefixes and suffixes have been computed along its root-to-leaf path, there are an additional $(d-c)$ prefix and suffix computations required to compute that leaf. Therefore, each leaf takes $\Theta((d-c)N)$ additional time outside of the shared computation, for a total of $\Theta(2^d(d-c)N)$ time.

Shared work: The remaining time of the algorithm is the amount of time it takes to compute the higher levels of the tree up to depth c given a c factor in space. Given a node v at depth c with position i such that $1 \leq i < c$, the amount of time it takes to compute the intermediate sums along the root-to-leaf path to v depends on the difference in the bit representation between i and $i-1$. Specifically, if i and $i-1$ differ in b bits, it takes bN additional time to store the intermediate sums for node i at depth c. In general, the number of nodes that differ in $b \in \{1, 2, \dots, c\}$ positions at depth c is 2^{c-b}. Therefore, the total time of computing the intermediate sums is

$$N \sum_{b=1}^{c} b 2^{c-b} \approx 2^{c+1} N = \Theta(2^c N).$$

Putting it together: Each leaf also requires $\Theta(N)$ time to add in the contribution. Therefore, the total time is $\Theta\left(\underbrace{2^c N}_{\text{shared}} + \underbrace{2^d (d-c) N}_{\text{independent}} + \underbrace{2^d N}_{\text{contribution}}\right)$. □

The time of the corners algorithm is lower bounded by $\Omega(2^d N)$ and minimized when $c = \Theta(d)$. Given $\Theta(N)$ space, the corners algorithm solves the excluded-sums problem in $O(2^d dN)$ time. Given $\Theta(dN)$ space, the corners algorithm solves the excluded-sums problem in $O(2^d N)$ time.

C.2 Pseudocode and proofs for BDBS-1D

```
BDBS-1D($A, N, k$)
    // Input: List $A$ of size $N$ and
    //  included-sum length $k$.
    // Output: List $A'$ of size $N$ where each
    // entry $A'[i] = A[i : i + k]$ for $i = 1, 2, \ldots N$.
    allocate $A'$ with $N$ slots
    $A_p = A; A_s = A$
    for $i = 1$ to $N/k$
        // $k$-wise prefix sum along $A_p$
        Prefix($A_p$, $(i-1)k + 1$, $ik$)
        // $k$-wise suffix sum along $A_s$
        Suffix($A_s$, $(i-1)k + 1$, $ik$)
    for $i = 1$ to $N$    // Combine into result
        if $i \bmod k = 0$
            $A'[i] = A_s[i]$
        else
            $A'[i] = A_s[i] \oplus A_p[i + k - 1]$
    return $A'$
```

Figure C-3: Pseudocode for the 1D included sum.

Lemma C.2 (Correctness in 1D) BDBS-1D *solves the included sums problem in 1 dimension.*

Proof. Consider a list A with N elements and box length k. We will show that for each $x = 1, 2, \ldots, N$, the output $A'[x]$ contains the desired sum. For $x \mod k = 1$, this holds by construction. For all other x, the previously defined prefix and suffix sum give the desired result. Recall that $A'[x] = A_p[x + k - 1] + A_s[x]$, $A_s[x] = A[x : \lceil (x + 1)/k \rceil \cdot k]$, and $A_p[x + k - 1] = A[\lfloor (x + k - 1)/k \rfloor \cdot k : x + k]$. Also note that for all $x \mod k \neq 1$, $\lfloor (x + k - 1)/k \rfloor = \lceil (x + 1)/k \rceil$.

Therefore,

$$\begin{aligned} A'[x] &= A_p[x + k - 1] + A_s[x] \\ &= A\left[x : \left\lceil \frac{x+1}{k} \right\rceil \cdot k\right] + A\left[\left\lfloor \frac{x+k-1}{k} \right\rfloor \cdot k : x + k\right] \\ &= A[x : x + k] \end{aligned}$$

which is exactly the desired sum. □

Lemma C.3 (Time and space in 1D) *Given an input array A of size N and box length k,* BDBS-1D *takes $\Theta(N)$ time and $\Theta(N)$ space.*

PROOF. The total time of the prefix and suffix sums is $O(N)$, and the loop that aggregates the result into A' has N iterations of $O(1)$ time each. Therefore, the total time of BDBS-1D is $\Theta(N)$. Furthermore, BDBS-1D uses two temporary arrays of size N each for the prefix and suffix, for total space $\Theta(N)$. $\square$

C.3 Pseudocode and proofs for box-complement

```
PREFIX-ALONG-DIM($\mathcal{A}, i$)
1  // Input: Tensor $\mathcal{A}$ ($d$ dimensions, side lengths $(n_1, \ldots, n_d)$), dimension $i$ to
   // do the prefix sum along.
   // Output: Modify $\mathcal{A}$ to do the prefix sum along dimension $i+1$,
   // fixing dimensions up to $i$.
2  // Iterate through coordinates by varying coordinates in dimensions $i+2, \ldots, d$
   // while fixing the first $i$ dimensions.
   // Blanks mean they are not iterated over in the outer loop
3  for   $\{\mathbf{x} = (x_1, \ldots, x_d) \in (\underbrace{n_1, \ldots, n_i}_{i}, \_, \underbrace{:, \ldots, :}_{d-i-1})\}$
4        // Prefix sum along row
         // (can be replaced with a parallel prefix)
5        for $\ell = 2$ to $n_{i+1}$
6             $\mathcal{A}[\underbrace{n_1, \ldots, n_i}_{i}, \ell, \underbrace{x_{i+2}, \ldots, x_d}_{d-i-1}] \oplus= \mathcal{A}[\underbrace{n_1, \ldots, n_i}_{i}, \ell - 1, \underbrace{x_{i+2}, \ldots, x_d}_{d-i-1}]$
```

Figure C-4: Prefix sum along all rows along a dimension with initial dimensions fixed.

The suffix sum along a dimension is almost exactly the same, so we omit it.

Lemma C.4 (Time of prefix sum) PREFIX-ALONG-DIM($\mathcal{A}, i$) *takes* $O\left(\prod_{j=i+1}^{d} n_j\right)$ *time.*

PROOF. The outer loop over dimensions $i+2, \ldots, d$ has $\max\left(1, \prod_{j=i+2}^{d} n_j\right)$ iterations, each with $\Theta(n_{i+1})$ work for the inner prefix sum. Therefore, the total time is $O\left(\prod_{j=i+1}^{d} n_j\right)$. □

```
ADD-CONTRIBUTION($\mathcal{A}, \mathcal{B}, i, \mathit{offset}$)
1  // Input: Input tensor $\mathcal{A}$, output tensor $\mathcal{B}$, fixing dimensions up to $i$.
   // Output: For all coords in $\mathcal{B}$, add the relevant contribution from $\mathcal{A}$.
2  for $\{(x_1, \ldots, x_d) \in (:, \ldots, :)\}$
3       if $x_{i+1} + \mathit{offset} \leq n_{i+1}$
4            $\mathcal{B}[x_1, \ldots, x_d] = \mathcal{A}[\underbrace{n_1, \ldots, n_i}_{i}, \underbrace{x_{i+1} + \mathit{offset}, x_{i+2}, \ldots, x_d}_{d-i}]$
```

Figure C-5: Adding in the contribution.

Lemma C.5 (Adding contribution) ADD-CONTRIBUTION *takes* $\Theta(N)$ *time.*

```
// **Input:** Tensor $\mathcal{A}$ of $d$ dimensions and side lengths
// $(n_1, \ldots, n_d)$ output tensor $\mathcal{B}$, side lengths of the
// excluded box $\mathbf{k} = (k_1, \ldots, k_d)$, $k_i \le n_i$ for all
// $i = 1, 2, \ldots, d$.
// **Output:** Tensor $\mathcal{B}$ with size and dimensions
// matching $\mathcal{A}$ containing the excluded sum.
$\mathcal{A}' = \mathcal{A}, \mathcal{A}_p = \mathcal{A}, \mathcal{A}_s = \mathcal{A}$    // Prefix and suffix temp
for $i = 1$ to $d$    // Current dimension-reduction step
    // **PREFIX STEP**
    // At this point, $\mathcal{A}_p$ should hold prefixes up to
    // dimension $i - 1$.
    $\mathcal{A}' = \mathcal{A}_p$
    // Save the input to the suffix step
    $\mathcal{A}_s = \mathcal{A}_p$
    // Do prefix sum along dimension $i$
    PREFIX-ALONG-DIM($\mathcal{A}', i - 1$)
    // Save prefix up to dimension $i$ in $\mathcal{A}_p$
    $\mathcal{A}_p = \mathcal{A}'$
    // Do included sum on dimensions $[i + 1, d]$
    for $j = i + 1$ to $d$
        BDBS-ALONG-DIM($\mathcal{A}', i - 1, j, \mathbf{k}$)
    // Add into result
    ADD-CONTRIBUTION($\mathcal{A}', \mathcal{B}, i, -1$)
    // **SUFFIX STEP**
    // Start with the prefix up until dimension
    // $i - 1$
    $\mathcal{A}' = \mathcal{A}_s$
    // Do suffix sum along dimension $i$
    SUFFIX-ALONG-DIM($\mathcal{A}', i - 1$)
    // Do included sum on dimensions $[i + 1, d]$
    for $j = i + 1$ to $d$
        BDBS-ALONG-DIM($\mathcal{A}', i - 1, j, \mathbf{k}$)
    // Add into result
    ADD-CONTRIBUTION($\mathcal{A}', \mathcal{B}, i - 1, k_i$)
```

Figure C-6: Pseudocode for the box-complement algorithm with parameters filled in. For the ith dimension-reduction step, the copy of temporaries only needs to copy the last $d-i+1$ dimensions due to the dimension reduction.

C.4 Additional experimental data

The data in this appendix was generated with the experimental setup described in Section 12.6.

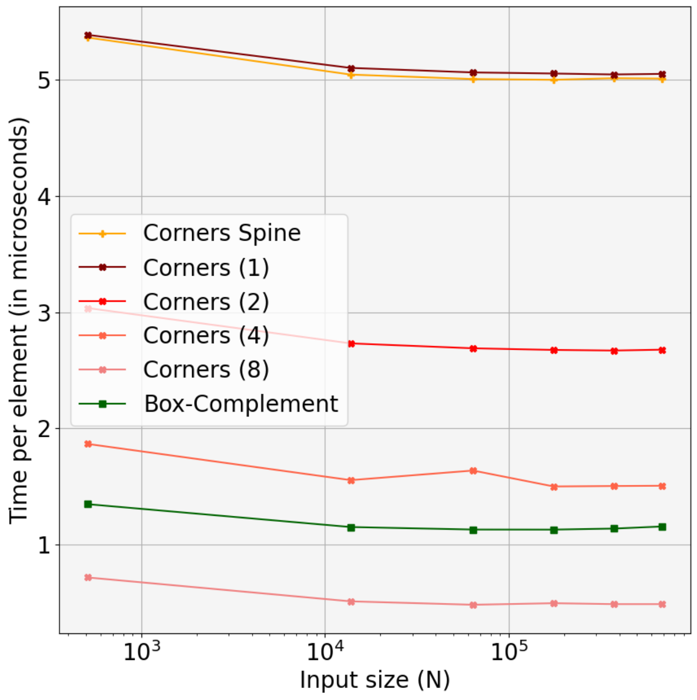


Figure C-7: Time per element of algorithms for strong excluded sums in 3D.

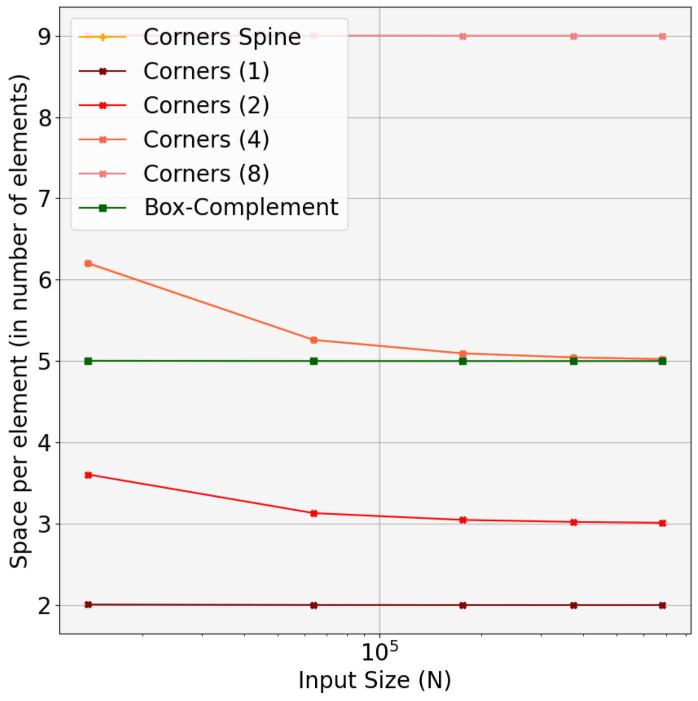


Figure C-8: Space per element of algorithms for strong excluded sums in 3D.

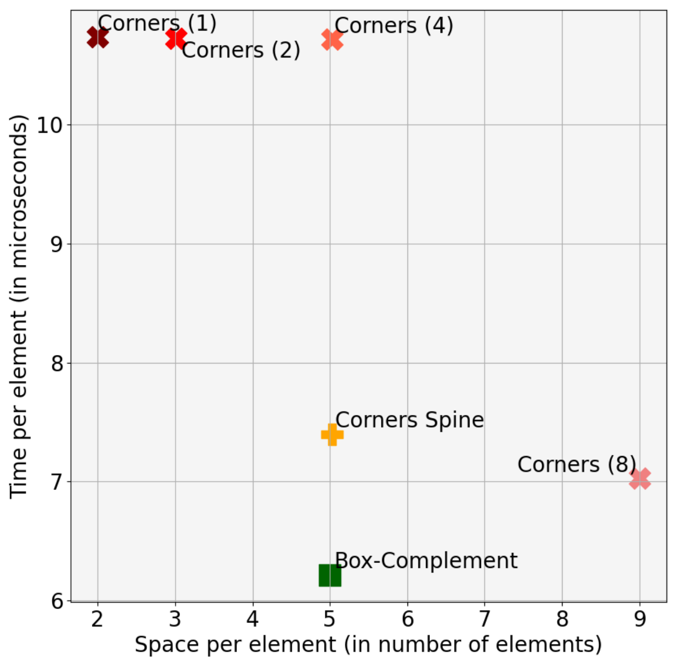


Figure C-9: Space and time per element of the corners and box-complement algorithms in 3 dimensions, with an artificial slowdown added to each numeric addition (or $\oplus$) that dominates the runtime.

Bibliography

[1] Ittai Abraham, James Aspnes, and Jian Yuan. Skip B-trees. In *OPODIS*, pages 366–380, 2006. 113

[2] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. Parallel batch-dynamic graph connectivity. In *SPAA*, page 381–392, 2019. 57

[3] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988. 19, 65, 113, 161, 164, 198, 258

[4] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. Green paging and parallel paging. In *SPAA*, page 493–495, 2020. 132

[5] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. Tight bounds for parallel paging and green paging. In *SODA*, 2021. 132

[6] Peter Ahrens, Helen Xu, and Nicholas Schiefer Schiefer. A fill estimation algorithm for sparse matrices and tensors in blocked formats. In *IPDPS*, 2018. 15, 83

[7] Peter J. Ahrens. A Parallel Fill Estimation Algorithm for Sparse Matrices and Tensors in Blocked Formats. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2019. 83

[8] Susanne Albers, Lene M. Favrholdt, and Oliver Giel. On paging with locality of reference. In *STOC*, pages 258–267, 2002. 29, 134, 145, 146

[9] Susanne Albers, Lene M. Favrholdt, and Oliver Giel. On paging with locality of reference. *Journal of Computer and System Sciences*, 70(2005):145–175, 2005. 131

[10] Amazon. Amazon web services. `https://aws.amazon.com/`, 2020. 18, 48, 76, 250

[11] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *VLDB*, 11(6):691–704, 2018. 57

[12] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *ISC*, pages 379–393, 2001. 112

[13] Spyros Angelopoulos, Reza Dorrigiv, and Alejandro López-Ortiz. On the separation and equivalence of paging strategies. In *SODA*, pages 229–237, 2007. 131, 134, 137, 138, 139

[14] Spyros Angelopoulos, Reza Dorrigiv, and Alejandro López-Ortiz. List update with locality of reference. In *LATIN*, pages 399–410, 2008. 133, 134, 137

[15] Spyros Angelopoulos, Marc P. Renault, and Pascal Schweitzer. Stochastic dominance and the bijective ratio of online algorithms. *Algorithmica*, 82(5):1101–1135, 2020. 137

[16] Spyros Angelopoulos and Pascal Schweitzer. Paging and list update under bijective analysis. In *SODA*, pages 1136–1145, 2009. 133, 134, 137, 138, 139, 146, 147, 148

[17] Spyros Angelopoulos and Pascal Schweitzer. Paging and list update under bijective analysis. *Journal of the ACM*, 60(2):1–18, 2013. 133

[18] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *IPDPS*, pages 22–31, 2016. 49, 76

[19] Lars Arge. External memory geometric data structures. Summer School on Massive Data Sets, 2002. 20

[20] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. 112

[21] Lars Arge, David Eppstein, and Michael T. Goodrich. Skip-webs: Efficient distributed data structures for multi-dimensional data sets. In *PODC*, pages 69–76, 2005. 112

[22] James Aspnes. Competitive analysis of distributed algorithms. In *Online Algorithms*, pages 118–146. Springer, 1998. 150

[23] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, 2007. 112

[24] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2008. 20

[25] Ariful Azad, Georgios A. Pavlopoulos, Christos A. Ouzounis, Nikos C. Kyrpides, and Aydın Buluç. HipMCL: A high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic Acids Research*, 46(6):e33, 2018. 49

[26] Brett W. Bader and Tamara G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, January 2008. 86, 92

[27] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. 49, 76

[28] Ilya Baran, Erik D. Demaine, and Mihai Pǎtraşcu. Subquadratic algorithms for 3SUM. *Algorithmica*, 50(4):584–596, 2008. 217

[29] Rémi Bardenet and Odalric-Ambrym Maillard. Concentration inequalities for sampling without replacement. *Bernoulli*, 21(3):1361–1385, August 2015. 104

[30] Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The Habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, page 735–736, 2009. 20

[31] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Application-controlled paging for a shared cache. *SIAM Journal on Computing*, 29(4):1290–1303, 2000. 150

[32] Rakesh D. Barve and Jeffrey S. Vitter. External memory algorithms with dynamically changing memory allocations. Technical report, Duke University, 1998. 161, 199

[33] Rakesh D. Barve and Jeffrey S. Vitter. A theoretical framework for memory-adaptive algorithms. In *FOCS*, pages 273–284, 1999. 161, 199

[34] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. 20, 112

[35] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015. 49

[36] Rick Beatson and Leslie Greengard. A short course on fast multipole methods. *Wavelets, Multilevel Methods and Elliptic PDEs*, pages 1–37, 1997. 234

[37] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. 132

[38] Shai Ben-David and Allan Borodin. A new measure for the study of on-line algorithms. *Algorithmica*, 11(1):73–91, 1994. 133

[39] Michael A. Bender, Jonathan W. Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In *PODS*, pages 289–302, 2016. 113, 114, 116

[40] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Dongdong Ge, Simai He, Haodong Hu, John Iacono, and Alejandro López-Ortiz. The cost of cache-oblivious searching. *Algorithmica*, 61(2):463–505, 2011. 199

[41] Michael A. Bender, Rezaul A. Chowdhury, Rathish Das, Rob Johnson, William Kuszmaul, Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Helen Xu. Closing the gap between cache-oblivious and cache-adaptive analysis. In *SPAA*, page 63–73, 2020. 15, 30, 159

[42] Michael A Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. Small refinements to the DAM can have big consequences for data-structure design. In *SPAA*, pages 265–274, 2019. 20, 46

[43] Michael A. Bender, Erik D. Demaine, Roozbeh Ebrahimi, Jeremy T. Fineman, Rob Johnson, Andrea Lincoln, Jayson Lynch, and Samuel McCauley. Cache-adaptive analysis. In *SPAA*, pages 135–144, 2016. 30, 159, 161, 164, 165, 166, 191, 199, 200, 202, 203, 206, 207, 208, 214, 270

[44] Michael A. Bender, Erik D. Demaine, and Martín Farach-Colton. Cache-oblivious B-trees. In *FOCS*, pages 399–409, 2000. 26, 36, 39, 59, 60, 63, 65, 69

[45] Michael A. Bender, Roozbeh Ebrahimi, Jeremy T. Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. Cache-adaptive algorithms. In *SODA*, pages 958–971, 2014. 30, 161, 164, 165, 166, 167, 168, 199, 200, 202, 203, 214, 278

[46] Michael A. Bender, Martín Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *SPAA*, pages 81–92, 2007. 112, 113, 114, 199

[47] Michael A. Bender, Martín Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to B$^\varepsilon$-trees and write-optimization. *:login; magazine*, 40(5):22–28, October 2015. 112, 114, 116

[48] Michael A. Bender, Martín Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty,

Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012. 112

[49] Michael A. Bender, Martín Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A Phillips, and Helen Xu. Write-optimized skip lists. In *PODS*, pages 69–78, 2017. 15, 111

[50] Michael A. Bender, Martín Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *PODS*, pages 233–242, 2006. 199

[51] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *SPAA*, pages 228–237, 2005. 61, 113

[52] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems*, 32(4):26, 2007. 65, 260

[53] Michael A. Bender, Bradley C. Kuszmaul, and William Kuszmaul. Linear probing revisited: Tombstones mark the demise of primary clustering. In *FOCS*, 2021. 255

[54] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, page 53–64, 2010. 21

[55] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In *OOPSLA*, page 81–96, 2009. 21

[56] Jonathan W. Berry, Matthew Oster, Cynthia A. Phillips, Steven Plimpton, and Timothy M. Shead. Maintaining connected components for infinite graph streams. In *BIGMINE*, pages 95–102, 2013. 51, 58

[57] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *IEEE Transactions on Parallel and Distributed Systems*, November 2021. 56, 57

[58] Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. Listing triangles. In *ICALP*, pages 223–234, 2014. 162

[59] Pierre Blanchard, Nicholas J. Higham, and Theo Mary. A class of fast and accurate summation algorithms. *SIAM Journal on Scientific Computing*, 42(3):A1541–A1557, 2020. 220, 221

[60] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. An experimental analysis of a compact graph representation. In *ALENEX*, 2004. 259

[61] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990. 31, 66, 74, 90, 220, 225, 241

[62] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996. 21

[63] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA*, pages 501–510, 2008. 199

[64] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*, pages 181–192, 2012. 21

[65] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 189–199, 2010. 21

[66] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. *Journal of the ACM*, 67(5):1–27, 2020. 21

[67] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996. 21, 91

[68] Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *HotPar*, page 4, 2009. 21

[69] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *WWW*, pages 595–602, 2004. 49, 76

[70] Joan Boyar, Faith Ellen, and Kim S. Larsen. Randomized distributed online algorithms against adaptive offline adversaries. *Information Processing Letters*, 161:105973, 2020. 150

[71] Joan Boyar, Lene M. Favrholdt, and Kim S. Larsen. The relative worst order ratio applied to paging. In *SODA*, pages 718–727, 2005. 133

[72] Joan Boyar, Lene M. Favrholdt, and Kim S. Larsen. Relative worst-order analysis: A survey. In *Adventures Between Lower Bounds and Higher Altitudes*, pages 216–230. Springer, 2018. 133

[73] Joan Boyar, Kim S. Larsen, and Morten N. Nielsen. The accommodating function: A generalization of the competitive ratio. *SIAM Journal on Computing*, 31(1):233–258, 2001. 133

[74] Derek Bradley and Gerhard Roth. Adaptive thresholding using the integral image. *Journal of Graphics Tools*, 12(2):13–21, 2007. 234, 236

[75] Gerth Stølting Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and J. Ian Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *SODA*, pages 1448–1456, 2010. 112, 113, 114

[76] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, pages 546–554, 2003. 20, 112, 113, 114, 116, 121

[77] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *STOC*, pages 307–315, 2003. 199

[78] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics*, 12:1–23, 2008. 199

[79] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dimitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, 2013. 57

[80] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing memory to meet multiclass workload response time goals. In *VLDB*, pages 328–341, 1993. 161, 199

[81] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On external memory graph traversal. In *SODA*, pages 859–860, 2000. 112, 113, 114

[82] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA*, pages 233–244, 2009. 85, 86, 110

[83] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *HPEC*, pages 1–7, 2018. 34, 57, 60

[84] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. *International Journal of High Performance Computing Applications*, 21(4):467–484, November 2007. 87, 107, 110

[85] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *CloudDB*, pages 1–8. ACM, 2012. 57

[86] Paul Callahan, Michael T. Goodrich, and Kumar Ramaiyer. Topology B-trees and their applications. In *WADS*, pages 381–392, 1995. 113

[87] Justus A. Calvin, Cannada A. Lewis, and Edward F. Valeev. Scalable task-based algorithm for multiplication of block-rank-sparse matrices. In *IA^3*, pages 1–8, 2015. 25, 86

[88] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *USTC*, 1994. 150

[89] George C. Caragea, Fuat Keceli, Alexandros Tzannes, and Uzi Vishkin. General-purpose vs. GPU: Comparison of many-cores on irregular workloads. In *HotPar*, 2010. 19

[90] Carlos Carvalho. The gap between processor and memory speeds. In *ICCA*, pages 27–34, 2002. 13

[91] Cascade lake. Available at https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html. 18

[92] Salvatore Catanese, Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Extraction and analysis of facebook friendship relations. In *Computational Social Networks: Mining and Visualization*, pages 291–324. Springer, 2012. 51

[93] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004. 49, 76

[94] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. *SIGARCH Computure Architecture News*, 34(2):264–276, May 2006. 30, 160

[95] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing*, 5(3):1–39, 2019. 26, 34

[96] Hongwei Cheng, William Y. Crutchfield, Zydrunas Gimbutas, Leslie F. Greengard, J. Frank Ethridge, Jingfang Huang, Vladimir Rokhlin, Norman Yarvin, and Junsheng Zhao. A wideband fast multipole method for the Helmholtz equation in three dimensions. *Journal of Computational Physics*, 216(1):300–325, 2006. 234, 238

[97] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012. 57

[98] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM SIGPLAN Notices*, 45(5):115, May 2010. 87

[99] Rezaul A. Chowdhury, Hai-Son Le, and Vijaya Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(3):495–510, 2010. 199

[100] Rezaul A. Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. In *SODA*, pages 591–600, 2006. 162

[101] Rezaul A. Chowdhury and Vijaya Ramachandran. The cache-oblivious Gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, 47(4):878–919, 2010. 162

[102] Rezaul A. Chowdhury, Muhibur Rasheed, Donald Keidel, Maysam Moussalem, Arthur Olson, Michel Sanner, and Chandrajit Bajaj. Protein-protein docking with F2Dock 2.0 and GB-Rerank. *PLoS One*, 8(3):e51307, 2013. 199

[103] Marek Chrobak and John Noga. LRU is better than FIFO. *Algorithmica*, 23(2):180–185, 1999. 134

[104] Valentina Ciriani, Paolo Ferragina, Fabrizio Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *FOCS*, pages 219–227, 2002. 113

[105] Ronald Coifman, Vladimir Rokhlin, and Stephen Wandzura. The fast multipole method for the wave equation: A pedestrian prescription. *IEEE Antennas and Propagation Magazine*, 35(3):7–12, 1993. 234

[106] Richard Cole and Vijaya Ramachandran. Efficient resource oblivious algorithms for multicores with false sharing. In *IPDPS*, pages 201–214, 2012. 199

[107] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990. 200, 216

[108] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. 21, 25, 36, 39, 41, 49, 60, 221, 252

[109] cppreference. std::inclusive_scan. Available at `https://en.cppreference.com/w/cpp/algorithm/inclusive_scan`, 2020. 220

[110] Franklin C. Crow. Summed-area tables for texture mapping. In *SIGGRAPH*, pages 207–212, 1984. 234, 236, 237

[111] Eric Darve. The fast multipole method: numerical implementation. *Journal of Computational Physics*, 160(1):195–240, 2000. 234

[112] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *PLDI*, pages 752–768, 2018. 56

[113] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, November 2011. 107

[114] Dean De Leo and Peter Boncz. Fast concurrent reads and updates with PMAs. In *GRADES-NDA*, 2019. 61

[115] Jeff Dean. Software engineering advice from building large-scale distributed systems. Available at `http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/people/jeff/stanford-295-talk.pdf`, 2009. 18, 24

[116] Erik D. Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002. 198

[117] Erik D. Demaine. Lecture 8: Ordered files and cache-oblivious priority queues (MIT 6.851), March 2012. 261

[118] Erik. D. Demaine, Martin. L. Demaine, Alan Edelman, Charles E. Leiserson, and Per-Olof Persson. Building blocks and excluded sums. *SIAM News*, 38(4):1–5, 2005. 31, 220, 235, 236, 237, 287

[119] Erik D. Demaine, Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Virginia Vassilevska Williams. Fine-grained I/O complexity via reductions: New lower bounds, faster algorithms, and a time hierarchy. In *ITCS*, pages 34:1–34:23, 2018. 162, 255

[120] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: Standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2008. 214

[121] Roman Dementiev and Peter Sanders. Asynchronous parallel disk sorting. In *SPAA*, pages 138–148, 2003. 215

[122] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968. 30, 134, 160

[123] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84, 1980. 30, 160

[124] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, July 2005. 13

[125] Luc Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, 2(3):597–609, August 1992. 112

[126] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *SPAA*, pages 293–304, 2017. 37, 62

[127] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *SPAA*, pages 393–404, 2018. 37, 62

[128] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *PLDI*, pages 918–934, 2019. 14, 26, 33, 34, 35, 36, 37, 45, 46, 48, 49, 53, 56, 57, 59, 60, 61, 62, 76

[129] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing*, 8(1):1–70, 2021. 21

[130] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *SODA*, pages 1300–1319, 2020. 57

[131] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. In *APOCS*, pages 129–143, 2021. 57, 58

[132] Dave Dice, Virendra J. Marathe, and Nir Shavit. Brief announcement: Persistent unfairness arising from cache residency imbalance. In *SPAA*, pages 82–83, 2014. 160

[133] Jack J. Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *International Journal of High Performance Computing Applications*, 17(2):125–131, May 2003. 109

[134] Jack J. Dongarra and Danny C. Sorensen. A portable environment for developing parallel FORTRAN programs. *Parallel Computing*, 5(1-2):175–186, 1987. 20

[135] Reza Dorrigiv. *Alternative Measures for the Analysis of Online Algorithms*. PhD thesis, Cheriton School of Computer Science, University of Waterloo, 2010. 134, 137

[136] Reza Dorrigiv and Alejandro López-Ortiz. A survey of performance measures for on-line algorithms. *SIGACT News*, 36:67–81, 2005. 133

[137] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. Weaver: A high-performance, transactional graph database based on refinable timestamps. *VLDB*, 9(11):852–863, 2016. 57

[138] David Durfee, John Peebles, Richard Peng, and Anup B. Rao. Determinant-preserving sparsification of SDDM matrices with applications to counting and sampling spanning trees. In *FOCS*, pages 926–937, 2017. 162

[139] David Ediger, Robert McColl, Jason Riedy, and David A. Bader. Stinger: High performance data structure for streaming graphs. In *HPEC*, pages 1–5, 2012. 34, 35, 56, 57, 60

[140] Paul Erdös and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959. 76

[141] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM Computer Communication Review*, 29(4):251–262, 1999. 25, 26, 34

[142] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103. Springer, 1996. 18

[143] Alexandra Fedorova, Margo I. Seltzer, and Michael D. Smith. Cache-fair thread scheduling for multicore processors. Technical Report TR-17-06, Harvard University, 2006. 132

[144] Guoyao Feng, Xiao Meng, and Khaled Ammar. DISTINGER: A distributed graph data structure for massive dynamic graph processing. In *BigData*, pages 1814–1822, 2015. 34, 57, 60

[145] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA*, page 1–11, 1997. 21

[146] Paolo Ferragina and Fabrizio Luccio. Batch dynamic algorithms for two graph problems. In *PARLE*, pages 713–724, 1994. 57

[147] Esteban Feuerstein and Alejandro Strejilevich de Loma. On-line multi-threaded paging. *Algorithmica*, 32(1):36–60, January 2002. 150

[148] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972. 24

[149] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59, 2004. 112, 113

[150] Sean Fraser. Computing Included and Excluded Sums Using Parallel Prefix. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2020. 221, 229

[151] Sean Fraser, Helen Xu, and Charles E. Leiserson. Work-efficient parallel algorithms for accurate floating-point prefix sums. In *HPEC*, pages 1–7, 2020. 16, 219

[152] Matteo Frigo and Steven G Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. 199

[153] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298, 1999. 30, 161, 162, 198, 199, 270, 272

[154] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):1–22, jan 2012. 30, 161, 270

[155] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Notices*, 33(5):212–223, May 1998. 20, 22, 23, 24

[156] Matteo Frigo and Volker Strumpen. Cache-oblivious stencil computations. In *ICS*, pages 361–366, 2005. 162

[157] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *GRADES*, pages 1–6, 2014. 34

[158] François Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303, 2014. 200, 216

[159] Ian P. Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for sat. In *AAAI*, pages 28–33, 1993. 256

[160] GMP FFT multiplication. Available at `https://gmplib.org/manual/FFT-Multiplication.html`. Accessed: 2018-02-01. 206

[161] Daniel Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662*, 2010. 113, 116

[162] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008. 256

[163] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012. 56

[164] Michael T. Goodrich and Roberto Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing, Aug. 2007. US Patent 7,257,711. 112

[165] Goetz Graefe. Write-optimized B-trees. In *Proceedings of the VLDB Endowment*, volume 30, pages 672–683, 2004. 112

[166] Goetz Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, March 2006. 112

[167] Goetz Graefe. A new memory-adaptive external merge sort. Private communication, July 2013. 161, 199

[168] Oded Green and David A. Bader. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *HPEC*, pages 1–6, 2016. 34, 57, 60

[169] Oded Green, Robert McColl, and David A. Bader. A fast algorithm for streaming betweenness centrality. In *PASSAT/SocialCom*, pages 11–20, 2012. 58

[170] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987. 234, 235

[171] Nail A. Gumerov and Ramani Duraiswami. *Fast Multipole Methods for the Helmholtz Equation in Three Dimensions*. Elsevier, 2005. 234, 238

[172] Mark Harris. How to access global memory efficiently in CUDA C/C++ kernels. NVIDIA Developer Blog, January 2013. 19

[173] Mark Harris. Using shared memory in CUDA C/C++. NVIDIA Developer Blog, January 2013. 19

[174] Mark Harris, Shubhabrata Sengupta, and John Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 39(39):851–876, Aug. 2007. 220, 231

[175] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Ordering heuristics for parallel graph coloring. In *SPAA*, page 166–177, 2014. 21

[176] Avinatan Hassidim. Cache replacement policies for multicore processors. In *ICS*, pages 501–509, 2010. 135, 137, 149, 150, 199

[177] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, page 145–156, 2010. 21

[178] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24(3):547–555, September 2005. 31, 220, 234

[179] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *OPODIS*, 2006. 113

[180] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. *SIROCCO*, pages 124–138, 2007. 112

[181] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2nd edition, 2020. 22

[182] Nicholas J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993. 31, 220, 221, 225, 227, 235

[183] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002. 220, 227

[184] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986. 220

[185] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, March 1963. 102

[186] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *STOC*, pages 326–333, 1981. 198

[187] Daniel Horn. Stream reduction operations for GPGPU applications. *GPU Gems*, 2, January 2005. 31, 220

[188] Eun-Jin Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, EECS Department, University of California, Berkeley, June 2000. 87, 88

[189] Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *ICCS*, pages 127–136, 2001. 84, 87, 95

[190] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. SPARSITY: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004. 87

[191] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available at http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf. 37, 48, 62, 76, 221, 222, 226

[192] Intel Corporation. Intel Intrinsics Guide. Available at https://software.intel.com/sites/landingpage/IntrinsicsGuide/, 2020. 229

[193] Internet live stats. Available at https://www.internetlivestats.com/one-second/#tweets-band. Accessed: 2021-02-01. 51

[194] Sandy Irani. Page replacement with multi-size pages and applications to web caching. In *STOC*, pages 701–710, 1997. 199

[195] Sandy Irani. Competitive analysis of paging. In *Online Algorithms: The State of the Art*, pages 52–73. Springer Berlin Heidelberg, 1998. 29

[196] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *ICALP*, pages 417–431, 1981. 26, 36, 39, 59, 60, 63

[197] Anand Iyer, Li Erran Li, and Ion Stoica. Celliq: Real-time cellular network analytics at scale. In *NSDI*, pages 309–322, 2015. 57

[198] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *GRADES*, pages 1–6, 2016. 57

[199] Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *PODC*, pages 131–140, 2009. 112

[200] JAVA BigInteger Class. Available at https://raw.githubusercontent.com/tbuktu/bigint/master/src/main/java/java/math/BigInteger.java. Accessed: 2018-02-01. 206

[201] Chris Jermaine, Anindya Datta, and Edward Omiecinski. A novel index supporting high volume data warehouse insertion. *Proceedings of the VLDB Endowment*, pages 235–246, 1999. 112, 114

[202] William Kahan. Further remarks on reducing truncation errors. *Communications of the ACM*, 8(1):40,48, 1965. 220, 227

[203] Tim Kaler. *Programming Technologies for Engineering Quality Multicore Software*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2020. 20, 21, 255

[204] Tim Kaler, William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. *ACM Transactions on Parallel Computing*, 3(1), July 2016. 21, 255

[205] Tim Kaler, Brian Wheatman, and Sarah Wooders. High-throughput image alignment for connectomics using frugal snap judgments: Poster. In *PPOPP*, page 433–434, 2019.

[206] Tim Kaler, Brian Wheatman, and Sarah Wooders. High-throughput image alignment for connectomics using frugal snap judgments. In *HPEC*, pages 1–9, 2020. 255

[207] Shahin Kamali and Alejandro López-Ortiz. A survey of algorithms and models for list update. In *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 251–266. Springer Berlin Heidelberg, 2013. 29

[208] Shahin Kamali and Helen Xu. Beyond worst-case analysis of multicore caching strategies. *arXiv preprint arXiv:2011.02046*, 2020. 131

[209] Shahin Kamali and Helen Xu. Multicore paging algorithms cannot be competitive. In *SPAA*, page 547–549, 2020. 15, 29, 132, 133, 137, 153

[210] Shahin Kamali and Helen Xu. Beyond worst-case analysis of multicore caching strategies. In *APOCS*, pages 1–15, 2021. 15, 29, 131

[211] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. A comparative study of blocking storage methods for sparse matrices on multicore architectures. In *CSE*, pages 247–256, 2009. 86, 93

[212] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Exploring the effect of block shapes on the performance of sparse kernels. In *IPDPS*, pages 1–8, May 2009. 27

[213] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Perfomance models for blocked sparse matrix-vector multiplication kernels. In *ICPP*, pages 356–364, 2009. 87

[214] Anna R. Karlin, Steven J. Phillips, and Prabhakar Raghavan. Markov paging. In *FOCS*, pages 208–217, 1992. 133

[215] Elaye Karstadt and Oded Schwartz. Matrix multiplication, a little faster. In *SPAA*, pages 101–110, 2017. 217

[216] Anil Kumar Katti and Vijaya Ramachandran. Competitive cache replacement strategies for shared cache environments. In *IPDPS*, pages 215–226, 2012. 132, 134, 137, 150, 199

[217] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *SC*, pages 323–334, 1992. 255

[218] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. In *SIGMOD*, pages 1149–1164, 2017. 57

[219] Peter Kirschenhofer and Helmut Prodinger. The path length of random skip lists. *Acta Informatica*, 31(8):775–792, 1994. 112

[220] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77:1–77:29, October 2017. 92, 93, 104, 109

[221] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 100(8):786–793, 1973. 220, 225

[222] Dennis Komm. *An Introduction to Online Computation*. Springer, 2016. 133

[223] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with Java STM. In *MULTIPROG*, 2010. 112

[224] Elias Koutsoupias and Christos H. Papadimitriou. Beyond competitive analysis. *SIAM Journal on Computing*, 30(1):300–317, 2000. 133

[225] Piyush Kumar. Cache oblivious algorithms. In *Algorithms for Memory Hierarchies*, pages 193–212. Springer Verlag, 2003. 198

[226] Pradeep Kumar and H. Howie Huang. G-store: High-performance graph store for trillion-edge processing. In *SC*, pages 830–841, 2016. 57

[227] Pradeep Kumar and H. Howie Huang. GraphOne: A data store for real-time analytics on evolving graphs. In *FAST*, pages 249–263, 2019. 57

[228] Rasmus Kyng and Sushant Sachdeva. Approximate Gaussian elimination for Laplacians: Fast, sparse, and simple. In *FOCS*, pages 573–582, 2016. 162

[229] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012. 34, 56, 60

[230] Richard E. Ladner, Ray Fortna, and Bao-hoang Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. *Experimental Algorithmics*, pages 78–92, 2002. 199

[231] Monica S. Lam and Martin C. Rinard. Coarse-grain parallel programming in Jade. In *PPoPP*, pages 94–105, 1991. 20

[232] Daniel Langr, Ivan Šimeček, and Tomáš Dytrych. Block iterators for sparse matrices. In *FedCSIS*, pages 695–704, October 2016. 88

[233] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, page 145–156, 2000. 18

[234] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, December 2002. 48, 76, 226, 250

[235] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004. 48, 76, 226, 250

[236] Quoc-Thai V Le. How Intel Advanced Vector Extensions 2 improves performance on server applications. Available at `https://software.intel.com/content/www/us/en/develop/ articles/how-intel-avx2-improves-performance-on-server-applications.html?language=en`, 2014. 229

[237] Doug Lea. A Java fork/join framework. In *JAVA*, pages 36–43, 2000. 20

[238] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. 21

[239] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, SPAA '10, page 303–314, 2010. 21, 255

[240] Dean De Leo and Peter A. Boncz. Teseo and the analysis of structural dynamic graphs. *VLDB*, 14(6):1053–1066, 2021. 40

[241] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. Available at `http://snap.stanford.edu/data`, June 2014. 49, 76

[242] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[243] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. *ACM SIGPLAN Notices*, 48(6):117–126, June 2013. 87

[244] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. Tree indexing on flash disks. In *ICDE*, pages 1303–1306, 2009. 112

[245] libstdc++: stl_algo.h source file. Available at https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01347.html. Accessed: 2018-02-13. 215

[246] Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Helen Xu. Cache-adaptive exploration: Experimental results and scan-hiding for adaptivity. In *SPAA*, pages 213–222, 2018. 15, 31, 201

[247] Hang Liu and H. Howie Huang. Enterprise: breadth-first graph traversal on GPUs. In *SC*, pages 1–12, 2015. 34

[248] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. A compiler framework for extracting superword level parallelism. In *PLDI*, page 347–358, 2012. 18

[249] Quanquan C. Liu. *Scalable and Efficient Graph Algorithms and Analysis Techniques for Modern Machines*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2021. 21, 25

[250] Alejandro López-Ortiz and Alejandro Salinger. Minimizing cache usage in paging. In *WAOA*, pages 145–158, 2012. 149, 199

[251] Alejandro López-Ortiz and Alejandro Salinger. Paging for multi-core shared caches. In *ITCS*, pages 113–127, 2012. 29, 132, 133, 134, 135, 136, 137, 139, 149, 153, 154, 199

[252] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010. 56

[253] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007. 57

[254] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *ICDE*, pages 363–374, 2015. 34, 56, 57, 60

[255] Devavret Makkar, David A. Bader, and Oded Green. Exact and parallel triangle counting in dynamic graphs. In *HiPC*, pages 2–12, 2017. 34, 58

[256] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010. 56

[257] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990. 132, 134, 154

[258] MySQL 5.7 Reference Manual. Chapter 14: The InnoDB storage engine. Available at `https://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html`. Accessed: 2021-11-24. 20

[259] William B. March and George Biros. Far-field compression for fast kernel summation methods in high dimensions. *Applied and Computational Harmonic Analysis*, 43(1):39–75, 2017. 234

[260] Paul A. Martin. Method and apparatus for implementing a lock-free skip list that supports concurrent accesses, December 2007. US Patent 7,308,448. 113

[261] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998. 226

[262] Alexander Matveev, Yaron Meirovitch, Hayk Saribekyan, Wiktor Jakubiuk, Tim Kaler, Gergely Odor, David Budden, Aleksandar Zlateski, and Nir Shavit. A multicore path to connectomics-on-demand. In *PPoPP*, page 267–281, 2017.

[263] Robert McColl, Oded Green, and David A. Bader. A new parallel algorithm for connected components in dynamic graphs. In *HiPC*, pages 246–255, 2013. 34, 58

[264] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. A performance evaluation of open source graph databases. In *PPAA*, pages 11–18, 2014. 57

[265] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys*, 48(2):1–39, 2015. 56

[266] Amanda McPherson. A conversation with Chris Mason on BTRfs. Available at `https://www.linuxfoundation.org/blog/a-conversation-with-chris-mason-on-btrfs/`, 2009. Accessed: 2021-11-24. 20

[267] Paul Menage. cgroups. Available at `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`. 160

[268] Charith Mendis and Saman Amarasinghe. Goslp: Globally optimized superword level parallelism framework. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), oct 2018. 18

[269] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. Web data commons - hyperlink graphs. Available at `http://webdatacommons.org/hyperlinkgraph/`, 2021.

[270] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the web – analyzed on different aggregation levels. *The Journal of Web Science*, 1(1):33–47, 2015.

[271] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002. 113

[272] Richard T. Mills. *Dynamic Adaptation to CPU and Memory Load in Scientific Applications*. PhD thesis, Department of Computer Science, The College of William and Mary, 2004. 161, 199

[273] Richard T. Mills, Andreas Stathopoulos, and Dimitrios S. Nikolopoulos. Adapting to memory pressure from within scientific applications on multiprogrammed COWs. In *IPDPS*, pages 71–80, 2004. 161, 199

[274] MIT Supercloud. Available at `https://supercloud.mit.edu/`, 2020. 225

[275] mkfs.btrfs manual page. Available at `https://btrfs.wiki.kernel.org/index.php/Manpage/mkfs.btrfs`. Accessed: 2021-11-24. 20

[276] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *MICRO*, pages 1–14, 2018. 25

[277] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. Incremental, iterative data processing with timely dataflow. *Communications of the ACM*, 59(10):75–83, 2016. 57

[278] David R. Musser. Introspective sorting and selection algorithms. *Journal of Software: Practice and Experience*, 27(8):983–993, August 1997. 215

[279] Mark E.J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, 2005. 25, 26, 34

[280] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013. 56

[281] Khang T. Nguyen. Introduction to Cache Allocation Technology in the Intel®Xeon®processor E5 v4 family. Available at `https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology`. 160

[282] Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973. 113

[283] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, May 2007. 93

[284] Diego Novillo. OpenMP and automatic parallelization in gcc. *Proceedings of the GCC Developers Summit*, 2006. 18

[285] Krzysztof Nowicki and Krzysztof Onak. Dynamic graph algorithms with batch updates in the massively parallel computation model. In *SODA*, pages 2939–2958, 2021. 57

[286] Charlene O'Hanlon. A conversation with John Hennessy and David Patterson: They wrote the book on computing. *Queue*, 4(10):14–22, December 2006. 20

[287] Jesper Holm Olsen and Søren Christian Skov. Cache-Oblivious Algorithms in Practice. Master's thesis, Department of Computer Science, University of Copenhagen, 2002. 215

[288] Patrick O'Neil, Edward Cheng, Dieter Gawlic, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996. 112

[289] Oracle. Setting up your data warehouse system. Available at https://docs.oracle.com/cd/B28359_01/server.111/b28314/tdpdw_system.htm#TDPDW003. Accessed: 2021-11-24. 20

[290] Boost Organization. Boost C++ libraries: Multiprecision. Available at https://www.boost.org/doc/libs/1_66_0/libs/multiprecision/doc/html/boost_multiprecision/tut/floats/cpp_bin_float.html, 2020. 227

[291] Rotem Oshman and Nir Shavit. The SkipTrie: Low-depth concurrent search without rebalancing. In *PODC*, pages 23–32, 2013. 112

[292] Rasmus Pagh and Francesco Silvestri. The input/output complexity of triangle enumeration. In *PODS*, page 224–233, 2014. 256

[293] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In *OOPSLA*, pages 1–19, 2016. 34, 56

[294] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydı n Buluç. Terrace: A hierarchical graph container for skewed dynamic graphs. In *SIGMOD*, page 1372–1385, 2021. 14, 26, 33

[295] HweeHwa Pang, Michael J. Carey, and Miron Livny. Memory-adaptive external sorting. In *VLDB*, pages 618–629, 1993. 30, 161, 199

[296] HweeHwa Pang, Michael J. Carey, and Miron Livny. Partially preemptible hash joins. In *COMAD*, pages 59–68, 1993. 30, 161, 199

[297] Thomas Papadakis, J. Ian Munro, and Patricio V. Poblete. Analysis of the expected search cost in skip lists. In *SWAT*, pages 160–172, 1990. 112

[298] Joon-Sang Park, Michael Penner, and Viktor K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004. 162

[299] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997. 13

[300] John Paul. Teradata 13 vs Teradata 14. Available at `http://teradata-thoughts.blogspot.com/2013/10/teradata-13-vs-teradata-14_20.html`, 2013. Accessed: 2021-11-24. 20

[301] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Pearson Education, 2006. 112

[302] Enoch Peserico. Paging with dynamic memory capacity. *arXiv preprint arXiv:1304.6007*, 2013. 199

[303] Filip Piekniewski. Robustness of power laws in degree distributions for spiking neural networks. In *IJCNN*, pages 2541–2546, 2009. 26

[304] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *SC*, pages 30–es, November 1999. 86, 92, 93, 260

[305] Keshav Pingali. Locality of reference and parallel processing. In *Encyclopedia of Parallel Computing*, pages 1051–1056. Springer US, 2011. 13

[306] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. Managing large graphs on multi-cores with graph awareness. In *USENIX ATC*, pages 41–52, 2012. 57

[307] Harald Prokop. Cache-Oblivious Algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999. 162, 198

[308] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Synthesizing parallel graph programs via automated planning. In *PLDI*, pages 533–544, 2015. 56

[309] William Pugh. Concurrent maintenance of lists. Technical Report CS-TR-2222.1, Deptartment of Computer Science, University of Maryland, College Park, 1990. 113

[310] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. 112

[311] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ISCA*, pages 381–391, 2007. 132

[312] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, pages 423–432, 2006. 132

[313] @raffi. New tweets per second record, and how!, August 2013. Available at `https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html`. 51, 77

[314] Sanguthevar Rajasekaran, Lance Fiondella, Mohamed Ahmed, and Reda A. Ammar. *Multicore Computing: Algorithms, Architectures, and Applications*. CRC Press, 2013. 18

[315] Tim Roughgarden. Beyond worst-case analysis. *Communications of the ACM*, 62(3):88–96, 2019. 28

[316] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. *ACM SIGPLAN Notices*, 34(8):72–83, 1999. 18

[317] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003. 73

[318] Jerome H. Saltzer. A simple linear model of demand paging performance. *Communications of the ACM*, 17(4):181–186, 1974. 133, 154

[319] Tao B. Schardl. *Performance Engineering of Multicore Software: Developing a Science of Fast Code for the Post-Moore Era*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2016. 20, 21, 23, 255

[320] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. The Cilkprof scalability profiler. In *SPAA*, page 89–100, 2015. 21, 23

[321] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. *ACM SIGPLAN Notices*, 52(8):249–265, 2017. 76, 226

[322] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding recursive fork-join parallelism into LLVM's intermediate representation. *ACM Transactions on Parallel Computing*, 6(4):1–33, 2019. 20, 48, 76, 226, 250

[323] Aaron Schild, Satish Rao, and Nikhil Srivastava. Localization of electrical flows. In *SODA*, pages 1577–1584, 2018. 162

[324] Russell Sears, Mark Callaghan, and Eric A. Brewer. Rose: Compressed, log-structured replication. *VLDB*, 1(1):526–537, 2008. 112

[325] Russell Sears and Raghu Ramakrishnan. bLSM: A general purpose log structured merge tree. In *SIGMOD*, pages 217–228, 2012. 112

[326] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995. 162

[327] Dipanjan Sengupta and Shuaiwen Leon Song. Evograph: On-the-fly efficient mining of evolving graphs on GPU. In *SC*, pages 97–119, 2017. 57

[328] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *EuroPar*, pages 319–333, 2016. 57

[329] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *VLDB*, 11(1):107–120, 2017. 57

[330] Nikita Shamgunov. The MemSQL in-memory database system. In *IMDM*, 2014. 112

[331] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, pages 505–516, 2013. 57

[332] Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *IPDPS*, pages 263–268, 2000. 112

[333] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *FOCS*, pages 605–614, 1999. 162

[334] Julian Shun. *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable*. Association for Computing Machinery and Morgan & Claypool, 2017. 20, 21, 22, 25

[335] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013. 26, 33, 36, 37, 45, 47, 48, 49, 56, 60, 62, 76

[336] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The Problem Based Benchmark Suite. In *SPAA*, page 68–70, 2012. 220, 225

[337] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *DCC*, pages 403–412, 2015. 37, 62, 76, 255

[338] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W Mahoney. Parallel local graph clustering. *VLDB*, 9(12):1041–1052, 2016. 37, 62

[339] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *ICDE*, pages 149–160, 2015. 21

[340] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts Essentials*. John Wiley & Sons, Inc., 2014. 134, 144

[341] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find with applications to incremental graph connectivity. In *EuroPar*, pages 561–573, 2016. 58

[342] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985. 15, 29, 123, 132, 137

[343] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982. 24

[344] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *IA^3*, pages 1–7, 2015. 92

[345] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *IPDPS*, pages 61–70, May 2015. 88, 93

[346] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, USA, 1997. 77

[347] Snapchat. Snap kit sdk 1.4, Jan 2020. Available at https://kit.snapchat.com/news/snap-kit-sdk-1-4. 51

[348] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004. 30

[349] Guy L. Steele Jr., Alexander T. Garthwaite, Paul A. Martin, Nir Shavit, Mark S. Moir, and David L. Detlefs. Lock-free implementation of concurrent shared object with dynamic node allocation and distinguishing pointer value, November 30 2004. US Patent 6,826,757. 113

[350] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, 1992. 132

[351] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969. 162, 200, 203, 216

[352] Alejandro Strejilevich de Loma. New results on fair multi threaded paging. *Electronic Journal of SADIO*, 1(1):21–36, 1998. 150

[353] Jaspal Subhlok, James M. Stichnoth, and Thomas O'Hallaron, David R.and Gross. Exploiting task and data parallelism on a multicomputer. In *PPoPP*, pages 13–22, 1993. 20

[354] G Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004. 132

[355] Xian-He Sun and Lionel M. Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, 1993. 22

[356] Xiaobai Sun and Nikos P. Pitsianis. A matrix version of the fast multipole method. *SIAM Review*, 43(2):289–300, 2001. 234

[357] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *SAC*, pages 1438–1445, 2004. 113

[358] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards large-scale graph stream processing platform. In *WWW*, pages 1321–1326, 2014. 57

[359] Yuan Tang, Rezaul A. Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *SPAA*, pages 117–128, 2011. 162

[360] Ernesto Tapia. A note on the computation of high-dimensional integral images. *Pattern Recognition Letters*, 32(2):197–201, 2011. 236, 237

[361] William F. Tinney and John W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967. 25, 35, 60, 73

[362] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994. 22

[363] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. Batch-parallel euler tour trees. In *ALENEX*, pages 92–106, 2019. 57

[364] John D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, pages 214–222, 1995. 113

[365] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000. 220

[366] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The Paralax infrastructure: Automatic parallelization with a helping hand. In *PACT*, pages 389–399, 2010. 18

[367] Jeffrey S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001. 20

[368] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. *ACM SIGOPS Operating Systems Review*, 51(2):237–251, 2017. 57

[369] Richard W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, EECS Department, University of California, Berkeley, January 2004. 25, 84, 86, 87, 88, 107

[370] Richard W. Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In *HotPar*. USENIX Association, 2010.

[371] Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521–530, January 2005. 15, 27, 84, 87

[372] Richard W. Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *SC*, pages 1–35, 2002. 27, 84, 86, 87

[373] Richard W. Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *HPCC*, pages 807–816, 2005. 27

[374] Jue Wang, Xiangyu Dong, Yuan Xie, and Norman P. Jouppi. Endurance-aware cache line management for non-volatile caches. *ACM Transactions on Architecture and Code Optimization*, 11(1):1–25, 2014. 160

[375] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single PC. In *USENIX ATC*, pages 387–401, 2015. 56

[376] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *SIGMOD*, pages 1813–1828, 2016. 49, 76

[377] Brian Wheatman. Image Alignment and Dynamic Graph Analytics: Two Case Studies of How Managing Data Movement Can Make (Parallel) Code Run Fast. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2019. 255

[378] Brian Wheatman and Helen Xu. Packed Compressed Sparse Row: A dynamic graph representation. In *HPEC*, 2018. 26, 40, 59, 60, 61, 64, 72, 257

[379] Brian Wheatman and Helen Xu. A parallel Packed Memory Array to store dynamic graphs. In *ALENEX*, pages 31–45, 2021. 14, 22, 24, 26, 27, 40, 59

[380] GCC Wiki. Automatic parallelization in GCC. Available at `https://gcc.gnu.org/wiki/AutoParInGCC#:~:text=Automatic%20parallelization%20distributes%20sequential%20code,constructs%20using%20the%20gomp%20library`, May 2012. 18

[381] David Williams. *Probability with Martingales.* Cambridge University Press, 1991. 175, 274

[382] Samuel Williams, Leonid Oliker, Richard W. Vuduc, John Shalf, Katherine A. Yelick, and James S. Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, March 2009. 85, 87

[383] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*, pages 887–898, 2012. 200, 216

[384] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *FOCS*, pages 645–654, 2010. 162

[385] Martin Winter, Rhaleb Zayer, and Markus Steinberger. Autonomous, independent management of dynamic graphs on GPUs. In *HPEC*, pages 1–7, 2017. 57

[386] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 174–183, 2009. 132

[387] Wenpu Xing and Ali Ghorbani. Weighted pagerank algorithm. In *Communication Networks and Services Research, 2004. Proceedings. Second Annual Conference on*, pages 305–314. IEEE, 2004. 37

[388] Helen Xu. Fill Estimation for Blocked Sparse Matrices and Tensors. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2018. 83, 107, 109

[389] Helen Xu, Sean Fraser, and Charles E. Leiserson. Multidimensional included and excluded sums. In *ACDA*, 2021. 16, 233

[390] Helen Xu, Sean Fraser, and Charles E. Leiserson. Multidimensional included and excluded sums. *arXiv preprint arXiv:2106.00124*, 2021. 233

[391] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017. 56

[392] Ke Yi. Dynamic indexability and the optimality of B-trees. *Journal of the ACM*, 59(4):21:1–21:19, August 2012. 112, 113, 114

[393] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-oblivious mesh layouts. *ACM Transactions on Graphic*, 24(3):886–893, 2005. 199

[394] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA*, pages 93–104, 2007. 199

[395] Neal E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, June 1994. 133

[396] Neal E. Young. Bounding the diffuse adversary. In *SODA*, pages 420–425, 1998. 133

[397] Neal E. Young. On-line paging against adversarially biased random inputs. *Journal of Algorithms*, 37(1):218–235, 2000. 133

[398] Neal E. Young. Online file caching. *Algorithmica*, 33(3):371–383, 2002. 133, 199

[399] A. N. Yzelman. Generalised vectorisation for sparse matrix-vector multiplication. In *IA^3*, pages 6:1–6:8, 2015. 86, 93, 110

[400] Hansjörg Zeller and Jim Gray. An adaptive hash join algorithm for multiuser environments. In *VLDB*, pages 186–197, 1990. 161, 199

[401] Weiye Zhang and Per-Äke Larson. A memory-adaptive sort (MASORT) for database systems. In *CASCON*, 1996. 161, 199

[402] Weiye Zhang and Per-Äke Larson. Dynamic memory adjustment for external mergesort. In *VLDB*, pages 376–385, 1997. 161, 199

[403] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *BigData*, pages 293–302, 2017. 34

[404] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015. 34

[405] Gernot Ziegler. Summed area ripmaps. GPU Technology Conference (talk). Available at `https://on-demand.gputechconf.com/gtc/2012/presentations/S0096-Summed-Area-Ripmaps.pdf`, 2012. 220, 235

[406] Uri Zwick. All pairs shortest paths in weighted directed graphs-exact and almost exact algorithms. In *FOCS*, pages 310–319, 1998. 162