Algorithms Incorporating Concurrency and Caching

by

Jeremy T. Fineman

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

ARCHIVES

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SEP 30 2009

LIBRARIES

© Massachusetts Institute of Technology 2009. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
September 4, 2009

Certified by ....
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by ....
Professor Terry P. Orlando
Chairman, Department Committee on Graduate Students
Electrical Engineering and Computer Science

Algorithms Incorporating Concurrency and Caching
by
Jeremy T. Fineman

Submitted to the Department of Electrical Engineering and Computer Science
on September 4, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis describes provably good algorithms for modern large-scale computer systems, including today's multicores. Designing efficient algorithms for these systems involves overcoming many challenges, including concurrency (dealing with parallel accesses to the same data) and caching (achieving good memory performance.)

This thesis includes two parallel algorithms that focus on testing for atomicity violations in a parallel fork-join program. These algorithms augment a parallel program with a data structure that answers queries about the program's structure, on the fly. Specifically, one data structure, called ***SP-ordered-bags***, maintains the series-parallel relationships among threads, which is vital for uncovering race conditions (bugs) in the program. Another data structure, called ***XConflict***, aids in detecting conflicts in a transactional-memory system with nested parallel transactions. For a program with work T_1 and span T_∞, maintaining either data structure adds an overhead of PT_∞ to the running time of the parallel program when executed on P processors using an efficient scheduler, yielding a total runtime of $O(T_1/P + PT_\infty)$. For each of these data structures, queries can be answered in $O(1)$ time.

This thesis also introduces the ***compressed sparse rows (CSB)*** storage format for sparse matrices, which allows both Ax and $A^T x$ to be computed efficiently in parallel, where A is an $n \times n$ sparse matrix with $nnz \geq n$ nonzeros and x is a dense n-vector. The parallel multiplication algorithm uses $\Theta(nnz)$ work and $\Theta(\sqrt{n}/\log n)$ span, yielding a parallelism of $\Theta(nnz/\sqrt{n}\log n)$, which is amply high for virtually any large matrix.

Also addressing concurrency, this thesis considers two scheduling problems. The first scheduling problem, motivated by transactional memory, considers randomized backoff when jobs have different lengths. I give an analysis showing that binary exponential backoff achieves makespan $V2^{\Theta(\sqrt{\log n})}$ with high probability, where V is the total length of all n contending jobs. This bound is significantly larger than when jobs are all the same size. A variant of exponential backoff, however, achieves makespan of $O(V \log V)$ with high probability. I also present the ***size-hashed backoff*** protocol, specifically designed for jobs having different lengths, that achieves makespan $O(V \log^3 \log V)$ with high probability.

The second scheduling problem considers scheduling n unit-length jobs on m unrelated machines, where each job may fail probabilistically. Specifically, an input consists of a set of n jobs, a directed acyclic graph G describing the precedence constraints among jobs, and a failure probability q_{ij} for each job j and machine i. The goal is to find a schedule that minimizes the expected makespan. I give an $O(\log\log(\min\{m,n\}))$-approximation for the case of independent jobs (when there are no precedence constraints) and an $O(\log(n+m)\log\log(\min\{m,n\}))$-approximation algorithm when precedence constraints form disjoint chains. This chain algorithm can be extended into one that supports precedence constraints that are trees, which worsens the approximation by another $\log(n)$ factor.

To address caching, this thesis includes several new variants of cache-oblivious dynamic dictionaries. A cache-oblivious dictionary fills the same niche as a classic B-tree, but it does so without tuning for particular memory parameters. Thus, cache-oblivious dictionaries optimize for all levels of a multilevel hierarchy and are more portable than traditional B-trees. I describe how to add concurrency to several previously existing cache-oblivious dictionaries. I also describe two new data structures that achieve significantly cheaper insertions with a small overhead on searches. The ***cache-oblivious lookahead array (COLA)*** supports insertions/deletions and searches in $O((1/B)\log N)$ and $O(\log N)$ memory transfers, respectively, where B is the block size, M is the memory size, and N is the number of elements in the data structure. The ***xDict*** supports these operations in $O((1/\varepsilon B^{1-\varepsilon})\log_B(N/M))$ and $O((1/\varepsilon)\log_B(N/M))$ memory transfers, respectively, where $0 < \varepsilon < 1$ is a tunable parameter.

Also on caching, this thesis answers the question: what is the worst possible page-replacement strategy? The goal of this whimsical chapter is to devise an online strategy that achieves the highest possible fraction of page faults / cache misses as compared to the worst offline strategy. I show that there is no deterministic strategy that is competitive with the worst offline. I also give a randomized strategy based on the most recently used heuristic and show that it is the worst possible page-replacement policy. On a more serious note, I also show that direct mapping is, in some sense, a worst possible page-replacement policy.

Finally, this thesis includes a new algorithm, following a new approach, for the problem of maintaining a topological ordering of a dag as edges are dynamically inserted. The main result included here is an $O(n^2 \log n)$ algorithm for maintaining a topological ordering in the presence of up to $m \leq n(n-1)/2$ edge insertions. In contrast, the previously best algorithm has a total running time of $O(\min\{m^{3/2}, n^{5/2}\})$. Although these algorithms are not parallel and do not exhibit particularly good locality, some of the data structural techniques employed in my solution are similar to others in this thesis.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

I would first like to thank Charles E. Leiserson, my advisor. When Charles announced to my entire group that I would be graduating this year, that was the first I had heard of it. At that point, I thought he had tired of having me around. Now that we're nearing the end, however, and Charles is requesting changes to this document, I realize that he must still want me around. In all seriousness, though, I would like to thank Charles for putting up with me for all these years and for providing guidance in both research and my career. And for putting his signature on the cover sheet.

I would like to thank all of my collaborators, including Kunal Agrawal, Michael A. Bender, Gerth Stølting Brodal, Aydın Buluç, Martin Farach-Colton, Christopher Y. Crutchfield, Erik D. Demaine, Zoran Dzunic, Yonatan Fogel, Matteo Frigo, John R. Gilbert, Seth Gilbert, John Iacono, David R. Karger, Bradley C. Kuszmaul, Stefan Langerman, Charles E. Leiserson, J. Ian Munro, Jelani Nelson, Jacob H. Scott, Jim Sukha, and Robert E. Tarjan. They were invaluable in developing the solutions described later in this thesis.

Michael deserves particular attention and gratitude. He worked with me on a large fraction of the research that went into this thesis, and he has been a good friend and mentor throughout.

I would also like to thank all the other members of the SuperTech group not already mentioned, including Angelina Lee and Edya Ladan Mozes, as well as the members of the theory group. I would also like to thank all my friends for contributing to many dinner parties, dessert parties, game nights, and other activities that kept me sane.

I extend particular gratitude to the delicious pig who became the pulled pork that got cooked in the pot that burned me. Were it not for you, pig, I would not have had those wonderful blisters on my fingers for a few weeks of peak thesis-typing time.

Last but not least, I would like to thank Artessa for doing her best to curtail my thesis efforts. Without her, perhaps I would have finished months earlier.

This work was supported in part by NSF Grants ACI-0324974, CCF-0541209, CCF-0621511, CNS-0305606, CNS-0540248, CNS-0615215, CSR-AES 0615215, OCI-0324974, the Singapore MIT Alliance, Google Inc., and the Intel Corporation. Any opinions, findings, conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of these funding organizations.

Contents

1 Introduction **1**

2 Series-Parallel Maintenance **9**

2.1 The SP-order algorithm 13
2.2 The SP-fast-bags algorithm 18
2.3 A model of parallel programs 23
2.4 The SP-ordered-bags algorithm 25
2.5 The global tier of SP-ordered-bags 29
2.6 The local tier of SP-ordered-bags 30
2.7 Correctness of SP-ordered-bags 33
2.8 Performance analysis 38
2.9 Related work on SP-maintenance 41
2.10 Concluding remarks 42

3 Nested Parallelism in Transactional Memory **43**

3.1 CWSTM framework 46
3.2 CWSTM semantics 47
3.3 A naive TM 50
3.4 CWSTM overview 51
3.5 CWSTM conflict detection 53
3.6 Trace maintenance 60
3.7 Highest active transaction 61
3.8 Supertraces 62
3.9 Ancestor queries 63
3.10 Performance claims 65
3.11 Concluding remarks 66

4 Scheduling Under Uncertainty **69**

4.1 Preliminaries 71
4.2 Independent jobs 74
4.3 Jobs with chain-like precedence constraints 80
4.4 Jobs with tree-like precedence constraints 85
4.5 Stochastic scheduling 85
4.6 Concluding remarks 86

5 **Contention Resolution with Heterogeneous Job Sizes** **89**
5.1 Traditional backoff with variable-sized jobs 91
5.2 Size-hashed backoff 94
5.3 Concluding remarks 101

6 **Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks** **103**
6.1 Conventional storage formats 105
6.2 The CSB storage format 107
6.3 Matrix-vector multiplication using CSB 109
6.4 Analysis 113
6.5 Experimental design 115
6.6 Experimental results 119
6.7 CSB Discussion 122

7 **Introduction to Cache-Oblivious Dictionaries** **127**
7.1 Previous cache-oblivious dictionaries 129

8 **Concurrent Cache-Oblivious B-Trees** **131**
8.1 The concurrent-cache model 132
8.2 Exponential CO B-tree 133
8.3 Packed-memory CO B-tree 137
8.4 Lock-free CO B-tree 143
8.5 Concurrent CO B-tree discussion 146

9 **The Lookahead Array: A Cache-Oblivious Dictionary With Faster Insertions** **147**
9.1 Cache-oblivious lookahead array (COLA) 148
9.2 COLA: experimental results 153

10 **The xDict: A CO B-Tree With Optimal Update/Query Tradeoff** **157**
10.1 Introducing the x-box 157
10.2 Sizing an x-box 160
10.3 Operating an x-box 160
10.4 Building a dictionary out of x-boxes 165
10.5 Final notes 166

11 **The Worst Page-Replacement Policy** **167**
11.1 Lower bounds 169
11.2 Most-recently used 170
11.3 Direct mapping 173
11.4 Concluding remarks 174

12 **Incremental Topological Ordering** **175**
12.1 Basic strategy 177
12.2 Algorithm for dense graphs 178
12.3 Analysis 182
12.4 Concluding remarks 184

13 **Conclusions** **187**

List of Figures

1-1 The machine model. 1
1-2 A chart showing the locality and parallelism exhibited by algorithms in each chapter of this thesis. 2

2-1 A dag representing a multithreaded computation. 10
2-2 The parse tree for the computation dag. 10
2-3 Comparison of serial, SP-maintenance algorithms 12
2-4 An English and Hebrew ordering. 14
2-5 The SP-order algorithm written in serial pseudocode. 16
2-6 An illustration of how SP-order operates at an S-node. 17
2-7 An illustration of how SP-order operates at a P-node. 17
2-8 A canonical Cilk parse tree. 19
2-9 The SP-bags algorithm described in terms of procedures. 20
2-10 The SP-bags algorithm written in serial pseudocode. 21
2-11 The SP-ordered-bags algorithm written in parallel pseudocode. 28
2-12 The SP-Precedes procedure for the SP-ordered-bags algorithm. 29
2-13 The split of a trace around a P-node in terms of a canonical Cilk parse tree. 31
2-14 An ordering of the new traces resulting from a steal. 32
2-15 The local-tier SP-fast-bags algorithm written in parallel pseudocode. 34

3-1 A simple fork-join program that does several parallel increments. 44
3-2 The series-parallel dag for the sample program. 44
3-3 The same program with the addition of transactions. 45
3-4 A legend for computation-tree figures. 47
3-5 A computation tree for the program given by Figure 3-1. 48
3-6 Pseudocode for a conflict-detection query. 52
3-7 Pseudocode instrumenting a memory access. 54
3-8 Code for cleanup up an aborted transaction. 55
3-9 Traces of a computation tree (a) before and (b) after a `steal` action. 56
3-10 Pseudocode for the XConflict algorithm. 57
3-11 The definition of arrows used to represent paths in Figures 3-12, 3-13 and 3-14. . . . 58
3-12 The three possible scenarios in which X is the nearest transactional ancestor of B that belongs to an active trace. 58
3-13 The possible scenarios in which the highest active transaction Y in U_x is an ancestor of X. 59
3-14 The scenario in which the highest active transaction Y in U_x is not an ancestor of X. 59

4-1 Approximation ratios for SUU. 71

5-1 Pseudocode for size-hashed backoff. 97

6-1 Average performance of Ax and $A^{\mathsf{T}}x$ operations on our benchmark suite. 104
6-2 Parallel matrix-vector multiplication using CSR. 105
6-3 Serial matrix-transpose-vector multiplication for CSR. 106
6-4 Pseudocode for CSB matrix-vector multiplication. 110
6-5 Pseudocode for subblockrow-vector multiplication used for CSB. 111
6-6 Pseudocode for the subblock-vector product used for CSB. 112
6-7 The effect of CSB block size on SpMv performance. 117
6-8 Structural information on the sparse matrices used in our experiments. 118
6-9 CSB_SPMV performance on Opteron (smaller matrices). 120
6-10 CSB_SPMV_T performance on Opteron (smaller matrices). 121
6-11 CSB_SPMV performance on Opteron (larger matrices). 122
6-12 CSB_SPMV_T performance on Opteron (larger matrices). 123
6-13 Average speedup of CSB on smaller test matrices. 123
6-14 Parallelism test for CSB_SPMV. 124
6-15 CSB_SPMV performance on Harpertown. 124
6-16 CSB_SPMV performance on Nehalem. 125
6-17 Serial performance comparison of SpMV for CSB and CSR. 125
6-18 Serial performance comparison of SpMV_T for CSB and CSR. 126
6-19 Performance comparison of parallel CSB_SPMV with Star-P. 126

7-1 Summary of known cache-oblivious dictionaries. 129
7-2 The van Emde Boas layout. 129

8-1 An exponential CO B-tree. 133
8-2 An example of an insert into an exponential CO B-tree. 134
8-3 The modified van Emde Boas layout of a node in the exponential CO B-tree. 134
8-4 A packed-memory CO B-tree. 138
8-5 An insertion into the packed-memory array. 140

9-1 Comparison of the COLA and B-tree for random insertions. 154
9-2 Comparison of the COLA and B-tree for sorted insertions. 154
9-3 Comparison of the COLA and B-tree for random searches. 155
9-4 Comparison of various insertion patterns into the COLA. 155

10-1 The recursive structure of an x-box. 158
10-2 Buffer sizes for an x-box. 158

12-1 An example of a clique, for which the simple algorithm performs $\Theta(k^2)$ work. . . . 178
12-2 Pseudocode for updating labels. 180

Chapter 1

Introduction

This thesis explores algorithms and data structures motivated by and designed for modern large-scale computer systems, such as multicore computers. Achieving good performance on these machines entails designing algorithms that achieve good parallelism, thereby utilizing multiple processors effectively, and that have good locality, exploiting the cache to attain good memory performance. The goal of this thesis is to study various techniques for parallel and cache-efficient (specifically, so-called cache-oblivious) algorithms. Many of the solutions described in this thesis are data structural in nature, and many of the solutions exploit amortization in the analysis. Much of this thesis is really a study of amortized data structures in the context of practical problems motivated by modern computers.

Recently, most hardware vendors have abandoned their singlecore chips in favor of multicore designs. Intel, for example, announced in 2005 that their focus would be on multicore chips. Although most common machines today have between 2 and 8 cores, a general belief is that the number of cores will double every silicon generation (see, for example, [28]). Hence, highly parallel algorithms are necessary to exploit the full power of these machines.

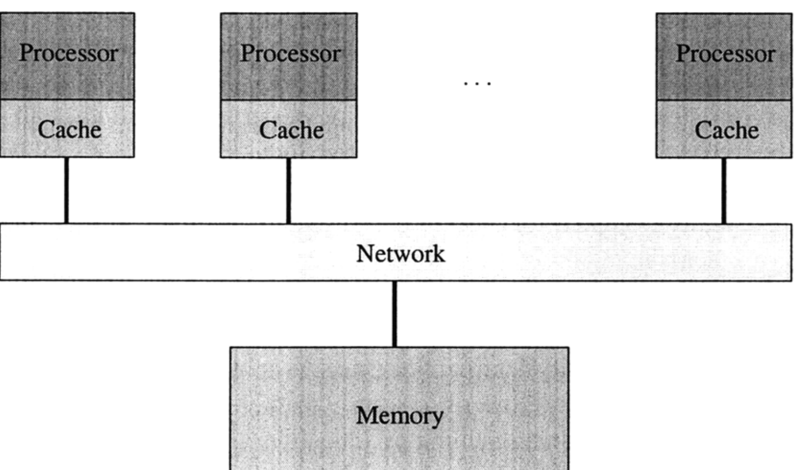


Figure 1-1: The machine model used for most of the algorithms in this thesis. Each processor is connected to a private fast memory, called the cache. Processors communicate through the shared main memory.

Throughout most of this thesis, I consider algorithms designed for a shared-memory machine, such as the one shown in Figure 1-1 (see also [117, Chapter 4]). As shown, each identical processor is connected to a private fast memory, called the ***cache***. Processors are connected to main memory

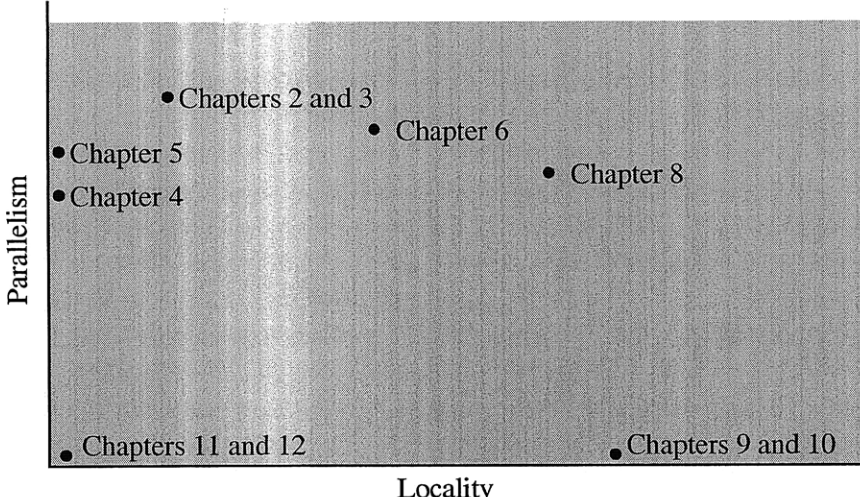


Figure 1-2: A chart showing how much locality and parallelism are expressed by each of the chapters in this thesis. The points here are subjective and should be taken as a guideline only. The x-axis here plots locality, with points towards the right exploiting the cache more effectively. The y-axis indicates concurrency or parallelism with higher points representing algorithms with more parallelism.

through some network, which is abstracted away throughout this thesis. Accessing the cache is much cheaper than accessing the shared memory, and thus good algorithms should exhibit locality to exploit the cache. In general, my algorithms are designed for a ***symmetric multiprocessor (SMP)***, in which all memory locations (not currently residing in a cache) have the same access cost by each processor. My performance analyses depend on the uniform access cost, but algorithmic correctness does not. These algorithms thus apply to nonuniform access machines (NUMA), and the presence of locality in the algorithms suggests good performance there as well. Moreover, although Figure 1-1 shows only a single level of cache and a slow memory, many of the algorithms presented in this thesis generalize effectively to multilevel memory or cache hierarchies. In fact, one of the reasons I consider cache-oblivious algorithms is that cache-oblivious algorithms automatically generalize to multilevel memory hierarchies.

Figure 1-1 is consistent with common multicore processors today, such as all present Intel and AMD chips. Other types of multicores exist, such as the Sony-Toshiba-IBM Cell Broadband Engine with heterogeneous processors, or the Cray MTA with no caches. Although the algorithms in this thesis are not specifically designed for those types of machines, many of the techniques presented here may still be applicable.

Figure 1-2 shows a subjective chart of the locality and parallelism expressed by the algorithms within this thesis. The goal is to achieve algorithms in the upper-right, like those in Chapters 6 and 8, achieving both good locality and parallelism. As a step in this direction, I also consider algorithms that achieve either parallelism or locality. Chapters 2–6 and Chapter 8 address concurrency, whereas Chapters 7–11 address caching and locality. Chapters 2–5 are more infrastructural in nature, whereas Chapters 6 and 12 address algorithms at the library or application level.

Contributions

One of the key difficulties in designing parallel algorithms is coping with concurrent accesses to shared data. In some cases (e.g., the algorithm in Chapter 6), parallel tasks may be scheduled such that no two contending tasks execute at the same time. In this case, however, the parallel algorithm

must be structured to facilitate this scheduling, which is not a trivial task.

Much of this thesis focuses on more complex interactions among processors. When two parallel tasks operate on the same data, some mechanism for handling critical sections is required, in the form of semaphores [83] (or locks), or, more recently, transactional memory [119]. Dealing with critical sections adds more difficulty to the implementation of parallel codes. If the critical sections are too large, then parallelism decreases. If the critical sections are smaller, then the program becomes more complex. Complex parallel codes are prone to bugs like race conditions and deadlock. Moreover, it is not generally well-understood how to design algorithms that achieve provably good performance bounds while using locks or transactions.

This thesis makes two major contributions with respect to concurrency control. First, I provide algorithms that help the programmer to write correct parallel codes using locks or transactions, either by uncovering race bugs in programs with locks (Chapter 2) or by supporting conflict detection in a highly parallel transactional-memory system (Chapter 3). Second, these are themselves examples of parallel algorithms that achieve provably good performance despite their use of locks internally. Moreover, the solutions here include a novel approach for amortizing the cost of locking against the span of the computation.

Another goal of this thesis is to construct algorithms with good locality, thereby efficiently using memory caches on the machine. For large data sets, the cost of memory accesses may dominate the total runtime of an algorithm. Fortunately, memory is transferred in large chunks, and so by organizing data cleverly (to introduce locality), we can exploit the cache and drive down the total cost of the memory accesses. Most notably, my work on caching focuses on cache-oblivious dictionaries in Chapters 7–10. Many of the parallel algorithms included in this thesis, moreover, are structured to achieve some level of locality (notably, the algorithm in Chapter 6, but also to a lesser degree those algorithms in Chapters 2 and 3).

As a whole, this thesis aims to study many techniques for designing provably good parallel algorithms, also including some that use locks, and cache-oblivious algorithms.

Chapters 2 and 3 addresses mutual exclusion for fork-join programs[1], providing tools that make it easier to guarantee program correctness. Both of these chapters describe solutions that are intended to be integrated into a parallel runtime system, essentially augmenting a parallel program with additional data structures to answer questions about the program as it is running. These algorithms essentially operate on input fork-join programs and augment them to address certain problems. Both of the parallel algorithms described in these chapters maintain a shared data structure that is accessed concurrently using locks for mutual exclusion. The analysis includes a novel technique for amortizing the cost of locking against the "span" of the underlying program. For a program with ***work*** (serial running time) T_1 and ***span*** (also called critical-path length or depth) T_∞, maintaining the centralized data structures in either of these chapters adds an overhead of $O(PT_\infty)$ to the running time of the parallel program when executed on P processors using an efficient scheduler. In contrast, the program without augmentation can be executed in $O(T_1/P + T_\infty)$ time. For both of these data structures, relevant queries can be answered in $O(1)$ time without acquiring any locks.

Chapter 2 address the problem of ***series-parallel maintenance***, determining whether the structure of the program allows for two particular tasks to execute in parallel. Specifically, are tasks u and v logically serial or parallel? Answering this question is a key component of race detectors tools for uncovering race bugs in the program. Chapter 2 describes three algorithms, the serial SP-order and SP-fast bags algorithms, and the parallel SP-ordered-bags algorithm. Consider a parallel program with T_1 work and T_∞ span. When augmented with SP-order or SP-fast-bags, the program

[1]Fork-join programs are a restricted class of parallel programs with well-structured parallelism. Languages such as Cilk [54], NESL [53], and libraries such as TBB [174] all provide fork-join parallelism.

can be executed in $O(T_1)$ time on a single processor while supporting queries in $O(1)$ time. Both of these algorithms can be combined to form the SP-ordered-bags algorithm. When that program is augmented with SP-ordered-bags, it can be executed in $O(T_1/P + PT_\infty)$ worst-case time on P processors. Here, queries can be answered in $O(1)$ time, in parallel, without acquiring any locks. Chapter 2 represents joint work with Michael A. Bender, Seth Gilbert, and Charles E. Leiserson.

Chapter 3 considers conflict detection in a transactional-memory system that contains nested parallel transactions. Previous work on transactional memory (e.g., [119]) focuses on transactions that represent serial computation. In contrast, Chapter 3 considers adding transactions that contain parallel code blocks (that in turn contain nested parallel transactions) to fork-join programs. Allowing nesting within transactions exacerbates the problem of discovering conflicts among transactions and determining when one transaction must abort. Chapter 3 presents an algorithm for conflict detection in this complex setting. The key data structure, called XConflict, supports these queries and achieves the same performance bounds as SP-ordered-bags. That is, a program with T_1 work and T_∞ span can be executed in $O(T_1/P + PT_\infty)$ worst-case time on P processors while augmented to maintain the XConflict data structure. Moreover, queries about conflicts among transactions can be answered in $O(1)$ time without acquiring any locks. Chapter 3 represents joint work with Kunal Agrawal and Jim Sukha.

Chapters 4 and 5 consider scheduling problems. Specifically, Chapter 4 deals with a distributed scheduling problem, whereas Chapter 5 studies various backoff protocols in the presence of tasks that have different sizes.

Chapter 4 addresses scheduling under uncertainty, first introduced in [146], where each job has a stochastic failure rate. Specifically, an input consists of n unit-length jobs and m machines, where each job-machine pair has an independent failure probability when executing the job for unit time. This failure probability models machines crashing or taking too long to complete. There may be precedence constraints among jobs modeled by a directed acyclic graph (dag). The problem is to assign jobs to machines so as to minimize the expected ***makespan***, which is the completion time of the last job to complete. A single job may be executed on multiple machines at the same time, thereby decreasing its aggregate failure probability. This thesis includes an $O(\log\log\mu)$-approximation when all jobs are independent (no precedence constraints), $O(\log(n+m)\log\log\mu)$-approximation when precedence constraints form chains, and $O(\log(n+m)\log(n)\log\log\mu)$-approximation when constraints form trees, where $\mu = \min\{m, n\}$. These results beat the previously best approximation ratios by at least an $O((\log n)/\log\log\mu)$ factor. The solutions I present use many classic tools like LP rounding and competitive analysis. Chapter 4 represents joint work with Christopher Y. Crutchfield, Zoran Dzunic, David R. Karger, and Jacob H. Scott.

Chapter 5 considers randomized backoff, where processes/jobs make competing attempts to access a shared resource, but only one can gain control of the resource at a time. Here, I consider jobs that have nonuniform size. If an access attempt fails due to contention, then that process waits for a random amount of time (determined by the backoff protocol) before trying again. In the context of this thesis, backoff with jobs having different sizes is primarily motivated by transactional memory, where the "jobs" correspond to transactions in a program, all of which access the same memory location. This thesis presents the first theoretical analysis of randomized backoff when jobs have different sizes. I consider the case here when all jobs enter the system at the same time. Chapter 5 includes several contributions. First, the analyses show that binary exponential backoff achieves a makespan of $V2^{\Theta(\sqrt{\log n})}$ with high probability for n jobs with total length V. This bound for jobs having different sizes is significantly worse than the $\Theta(n\log n)$ bound achieved when all jobs have unit size [43]. This bound also suggests that *binary* exponential backoff is not a good choice for some transactional-memory systems. I also show that exponential backoff with a slower rate of backoff achieves $O(V\log V)$ makespan. Finally, I improve the bounds further

by providing the ***size-hashed backoff*** scheme that is designed to work with a particular class of functions. I show constructively a class of functions allowing size-hashed backoff to complete with makespan $O(V\sqrt{\log V}(\log\log V)^2)$ and prove the existence of functions that would allow size-hashed backoff to complete with makespan $O(V(\log\log V)^3)$. Chapter 5 represents joint work with Michael A. Bender and Seth Gilbert.

Chapter 6 returns to a fork-join parallel algorithm, but now at the application level instead of the runtime-system level. This chapter introduces a storage format for sparse matrices, called ***compressed sparse blocks (CSB)***, which allows both Ax and A^Tx to be computed efficiently in parallel, where A is an $n \times n$ sparse matrix with $nnz \geq n$ nonzeros and x is a dense n-vector. The parallel multiplication algorithms use $\Theta(nnz)$ work and $\Theta(\sqrt{n}\lg n)$ span, yielding a parallelism of $\Theta(nnz/\sqrt{n}\lg n)$, which is amply high for virtually an large matrix. The storage requirement for CSB is essentially the same as that for the more-standard compresses-sparse-rows (CSR) [175] format, for which computing Ax in parallel is easy but A^Tx is difficult. Benchmark results indicate that on one processor, the CSB algorithm for Ax and A^Tx run just as fast as the CSR algorithm for Ax, but the CSB algorithm also scales up linearly with processors until limited by offchip memory bandwidth. Chapter 6 represents joint work with Aydın Buluç, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson

One interesting aspect CSB is that it uses a recursive layout. Typically recursive layouts are used to exploit locality, but here the recursive layout allows for parallelism. Moreover, the multiplication algorithm introduces parallelism by duplicating shared-memory locations, thereby reducing the need for concurrency control on these locations. The analyses amortizes the cost of creating these extra memory locations against the cost of the actual multiplication, yielding a low overhead. Although I do not analyze the cache efficiency of CSB, the experimental study as well as the recursive layout suggest that CSB does achieve good spatial locality.

Chapters 8–11 address cache performance, including various cache-oblivious dynamic dictionaries in Chapters 8–10. A ***dynamic dictionary*** is a data structure that supports insertions, deletions, and predecessor queries on keyed data. Dynamic dictionaries that achieve good memory performance are the key data structures for filesystems and databases. The most famous memory-efficient dynamic dictionary is the B-tree [30, 70], which supports all three operations with $O(\log_B N)$ memory transfers, where B is the ***block size***, or number of records moved in each memory transfer. A ***cache-oblivious*** dynamic dictionary is a dictionary that achieves good performance without using the parameter B in the data structure. Several cache-oblivious dynamic dictionaries (e.g., [34, 35, 37, 41]), also called cache-oblivious B-trees, already exist. These previous cache-oblivious dictionaries, however, are all serial in nature. Chapter 7 provides a more detailed introduction to cache-oblivious dictionaries as well related work.

Chapter 8 provides cache-oblivious B-trees that support concurrent operations. These data structures build on previous versions of cache-oblivious B-tree and obtain similar performance bounds to the original data structures when operating serially. This chapter uses two main types of tools to facilitate concurrent operations. First, introducing randomization into the underlying data structures reduces the likelihood of overlapping expensive update operations. Second, the concurrent data structures make use of monotonic data movement. That is to say, the typical cache-oblivious B-trees move data in memory on update operations. The concurrent versions described in Chapter 8 aim to restrict this movement so that the address of a particular record can only increase over time, thereby facilitating concurrent operations. Chapter 8 represents joint work with Michael A. Bender, Seth Gilbert, and Bradley C. Kuszmaul.

Although the concurrent cache-oblivious B-trees are not the only parallel cache-oblivious algorithms in the literature, to my knowledge, these are the only ones that directly cope with multiple processors concurrently accessing the same data. Blelloch, Gibbons, and Simhadri [52], for exam-

ple, provide general theorems stating that if an algorithm has low span and achieves good serial cache performance, then it achieves good parallel cache performance. Their results, however, assume that caches are noninterfering. There are also many examples of parallel cache-oblivious algorithms [51, 52, 67, 68, 100], but these typically also deal with computations where the data can be partitioned across processors, and there is little contention on shared locations.

Chapters 9 and 10 include dynamic cache-oblivious dictionaries that support a better insertion bound at a small cost to searches. The cache-oblivious lookahead array, described in Chapter 9, achieves inserts in $O((\log N)/B)$ amortized memory accesses while still supporting searches in $O(\log N)$ worst-case memory transfers. That is, insertions are improved by an $O(B/\log B)$ factor whereas searches are worsened by only a $O(\log B)$ factor. The xDict, described in Chapter 10, achieves the optimal tradeoff bound (proved in [59]), namely inserts in $O((1/\varepsilon B^{1-\varepsilon})\log_B(N/M))$ and searches in $O((1/\varepsilon)\log_B(N/M))$ memory transfers. These data structures are the first to incorporate the technique of fractional cascading [64] into cache-oblivious data structures. The cache-oblivious lookahead array represents joint work with Michael A. Bender, Martin Farach-Colton, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson, and the xDict represents joint work with Gerth Stølting Brodal, Erik D. Demaine, John Iacono, Stefan Langerman, and J. Ian Munro.

Chapter 11 also addresses caching, but with a paradoxical goal. In particular, this chapter aims to devise an online page-replacement strategy that achieves the highest possible fraction of page faults (or cache misses) as compared to the (worst) offline strategy. The goal here is not entirely serious, but the analyses may still be interesting in their own right. Since the aim is to reduce locality, and this chapter does not address parallelism, Figure 1-2 shows this chapter in the lower-left of the chart. I show that randomization is required to have a strategy that is competitive with the worst offline strategy. I also give a randomized strategy based on the most recently used heuristic and prove that it is the worst possible replacement policy. I also show that direct mapping is, in some sense, the worst page-replacement policy. This last result suggests that caches should use at least some degree of associativity. Chapter 11 represents joint work with Kunal Agrawal and Michael A. Bender.

Finally, Chapter 12 addresses dynamically maintaining a topological ordering in a dag. In particular, let $G = (V, E)$ be a dag, and let $n = |V|$ and $m = |E|$. We say that a total ordering on vertices is a ***topological ordering*** if for every edge $(u, v) \in E$, we have u appearing before v in the total ordering. This chapter considers the problem of maintaining a topological ordering dynamically as the underlying graph changes. Specifically, I address the incremental problem, in which we begin with an empty n-node graph $G = (V, \emptyset)$ and edges are added one at a time. Throughout this process, a data structure maintains a topological ordering of the graph G, allowing for queries about relationships among vertices. Chapter 12 presents a new algorithm that has total cost $O(n^2 \log n)$ for maintaining the ordering across m edge insertions. At the heart of the algorithm is a new approach for maintaining the ordering. Rather than attempting to place nodes in an ordered list (as in previous approaches), we assign each node a label that induces the ordering and can be updated efficiently. When the graph is dense, this algorithm is more efficient than existing algorithms. By way of contrast, the best prior algorithms achieve only $O(\min\{m^{1.5}, n^{2.5}\})$ cost. These algorithms are all serial, and there is likely little locality. Chapter 12 represents joint work with Michael A. Bender and Seth Gilbert.

With the exception of the scheduling algorithms in Chapters 4 and 5 and the page-replacement problem in Chapter 11, all of the algorithms in this thesis incorporate data structures and amortization. These data structures have many similar themes that unify the topic areas. Some of the cache-oblivious data structures, particular the concurrent B-trees (Chapter 8) and the xDict (Chapter 10) use recursive layouts. A recursive layout also occurs in the parallel CSB algorithm in Chapter 6, but the goal in CSB is to provide parallelism, not locality.

Another theme is maintaining a total order. SP-ordered-bags (Chapter 2) as well as XConflict

(Chapter 3) have components that maintain total orders of nodes in the underlying dag. Similarly, a topological order (Chapter 12) is a total ordering. In fact, SP-order maintains two topological orderings of the underlying computation dag. The cache-oblivious dictionaries also maintain total orders by keeping data in sorted order in components of the concurrent CO B-trees (Chapter 8) and the COLA (Chapter 9) and xDict (Chapter 10). Moreover, the underlying order-maintenance data structure [79, 81] used in SP-ordered-bags and XConflict is reminiscent of the packed-memory array [41, 123] employed by many cache-oblivious dictionaries including those in Chapter 8. Both data structures aim to order data, and analysis of each uses a similar amortization argument (although the latter is more complicated). Also common to these orderings is labels. The order-maintenance data structure dynamically labels objects to permit efficient queries. In the same vein, the incremental topological ordering algorithm (Chapter 12) labels nodes in the dag to infer the topological ordering.

Chapter 2

Series-Parallel Maintenance

This chapter shows how to maintain the series-parallel (***SP***) relationships between logical strands in a multithreaded program "on the fly." That is, given two logical strands, I show how to answer the question of whether the strands may run in parallel, and I show how to answer this question while the multithreaded program is running (rather than by after-the-fact or static analysis). This problem arises as the principal data-structuring issue in dynamic race detectors [65, 84, 91, 153, 163], as a race can only occur when two strands may access the same memory in parallel. In this chapter, I show that for fork-join programming models, such as MIT's Cilk system [54, 99, 185], this data-structuring problem can be solved asymptotically optimally. I also give the first provably efficient parallel algorithm for the problem. This chapter represents joint work with Michael A. Bender, Seth Gilbert, and Charles E. Leiserson, previously appearing in [48].

Both this chapter and the next discuss algorithms that are intended to be integrated into a parallel runtime system, essentially augmenting a parallel program with additional data structures to answer questions about the program as it is running. The goal of these algorithms is to improve the parallel-programming platform, thereby moving some complexities away from the programmer. Specifically, these chapters discusses algorithms that help the programmer to deal with atomicity, coping with concurrent accesses to shared data. These algorithms are designed to take input parallel programs and augment them to address certain problems. The "series-parallel maintenance" problem addressed in this chapter is relevant to race-detection tools for verifying program correctness under concurrent accesses to data. The "conflict-detection" problem in Chapter 3 is a component for guaranteeing atomicity in a transactional-memory system.

This chapter considers programs written in dynamic multithreaded languages such as Cilk [54] or NESL [53] or multithreaded libraries like Hood [56]. In such languages, a programmer specifies dynamic parallelism using linguistic constructs such as "fork" and "join," "spawn" and "sync," or "parallel" blocks. Dynamic multithreaded languages allow the programmer to specify a program's parallel structure abstractly, that is, permitting (but not requiring) parallel in specific locations of the program. We restrict our attention further to programs that have only properly nested parallelism, i.e., one that can be represented by a series-parallel dag or parse tree, to be described shortly.

Following [91] we model the execution of a multithreaded program as a directed acyclic graph, or ***computation dag***, where nodes are either ***forks*** or ***joins*** and edges are ***strands***. Such a dag is illustrated in Figure 2-1. A fork node has a single incoming edge and multiple outgoing edges. A join node has multiple incoming edges and (at most) a single outgoing edge. Strands (edges) represent blocks of serial execution. We model the performance of our parallel algorithms in terms of work and span [72, Ch. 27]:

- The ***work*** of a computation, denoted by T_1, is the total running time of all its individual

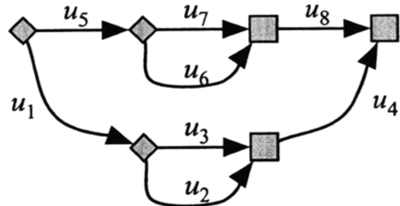


Figure 2-1: A dag representing a multithreaded computation. The edges represent strands, labeled $u_1, u_2, \ldots u_8$. The diamonds represent forks, and the squares indicate joins.

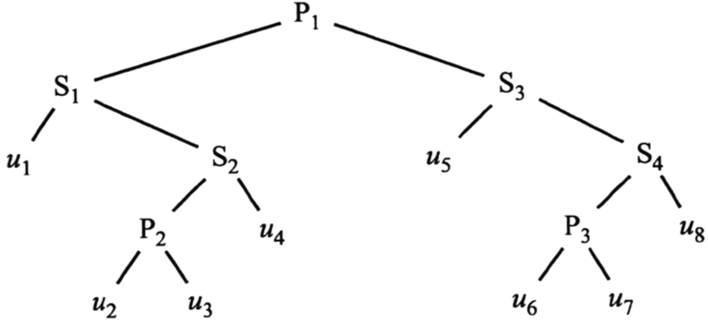


Figure 2-2: The parse tree for the computation dag shown in Figure 2-1. The leaves are the strands in the dag. The S-nodes indicate series relationships, and the P-nodes indicate parallel relationships.

strands, or equivalently, the running time on 1 processor.

- The ***span***[1] of a computation, denoted by T_∞, is the length of the longest chain of dependencies, or equivalently, its running time on an infinite number of processors assuming no scheduling overhead.

The ***parallelism*** of a computation is the ratio T_1/T_∞.

For fork-join programming models, where every fork has a corresponding join that unites the forked strands, the computation dag has a structure that can be represented efficiently by a ***series-parallel (SP) parse tree*** [91]. In the parse tree each internal node is either an ***S-node*** or a ***P-node*** and each leaf is a strand of the dag.[2] Figure 2-2 shows the parse tree corresponding to the computation dag from Figure 2-1. If two subtrees are children of the same S-node, then the parse tree indicates that (the subcomputation represented by) the left subtree must execute before (that of) the right subtree, and we say that strands in the left subtree ***logically precede*** strands in the right subtree. If two subtrees are children of the same P-node, then the parse tree indicates that the two subtrees may execute in parallel, and we say that the strands in the left subtree operate ***logically in parallel*** with those in the right subtree.

An SP parse tree can be viewed as an *a posteriori* execution of the corresponding computation dag, but our algorithms must operate while the dag, and hence the parse tree, is unfolding dynamically. The way that the parse tree unfolds depends on a ***scheduler***, which determines which strands execute where and when on a finite number of processors. A partial execution corresponds to a subtree of the parse tree that obeys the series-parallel relationships, namely, that a right subtree of an S-node cannot be present unless the corresponding left subtree has been fully ***elaborated***, or unfolded with all leaf strands executed. Both subtrees of a P-node, however, can be partially elabo-

[1]The literature also uses the terms ***depth*** [50] and ***critical-path length*** [54].

[2]We assume without loss of generality that all SP parse trees are full binary trees, that is, each internal node has exactly two children.

rated. In a language like Cilk, a serial execution unfolds the parse tree in the manner of a left-to-right walk. For example, in Figure 2-2 a serial execution executes the strands in the order of their indices.

A typical serial on-the-fly determinacy-race detector simulates the execution of the program as a left-to-right walk of the parse tree while maintaining various data structures for determining the existence of races. The type of race detector we are discussing considers a single *a posteriori* SP parse tree (as generated from the simulated execution) and reports a race whenever there exists a valid scheduling of this parse tree that contains a race condition.

The core data structure of the race detector maintains the series-parallel relationships between the currently executing strand and previously executed strands. Specifically, the race detector must determine whether the current strand is operating logically in series or in parallel with certain previously executed strands. We call a dynamic data structure that maintains the series-parallel relationship between strands an ***SP-maintenance*** data structure. The data structure supports insertion, deletion, and ***SP queries***: queries as to whether two nodes are logically in series or in parallel.

The Nondeterminator [65, 91] race detectors use a variant of Tarjan's [187] least-common-ancestor algorithm, called the ***SP-bags*** algorithm, as the basis of their SP-maintenance data structure. To determine whether a strand u_i logically precedes a strand u_j, denoted $u_i \prec u_j$, the SP-bags algorithm can be viewed intuitively as inspecting their least common ancestor $\text{lca}(u_i, u_j)$ in the parse tree to see whether it is an S-node with u_i in its left subtree. Similarly, to determine whether a strand u_i operates logically in parallel with a strand u_j, denoted $u_i \parallel u_j$, the SP-bags algorithm checks whether $\text{lca}(u_i, u_j)$ is a P-node. Observe that an SP relationship exists between any two nodes in the parse tree, not just between strands (leaves). For example, in Figure 2-2 we have $u_1 \prec u_4$, because $S_1 = \text{lca}(u_1, u_4)$ is an S-node and u_1 appears in S_1's left subtree. We also have $u_1 \parallel u_6$, because $P_1 = \text{lca}(u_1, u_6)$ is a P-node. The (serially executing) Nondeterminator race detectors perform SP-maintenance operations whenever the program being tested forks, joins, or accesses a shared-memory location. The amortized cost for each of these operations is $O(\alpha(v, v))$, where α is Tarjan's functional inverse of Ackermann's function and v is the number of shared-memory locations used by the program. As a consequence, the asymptotic running time of performing determinacy-race detection is $O(T_1 \alpha(v, v))$, where T_1 is the work of the original computation.

The SP-bags algorithm has two shortcomings. The first is that it slows the asymptotic running time by a factor of $\alpha(v, v)$. This factor is nonconstant in theory but is nevertheless close enough to constant in practice that this deficiency is minor. The second, more important, shortcoming is that the SP-bags algorithm relies heavily on the serial nature of its execution, and hence it appears difficult to parallelize.

Some early SP-maintenance algorithms use labeling schemes without centralized data structures. These labeling schemes are easy to parallelize but unfortunately are much less efficient than the SP-bags algorithm. Examples of such labeling schemes include the ***English-Hebrew*** scheme [163] and the ***offset-span*** scheme [153]. These algorithms generate labels for each strand on the fly, but once generated, the labels remain static. By comparing labels, these SP-maintenance algorithms can determine whether two strands operate logically in series or in parallel. One of the reasons for the inefficiency of these algorithms is that label lengths increase linearly with the number of forks (English-Hebrew) or with the depth of fork nesting (offset-span).

Contributions

This chapter presents two serial SP-maintenance algorithm that improve on the original SP-bags algorithm. The ***SP-order*** algorithm, which is inspired by the English-Hebrew scheme, removes the inverse Ackermann's $\alpha(v, v)$ factor from the amortized running time of SP-bags operations,

Algorithm	Space per node	Time per	
		Strand creation	Query
English-Hebrew [163]	$\Theta(f)$	$\Theta(1)$	$\Theta(f)$
Offset-Span [153]	$\Theta(d)$	$\Theta(1)$	$\Theta(d)$
SP-Bags [91]	$\Theta(1)$	$\Theta(\alpha(v,v))$*	$\Theta(\alpha(v,v))$*
SP-Fast-Bags	$\Theta(1)$	$\Theta(1)$*	$\Theta(1)$
SP-Order	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

f = number of forks in the program
d = maximum depth of nested parallelism
v = number of shared locations being monitored

Figure 2-3: Comparison of serial, SP-maintenance algorithms. An asterisk (*) indicates an amortized bound. The function α is Tarjan's functional inverse of Ackermann's function.

making all operations $O(1)$ time in the worst case. Rather than using static labels, SP-order maintains labels with an order-maintenance data structure [33, 79, 81, 192]. The ***SP-fast-bags*** algorithm also achieves $O(1)$ time for SP-maintenance operations, but the bound is amortized. SP-fast-bags directly improves the SP-bags algorithm using data-structuring techniques from Gabow and Tarjan [101]. Figure 2-3 compares the serial space and running times of SP-order and the SP-fast-bags with the other algorithms. Although SP-fast-bags only achieves an amortized bound, and hence is qualitatively dominated by SP-order, its data structure allows its bags to be "split" (which turns out to be useful in the context of our parallel algorithm).

Our parallel SP-maintenance algorithm ***SP-ordered-bags***, which is designed to run with a Cilk-like work-stealing scheduler [55, 99], is the first of its kind which is provably efficient. SP-ordered-bags consists of two tiers: a ***global tier*** based on parallelizing SP-order, and a ***local tier*** based on SP-fast-bags. Suppose that a fork-join program has T_1 work and T_∞ span. Whereas an efficient Cilk-like scheduler executes this computation in asymptotically optimal $T_P = O(T_1/P + T_\infty)$ expected time on P processors, SP-ordered-bags executes the computation in $O(T_1/P + PT_\infty)$ worst-case time on P processors while maintaining SP relationships. Thus, whereas the underlying computation achieves linear speedup when $P = O(T_1/T_\infty)$ — the parallelism dominates the number of processors — SP-ordered-bags achieves linear speedup when $P = O(\sqrt{T_1/T_\infty})$ — the square root of the parallelism dominates the number of processors.

These results represent joint work with Michael A. Bender, Seth Gilbert, and Charles E. Leiserson, some of which appeared earlier in a conference paper [48]. The conference paper described the SP-order and a less-efficient variant of the parallel SP-ordered-bags algorithm, called SP-hybrid. This thesis describes the new SP-fast-bags algorithm, as well as the improved parallel algorithm SP-ordered-bags. In particular, for a program with T_1 work, span T_∞, and n strands, the SP-hybrid algorithm runs in $O((T_1/P + PT_\infty) \lg n)$ time in expectation, whereas SP-ordered-bags trims the $\lg n$ factor and makes the bound worst case. These improved results are also given in my Master's thesis [93].

Chapter outline

The remainder of this chapter is organized as follows. I present the SP-order algorithm in Section 2.1 and the SP-fast-bags algorithm in Section 2.2. Section 2.3 describes the performance model

in the presence of locks and features of the scheduler necessary to achieve good performance for SP-ordered-bags. Section 2.4 presents an overview of the parallel SP-ordered-bags algorithm. Section 2.5 describes the organization of the global tier of SP-ordered-bags in more detail, and Section 2.6 describes the local tier. Section 2.7 provides a proof of correctness, and Section 2.8 analyzes the performance of SP-ordered-bags. Finally, Section 2.9 reviews related work.

2.1 The SP-order algorithm

This section presents the serial SP-order algorithm. I begin by discussing how an SP parse tree, provided as input to SP-order, is created. I then review the concept of an English-Hebrew ordering [163], showing that two linear orders are sufficient to capture SP relationships. I show how to maintain these linear orders on the fly using order-maintenance data structures [33, 79, 81, 192]. Finally, I give the SP-order algorithm itself and show that each SP-maintenance operation takes $O(1)$ time in the worst case. I conclude that any fork-join program running in T_1 time on a single processor can be checked on the fly for determinacy races in $O(T_1)$ time.

The input to SP-order

SP-order takes as input a fork-join multithreaded program expressed as an SP parse tree. In a real implementation, such as a race detector, the parse tree unfolds dynamically and implicitly as the multithreaded program executes, and the particular unfolding depends on how the program is scheduled on the multiprocessor computer. For ease of presentation, however, we assume that the program's SP parse tree unfolds according to a left-to-right tree walk. During this tree walk, SP-order maintains SP relationships "on the fly" in the sense that it can immediately respond to SP queries between any two executed strands. At the end of the section, we relax the assumption of left-to-right unfolding and argue that no matter how the parse tree unfolds, SP-order can maintain SP relationships on the fly.

English and Hebrew orderings

SP-order uses two total orders to determine whether strands are logically parallel, an ***English order*** and a ***Hebrew order***. In the English order, the nodes in the left subtree of a P-node precede those in the right subtree of the P-node. In the Hebrew order, the order is reversed: the nodes in the right subtree of a P-node precede those in the left. In both orders, the nodes in the left subtree of an S-node precede those in the right subtree of the S-node.

Figure 2-4 shows English and Hebrew orderings for the strands in the parse tree from Figure 2-2. Notice that if u_i belongs to the left subtree of an S-node and u_j belongs to the right subtree of the same S-node, then we have $E[u_i] < E[u_j]$ and $H[u_i] < H[u_j]$. In contrast, if u_i belongs to the left subtree of a P-node and u_j belongs to the right subtree of the same P-node, then $E[u_i] < E[u_j]$ and $H[u_i] > H[u_j]$.

The English and Hebrew orderings capture the SP relationships in the parse tree. Specifically, if one strand u_i precedes another strand u_j in both orders, then we have $u_i \prec u_j$ in the parse tree (or multithreaded dag). If u_i precedes u_j in one order but u_i follows u_j in the other, then we have $u_i \parallel u_j$. For example, in Figure 2-4, we have $u_1 \prec u_4$, because $1 = E[u_1] < E[u_4] = 4$ and $5 = H[u_1] < H[u_4] = 8$. Similarly, we can deduce that $u_1 \parallel u_6$, because $1 = E[u_1] < E[u_6] = 6$ and $5 = H[u_1] > H[u_6] = 3$. The following lemma, originally proved by Nudler and Rudolph [163], shows that this property always holds.

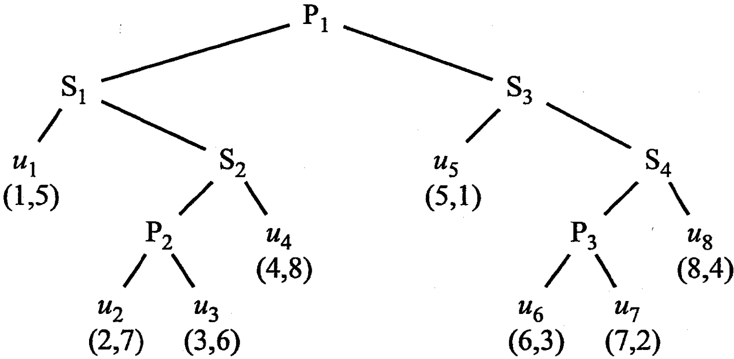


Figure 2-4: An English ordering E and a Hebrew ordering H for the strands in the parse tree from Figure 2-2. Under each strand u is an ordered pair $(E[u], H[u])$ giving its index in each of the two orders.

Lemma 2.1 *Let E be an English ordering of the strands of an SP-parse tree, and let H be a Hebrew ordering. Then, for any two strands u_i and u_j in the parse tree, we have $u_i \prec u_j$ in the parse tree if and only if $E[u_i] < E[u_j]$ and $H[u_i] < H[u_j]$.*

Proof. ($\Rightarrow$) Suppose that $u_i \prec u_j$, and let $X = \text{lca}(u_i, u_j)$. Then, X is an S-node in the parse tree, the strand u_i resides in X's left subtree, and u_j resides in X's right subtree. In both orders, the strands in the X's left subtree precede those in X's right subtree, and hence, we have $E[u_i] < E[u_j]$ and $H[u_i] < H[u_j]$.

($\Leftarrow$) Suppose that $E[u_i] < E[u_j]$ and $H[u_i] < H[u_j]$, and let $X = \text{lca}(u_i, u_j)$. Since we have $E[u_i] < E[u_j]$, strand u_i must appear in X's left subtree, and u_j must appear in X's right subtree. By definition of a Hebrew ordering, X must be an S-node, and hence we have $u_i \prec u_j$. □

We can restate Lemma 2.1 as follows.

Corollary 2.2 *Let E be an English ordering of the strands of an SP-parse tree, and let H be a Hebrew ordering. Then, for any two strands u_i and u_j in the parse tree with $E[u_i] < E[u_j]$, we have $u_i \parallel u_j$ if and only if $H[u_i] > H[u_j]$.* □

Labeling a static SP parse tree with an English-Hebrew ordering is easy enough. To compute the English ordering, perform a depth-first traversal visiting left children of both P-nodes and S-nodes before visiting right children (an ***English walk***). Assign label i to the ith strand visited. To compute the Hebrew ordering, perform a depth-first traversal visiting right children of P-nodes before visiting left children but left children of S-nodes before visiting right children (a ***Hebrew walk***). Assign labels to strands as before.

In race-detection applications, however, one must generate "on-the-fly" orderings as the parse tree unfolds. If the parse tree unfolds according to an English walk, then computing an English ordering is easy. Unfortunately, computing a Hebrew ordering on the fly during an English walk is problematic. In the Hebrew ordering the label of a strand in the left subtree of a P-node depends on the number of strands in the right subtree. In an English walk, however, this number is unknown until the right subtree has unfolded.

Nudler and Rudolph [163], who introduced English-Hebrew labeling for race detection, addressed this problem by using large strand labels. Unfortunately, the number of bits in a label in their scheme can grow linearly in the number of P-nodes in the SP parse tree. Although they give a heuristic for reducing the size of labels, manipulating large labels is the performance bottleneck in their algorithm.

The SP-order data structure

Rather than using static labels, one can employ order-maintenance data structures [33,79,81,192] to maintain the English and Hebrew orders . In order-maintenance data structures, the labels inducing the order change during the execution of the program. An order-maintenance data structure is an abstract data type that supports the following operations:

- OM-PRECEDES(L, X, Y): Return TRUE if X precedes Y in the ordering L. Both X and Y must already exist in the ordering L.
- OM-INSERT$(L, X, Y_1, Y_2, \ldots, Y_k)$: In the ordering L, insert new elements $Y_1, Y_2, \ldots, Y_k$, in that order, immediately after the existing element X.

The OM-PRECEDES operation can be supported in $O(1)$ worst-case time. The OM-INSERT operation can be supported in $O(k)$ worst-case time, where k is the number of elements being inserted.

The SP-order data structure consists of two order-maintenance data structures to maintain English and Hebrew orderings.[3] With the SP-order data structure, the implementation of an SP-maintenance algorithm is remarkably simple.

Pseudocode for SP-order

Figure 2-5 gives serial pseudocode for SP-order. The code performs a left-to-right tree walk of the input SP parse tree, executing strands on the fly as the parse tree unfolds. In lines 1–3, the code handles a leaf in the SP parse tree. In a race-detection application, queries of the two order-maintenance data structures are performed in the EXECUTESTRAND function, which represents the computation of the program under test. Typically, a determinacy-race detector [91] performs $O(1)$ queries for each memory access of the program under test.

As the tree walk encounters each internal node of the SP parse tree, it performs OM-INSERT operations into the English and Hebrew orderings. In line 4, we update the English ordering for the children of the node X and insert X's (left and right) children after X with X's left child appearing first. Similarly, we update the Hebrew ordering in lines 5–7. For the Hebrew ordering, we insert X's children in different orders depending on whether X is an S-node or a P-node. If X is an S-node, handled in line 6, we insert X's left child and then X's right child after X in the Hebrew order. Figure 2-6 illustrates the insertions at an S-node. If X is a P-node, on the other hand, X's left child follows X's right child. Figure 2-7 illustrates these insertions. In lines 8–9, the code continues to perform the left-to-right tree walk. We determine whether X precedes Y, shown in lines 10–11, by querying the two order-maintenance structures using the order-maintenance query OM-PRECEDES.

Correctness of SP-order

The following lemma demonstrates that SP-ORDER produces English and Hebrew orderings correctly.

Lemma 2.3 *At any point during the execution of* SP-ORDER *on an SP parse tree, the order-maintenance data structures Eng and Heb maintain English and Hebrew, respectively, orderings of the nodes of the parse tree that have been visited thus far.*

Proof. Consider an internal node Y in the SP parse tree. We focus first on the *Heb* data structure. Assume that Y is an S-node. We must prove that all the nodes in Y's left subtree precede all the

[3] In fact, the English ordering can be maintained implicitly during a left-to-right tree walk. For conceptual simplicity, however, we use explicit order-maintenance data structures for both orderings.

```
SP-ORDER(X)
1  if ISLEAF(X)
2     then EXECUTESTRAND(X)
3          return

   ▷ X is an internal node
4  OM-INSERT(Eng, X, left[X], right[X])

5  if ISSNODE(X)
6     then OM-INSERT(Heb, X, left[X], right[X])
7     else  OM-INSERT(Heb, X, right[X], left[X])

8  SP-ORDER(left[X])
9  SP-ORDER(right[X])

SP-PRECEDES(X, Y)
10  if OM-PRECEDES(Eng, X, Y) and OM-PRECEDES(Heb, X, Y)
11     then return TRUE
12  return FALSE
```

Figure 2-5: The SP-order algorithm written in serial pseudocode. The SP-ORDER procedure maintains the relationships between strand nodes in an SP parse tree, which can be queried using the SP-PRECEDES procedure. Each internal node X in the parse tree contains pointers to a left child *left*$[X]$ and a right child *right*$[X]$. The query ISSNODE determines whether a node is an S-node or a P-node, and the query ISLEAF determines whether the node is a leaf. The order-maintenance data structures *Eng* and *Heb* represent the English and Hebrew, respectively, orderings being constructed. The EXECUTESTRAND procedure executes the strand. We assume that the root of the SP parse tree is added when the data structure is initialized.

nodes in Y's right subtree in the *Heb* ordering. We do so by showing that this property is maintained as an invariant during the execution of the code. The only place that the *Heb* data structure is modified is in lines 6–7. Suppose that the invariant is maintained before SP-ORDER is invoked on a node X. There are four cases:

1. $X = Y$: Trivial.
2. X resides in the left subtree of Y: We already assume (inductively) that X precedes all the nodes in Y's right subtree. Regardless of whether X is an S-node or a P-node, X's children are inserted immediately after X in *Heb*. (Only their order of insertion varies.) Hence, *left*$[X]$ and *right*$[X]$ also precede all the nodes in Y's right subtree.
3. X resides in the right subtree of Y: The same argument applies as in Case 2.
4. X lies outside of the subtree rooted at Y: Inserting X's children anywhere in the data structure cannot affect the invariant.

If, instead, Y is a P-node, the argument is analogous, except that the order is reversed. That is, we must prove that all the nodes in Y's right subtree precede all the nodes in Y's left subtree in the *Heb* ordering. Again, we do so by showing that this property is maintained as an invariant during the execution of the code. Suppose that the invariant is maintained before SP-ORDER is invoked on a node X. There are again four cases, similar to the ones above.

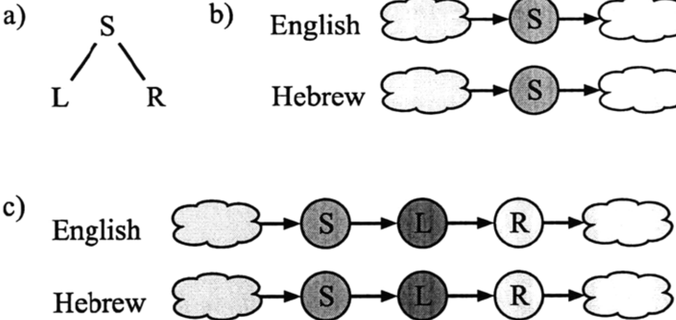


Figure 2-6: An illustration of how SP-order operates at an S-node. (a) A simple parse tree with an S-node S and two children L and R. (b) The order structures before traversing to S. The clouds represent the rest of the order structure, which does not change when traversing to S. (c) The result of the inserts after traversing to S. The left child L and then the right child R are inserted after S in both lists.

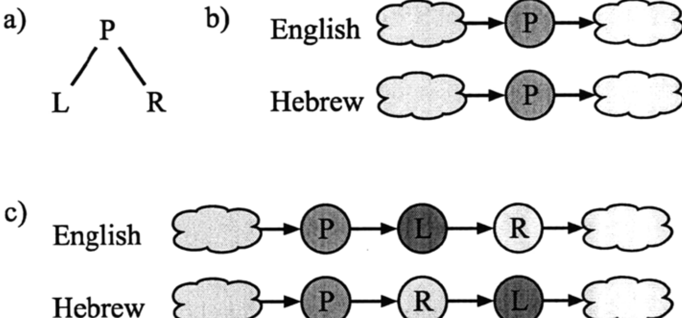


Figure 2-7: An illustration of how SP-order operates at a P-node. (a) A simple parse tree with a P-node P and two children L and R. (b) The order structures before traversing to P. The clouds are the rest of the order structure, which does not change when traversing to P. (c) The result of the inserts after traversing to P. The left child L then the right child R are inserted after P in the English order, and R then L are inserted after P in the Hebrew order.

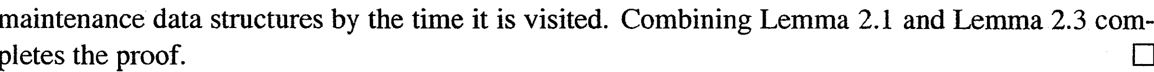
1. $X = Y$: Trivial.
2. X resides in the left subtree of Y: We already assume (inductively) that X comes *after* all the nodes in Y's right subtree. Regardless of whether X is an S-node or a P-node, X's children are inserted immediately after X in *Heb*. (Only their order of insertion varies.) Hence, *left*$[X]$ and $right[X]$ also come after all the nodes in Y's right subtree.
3. X resides in the right subtree of Y: The same argument applies as in Case 2.
4. X lies outside of the subtree rooted at Y: Inserting X's children anywhere in the data structure cannot affect the invariant.

The argument for the *Eng* data structure is analogous, except that there is no need to distinguish whether Y is a P-node or S-node separately. □

The next theorem shows that SP-PRECEDES works correctly.

Theorem 2.4 *Consider any point during the execution of the* SP-ORDER *procedure on an SP parse tree, and let u_i and u_j be two strands that have already been visited. Then, the procedure* SP-PRECEDES(u_i, u_j) *correctly returns* TRUE *if we have $u_i \prec u_j$ and* FALSE *otherwise.*

Proof. The SP-ORDER procedure inserts a node X into the *Eng* and *Heb* linear orders when it visits X's parent and before executing SP-ORDER(X). Thus, any strand is already in the order-


maintenance data structures by the time it is visited. Combining Lemma 2.1 and Lemma 2.3 completes the proof. □

Performance of SP-order

We now analyze the running time of the SP-order algorithm.

Theorem 2.5 *Consider a fork-join multithreaded program having a parse tree with n leaves. Then, each SP-operation costs $O(1)$ time in the worst case, and the total time for on-the-fly construction of the SP-order data structure is $O(n)$.*

Proof. Each SP-maintenance operation runs in $O(1)$ worst-case time, because it performs at most two order-maintenance operations and constant other work. Since a parse tree with n leaves has at most $O(n)$ nodes, the theorem follows. □

The following corollary explains that SP-order can be used to make an efficient, on-the-fly race detector. A "determinacy race" occurs whenever two logically parallel strands access the same memory location, and at least one of those memory accesses is a write.

Corollary 2.6 *A fork-join multithreaded program with running time T_1 on a single processor can be checked for determinacy races using SP-order in $O(T_1)$ time.*

Proof. Implement Feng and Leiserson's algorithm [91] using SP-order instead of SP-bags. □

SP-order may trivially be incorporated into more complicated race detectors, in particular those that recognize mutual exclusion (i.e., locks) within the code. This chapter, however, focuses only on the SP-maintenance algorithm. For more details on incorporating SP-order (and SP-ordered bags) into a race detector that supports locks, see Fineman's Master's thesis [93].

To conclude this section, observe that SP-order can be made to work on the fly no matter how the input SP parse tree unfolds. Not only can lines 8–9 of Figure 2-5 be executed in either order, the basic recursive call can be executed on nodes in any order that respects the parent-child and SP relationships. For example, one can unfold the parse tree in essentially breadth-first fashion at P-nodes as long as the left subtree of an S-node is fully expanded before its right subtree is processed. An examination of the proof of Lemma 2.3 shows why we have this flexibility. The invariant in the proof considers only a node and its children. If we expand any single node, its children are inserted into the order-maintenance data structures in the proper place independent of what other nodes have been expanded.

2.2 The SP-fast-bags algorithm

This section describes SP-fast-bags, our improved version of Feng and Leiserson's serial SP-bags algorithm [91]. The SP-bags algorithm is most easily described on parse trees having a canonical structure, and we begin by describing this structure. I then describe Feng and Leiserson's SP-bags algorithm. Next, I present an improvement to the underlying data structure of SP-bags, which yields our SP-fast-bags algorithm. The improved data structure allows PRECEDES queries to be supported in worst-case constant time and INSERT operations in amortized constant time. Although the SP-order algorithm from Section 2.1 achieves constant-time operations in the worst case, rather than just amortized bounds, we shall see in Section 2.4 how SP-fast-bags can be used together with SP-order to implement the parallel SP-ordered-bags algorithm.

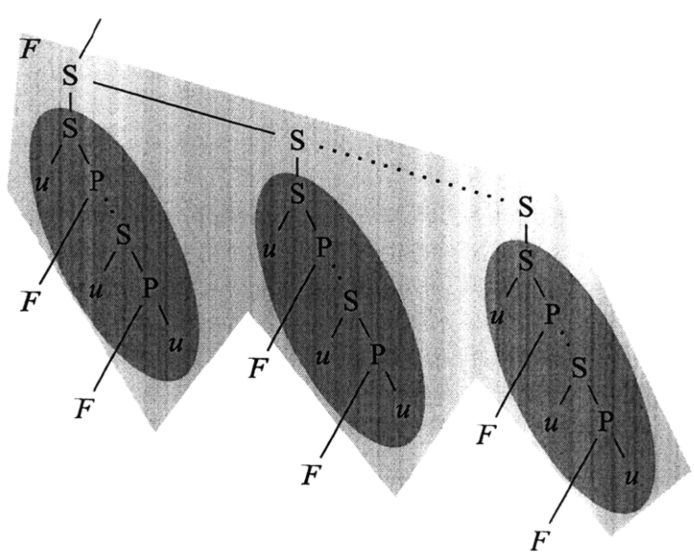


Figure 2-8: A Cilk parse tree for a generic Cilk procedure. The notation F represents the parse tree of a spawned procedure, and u represents a strand. All the nodes in the shaded area belong to the generic procedure, while all the nodes in the ovals belong to a single sync block.

Figure 2-8 shows the canonical "Cilk parse tree" as given by Feng and Leiserson [91]. A ***Cilk parse tree*** can be partitioned into truncated subtrees called procedures. A ***procedure*** is either a single strand (not shown), or it includes a right chain, called the ***spine***, of one or more S-nodes whose left children are "sync blocks." In the figure, the spine nodes are those S-nodes in the light-gray region. For convenience, we omit the right child of the rightmost S-node in the spine — this child may be thought of as a strand containing no work. A ***sync block*** (the dark-gray region) consists of either a single strand (not shown) or a right chain of alternating S- and P-nodes, beginning with an S-node and ending with a P-node. The left children of all S-nodes in the sync block are strands. The left children of all P-nodes in the sync block are procedures. We call these (left-child) procedures the ***child procedures*** or ***children*** of the procedure containing the parental P-nodes. When some (but not all) nodes in a procedure have been traversed, we say that the procedure is ***active***. When all nodes in a procedure (and hence all descendant nodes) have been traversed, we call the procedure ***completed***.

The form of the Cilk parse tree is slightly more restrictive than that of a generic fork-join program in Figure 2-2: at any given time, all the active children of a procedure (which are children of the same sync block) share the same join point — the next S-node in the spine. Any SP parse tree, however, can be represented as a Cilk parse tree by adding additional S- and P-nodes and empty strands. This transformation does not increase the size of the parse tree, and hence the work, by more than a constant factor. The span also increases by at most a constant factor assuming that the original computation has two-way (or constant-way) forks or, equivalently, assuming the children of P-nodes in the original SP parse tree are either S-nodes or strands. This constant-factor increase to span also holds for k-way fork (for nonconstant k), under the reasonable assumption that a k-way fork requires $\Theta(k)$ steps of setup (i.e., adds $\Theta(k)$ to the span).

We use the term "Cilk parse tree" because programs written in the Cilk multithreaded programming language [54, 99, 185] have this structure. Moreover, the terms "sync block" and "procedure" are consistent with Cilk terminology and linguistic constructs. It is not necessary, however, to understand these terms in the context of Cilk; our algorithms operate on the parse tree and are applicable to programs written in other languages.

start a procedure F (on a fork):
$S_F \leftarrow$ MAKE-SET(F)
$P_F \leftarrow \emptyset$

complete a procedure F' with parent F:
$P_F \leftarrow$ UNION($P_F, S_{F'}$)
$S_{F'} \leftarrow \emptyset$

traverse a spine node in a procedure F (on a join):
$S_F \leftarrow$ UNION(S_F, P_F)
$P_F \leftarrow \emptyset$

Figure 2-9: The SP-bags algorithm described in terms of procedures, as taken from [91]. Whenever one of three actions occurs during the serial, depth-first execution of a Cilk parse tree, the operations in the figure are performed.

The SP-bags algorithm

Feng and Leiserson's SP-bags algorithm [91] uses the classical disjoint-set data structure with "union by rank" and "path compression" heuristics [71, 186, 188]. The data structure maintains a collection of disjoint sets and provides three operations:

1. MAKE-SET(x) creates a new set whose only member is x.
2. UNION(x, y) unites the sets containing x and y.
3. FIND(x) returns a representative for the set containing x.

On a single processor, this data structure supports m operations on n elements in $O(m\alpha(m,n))$ time, where α is Tarjan's functional inverse of Ackermann's function. [71, 188]

Whereas the SP-order algorithm builds an SP-maintenance data structure on parse-tree nodes (specifically, on strands), the SP-bags algorithm builds a dynamic data structure on procedures. For each active procedure, the SP-bags algorithm maintains two ***bags*** (sets) of procedures with the following contents at any given time:

- The ***S-bag*** S_F of a procedure F contains the descendant procedures of F that logically precede the currently executing strand. (The descendant procedures of F include F itself.)
- The ***P-bag*** P_F of a procedure F contains the completed descendant procedures of F that operate logically in parallel with the currently executing strand.

Note that since these bags are defined on procedures, not nodes, the right child of a P-node and both children of an S-node essentially "inherit" the bags used by their parent node.

As SP-bags performs a left-to-right tree walk, it inserts procedures into the bags, unions the bags, and queries as to what type of bag a procedure belongs to. Figures 2-9 and 2-10 give the SP-bags algorithm in terms of the procedures [91] and the parse tree, respectively. Whenever a new procedure F (entering the left subtree of a P-node) is started, new bags are created. The bag S_F is initially set to contain F, and P_F is set to be empty. Whenever a procedure F' with parent procedure F completes (going from the left subtree to the right subtree of a P-node in F), the contents of $S_{F'}$ are unioned into P_F, since the descendents of F' can execute in parallel with the remainder of the sync block in F. When traversing a spine node (completing a sync block, or, equivalently, returning from any internal S-node) in F, the bag P_F is emptied into S_F, since all of F's executed descendents precede any future strands in F.

```
SP-BAGS(X, F)
    ▷ X is an SP-parse-tree node, and F is a procedure.
 1  if ISLEAF(X)
 2     then ▷ X is a strand
 3          F ← F ∪ {X}
 4          EXECUTESTRAND(X, F)
 5          return

 6  if ISSNODE(X)
 7     then SP-BAGS(left[X], F)
 8          SP-BAGS(right[X], F)
 9          S_F ← UNION(S_F, P_F)
10          P_F ← ∅
11     else ▷ X is a P-node
12          F' ← NEWPROCEDURE()
13          SP-BAGS(left[X], F')
14          P_F ← UNION(P_F, S_F')
15          S_F' ← ∅
16          SP-BAGS(right[X], F)
```

Figure 2-10: The SP-bags algorithm written in serial pseudocode which operates on a Cilk parse trees, such as that in Figure 2-8. SP-BAGS accepts as arguments a parse-tree node X and the procedure F to which X belongs. Every internal node X in the parse tree contains a pointer to its left child *left*$[X]$ and its right child *right*$[X]$. Whether a node is an S-node or a P-node can be queried with ISSNODE. Whether the node is a leaf can be queried with ISLEAF. NEWPROCEDURE$[F']$ creates a new procedure object F' with associated S-bag $S_{F'}$ and P-bag $P_{F'}$, initialized as $S_{F'} \leftarrow$ MAKE-SET$[F']$ and $P_{F'} \leftarrow \emptyset$, respectively.

SP-bags supports SP-PRECEDES on two strands provided that one of the strands is the currently executing strand. To query the relationship of a previously executed strand u_i and the currently executing strand u_j, simply find whether u_i is contained in an S- or P-bag. An S-bag indicates that $u_i \prec u_j$, and a P-bag indicates that $u_i \parallel u_j$. The "currently executing" restriction on u_j does not exist in the SP-order algorithm, but it arises here due to the way in which bags merge dynamically.

The correctness of SP-bags is captured by the following lemma. Since Feng and Leiserson [91] prove a similar lemma, we omit the proof. Note that although the algorithm given in Figure 2-10 is presented in terms of a general SP parse tree, it is only guaranteed to be correct on Cilk parse trees.

Lemma 2.7 *Consider any point during the execution of SP-bags on a Cilk parse tree. Let u_i be an already visited strand, and let u_j be the currently executing strand. Then, we have $u_i \parallel u_j$ if and only if u_i belongs to some P-bag. Conversely, we have $u_i \prec u_j$ if and only if u_i belongs to some S-bag.* □

When run on a parse tree with T_1 work and n P-nodes (or procedures), Feng and Leiserson's SP-bags performs $O(T_1)$ SP queries in the worst case. Since each query takes amortized $O(\alpha(T_1, n))$ amortized time, SP-bags runs in $O(\alpha(T_1, n)T_1)$ time.

Improving the SP-bags algorithm

The SP-fast-bags algorithm improves on the SP-bags algorithm by using a more efficient underlying disjoint-sets data structure. Gabow and Tarjan [101] describe a disjoint-sets data structure that runs in amortized-constant time per operation when the n elements being unioned are ordered *a priori* from $0, 1, \ldots, n-1$ and unions are of the form UNION$(i-1, i)$.[4] We show that SP-bags adheres to this structure. A slight variant of Gabow and Tarjan's data structure has worst-case constant-time FIND set-membership queries and amortized constant-time UNION and MAKE-SET operations.[5]

Gabow and Tarjan's disjoint-sets data structure essentially divides the n elements into groups of size $\Theta(\lg n)$ and uses a bit-vector to represent set membership for that group. In a machine model where a word containing $O(\lg n)$ bits can be operated on in constant-time [71, Section 2.2], clever bit arithmetic can be used to perform UNION and FIND operations for elements belonging to the same group in constant time. Set membership among the $O(n/\lg n)$ groups is maintained with the simple linked-list implementation from [71, Section 21.2] with the "weighted-union heuristic," where each element maintains a pointer to the set representative. Since the linked-list data structure takes amortized $O(\lg n)$ time per UNION operations, operation on $O(n/\lg n)$ groups gives a total of $O(n)$ time.

We now show that SP-bags has the union structure required by Gabow and Tarjan's data structure. Consider the English ordering of procedures in the parse tree (corresponding to the left-to-right tree walk performed by SP-bags), where a procedure is listed when it first becomes active. We index these procedures $F_1, F_2, \ldots, F_n$ according to the English ordering.[6] The following lemma shows that at any point, all S- and P-bags contain contiguous procedures. Thus, all union operations effectively have the form UNION(F_{i-1}, F_i), and we can employ Gabow and Tarjan's data structure.

Lemma 2.8 *All* UNION*s performed by SP-bags effectively have the form* UNION(F_{i-1}, F_i).

Proof. We claim that the S- and P-bags corresponding to a procedure F_i contain the procedures $S_{F_i} = \{F_i, F_{i+1}, \ldots, F_j\}$ and $P_{F_i} = \{F_{j+1}, F_{j+2}, \ldots, F_k\}$, for some j and k satisfying $i \le j \le k$.

[4] In fact, their algorithm is more general, but SP-bags follows this special case.

[5] For more details, we refer the reader to [93].

[6] Since the procedures are indexed by execution order, numbering the procedures on the fly is trivial.

As long as this property holds across the execution of the algorithm, the lemma follows. We prove this claim by induction on UNION operations.

As a base case, consider the bags S_{F_i} and P_{F_i} on creation. We have $S_{F_i} = \{F_i\}$ and $P_{F_i} = \emptyset$, which satisfies the claim. Next, consider a UNION. There are two cases.

Case 1. Suppose that a UNION occurs while traversing a spine node in a procedure F_i. Then, by assumption we have $S_{F_i} = \{F_i, F_{i+1}, \ldots, F_j\}$ and $P_{F_i} = \{F_{j+1}, F_{j+2}, \ldots, F_k\}$. Thus, the result of the UNION is $S_{F_i} = \{F_i, F_{i+1}, \ldots, F_k\}$ and $P_{F_i} = \emptyset$.

Case 2. Suppose that a UNION occurs because a child procedure $F_{i'}$ of F_i completes. Then, we must have $F_{i'} = F_{k+1}$, because $F_{i'}$ is the first procedure to follow F_k in the English ordering of procedures. Moreover, since we assume that $S_{F_{i'}} = \{F_{i'}, F_{i'+1}, \ldots, F_{k'}\}$, we end with $P_{F_i} = \{F_{j+1}, F_{j+2}, \ldots, F_k, F_{k+1}, \ldots, F_{k'}\}$. □

Given Lemma 2.8, we could apply Gabow and Tarjan's [101] data structure as a black box to achieve a serial SP-fast-bags algorithm that runs in amortized constant-time per operation. For SP-ordered-bags analysis in Section 2.8, however, where the amortization occurs in the data structure matters. In particular, we require a variant [93] that achieves worst-case constant-time FIND and MAKE-SET operations, and amortized constant-time UNION operations.

2.3 A model of parallel programs

This section first describes the performance model for parallel programs with locks and next describes features of the scheduler necessary for correctness and performance.

Our performance model assumes that the parallel algorithms execute on a shared-memory machine, as discussed in Chapter 1. The machine supports the sequential consistency [137] memory model, which is to say, for any execution of the program, there is some sequential order of memory accesses that is consistent with the program order and the values observed by the program execution. Read operations always complete in constant time, even if there are many reads to the same location. (Thus, we do not model memory congestion in the underlying machine.) We assume that concurrent-write operations on memory must queue and complete in arrival order. In the case of a tie, an adversary chooses which write operation proceeds. If a read is concurrent with many writes, then the read succeeds in constant time and gets a valid value (the value before or after the write that completes on the same step).

Since the SP-ordered-bags algorithm presented in this chapter and the XConflict algorithm presented in Chapter 3 use locks to enforce mutual exclusion, we must model how locking affects performance. In particular, whenever a processor holds a lock, other processors may be stalled waiting for that same lock. We call this idle time spent waiting for the lock ***waiting time***. During any step in which the lock is held by a processor, our model assumes the worst case: that all $P-1$ other processors are waiting for the lock. For example, if a processor acquires the lock, performs 5 steps of real work, and then releases the lock, we assume that $P-1$ other processors were waiting for the lock during these steps, and hence these 5 steps induce a waiting time of $5(P-1)$. Once the other processors acquire the lock, they also cause waiting time proportional to the number of steps they hold the lock. Thus, if the lock is held for L steps in total (summing across all processors), there may be $\Theta(PL)$ waiting time. To see how such a large waiting time can occur, consider a program in which P parallel strands all simultaneously try to acquire a lock, perform k steps of real work while holding the lock, and then release the lock. When the first processor acquires the lock, there are $P-1$ other processors waiting, inducing a waiting time of $k(P-1)$. When the second processor acquires the lock, there are $P-2$ processors waiting for a waiting time of $k(P-2)$. Summing across the waiting time introduced by all processors, we get a total waiting time of $\Theta(kP^2)$.

In this example, each of the P processors holds the lock for k steps, so the lock is held for a total of $L = kP$ steps.

Locks also serialize operations. If a single lock is held for L steps in total over the entire program (again, summing across all the processors), then the running time of the program must be at least L. This bound is straightforward, because none of the work performed while holding the lock can occur in parallel.

Consider a multithreaded program with work T_1 and span T_∞ in which a single lock is held for L steps across the course of the computation. By introducing $\Theta(PL)$ waiting time, we induce an ***apparent work***, the real work plus waiting time, of $T_1 + \Theta(PL)$. Similarly, the serialization length L induces an ***apparent span*** of $\Theta(T_\infty + L)$.

The scheduler

SP-ordered-bags and XConflict accept as input fork-join multithreaded programs expressed SP parse trees. The parse tree unfolds dynamically as the program executes, and a particular unfolding depends on how the program is scheduled. These algorithms can only operate correctly for schedulers that unfold the parse tree in a "Cilk-like" manner. Specifically, we exploit properties of an efficient "work-stealing" scheduler (such as the one used by Cilk) both for correctness and efficiency.

Our algorithms are designed for a "work-stealing" scheduler, which executes any multithreaded computation having work T_1 and span T_∞ in the asymptotically optimal $O(T_1/P + T_\infty)$ expected time on P processors. The idea behind work stealing is that when a processor runs out of its own work to do, it becomes a ***thief*** and "steals" work from another processor. The steals occurring during an execution by a work-stealing scheduler break the computation, and hence the computation's SP parse tree, into a set of truncated subtrees called "traces," where each trace consists of a set of instructions all executed by the same processor.

More specifically, ***Cilk-like work stealing*** operates as follows. A processor initially "owns" some subtree of the parse tree, which it unfolds in a left-to-right order. When a processor first traverses a P-node and begins walking its left subtree, the (unvisited) right child of that P-node becomes ***stealable***. Only the right children of P-nodes are ever stealable, and they are only stealable while the left subtree is partially elaborated and the right subtree is unvisited. When any processor first visits a stealable node, that node ceases to be stealable. When a processor has no work remaining in its subtree, it selects a victim processor and steals the subtree rooted at oldest (highest) stealable node from that victim processor. This step essentially partitions the parse tree t of the victim processor into three pieces: the subtree t_R rooted at the stealable node, the subtree t_L rooted at left sibling of the stolen node, and the truncated tree t' resulting from removing t_L and t_R from t. The subtree t_R and t_L are owned by the thief and victim, respectively. The remaining subtree t', which includes those nodes that logically follow both t_R and t_L, is awarded to the processor that traverses the last node in $t_R \cup t_L$. Whereas our conflict-detection algorithm in Chapter 3 maintains these subtrees, our series-parallel maintenance algorithm in Chapter 2 divides the computation into *five* subtrees.

Our algorithms rely on Cilk-like work stealing for correctness (and performance). Specifically, such schedulers obey the following property.

Property 2.9

- *A processor expands the subtree it is working on in a depth-first, left-to-right manner, and steals only when its subtree has been fully expanded.*

- *Whenever a thief processor steals work from a victim processor, the work stolen corresponds to the subtree rooted at a stealable node (i.e., the right subtree of a P-node whose left subtree is partially elaborated). Moreover, it selects a* highest *stealable node. That is, if the thief steals the right subtree of a P-node X, then X has no ancestor P-node with an unvisited (i.e., stealable) right child.*

For performance, we also require the mechanism for selecting a victim to be efficient. In particular, the number of steals must be bounded by $O(PT_\infty)$, which is true for a work-stealing scheduler that selects a random victim [27], such as the one used by the Cilk runtime system. We say that a work-stealing scheduler is a ***Cilk-like work-stealing scheduler*** if it satisfies Property 2.9, and that it is an efficient (Cilk-like) work-stealing scheduler if the number of steals is bounded by $O(PT_\infty)$. The Cilk scheduler (which matches Arora, Blumofe, and Plaxton's scheduler [27]) and the Intel Thread Building Blocks (TBB) scheduler [174] are both examples of Cilk-like work-stealing schedulers. Not all work-stealing schedulers, however, are Cilk-like — the Hood scheduler [56], for example, is a work-stealing scheduler that is *not* Cilk-like. We guarantee correctness on Cilk and TBB, but not on Hood. Moreover, not all Cilk-like work-stealing schedulers are what we term efficient — the Cilk scheduler is, but the TBB scheduler is not [184].

2.4 The SP-ordered-bags algorithm

This section provides an overview of the parallel SP-maintenance algorithm SP-ordered-bags. I first give a lower bound on the performance of a naive parallelization of the SP-order algorithm. I discuss how a Cilk parse tree is provided as input to SP-ordered-bags and explain some of the properties of an efficient scheduler that SP-ordered-bags exploits. I then sketch the two-tier structure of the algorithm, which combines elements of SP-order from Section 2.1 and SP-fast-bags from Section 2.2. (Details of these two tiers are presented in Sections 2.5 and 2.6.) Finally, I outline the SP-ordered-bags algorithm itself and present pseudocode for its implementation.

The SP-ordered-bags algorithm provides weaker query semantics than the serial SP-order algorithm. These semantics are exactly what are provided by SP-bags and what are required for on-the-fly determinacy-race detection. Whereas SP-order allows precedence queries between any two strands that have been unfolded in the parse tree, SP-ordered-bags requires that one of the strands be a currently executing strand.

A naive parallelization of SP-order

A straightforward way to parallelize the SP-order algorithm is to share the SP-order data structure among the processors that are executing the input fork-join program. In Section 2.1, we showed that the algorithm's correctness does not depend on the order of the parse tree's execution. Unfortunately, processors may interfere with each other as they modify the data structure, and thus some method of synchronization must be employed to provide mutual exclusion.

Suppose that we handle mutual exclusion through the use of locks. Specifically, suppose that each processor obtains a global lock prior to every OM-INSERT or OM-PRECEDES operation on the shared SP-order data structure, releasing the lock when the operation is complete. Although this parallel version of SP-order is correct, the locking can introduce significant performance penalties.

Since there can be as many as $\Theta(T_1)$ SP-order operations, and each operation causes a processor to hold the lock for $\Theta(1)$ amortized steps, the lock might be held for $L = \Theta(T_1)$ steps. (Recall the performance model in Section 2.3.) Thus, the apparent work becomes $\Theta(PT_1)$, and the apparent span length becomes $\Theta(T_1)$. Consequently, in this worst-case scenario, the program executes in

$\Omega(T_1)$ time on P processors, which shows no asymptotic improvement over the serial SP-order algorithm.

Of course, common programs may not realize such a pessimistic bound. Nevertheless, locking can significantly inhibit the scalability of a parallel algorithm, and we would like provable guarantees on scalability.

Overview of SP-ordered-bags

The SP-ordered-bags algorithm uses a two-tiered hierarchy with a global tier and a local tier in order to overcome the scalability problems with lock synchronization. The computation is dynamically divided into "traces," where each trace encapsulates a part of the computation that is performed by a single process. The global tier maintains the ordering among traces, while the local tier maintains the ordering within a trace. Thus, the global tier is subject to concurrent accesses, while (essentially), the local tier is accessed only sequentially. As a result, we use a variant of SP-Order, with locking, for the global tier, and a variant of SP-fast-bags for the local tier. (An advantage of using SP-fast-bags at the local tier is that it facilitates the dynamic management of traces.)

In more detail, the SP-ordered-bags performs a parallel walk of the input SP parse tree, partitioning the strands into traces on the fly, where each trace consists of strands that execute on the same processor. The task of maintaining the ordering is then divided among the two tiers. The goal of this two-tier structure is to reduce the synchronization delays for shared data structures, that is, processors wasting their time by waiting on locks. SP-ordered-bags's shared global tier minimizes synchronization delays in two ways. First, a lock-free scheme is employed so that OM-PRECEDES can execute on the shared data structure without waiting for operations that modify the data structure to complete. Second, the number of insertions is reduced to $O(PT_\infty)$, thereby reducing the maximum apparent work for insertions to $O(P^2T_\infty)$, since at most $P-1$ processors need to wait during the work of any insertion.

For the purposes of explaining how SP-ordered-bags works, we maintain traces explicitly. Formally, we define a ***trace*** U to be a (dynamic) set of strands that have been executed on a single processor. The ***computation*** $\mathcal{C}$ is a dynamic collection of disjoint traces, $\mathcal{C} = \{U_1, U_2, \ldots, U_k\}$. Initially, the computation consists of a single empty trace. As the computation unfolds, each strand is inserted into a trace.

Whenever the work-stealing scheduler causes a steal from a victim processor that is executing a trace U, SP-ordered-bags splits U into five subtraces $\langle U^{(1)}, U^{(2)}, U^{(3)}, U^{(4)}, U^{(5)}\rangle$, which Section 2.6 explains in detail, modifying the computation $\mathcal{C}$ as follows:

$$\mathcal{C} \leftarrow \mathcal{C} - U \cup \{U^{(1)}, U^{(2)}, U^{(3)}, U^{(4)}, U^{(5)}\} .$$

Consequently, if the work-stealing scheduler performs s steals, $|\mathcal{C}| = 4s+1$. Since an efficient work-stealing scheduler provides a bound of $O(PT_\infty)$ steals with high probability, the expected size of $\mathcal{C}$ is $O(PT_\infty)$. The principal use of the SP-bags algorithm from [91] is that it enables efficient splitting, as will be explained in Section 2.6.

Details of the two tiers of SP-ordered-bags will be presented in Sections 2.5 and 2.6. For now, it is sufficient to understand the operations each tier supports. The global tier supports the operations OM-INSERT and OM-PRECEDES on English and Hebrew orderings. In addition, the global tier supports an OM-MULTI-INSERT operation, which inserts several items into an order-maintenance data structure. The local tier supports LOCAL-INSERT and LOCAL-PRECEDES on a local (SP-bags) data structure. It supports an operation SPLIT, which partitions the strands in a trace when a steal occurs. It also supports an operation FIND-TRACE, which returns the current trace to which a strand

belongs. The implementation of all the local-tier operations must be such that many FIND-TRACE operations can execute concurrently.

Figure 2-11 presents parallel pseudocode for the SP-ordered-bags algorithm, with details of the local-tier operations omitted. As in the SP-order algorithm, SP-ordered-bags performs a left-to-right walk of the SP parse tree, executing strands as the parse tree unfolds. Each strand is inserted into a trace, which is local to the processor executing the strand. The structure of the trace forms the local tier of the SP-ordered-bags algorithm and is described further in Section 2.6. The full SP-ordered-bags algorithm can be obtained by merging Figure 2-11 with the SP-parse-tree walk performed by the local-tier algorithm.

SP-ordered-bags associates each node in the SP parse tree with a single trace by accepting a trace U as a parameter in addition to a node X, indicating that the descendant strands of X should be inserted into the trace U. When SP-ORDERED-BAGS(X, U) completes, it returns the trace with which to associate the next node in the walk of the parse tree. In particular, for an S-node X, the trace U' returned from the walk of the left subtree in line 8 is passed to the walk of X's right subtree in line 9. The same is true for P-nodes, unless the right subtree has been stolen; see lines 13 and 16.

Lines 1–5 deal with the case where X is a leaf and therefore a strand. As in SP-ORDER, the queries to the SP-maintenance data structure occur in the EXECUTESTRAND procedure. In our analysis in Section 2.8, we shall assume that the number of queries is at most the number of instructions in the strand. The strand is inserted into the provided trace U in line 3 before executing the strand in line 4. Lines 6–10 and lines 12–28 handle the cases where X is an S- or P-Node, respectively. For both P-nodes and S-nodes, the procedure walks to X's left then right subtree. For an S-node, however, the left subtree must be fully expanded before walking to the right subtree.

During the time that a P-node is being expanded, a steal may occur. Specifically, while the current processor walks the left subtree of the P-node, another processor may steal (the walking of) the right subtree. When a steal is detected (line 14 — EXECUTINGSERIALLY() returns FALSE),[7] the current trace is split into five traces — $U^{(1)}$, $U^{(2)}$, $U^{(3)}$, $U^{(4)}$, and $U^{(5)}$ — with a call to the SPLIT procedure. This SPLIT procedure, and the partitioning into subtraces, is described further in Section 2.6. The SP-ordered-bags algorithm proceeds to order the traces, inserting the five new traces into the global SP-maintenance data structures. The *Eng* order maintains the English ordering of the traces, as follows:

$$\langle U^{(1)}, U^{(2)}, U^{(3)}, U^{(4)}, U^{(5)} \rangle .$$

Similarly, the *Heb* order maintains the Hebrew ordering of the traces:

$$\langle U^{(1)}, U^{(4)}, U^{(3)}, U^{(2)}, U^{(5)} \rangle .$$

We use a global lock to serialize these trace constructions and insertions into the shared order-maintenance data structure. For correctness, we require not only that a stolen node have no ancestors with stealable children (see Property 2.9), but we also require that all work associated with ancestral steals (such as splitting traces) complete before performing the work for the newly stolen node. Whereas the scheduler guarantees that steals occur in the proper order, our algorithm must enforce the property for the work occurring on steals. To enforce this condition, we introduce two fields for each P-node in the parse tree. The field $stolen[X]$ is initially set to FALSE; when the work associated with stealing X's right has completed, $stolen[X] \leftarrow$ TRUE (line 25). The field $PNodeParent[X]$ stores the nearest P-node ancestor Y of X for which X is in Y's left subtree.[8] In particular, our

[7]Although the function EXECUTINGSERIALLY may be provided by the runtime system, as it is in Cilk with the "SYNCHED" function, runtime-system support is not required. In particular, one need simply check whether U' has been written. If it has, then the first **do** block has executed completely.

[8]Maintaining *PNodeParent* is omitted from the pseudocode for compactness, but it is easily maintained by passing

```
SP-ORDERED-BAGS(X, U)
     ▷ X is a SP-parse-tree node, and U is a trace
 1   if ISLEAF(X)
 2      then ▷ X is a strand
 3           U ← U ∪ {X}
 4           EXECUTESTRAND(X)
 5           return U
 6   if ISSNODE(X)
 7      then ▷ X is an S-node
 8           U′ ← SP-ORDERED-BAGS(left[X], U)
 9           U″ ← SP-ORDERED-BAGS(right[X], U′)
10           return U″
     ▷ X is a P-node
11   stolen[X] = FALSE
12   in parallel
13        do U′ ← SP-ORDERED-BAGS(left[X], U)
14        do if EXECUTINGSERIALLY()
15              then ▷ the recursive call on line 12 has completed
16                   U″ ← SP-ORDERED-BAGS(right[X], U′)
17                   return U″
18              else ▷ A steal has occurred
19                   wait until stolen[PNodeParent[X]] = TRUE
20                   ACQUIRE(lock)
21                     create new traces U^(1), U^(2), U^(4), and U^(5)
22                     OM-MULTI-INSERT(Eng, U^(1), U^(2), U, U^(4), U^(5))
23                     OM-MULTI-INSERT(Heb, U^(1), U^(4), U, U^(2), U^(5))
24                     SPLIT(U, X, U^(1), U^(2))
25                     stolen[X] ← TRUE
26                   RELEASE(lock)
27                   SP-ORDERED-BAGS(right[X], U^(4))
28   return U^(5)
```

Figure 2-11: The SP-ordered-bags algorithm written in parallel pseudocode, with the local-tier operations omitted. SP-ORDERED-BAGS accepts as arguments an SP-parse-tree node X and a trace U, and it returns a trace. The algorithm is essentially a tree walk that carries with it a trace U into which new strands are inserted. The **in parallel** keyword indicates that each of the following **do** blocks may execute in parallel; all subsequent code (line 28) does not execute until both parallel **do** blocks complete. The EXECUTINGSERIALLY function (line 14) returns TRUE if the first **do** block (i.e., the recursive call in line 16) has executed completely; when it returns FALSE, a steal has occurred. The $PNodeParent[X]$ attribute gives the nearest P-node ancestor of X, and the $stolen[X]$ attribute is updated by the algorithm to indicate whether the work occurring on a steal of X's right child has completed. An OM-MULTI-INSERT(L, A, B, U, C, D) inserts the objects A, B, C, and D before and after U in the order-maintenance data structure L. The *Eng* and *Heb* data structures maintain the English and Hebrew orderings of traces. The SPLIT procedure uses node X to partition the existing strands in trace U into three sets, leaving one of the sets in U and placing the other two into $U^{(1)}$ and $U^{(2)}$.

```
SP-PRECEDES(u_i, u_j)
29  U_i ← FINDTRACE(u_i)
30  U_j ← FINDTRACE(u_j)
31  if U_i = U_j
32     then return LOCAL-PRECEDES(u_i, u_j)
33  if OM-PRECEDES(Eng, U_i, U_j) and
           OM-PRECEDES(Heb, U_i, U_j)
34     then return TRUE
35  return FALSE
```

Figure 2-12: The SP-Precedes procedure for the SP-ordered-bags algorithm given in Figure 2-11. SP-PRECEDES accepts two strands u_i and u_j, where u_j must be a currently executing strand, and returns TRUE if $u_i \prec u_j$. FINDTRACE and LOCAL-PRECEDES are local-tier operations to determine what trace a strand belongs to and the relationship between strands in the same trace, respectively.

algorithm simply checks that the trace splitting for X (in lines 21–25) occurs only after Y (and, inductively, all ancestors of X) has been dealt with.

If a steal does not occur, we execute lines 16–17. Notice that if a steal does not occur anywhere in the subtree rooted at some node X, then we execute only lines 1–17 for the walk of this subtree. Thus, all descendant strands of X belong to the same trace, thereby satisfying the requirement that a trace be a set of strands that execute on the same processor.

The pseudocode for SP-PRECEDES is shown in Figure 2-12. A SP-PRECEDES query for strands u_i and u_j first examines the order of their respective traces. If the two strands belong to the same trace, the local-tier (SP-bags) data structure determines whether u_i precedes u_j. If the two strands belong to different traces, the global-tier SP-order data structure determines the order of the two traces.

2.5 The global tier of SP-ordered-bags

As introduced in Section 2.4, the global tier is essentially a shared SP-order data structure with locking mediating concurrent operations. This section describes the global tier in more detail. We show how to support concurrent queries without locking, leaving only insertions as requiring locking.

We focus on making OM-PRECEDES operations on the global tier run efficiently without locking, because the number of concurrent queries may be large. If we were to lock the data structure for each of Q queries, each query might be forced to wait for insertions and other queries, thereby increasing the apparent work by as much as $\Theta(QP)$ and nullifying the advantages of P-way parallelism. Thus, we lock the entire global tier when an insertion occurs, but use a lock-free implementation for the queries, which are likely to be more numerous.

The global tier is implemented using an $O(1)$-amortized-time order-maintenance data structure such as those described in [33, 81, 192]. The data structure keeps a doubly linked list[9] of items and assigns an integer label to each inserted item. The labels are used to implement OM-PRECEDES:

an extra parameter, identifying the relevant P-node, into the SP-ORDERED-BAGS calls.

[9]Actually, a two-level hierarchy of lists is maintained, but this detail is unnecessary to understand the basic workings of lock-free queries, and the one-level scheme we describe can be readily extended. A one-level scheme yields the same correctness guarantees, but inserts have an amortized cost of $O(\log n)$ for an n-element list instead of the desired $O(1)$.

to compare two items in the linear order, we compare their labels. When OM-INSERT adds a new item to the dynamic set, it assigns the item a label that places the item into its proper place in the linear order.

Sometimes, however, an item must be placed between two items labeled i and $i+1$, in which case this simple scheme does not work. At this point, the data structure relabels some items so that room can be made for the new item. We refer to the dynamic relabeling that occurs during an insertion as a ***rebalance***. Depending on how "bunched up" the labels of existing items are, the algorithm may need to relabel a different number of items during one rebalance than another. In the worst case, nearly all of the items may need to be relabeled.

When implementing a rebalance, therefore, the data structure may stay locked for an extended period of time. The goal of the lock-free implementation of OM-PRECEDES is to allow these operations to execute quickly and correctly even in the midst of rebalancing. We modify the order-maintenance data structure to contain two sets of labels—an item x has labels $label_1[x]$ and $label_2[x]$. Implementation of a rebalance maintains the following three properties:

- When no rebalance is in progress, $label_1[x] = label_2[x]$ for all items x in the list, and the labels respect the total order (i.e., $label_i[x] < label_i[y]$ if and only if $x \prec y$).
- At any instant in time (during a rebalance), at least one set of labels is consistent with the total order.
- A concurrent query can detect whether a rebalance in progress has corrupted its view of the linear order.

We use a counter (which starts at 1) to support the third property. When the counter is odd, the set of $label_1$ respects the total order. When the counter is even, the set of $label_2$ is valid. The algorithm actually proceeds in five phases, two of which implement the normal rebalance:

1. Determine the range of items to rebalance.
2. Assign the desired label to each item's $label_2$.
3. Increment the counter indicating that a concurrent query should read the $label_2$'s.
4. Assign the desired label to each item's $label_1$.
5. Increment the global counter indicating that the rebalance has completed and that a concurrent query should read the $label_1$.

This rebalancing strategy modifies each item twice while guaranteeing that a concurrent read obtains a consistent view of the linear order.

OM-PRECEDES query checks the counter to determine whether a rebalance is in progress. To compare items X and Y, it first determines the parity of the counter, then examines the appropriate labels of X and Y, and finally checks the counter again. If the counter has not changed between readings, then the query attempt ***succeeds***, and the order of labels determines the order of X. Otherwise, the query attempt ***fails*** and is repeatedly retried until it succeeds.

Since queries attempts can fail, the repeated attempts may increase the apparent work and the apparent span of the computation. Section 2.8 bounds these increases.

2.6 The local tier of SP-ordered-bags

This section describes the local tier of the SP-ordered-bags algorithm. I show how a trace running locally on a processor can be split when a steal occurs. By using the SP-fast-bags algorithm to implement the trace data structure, a split can be implemented in $O(1)$ time. Finally, I show that these data structures allow the series-parallel relationship to be determined between a currently running strand and any other previously executed or currently executing strand.

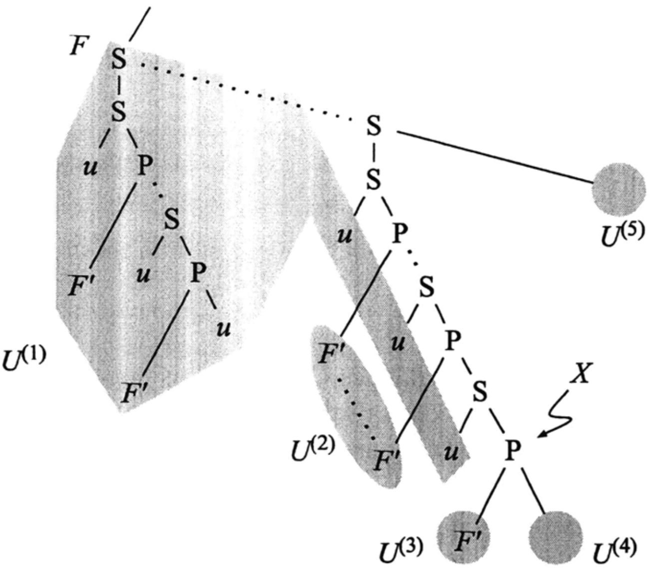


Figure 2-13: The split of a trace U around a P-node X in terms of a canonical Cilk parse tree (see Figure 2-8). The notation F' represents the parse tree of a spawned procedure, and u represents a strand. The tree walk of U is executing in $\mathit{left}[X]$ when the subtree rooted at $\mathit{right}[X]$ is stolen by a thief processor. The shaded regions contain the nodes belonging to each of the subtraces produced by the split. The two circles not enclosing any text indicate portions of the parse tree that have not yet been visited by the tree walk of U.

Splitting traces

Besides maintaining the SP relationships within a single trace, the local tier of the SP-ordered-bags algorithm supports the splitting of a trace into subtraces. A split of a trace U occurs when the processor executing U becomes the victim of a steal. The work stolen corresponds to the right subtree of the P-node X that is highest in the SP-parse tree walked by the victim. When a trace U is split around a P-node X, the local tier creates five subtraces:[10]

1. $U^{(1)} = \{u \in U : u \prec X\}$, the strands that precede X.
2. $U^{(2)} = \{u \in U : u \parallel X \text{ and } u \notin \mathit{descendants}(X)\}$, the strands parallel to X that do not belong to a subtree of X.
3. $U^{(3)} = \{u \in U : u \in \mathit{descendants}(\mathit{left}[X])\}$, the strands in X's left subtree.
4. $U^{(4)} = \{u \in U : u \in \mathit{descendants}(\mathit{right}[X])\}$, the strands in X's (stolen) right subtree. This set is initially empty.
5. $U^{(5)} = \{u \in U : X \prec u\}$, the strands that follow X. This set is also initially empty.

We call these properties the ***subtrace properties*** of U.

The SPLIT procedure from Figure 2-11 implements the split. Since $U^{(4)}$ and $U^{(5)}$ are initially empty, they are not provided as parameters to the SPLIT procedure in line 24 of the SP-ORDERED-BAGS pseudocode from Figure 2-11. The set $U^{(3)}$ is simply those strands that remain in U after those from $U^{(1)}$ and $U^{(2)}$ have been split off.

Let us look at these subtraces in terms of the parse tree. (Recall that Figure 2-8 shows the canonical Cilk parse tree.) Figure 2-13 shows the subtraces formed when a processor steals the tree walk rooted at $\mathit{right}[X]$. Since all the strands contained in $U^{(1)}$ have been executed, no more changes to this subtrace will occur. Similarly, the strands contained in $U^{(2)}$ have already been executed. The subtrace $U^{(3)}$ is partially populated, and the processor executing the walk of U will

[10]In fact, the subtraces $U^{(2)}$ and $U^{(3)}$ can be combined, but we keep them separate to simplify the proof of correctness.

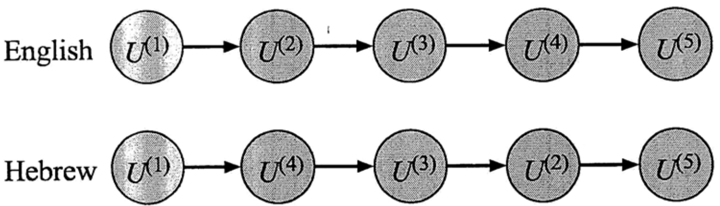


Figure 2-14: An ordering of the new traces resulting from a steal as shown in Figure 2-13. Each circle represents a trace.

continue to put strands into $U^{(3)}$. The subtrace $U^{(4)}$, which is initially empty, corresponds to the strands encountered during the thief processor's tree walk. The subtrace $U^{(5)}$, which is also initially empty, represents the next spine node / the start of the next sync block in the procedure.

When the subtraces are created, they are placed into the global tier using the concurrent SP-order algorithm. The ordering of the traces resulting from the steal in Figure 2-13 is shown in Figure 2-14. All the strands in $U^{(1)}$ precede those in $U^{(3)}$, $U^{(4)}$, and $U^{(5)}$. Similarly, all the strands (to be visited) in $U^{(5)}$ serially follow those in $U^{(1)}$, $U^{(2)}$, $U^{(3)}$, and $U^{(4)}$. Thus, we place $U^{(1)}$ first and $U^{(5)}$ last in both the English and Hebrew orders. Since any pair of strands drawn from distinct subtraces $U^{(2)}$, $U^{(3)}$, and $U^{(4)}$ operate logically in parallel, we place $U^{(2)}$, $U^{(3)}$, and $U^{(4)}$ in that order into the English ordering and $U^{(4)}$, $U^{(3)}$, and $U^{(2)}$ in that order into the Hebrew ordering. Although there is no clear relationship among all the strands in $U^{(1)}$ and $U^{(2)}$, since neither of these traces contains unexecuted strands, SP-ordered-bags never compares them.

The SP-fast-bags algorithm can be adapted to implement the local-tier operations required by SP-ordered-bags, namely LOCAL-INSERT, LOCAL-PRECEDES, FIND-TRACE, and SPLIT. All these operations, except FIND-TRACE, are executed only by the single processor working on a trace. The FIND-TRACE operation, however, may be executed by any processor, and thus the implementation must operate correctly in the face of multiple FIND-TRACE operations.

The SP-fast-bags implementation used by SP-ordered-bags follows that of Section 2.2, except we must additionally support the SPLIT operation. At the time of a split, the subtraces $U^{(1)}$, $U^{(2)}$, and $U^{(3)}$ may all contain many strands. Thus, splitting them off from the trace U may take substantial work. Fortunately, SP-fast-bags overcomes this difficulty by allowing a split to be performed in $O(1)$ time.

Consider the S- and P-bags at the time a strand (i.e., the right subtree of the P-node X in Figure 2-13) in the top-level procedure F is stolen and the five subtraces $U^{(1)}$, $U^{(2)}$, $U^{(3)}$, $U^{(4)}$, and $U^{(5)}$ are created. The S-bag of F contains exactly the strands in the subtrace $U^{(1)}$. Similarly, the P-bag of F contains exactly the strands in the subtrace $U^{(2)}$. The SP-fast-bags data structure supports moving these two bags to the appropriate subtrace using only $O(1)$ pointer updates. The subtrace $U^{(3)}$ owns all the other S- and P-bags that belonged to the original trace U, and thus nothing more need be done, since $U^{(3)}$ directly inherits U's strands. The subtraces $U^{(4)}$ and $U^{(5)}$ are created with empty S- and P-bags. (Although $U^{(4)}$ and $U^{(5)}$ belong to the same procedure as $U^{(1)}$, we modify SP-fast-bags to treat them as new procedures.) Thus, the split can be performed in $O(1)$ time, since only $O(1)$ bookkeeping needs to be done including updating pointers.

We must make one additional change to SP-fast-bags to get an efficient SP-ordered-bags algorithm. The analysis in Section 2.8 bounds the number of steals by arguing that each time a steal occurs, we make headway in the critical path of the computation. Since a UNION is amortized, we may increase the critical path of the computation. In particular, the linked-list disjoint-set component of a UNION may take up to $\Theta(n/\lg n)$ time, where n is the number of strands. To compensate for this amortization, we also provide a FAST-UNION(x, y) operation that simply points the repre-

sentative of the set containing y at the representative of the set containing x. This operation takes constant time. The FIND operation, however, must now follow more than one pointer to find the set representative. In Section 2.8, we argue that FIND operations are still $O(1)$ in the worst case.

We call the FAST-UNION instead of the UNION operation whenever the procedure making the call is ready to be stolen. Specifically, when SP-fast-bags performs a union of two sets with a call to ADAPTABLE-UNION, we repeatedly perform a constant number of steps from the slow UNION operation and test whether the current procedure is ready to be stolen. If the procedure is ready to be stolen, we instead perform a FAST-UNION.

Figure 2-15 gives parallel pseudocode for the local-tier SP-fast-bags algorithm. This version of SP-fast-bags returns a procedure ID into which future strands should be inserted, which is conceptually similar to SP-ordered-bags from Figure 2-11. Whenever a steal occurs, as shown in lines 20–24, we create new "procedures" to handle the traces $U^{(4)}$ and $U^{(5)}$. The new procedure, corresponding to trace $U^{(5)}$, is returned to be handled by the appropriate ancestor S-node. This version of SP-fast-bags also uses the FAST-UNION operations where appropriate. Whenever the parent of a procedure F' has been stolen, F' is available to be stolen, and a FAST-UNION is preferred. We use the field $parentstolen[F']$ to indicate this fact. If we call an ADAPTABLE-UNION(F', x, y), then the UNION periodically checks against $parentstolen[F']$ to see whether it should switch to the FAST-UNION.

Even though Figure 2-15 gives a parallel implementation of SP-fast-bags, the code still reflects a serial algorithm. That is, SP-fast-bags is still only correct if run on a single processor — this version of SP-fast-bags allows parallelization only when run in the context of the local tier of SP-ordered-bags. Also, as with the SP-fast-bags from Figure 2-10, we do not require any locking. Since any particular bag or set is modified by only a single processor, there is no contention to worry about.

To attain the full SP-ordered-bags algorithm, the two parse-tree walks from Figures 2-11 and 2-15 must be merged together.

2.7 Correctness of SP-ordered-bags

This section proves the correctness of the SP-ordered-bags algorithm. I begin by showing that the traces maintained by SP-ordered-bags are consistent with the subtrace properties defined in Section 2.6. I then prove that the traces are ordered correctly to determine SP relationships. Finally, I conclude that SP-ordered-bags properly responds to SP queries.

Due to the way the splits work, we can no longer prove a theorem as general as Lemma 2.1. That is to say, we can only accurately derive the relationship between two strands if one of them is a currently executing strand.[11] Although this result is weaker than for the serial algorithm, we do not need anything stronger for a race detector. Furthermore, these are exactly the semantics provided by the lower-tier SP-bags algorithm.

Correctness for SP-bags is established in [91]. The only significant difference between our version of SP-fast-bags from Section 2.6 and the version in [91] is that we may spawn new instances of the SP-fast-bags algorithm when a steal occurs. The new instances result from the creation of new bags given in lines 20–24 of Figure 2-15. These instances correspond to the subtraces $U^{(4)}$ and $U^{(5)}$ from Figure 2-13, which are subtrees of the parse tree. Since SP-fast-bags operates correctly on a Cilk parse tree, it operates correctly on a subtree as well, and we do not give a new correctness proof here. Instead, we concentrate on correctness of the global tier and the SP-ordered-bags algorithm as a whole.

[11] Specifically, we cannot determine the relationship between strands in $U^{(1)}$ and $U^{(2)}$, but we can determine the relationship between any other two traces.

```
SP-FAST-BAGS(X, F)
      ▷ X is an SP-parse-tree node, and F is a procedure ID.
 1  if ISLEAF(X)
 2     then ▷ X is a strand
 3           F ← F ∪ {X}
 4           EXECUTESTRAND(X, F)
 5           return F

 6  if ISSNODE(X)
 7     then F ← SP-FAST-BAGS(left[X], F)
 8           F_return ← SP-FAST-BAGS(right[X], F)
 9           S_F ← ADAPTABLE-UNION(F, S_F, P_F)
10           P_F ← ∅
11           return F_return

      ▷ X is a P-node
12  F' ← NEWPROCEDURE()
13  parentstolen[F'] ← FALSE
14  in parallel
15       do spawn SP-FAST-BAGS(left[X], F')
16       do if EXECUTINGSERIALLY()
17             then P_F ← ADAPTABLE-UNION(F, P_F, S_F')
18                  S_F' ← ∅
19                  return SP-FAST-BAGS(right[X], F)
20             else ▷ A steal occurred.
21                  parentstolen[F'] ← TRUE
22                  F ← NEWPROCEDURE()
23                  SP-FAST-BAGS(right[X], F)
24                  return NEWPROCEDURE()
```

Figure 2-15: The local-tier SP-fast-bags algorithm written in parallel pseudocode to operate on the canonical Cilk parse tree from Figure 2-8. This implementation is similar to Figure 2-10 except for the implicit addition of a FAST-UNION where necessary. The boolean *parentstolen*$[F']$ indicates whether the parent procedure of F' has been stolen (meaning that F' is available to be stolen). As with SP-ordered-bags, this version of SP-fast-bags returns a function into which future strands should be inserted.

The following lemma shows that when a split occurs, the subtraces are consistent with the subtraces properties given in Section 2.6.

Lemma 2.10 *Let U_i be a trace that is split around a P-node X. Then, the subtrace properties of U_i are maintained as invariants by* SP-ORDERED-BAGS.

Proof. The subtrace properties of U_i hold at the time of the split around the P-node X, when the subtraces were created, by definition. If a subtrace is destroyed by splitting, the property holds for that subtrace vacuously.

Consider any strand u at the time it is inserted into some trace U. Either U is a subtrace of U_i or not. If not, then the properties hold for the subtrace U_i vacuously. Otherwise, we have five cases.

Case 1: $U = U_i{}^{(1)}$. This case cannot occur. Since $U_i{}^{(1)}$ is mentioned only in lines 19–28 of Figure 2-11, it follows that $U_i{}^{(1)}$ is never passed to any call of SP-ORDERED-BAGS. Thus, no strands are ever inserted into $U_i{}^{(1)}$.

Case 2: $U = U_i{}^{(2)}$. Like Case 1, this case cannot occur.

Case 3: $U = U_i{}^{(3)}$. We must show that $U_i{}^{(3)} = \{u : u \in \mathit{descendants}(\mathit{left}[X])\}$. The difficulty in this case is that when the trace U_i is split, we have $U_i = U_i{}^{(3)}$, that is, U_i and $U_i{}^{(3)}$ are aliases for the same set. Thus, we must show that the invariant holds for all the already forked instances of SP-ORDERED-BAGS that took U_i as a parameter, as well as those new instances that take $U_i{}^{(3)}$ as a parameter. As it turns out, however, no new instances take $U_i{}^{(3)}$ as a parameter, because (like Cases 1 and 2) $U_i{}^{(3)}$ is neither passed to SP-ORDERED-BAGS nor returned.

Thus, we are left to consider the already forked instances of SP-ORDERED-BAGS that took U_i as a parameter. One such instance is the outstanding SP-ORDERED-BAGS$(\mathit{left}[X], U_i)$ in line 13. If $u \in \mathit{descendants}(\mathit{left}[X])$, then we are done, and thus, we only need consider the forks SP-ORDERED-BAGS(Y, U_i), where Y is an ancestor of the P-node X. We use induction on the ancestors of X, starting at $Y = \mathit{parent}(X)$ to show that SP-ORDERED-BAGS(Y, U_i) does not pass U_i to any other calls, nor does it return U_i. For the base case, we see that SP-ORDERED-BAGS(X, U_i) returns $U_i{}^{(5)} \neq U_i{}^{(3)}$.

For the inductive case, consider SP-ORDERED-BAGS(Y, U_i). We examine the locations in the pseudocode where this procedure can resume execution. If Y is an S-node, then this procedure can be waiting for the return from SP-ORDERED-BAGS$(\mathit{left}[Y], U_i)$ in line 8 or the return from SP-ORDERED-BAGS$(\mathit{right}[Y], U_i)$ in line 9. In the first situation, our inductive hypothesis states that SP-ORDERED-BAGS$(\mathit{left}[Y], U_i)$ does not return U_i, and hence, we neither pass U_i to the right child nor do we return it. The second situation is similar.

Instead, suppose that Y is a P-node. Since steals occur from the top of the tree, we cannot resume execution at line 16, or else SP-ORDERED-BAGS$(\mathit{right}[Y], U_i)$ would have already been stolen. We can be only at either line 17 or line 28. If we are at line 17, our inductive assumption states that SP-ORDERED-BAGS$(\mathit{right}[Y], U_i)$ does not return U_i, and thus we do not return U_i either. Otherwise, we are at line 28, and we return the $U^{(5)}$ resulting from some split.

Case 4: $U = U_i{}^{(4)}$. We must show that $U_i{}^{(4)} = \{u : u \in \mathit{descendants}(\mathit{right}[X])\}$. The only place where $U_i{}^{(4)}$ is passed to another SP-ORDERED-BAGS call, and hence used to insert a strand, is line 27. No matter what SP-ORDERED-BAGS$(\mathit{right}[X], U_i{}^{(4)})$ returns, the call to SP-ORDERED-BAGS(X, U_i) does not return $U_i{}^{(4)}$; it returns $U_i{}^{(5)}$. Thus, the only strands that can be inserted into $U_i{}^{(4)}$ are descendants of $\mathit{right}[X]$, which matches the semantics of $U^{(4)}$.

Case 5: $U = U_i{}^{(5)}$. We must show that $U_i{}^{(5)} = \{u \in U_i : X \prec u\}$. The subtrace $U_i{}^{(5)}$ is used only in the return from SP-ORDERED-BAGS(X, U_i) on line 28. As seen in lines 6–10 and lines 16–17, SP-ORDERED-BAGS passes the trace returned from a left subtree to a right subtree. Thus, the only SP-ORDERED-BAGS calls that have any possibility of inserting into $U_i{}^{(5)}$ are the right descendants of X's ancestors. When a split occurs (and hence when a steal occurs), by the

properties (Property 2.9) of the work-stealing scheduler, it occurs at the topmost P-node of a trace. Thus, the only ancestors of X with unelaborated right subtrees are S-nodes. It follows that $\mathrm{lca}(u, X)$ is an S-node, and hence $X \prec u$. □

The following lemma shows that the *Eng* and *Heb* orderings maintained by SP-ordered-bags are sufficient to determine the relationship between traces.

Lemma 2.11 *Let Eng and Heb be the English and Hebrew orderings, respectively, maintained by the global tier of SP-ordered-bags. Let u_j be a currently executing strand in the trace U_j, and let u_i be any strand in a different trace $U_i \neq U_j$. Then, we have $u_i \prec u_j$ if and only if $Eng[U_i] < Eng[U_j]$ and $Heb[U_i] < Heb[U_j]$.*

Proof. The proof is by induction on the number of splits during the execution of SP-ordered-bags. Consider the time that a trace U is split into its five subtraces. If neither U_i nor U_j is one of the resulting subtraces $U^{(1)}, U^{(2)}, \ldots, U^{(5)}$, then the split does not affect U_i or U_j, and the lemma holds holds trivially.

Suppose that $U_i \in \{U^{(1)}, U^{(2)}, \ldots, U^{(5)}\}$, but $U_j \notin \{U^{(1)}, U^{(2)}, \ldots, U^{(5)}\}$. Then, U_i and U_j have the same relationship they did before the split, because we insert the subtraces $U^{(1)}$, $U^{(2)}$, $U^{(4)}$, and $U^{(5)}$ contiguously with $U = U^{(3)}$ in the English and Hebrew orderings. Similarly, if we have $U_i \notin \{U^{(1)}, \ldots, U^{(5)}\}$, but $U_j \in \{U^{(1)}, U^{(2)}, \ldots, U^{(5)}\}$, then the lemma holds symmetrically.

Thus, we are left with the situation where $U_i \in \{U^{(1)}, U^{(2)}, \ldots, U^{(5)}\}$ can be any trace, but the other trace is restricted to $U_j \in \{U^{(3)}, U^{(4)}, U^{(5)}\}$. We can ignore the case when $U_i = U_j$, because the lemma assumes that $U_i \neq U_j$, as well as the cases when $U_j \in \{U^{(1)}, U^{(2)}\}$, because u_j is a currently executing strand. We consider the remaining twelve cases in turn.

Case (1,3): $U_i = U^{(1)}$ and $U_j = U^{(3)}$. We apply Lemma 2.10 to conclude that $u_i \prec X$ for some P-node X and $u_j \in \mathit{descendents}(\mathit{left}[X])$, which implies that $u_i \prec u_j$. We also have that $Eng[U^{(1)}] < Eng[U^{(3)}]$ and $Heb[U^{(1)}] < Heb[U^{(3)}]$, which matches the claim.

Case(1,4): Similar to case (1,3).

Case(1,5): $U_i = U^{(1)}$ and $U_j = U^{(5)}$. We apply Lemma 2.10 to conclude that $u_i \prec X$ for some P-node X and that $X \prec u_j$, and hence $u_i \prec u_j$. We also have that $Eng[U^{(1)}] < Eng[U^{(5)}]$ and $Heb[U^{(1)}] < Heb[U^{(5)}]$, which matches the claim.

Case (2,3): $U_i = U^{(2)}$ and $U_j = U^{(3)}$. Lemma 2.10 allows us to conclude that $u_i \in \{u \in U : u \parallel X \text{ and } u \notin \mathit{descendants}(X)\}$ for some P-node X and that strand $u_j \in \mathit{descendants}(\mathit{left}[X])$, which means that $u_i \parallel u_j$. We also have that $Eng[U^{(2)}] < Eng[U^{(3)}]$ and $Heb[U^{(2)}] > Heb[U^{(3)}]$, which matches the claim.

Case(2,4): Similar to case (2,3).

Case(2,5): $U_i = U^{(2)}$ and $U_j = U^{(5)}$. As $Eng[U^{(2)}] < Eng[U^{(5)}]$ and $Heb[U^{(2)}] < Heb[U^{(5)}]$, we must show that $u_i \prec u_j$. The simple fact here is that $U^{(5)}$ logically follows *all* other strands in U (as shown pictorially in Figure 2-13), but that is not directly stated in the subtrace properties, so we will prove it more formally here. Lemma 2.10 implies that $X \prec u_j$, and hence $S = \mathrm{lca}(X, u_j)$ is an S-node such that X and u_j are contained in S's left and right subtrees, respectively. If u_i is in S's left subtree, then $u_i \prec u_j$, and we have shown what we need to show. We also infer that u_i cannot be in S's right subtree (specifically, $\mathrm{lca}(X, u_i) \neq S$), since Lemma 2.10 states that $\mathrm{lca}(X, u_i)$ is a P-node. The remaining option to consider is that u_i not be a descendant of S. We will show by contradiction that this case cannot occur. We therefore consider $\mathrm{lca}(u_i, S)$. As $\mathrm{lca}(u_i, S) = \mathrm{lca}(u_i, X)$, and $u_i \parallel X$, we conclude that $\mathrm{lca}(u_i, S)$ must be a P-node. Note that by the structure of the Cilk parse tree, S is a spine node (as spine nodes are the only S-nodes with P-nodes in their left subtrees). Moreover, also by the structure of the Cilk parse tree, the nearest P-node ancestor of a spine node is a right parent. Let X' be the nearest P-node ancestor of S. Then either $X' = \mathrm{lca}(u_i, S)$, or X'

is a descendant of $\text{lca}(u_i, S)$. Given the left-to-right execution of Property 2.9 and the fact that u_j (located in the left subtree of X') is currently executing, we have two possibilities. If the right child of X' is stealable, then Property 2.9 implies that the current split is not around X, which generates a contradiction. If the right child of X' has already been stolen, then the right subtree of X' does not belong to the trace U, which is also a contradiction. As all other possibilities have been eliminated, we thus conclude that $\text{lca}(u_i, u_j) = S$, and hence $u_i \prec u_j$.

Case(3,4): $U_i = U^{(3)}$ and $U_j = U^{(4)}$. Lemma 2.10 states that $u_i \in \mathit{descendants}(\mathit{left}[X])$ and $u_j \in \mathit{descendants}(\mathit{right}[X])$ for some P-node X, and hence $\text{lca}(u_i, u_j) = X$ and $u_i \parallel u_j$. We also have that $\mathit{Eng}[U^{(3)}] < \mathit{Eng}[U^{(4)}]$ and $\mathit{Heb}[U^{(3)}] > \mathit{Heb}[U^{(4)}]$, which matches the claim.

Case(3,5): $U_i = U^{(3)}$ and $U_j = U^{(5)}$. Lemma 2.10 states that $u_i \in \mathit{descendants}(\mathit{left}[X])$ and $X \prec u_j$. It follows that $\text{lca}(X, u_j)$ is an S-node with X belonging to the left subtree, and hence $\text{lca}(u_i, u_j)$ is an S node with u_i belonging to the left subtree. We conclude that $u_i \prec u_j$, which matches the ordering $\mathit{Eng}[U^{(3)}] < \mathit{Eng}[U^{(5)}$ and $\mathit{Heb}[U^{(3)}] < \mathit{Heb}[U^{(5)}]$.

Case(4,3): Symmetric with case (3,4).

Case(4,5): Similar to case (3,5).

Case(5,3): Symmetric with case (3,5).

Case(5,4): Symmetric with case (4,5). □

We are now ready to prove that SP-ordered-bags returns the correct result for an SP-PRECEDES operation run on a currently executing strand and any other strand.

Theorem 2.12 *Consider any point during the execution of* SP-ORDERED-BAGS *on an SP parse tree. Let u_i be a strand that has been visited, and let u_j be a strand that is currently executing. Then, the procedure* SP-PRECEDES(u_i, u_j) *correctly returns* TRUE *if $u_i \prec u_j$ and* FALSE *otherwise.*

Proof. The SP-ORDERED-BAGS procedure inserts a strand u into a trace U before executing u, and therefore when a strand executes, it belongs to some trace. Furthermore, the English and Hebrew orderings *Eng* and *Heb*, respectively, contain all traces that contain any strands.

First, consider the case in which u_i and u_j do not change traces during the execution of SP-PRECEDES. If u_i and u_j belong to the same trace, then SP-PRECEDES returns the correct result as the result of a query on the local tier. If u_i and u_j belong to different traces, then Lemma 2.11 shows that the correct result is returned.

We must also show that SP-PRECEDES returns the correct result if the traces for either u_i or u_j are split during the execution of the SP-PRECEDES query. Only a single SPLIT may be in progress at a time because of the global lock used by SP-ORDERED-BAGS. We consider the state of the traces at the instant in time at which the last SPLIT *completed* before the start of SP-PRECEDES. That is, if there is no SPLIT in progress, then we consider the state of traces at the time SP-PRECEDES begins. If there is a SPLIT in progress when SP-PRECEDES begins, then we consider the state of traces just before *that* SPLIT began.

Suppose that we have u_i and u_j belong to different traces at this time. Consider the code given in Figure 2-12. There is no way to get $U_i = U_j$ in the test in line 31. Moreover, we have that Lemma 2.11 applies to give us the correct result.

Suppose instead that at the start of the SPLIT, we have u_i and u_j belong to the same trace U. A SPLIT may be in progress. Since u_j is still executing, and it belongs to some trace already, it follows from the subtrace properties that u_j can only be a part of a $U^{(3)} = U$ resulting from a SPLIT. Thus, the trace for u_j cannot change across the execution of SP-PRECEDES. Similarly, u_i can belong to one of $\{U^{(1)}, U^{(2)}, U^{(3)}\}$, but it cannot belong to $U^{(4)}$ or $U^{(5)}$, since it already exists at the time of the SPLIT. Given these facts, it does not matter whether we get $U_i = U_j$ in line 31. If $U_i = U_j$ and $u_i, u_j \in U^{(3)}$, then LOCAL-PRECEDES returns the correct result. If $U_i = U_j$ and $u_i \in U^{(1)}$,

then LOCAL-PRECEDES still returns the correct result, since u_i belongs to an S-Bag. Similarly, if $U_i = U_j$ and $u_i \in U^{(2)}$, then LOCAL-PRECEDES returns the correct result, since u_i belongs to a P-Bag. From Lemma 2.11, the OM-PRECEDES returns the correct result in either of these two cases. □

2.8 Performance analysis

This section analyzes the SP-ordered-bags algorithm run on a fork-join program. Suppose that the program has T_1 work and a span of T_∞. When executed on P processors using a round-robin work-stealing scheduler (that satisfies Property 2.9), SP-ordered-bags runs in $O(T_1/P + PT_\infty)$ time in the worst case.

First, I show that the local-tier operations LOCAL-PRECEDES (implemented as a FIND in the disjoint-sets data structure) takes $O(1)$ time in the worst case. Recall that we modify SP-fast-bags and the disjoint-sets data structure for SP-ordered-bags in Section 2.6 by introducing this notion of a FAST-UNION. Whereas the Gabow and Tarjan data structure has all elements pointing at the representative, a set resulting from a FAST-UNION does not. As a result, it is not immediately obvious that a FIND takes $O(1)$ time in the worst case. To prove this fact, we exploit the structure of the unions performed by the local tier of SP-ordered-bags. If a set has all elements pointing at the representative, then we say that the depth of the set is 1. When the FAST-UNION is performed, the depth may increase, but we bound this increase.

Lemma 2.13 *For a procedure F that is ready to be stolen in an execution of SP-ordered-bags, the depth of the sets for S_F and P_F are at most 2 and 3, respectively.*

Proof. Proof by induction on the unions involving these bags.

Any child procedure F' is not ready to be stolen, because Property 2.9 (see Section 2.4) requires parents to be stolen before their children. Thus, since no FAST-UNIONs are performed on F', the depth of $S_{F'}$ is 1 when F' returns to F.

The ADAPTABLE-UNION($F, P_F, S_{F'}$) is only performed in line 18 of Figure 2-15. Since the modified ADAPTABLE-UNION points the representative of $S_{F'}$ to the representative for P_F, this union does not increase the depth of P_F past 2.

The only point at which we union with S_F is line 9. Since the representative for P_F is pointed to the representative of S_F, and the depth of P_F is at most 2 by the inductive assumption, we have that the depth of S_F does not increase past 3. □

Given Lemma 2.13, we have that each FIND takes $O(1)$ worst-case time, which gives us the following corollary.

Corollary 2.14 *A* LOCAL-PRECEDES *performed by SP-ordered-bags takes $O(1)$ worst-case time.* □

To prove the desired bound on the entire SP-ordered-bags execution, we must first bound the number of steals performed. The following theorem, similar to Theorem 9 from [27], applies to our environment when using an efficient work-stealing scheduler, such as the Cilk scheduler. We focus specifically on the case where the work stealing occurs, and show that this is still efficient in our context. We assume that processors are moving at roughly the same speed. That is, all processors execute $\Theta(1)$ instructions per unit time.

Theorem 2.15 *Consider any fork-join program with T_1 work and span T_∞. When executed on P processors using a Cilk-like work-stealing scheduler that chooses victims to steal from at random, SP-ordered-bags has $O(PT_\infty)$ steal attempts in expectation. Moreover, for any $\varepsilon > 0$, the number of steal attempts is $O(P(T_\infty + \lg(1/\varepsilon)))$ with probability at least $1 - \varepsilon$.*

Proof. Since the proof is virtually identical to one given by Arora, Blumofe, and Plaxton [27], only a sketch of the proof is given here. We assign potentials to each ***ready*** strand — those strands that are ready to be executed. In particular, let $d(u)$ be the depth of a strand u, or the distance of u from the start of the computation. Then

$$\phi(u) = \begin{cases} 3^{2(T_\infty - d(u))-1} & \text{if } u \text{ is assigned to some processor;} \\ 3^{2(T_\infty - d(u))} & \text{otherwise.} \end{cases}$$

The potential only decreases over time. Whenever a strand is stolen, it is assigned to a processor. Whenever a processor completes an assigned strand, it enables up to two children which are deeper in the computation. Either of these actions decreases the total potential.

To bound the number of steals, group potentials by processors owning the strands. The crux of the argument is that whenever a thief processor tries to steal from a victim processor q, the victim loses a constant factor of its potential. It turns out that the next strand stolen from a particular processor q contributes a constant fraction of that processor's potential. Thus, if a successful steal occurs, the potential decreases by a constant fraction of q's potential. Similarly, if q does not have any strands ready to be stolen, then completing the current strand also decreases q's potential by a constant fraction. Therefore, even if the steal attempt is not successful, we know that the potential decreases.

Following the argument by Aurora *et al.*, we apply a balls-in-bins argument to argue that each contiguous ***round*** of P steals reduces the total potential by a constant factor with constant probability. A Chernoff bound across the rounds gives a high-probability result.

In our case, the work completed during any step may not be real work towards the original fork-join program. To compensate, we blow up each instruction by a factor of r, where r is the worst-case cost of the $O(1)$ SP-PRECEDES (without retrying the OM-PRECEDES) queries performed at each instruction and the cost of the local-tier SP-maintenance operation when the strand is ready to be stolen. Since the balls-in-bins argument relies on the fact that we make progress towards the critical path when a strand is being worked on or stolen, we care only about the blowup from SP-maintenance that occurs at this time.

The two SP-maintenance operations that pose the greatest challenge are the UNION, which may need to make a lot of updates, and the OM-PRECEDES operations, which may retry several times. For the former, observe that since FAST-UNION is performed when the strand is ready to be stolen, the blowup (at steal-attempt time) from this operation is at most $O(1)$. As for OM-PRECEDES, recall that we assume processors are moving at the same speed. We assume that the updates performed on a steal take a sufficiently long (constant) amount of time that an OM-PRECEDES only needs to abort once. (If not, we can make SP-ordered-bags wait for a constant amount of time while holding the global lock, without impacting the asymptotic performance.) We, therefore, effectively have a new computation with span $T'_\infty \leq rT_\infty$, with $r = O(1)$, and the potential and balls-in-bins arguments from Arora *et al.* still apply. □

Next, I show that the entire SP-ordered-bags algorithm performs well.

Theorem 2.16 *Suppose that a fork-join program has T_1 work and span T_∞. When executed on P processors using a Cilk-like work-stealing scheduler that chooses victims to steal from at random,*

SP-ordered-bags runs in $O(T_1/P + PT_\infty)$ expected time. Moreover, for any $\varepsilon > 0$, SP-ordered-bags runs in $O(T_1/P + P(T_\infty + \lg(1/\varepsilon)))$ time with probability at least $1 - \varepsilon$.

Proof. We use an accounting argument similar to [55], except with seven buckets, instead of three. Each bucket corresponds to a type of task that a processor can be doing during a step of the algorithm. For each time step, each processor places one dollar in exactly one bucket. If the execution takes time T_P, then at the end the total number of dollars in all of the buckets is PT_P. Thus, if we sum up all the dollars in all the buckets and divide by P, we obtain the running time.

The analysis depends on the number s of successful steals during the execution of the SP-ordered-bags algorithm. Theorem 2.15 shows that the expected value of s. The seven buckets are as follows:

B_1: The work of the original computation excluding costs added by SP-ordered-bags. We have that $|B_1| = T_1$, because a processor places one dollar in the work bucket whenever it performs work on the input program.

B_2: The work for global-tier insertions, including the cost for splits. SP-ordered-bags performs an OM-INSERT operation, serially, for each steal. The amortized time required to perform s operations in the order-maintenance data structure is $O(s)$. Thus, $|B_2| = O(s)$.

B_3: The work for the local-tier SP-maintenance operations. Since there are $O(1)$ SP-fast-bags operations for each instruction in the computation, and each SP-fast-bags operation costs $O(1)$ amortized time, we have $|B_3| = O(T_1)$.

B_4: The waiting time for the global lock on global-tier OM-INSERT operations. When one processor holds the lock, at most $O(P)$ processors can be waiting. Since $O(1)$ insertions occurs for each steal, we have $|B_4| = O(Ps)$.

B_5: The work wasted on failed and retried global-tier queries. Since a single insertion into the order-maintenance structure can cause at most $O(1)$ queries to fail on each processor and the number of insertions is $O(s)$, we conclude that $|B_5| = O(Ps)$.

B_6: Steal attempts while the global lock is not held by any processors. We use Theorem 2.15 to conclude that $|B_6| = O(PT_\infty)$ in expectation, or $|B_6| = O(P(T_\infty + \lg(1/\varepsilon)))$ with probability at least $1 - \varepsilon$.

B_7: Steal attempts while the global lock is held by some processor. The global lock is held for $O(s)$ time in total, and in the worst case, all processors try to steal during this time. Thus, we have $|B_7| = O(Ps)$.

To conclude the proof, observe that $s \leq |B_6|$, because the number of successful steals is less than the number of steal attempts. Summing up all the buckets yields $O(T_1 + P|B_6|)$ at the end of the computation, and hence, dividing by P, we obtain an expected running time of $O(T_1/P + PT_\infty)$ and the corresponding high probability bound. □

It turns out that we can modify the work-stealing scheduler to improve the worst-case performance of SP-ordered-bags. In particular, we modify the scheduler to perform steal attempts in a round-robin order instead of randomly. To perform round-robin steal attempts, we lock a global list of processors on each steal attempt. Since we lock on successful steals for SP-ordered-bags anyway, this additional locking does not hurt us. (Randomized stealing makes sense in the context of an arbitrary multithreaded program, because ordinarily steals are not serialized.)

Given the round-robin stealing policy, we can state a worst case bound on the number of steals, similar to Theorem 2.15.

Theorem 2.17 *Consider any fork-join program with T_1 work and span T_∞. When executed on P processors using the round-robin, Cilk-like work-stealing scheduler (i.e., one obeying Property 2.9), SP-ordered-bags has $O(PT_\infty)$ steal attempts in the worst case.*

Proof. We use the same potential function as in Theorem 2.15. We group P contiguous steal attempts into rounds. In a round, a steal attempt occurs on each processor. When attempting to steal from a particular processor, its potential decreases by a constant factor. Thus, in a round, the potential of the entire system decreases by a constant factor. There can be $O(T_\infty)$ such rounds, for a total of $O(PT_\infty)$ steal attempts in the worst case. □

Applying Theorem 2.17 to bound the number s of successful steals in Theorem 2.16, we achieve the following worst-case bound.

Theorem 2.18 *Consider any fork-join program with T_1 work and span T_∞. When executed on P processors using the round-robin, Cilk-like work-stealing scheduler, SP-ordered-bags runs in $O(T_1/P + PT_\infty)$ time in the worst case.* □

Finally, I show that SP-ordered-bags is scalable. In particular, the following corollary states that SP-ordered-bags achieves linear speedup when $P = O(\sqrt{T_1/T_\infty})$. In contrast, the underlying fork-join program achieves linear speedup on a work-stealing scheduler when $P = O(T_1/T_\infty)$, which is optimal. Although $\sqrt{T_1/T_\infty}$ is not optimal, it is reasonably large.

Corollary 2.19 *SP-ordered-bags achieves linear speedup when $P = O(\sqrt{T_1/T_\infty})$.*

Proof. When $P = O(\sqrt{T_1/T_\infty})$, Theorems 2.16 and 2.18 state that SP-ordered-bags runs in $O\left(T_1/P + \sqrt{T_1/T_\infty} \cdot T_\infty\right) = O\left(T_1/P + T_1/\sqrt{T_1/T_\infty}\right) = O(T_1/P)$. □

2.9 Related work on SP-maintenance

This section summarizes related work on SP-maintenance and order-maintenance data structures.

Nudler and Rudolph [163] introduced the English-Hebrew labeling scheme for SP-maintenance. Each strand is assigned two labels, similar to the labeling in this paper. They do not, however, use a centralized data structure to reassign labels. Instead, label sizes grow proportionally to the maximum concurrency of the program. Mellor-Crummey [153] proposed an "offset-span labeling" scheme, which has label lengths proportional to the maximum nesting depth of forks. Although it uses shorter label lengths than the English-Hebrew scheme, the size of offset-span labels is not bounded by a constant as it is in our scheme.

The first order-maintenance data structure was published by Dietz two decades ago [79]. It supports insertions and deletions in $O(\lg n)$ amortized time and queries in $O(1)$ time. Tarjan observed [81] that updates could be supported in $O(1)$ amortized time, and the same result was obtained independently by Tsakalidis [192]. Dietz and Sleator [81] proposed two data structures, one that supports insertions and deletions in $O(1)$ amortized time and queries in $O(1)$ worst-case time and another that supports all operations in $O(1)$ worst-case time. Bender, Cole, Demaine, Farach-Colton, and Zito [33] gave two simplified data structures whose asymptotic performance matches the data structures from [81]. Their paper also presents an implementation study of the amortized data structure.

A special case of the order-maintenance problem is the the ***online list-labeling problem*** [22,80, 82,123], also called the ***file maintenance problem*** [197–200]. In online list labeling, we maintain a mapping from a dynamic set of n elements to the integers in the range from 1 to u (***tags***), such that the order of the elements matches the order of the corresponding tags. Any solution to the online list-labeling problem yields an order-maintenance data structure. The reverse is not true, however, because there exists an $\Omega(\lg n)$ lower bound on the list-labeling problem [80, 82]. In

file maintenance, we require that $u = O(n)$, since this restriction corresponds to the problem of maintaining a file densely packed and defragmented on disk.

Labeling schemes have been used for other combinatorial problems such as answering least-common-ancestor queries [1, 15, 17, 126] and distance queries used for routing [14, 16, 26, 102, 132, 189]. Although these problems are reminiscent of the order-maintenance problem, most solutions focus on reducing the number of bits necessary to represent the labels in a static (offline) setting.

Anderson and Woll [20] discuss concurrent union-find operations using path compression (with path halving) and union by rank. Whereas they consider multiple finds and multiple unions occurring concurrently, however, our problem is confined to single unions and multiple finds occurring concurrently.

2.10 Concluding remarks

This chapter has focused on provably efficient parallel algorithms for SP-maintenance. As a practical matter, the algorithms are likely to perform faster than the worst-case bounds indicate, because it is rare that every lock access sees contention proportional to the number of processors. This observation can be used in practical implementations to simplify the coding of the algorithms and yield somewhat better performance in the common case. Nevertheless, we contend that the predictability of provably efficient software gives users less frustrating experiences. Giving up on provable performance is an engineering decision that should not be taken lightly. We also believe that provably efficient algorithms are scientifically interesting in their own right.

As we were writing the earlier conference version of the SP-ordered-bags paper [48], we repeatedly confronted the issue of how an amortized data structure interacts with a parallel scheduler. Standard amortized analysis could be applied to analyze the work of a computation, but we could not use amortization to analyze the critical path and had to settle for worst-case bounds. Luckily, since the conference version, we discovered that we could remove the amortized difficulties from the SP-ordered-bags algorithm. The issue of dealing with amortization in a parallel setting, however, has not been solved. Moreover, we were surprised that we needed to reprise the elaborate work-stealing analysis from [55] (with seven buckets, no less!) in order show that SP-ordered-bags is efficient. Are there general techniques that can allow us to develop provably good parallel algorithms without repeatedly subjecting ourselves (and readers) to such intricate and difficult mathematical arguments?

We were also surprised to see that using a round-robin work-stealing scheduler resulted in a better worst-case running time than the randomized work-stealing scheduler. When running a program that does not serialize steals as SP-ordered-bags or XConflict do, the randomized work stealer performs better. Is there some balance that can be achieved here? A Cilk program (that does not contain locks) with T_1 work and span T_∞ runs on the randomized work-stealing scheduler in $O(T_1/P+T_\infty)$ time in expectation. Is there a scheduler that results in a running time like $O(T_1/P + P^\varepsilon T_\infty)$ in the worst case?

Chapter 3

Nested Parallelism in Transactional Memory

This chapter investigates adding transactions with nested parallelism and nested transactions to a dynamically multithreaded parallel programming language that generates only series-parallel programs. I describe XConflict, a data structure that facilitates conflict detection for a software transactional memory system which supports transactions with nested parallelism and unbounded nesting depth. For languages that use a Cilk-like work-stealing scheduler, XConflict answers concurrent conflict queries in $O(1)$ time and can be maintained efficiently. In particular, for a program with T_1 work and span T_∞, the running time on P processors of the program augmented with XConflict is only $O(T_1/P + PT_\infty)$. In addition, we use XConflict as the basis for the CWSTM runtime-system design for software transactional memory, supporting unbounded nesting of parallelism and transactions. This chapter represent joint work with Kunal Agrawal and Jim Sukha, previously appearing in [8].

Using XConflict, I describe CWSTM, a runtime-system design for software transactional memory which supports transactions with nested parallelism and unbounded nesting depth of transactions. The CWSTM design provides transactional memory with eager updates, eager conflict detection, strong atomicity, and lazy cleanup on aborts. In the restricted case when no transactions abort and there are no concurrent readers, CWSTM executes a transactional computation on P processors also in time $O(T_1/P + PT_\infty)$. Although this bound holds only under rather optimistic assumptions, to our knowledge, this result is the first theoretical performance bound on a TM system that supports transactions with nested parallelism which is independent of the maximum nesting depth of transactions.

Transactional memory (TM) [119] represents a collection of hardware and software mechanisms that help provide a transactional interface for accessing memory to programmers writing parallel code. Recently, TM has been an active area of study; for example, researchers have proposed many designs for transactional memory systems, with support for TM in hardware (e.g. [18, 115, 156]), in software (e.g., [5, 118, 147]), or hybrids of hardware and software (e.g., [75, 135]). A typical TM runtime system executes transactions optimistically, aborting and rolling back transactions that "conflict" to guarantee that transactions appear to execute atomically.

Most work on transactional memory focuses exclusively on supporting transactions in programs that use persistent threads (e.g., pthreads). TM systems for such an environment are designed assuming that transactions execute serially, since the overhead of creating or destroying a pthread naturally discourages programmers from having nested parallelism inside a transaction. Furthermore, the special case of serial transactions greatly simplifies conflict detection for TM; typically,

```
PARALLELINCREMENT()
1  x ← 0
2  in parallel
3       do x ← x + 1
4       do x ← x + 10
5           in parallel
6               do x ← x + 100
7               do x ← x + 1000
8  print x
```

Figure 3-1: A simple fork-join program that does several parallel increment of a shared variable. The **in parallel** keyword indicates that each of the following **do** blocks may execute in parallel; all subsequent code (line 8) does not execute until both parallel **do** blocks (line 3 and lines 4–7) complete. This program contains several races — assuming sequential consistency, valid outputs are $x \in \{1, 11, 101, 110, 111, 1001, 1010, 1011, 1101, 1110, 1111\}$.

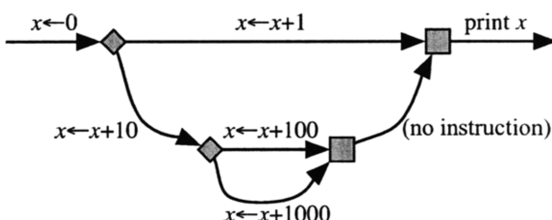


Figure 3-2: The series-parallel dag for the sample program given in Figure 3-1. Edges correspond to instructions in the program. Diamonds and squares correspond to the start and end, respectively, of parallel constructs.

the TM runtime detects a ***conflict*** between two distinct active transactions if they both access the same object ℓ, at least one transaction tries to write to ℓ, and both transactions are executed on different threads. This last condition is relevant for TM systems that support ***nested transactions***, where transactions may contain other transactions. Conceptually, when transactions execute serially, two active transactions executing on the same thread are allowed to access the same object because one must be nested inside the other.

Another way to write parallel programs, however, is to use dynamic multithreaded languages such as Cilk [54, 185] or NESL [53] or multithreaded libraries like Hood [56], as discussed in Chapter 2. A natural question arises: how can transactions be added to a dynamic multithreaded language such as Cilk?

To pose the problem more concretely, consider the series-parallel program shown in Figure 3-1, which performs parallel increments to a shared variable. Figure 3-2 gives the corresponding series-parallel dag for the program. One natural way to add transactions to a series-parallel program is by wrapping segments of the program in atomic blocks, as illustrated by Figure 3-3. As shown, it is easy to generate transactions (e.g., X_3) with nested parallelism and nested transactions (e.g., X_4). How does a TM system execute the program in Figure 3-3?

This chapter investigates adding transactions to a dynamic multithreaded language that generates only series-parallel programs. I focus on a provably efficient TM system that supports un-

```
XPARALLELINCREMENT()
 1  atomic {                                          ▷ Transaction X_1
 2     x ← 0
 3     in parallel
 4        do atomic {x ← x + 1}                       ▷ X_2
 5        do atomic {                                 ▷ X_3
 6              x ← x + 10
 7              in parallel
 8                 do x ← x + 100
 9                 do atomic {x ← x + 1000}           ▷ X_4
10        }                                           ▷ End X_3
11  }                                                 ▷ End X_1
12  print x
```

Figure 3-3: The program from Figure 3-1 with the addition of some transactions, denoted by **atomic**{.} blocks. The triangle denotes a comment. Since atomic blocks are not placed around *all* increments, this program still permits multiple outputs — valid outputs are 111 and 1111. The (symmetric) 1011 is excluded due to strong atomicity.

bounded nesting and parallelism. That is, we want a TM system with a bound on a program's completion time that is independent of the maximum nesting depth of transactions. It turns out that TMs that perform work on every transaction commit proportional to the size of the transaction's "readset" or "writeset" cannot support an unbounded nesting depth efficiently. Generally, TM with lazy conflict detection requires work proportional to the size of the transactions readset, and TM with lazy updates require work proportional to the size of the transaction's writeset. Thus, we focus on TM with eager conflict detection and eager updates, since both require only a constant amount of work on every commit.

Contributions

A key component of a TM system with nested parallelism is the conflict-detection scheme. I describe the semantics for TM with eager conflict detection for series-parallel computations with transactions. I present XConflict, a data structure that a software TM system can use to query for conflicts when implementing these semantics. For Cilk-like work-stealing schedulers, the XConflict answers concurrent queries in $O(1)$ time and can be maintained efficiently. In particular, consider a program with T_1 work and a span of T_∞. Then, the running time on P processors of the program augmented with XConflict is only $O(T_1/P + PT_\infty)$. In comparison, with high probability, Cilk executes the program without XConflict in the asymptotically optimal time $O(T_1/P + T_\infty)$. These two bounds imply that maintaining the XConflict data structure does not asymptotically increase the running time of the program, compared to Cilk, when $\sqrt{T_1/T_\infty} \gg P$.

This chapter also describe CWSTM, a design for a software TM system with eager updates that uses the XConflict data structure. CWSTM provides strong atomicity and supports lazy cleanup on aborts (i.e., when a transaction X aborts, other transactions can help roll back the objects modified by X). The XConflict bounds translate to CWSTM in a restricted case when there are no concurrent readers (all memory accesses are treated as writes) and there are no transaction abort. If the underlying transaction-free program has T_1 work and T_∞ span, then the CWSTM executes the transactional

program in time $O(T_1/P + PT_\infty)$ when run on P processors. At first glance, these bounds might seem uninteresting due to the restrictions. It is difficult, however, for any TM system to provide any nontrivial bounds on completion time in the presence of aborts, since the system might redo an arbitrary amount of work. Moreover, TM with eager conflict detection that allows more than a constant number of shared readers to an object can potentially lead to memory contention; thus, even if there are no conflicts on that object, it seems difficult to provide efficient worst-case theoretical bounds.

XConflict and CWSTM represent joint work with Kunal Agrawal and Jim Sukha, previously appearing in [8].

The remainder of this chapter is organized as follows. We use a computation tree to model a computation with nested parallel transactions; Section 3.1 describes the computation tree and how the CWSTM runtime system maintains this computation tree online. Section 3.2 defines our CWSTM semantics. I show in Section 3.3 that a naive conflict-detection algorithm has poor worst-case performance. Section 3.4 describes the high-level design of CWSTM and its use of XConflict for conflict-detection. Section 3.5 gives an overview of the XConflict algorithm. Sections 3.6–3.9 provide details on data structures used by XConflict. Finally, Section 3.10 claims that XConflict, and hence CWSTM, is efficient for programs that experience no conflicts or contention.

3.1 CWSTM framework

This section presents the computation-tree framework that we use to model CWSTM program executions. A program execution is modeled as an ordered tree, called the computation tree (as in [9]). This computation tree is similar to the SP parse tree of Chapter 2, except there are some modifications to accommodate transactions. The computation tree is not given *a priori* (i.e., from a static analysis of the program); rather, it unfolds dynamically as the program executes. Moreover, nondeterminism in the program may result in different computation trees. Constructing the computation tree on the fly as the program executes is not difficult and thus not described in full in this thesis. A partial program execution corresponds to partial traversal of the computation tree.

A computation tree has two types of nodes: leaf nodes correspond to single memory operations, while internal nodes model the nested parallel control structure of the program as well as the structure of nested transactions. As in the SP parse tree (Chapter 2), an internal node is either an S-node or a P-node. An S-node denotes series composition of the subtrees (i.e., the subtrees must execute in left-to-right order), whereas a P-node denotes parallel composition. Transactions are specified in a computation tree by marking a subset of S-nodes as transactions. A computation-tree node B is a descendant of a particular transaction node X if and only if B is contained in X's transaction. Consequently, a transaction Y is ***nested*** inside a transaction X if X is an ancestor of Y in the computation tree. We consider only computations that have closed-nested transactions.

In our canonical computation tree, all P-nodes have exactly 2 nontransactional S-nodes as children, while S-nodes can have an arbitrary number of children. In contrast, for the SP parse tree in Chapter 2, all nonleaf nodes have 2 children. This modification allows for simpler presentation in this chapter. In addition, we require that no nontransactional S-node has a child nontransactional S-node. Thus, it follows that all nontransactional S-nodes are children of P-nodes (or the root of the tree). For convenience, we treat the root of the computation tree as both a transactional and a nontransactional S-node.

We define several notations for the computation tree $\mathcal{C}$. Let $\texttt{root}(\mathcal{C})$ denote the tree's root node. For any node $B \neq \texttt{root}(\mathcal{C})$, $\texttt{parent}[B]$ denotes B's parent in $\mathcal{C}$. If B is an internal node of $\mathcal{C}$, then $\texttt{children}(B)$ is the ordered set of B's children, $\texttt{ances}(B)$ is the set of B's ancestors and $\texttt{desc}(B)$ is B's descendants. In this chapter, whenever we refer to the set of ancestors or descendants of a

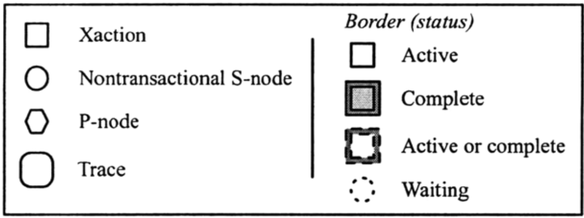


Figure 3-4: A legend for computation-tree figures.

node B, we include B in this set.

For any node $B \neq \texttt{root}(\mathcal{C})$, we define the transactional parent $\texttt{xparent}[B]$ as $Z = \texttt{parent}[B]$ if Z is a transaction or $\texttt{root}(\mathcal{C})$, and as $\texttt{xparent}[Z]$ otherwise. Similarly, we define nontransactional S-node parent $\texttt{nsParent}[B]$ as $Z = \texttt{parent}[B]$ if Z is a nontransactional S-node or $\texttt{root}(\mathcal{C})$, or $\texttt{nsParent}[Z]$ otherwise.

At any point during the computation-tree traversal, each node B in the computation tree has a ***status***, denoted by $\texttt{status}[B]$. The status can be one of PENDING, PENDING_ABORT, COMMITTED, or ABORTED. A leaf is ***complete*** if the corresponding operation has been executed. An internal node can complete only if all nodes in its subtree are complete. Thus, a complete node corresponds to the root of a subtree that will not unfold further, and hence a node is complete if and only if its status is COMMITTED or ABORTED. A node is ***active*** (having status PENDING or PENDING_ABORT) if it has any unexecuted descendants. Once a node is complete, it can never become active again.

Any execution of the computation tree has the invariant that at any time, the set of active nodes in the computation tree also forms a tree, with the leaves of this active tree being the set of ***ready*** nodes. Only a node that is ready can be traversed to "discover" a new child node. (Discovering a new child node corresponds to executing an instruction in the program: a read or write creates new leaf below the current S-node, a transactional begin creates a new transactional S-node, and a fork statement creates a new P-node along with its two S-node children.) When a ready transactional S-node completes, its parent becomes ready. When a ready nontransactional S-node Z completes, if Z's sibling nontransactional S-node is already complete, then Z's parent (which is a P-node) completes, and Z's grandparent becomes ready.

Figure 3-5 shows the structure of the computation tree after an execution of the code from Figure 3-1 in which transaction X_2 aborts once. If other aborts (and retries) occur, the computation tree would have additional subtrees.

3.2 CWSTM semantics

This section describes CWSTM semantics, a semantics for a generic transactional memory system with nested parallel transactions and eager conflict detection. I describe these semantics operationally, in terms of a "readset" and "writeset" for each transaction. In particular, I define conflicts and describe transaction commits and aborts abstractly using readsets and writesets. Later, in Section 3.3, I give a simple design for a TM that provides these semantics. In Section 3.4, we improve the simple TM and present the the CWSTM design.

At any point during the program execution, the ***readset*** of a transaction X is the set of objects ℓ that X has "accessed". Similarly, the ***writeset*** is a set of objects ℓ that X has written to. Operationally, readsets and writesets change as follows. A transaction begins with an empty readset and empty writeset. Whenever a successful `read` of ℓ_1 occurs in a memory-operation (leaf) node u,

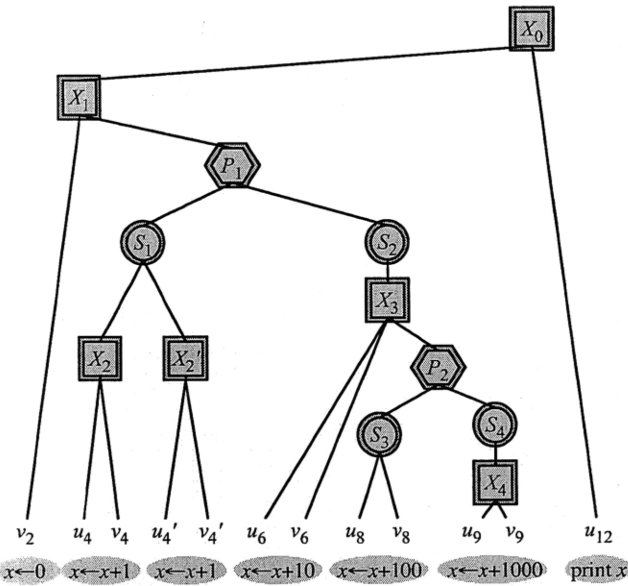


Figure 3-5: A computation tree for an execution of the program given by Figure 3-1, in which transaction X_2 aborted once and was retried as the transaction X_2'. The root X_0 does not correspond to any transaction in the program — it is just the S-node root of the tree. Each increment to x on line j of the program decomposes into two atomic memory operations: a `read` u_j, and a `write` v_j. The corresponding code is shown in a gray oval under the accesses.

ℓ_1 is added to $\texttt{xparent}[u]$'s readset. Similarly, whenever a successful `write` of ℓ_2 occurs in a memory-operation node u, ℓ_2 is added to $\texttt{xparent}[u]$'s readset and writeset. A `read` or a `write` to ℓ by an operation u "observes" the value associated with the write stored in the writeset of Z, where Z is the nearest transactional ancestor of u that contains ℓ in its writeset. For consistency, the writeset of the computation-tree's root contains all objects. When a transaction X commits, its readset and writeset are merged into $\texttt{xparent}[X]$'s readset and writeset, respectively.

A transactional memory with eager conflict detection must test for conflict before performing each `read` or `write`. An access is unsuccessful if it generates a transactional conflict. TM systems with serial, closed-nested transactions report conflicts when two active transactions on different threads are accessing the same object ℓ, and one of those accesses is a write. Thus, only a single active transaction is allowed to contain ℓ in its writeset at one time. For CWSTM semantics, we generalize this definition of conflict in a straightforward manner. At any point in time, let $\texttt{readers}(\ell)$ and $\texttt{writers}(\ell)$ be the sets of *active* transactions that have object ℓ in their readsets or writesets, respectively. Then, we define conflicts as follows:

Definition 3.1 *At any point in time, a memory operation v generates a* ***conflict*** *if*

1. *v reads object ℓ, and $\exists X \in \texttt{writers}(\ell)$ such that $X \notin \texttt{ances}(v)$, or*
2. *v writes to object ℓ, and $\exists X \in \texttt{readers}(\ell)$ such that $X \notin \texttt{ances}(v)$.*

If there is such a transaction X, then we say that v conflicts with X. If v belongs to the transaction X', then we say that X and X' conflict with each other.

If a memory operation v would cause a conflict between $X = \texttt{xparent}[v]$ and another transaction X', then v triggers an abort of either X or X' (or both). Say X is aborted. An abort of a transaction X changes $\texttt{status}[X]$ from `PENDING` to `PENDING_ABORT`, and also changes the

status of any PENDING (nested) transaction Y in the subtree of X to PENDING_ABORT. In general, a PENDING_ABORT transaction X that is also ready can only complete by changing its status to ABORTED. Conceptually, when a transaction X is ABORTED, CWSTM semantics discards X's writeset and readset. Since X is no longer active after this action occurs, the action also conceptually removes X from readers(ℓ) and writers(ℓ) for all objects ℓ. Note that in CWSTM, if v causes a conflict, and the runtime chooses to abort $X' \neq$ xparent[v], then the conflict is not fully resolved until status[X'] has changed to ABORTED.

Consider a computation subtree rooted at a transaction X with status[X] = PENDING. Since we allow only closed-nested transactions, if every child of X has completed, CWSTM can commit X, i.e., change X's status from PENDING to COMMITTED, and merge X's readset and writeset into those of xparent[X].

Code example

We can now discuss how the CWSTM semantics constrain the possible outputs of the program in Figure 3-3. Since parallelism is allowed in transactions, we must consider the scoping of atomicity. In particular, the $x \leftarrow x + 1$ in line 4 and the code block in lines 6–9 must appear as though one executes entirely before the other. If the **atomic** statements in lines 4 and 5 were removed, then these two blocks could interleave arbitrarily, even though the entire procedure is protected by an **atomic** statement in line 1. Basically, the atomicity applies only when comparing two blocks of code belonging to different transactions (protected by different atomic statements), not parallel blocks within the same transaction (protected by the same atomic statement).

Conflict as stated in Definition 3.1 naturally enforces strong atomicity [57]. Strong atomicity implies that although line 8 is not atomic, it cannot perform its write between line 9's read and write. In terms of the computation tree in Figure 3-5, after u_9 performs a read of x, it adds x to the readset of X_4; thus, after u_9 occurs but before X_4 commits, if v_8 tries to write to x, it will cause a conflict with X_4. We can, however, have line 8 read x, line 9 read and write x and commit, and then line 8 write x. This interleaving can occur because when u_8 happens, it adds x to the readset of X_3, and u_9 and v_9 can subsequently happen because they are both descendants of X_3 in the computation tree. This behavior means that the increment of 1000 can be "lost" (by being overwritten) but the increment of 100 cannot. Another way of describing strong atomicity is that each memory operation is viewed as a transaction.

Semantic guarantees

The CWSTM semantics maintains the invariant that a program execution is always conflict-free, according to Definition 3.1. One can show that when transactions have nested parallel transactions, TM with eager conflict detection according to Definition 3.1 satisfies the transactional-memory model of *prefix race-freedom* defined in [9].[1] As shown in [9], prefix race-freedom and serializability are equivalent if one can safely "ignore" the effects aborted transactions. Note that this equivalence may not hold in TM systems with explicit retry constructs that are visible to the programmer.

Definition 3.1 directly implies the following lemma about a conflict-free execution.

Lemma 3.2 *For a conflict-free execution, the following invariants hold for any object ℓ:*

[1] The proof is a special case of the proof for the operational model described in [9], without any open-nested transactions.

1. *All transactions* $X \in$ `writers`(ℓ) *fall along a single root-to-leaf path in* C. *There is thus a unique transaction* $Y \in$ `writers`(ℓ), *denoted by* `lowest(writers`(ℓ)`)`, *such that* `writers`$(\ell) \subseteq$ `ances`(Y).
2. *All transactions* $X \in$ `readers`(ℓ) *are either along the root-to-leaf path induced by the writers or are descendants of the* `lowest(writers`(ℓ)`)`. □

We use Lemma 3.2 to argue that one can check for conflicts for a memory operation u by looking at one writer and only a small number of readers. Since all the transactions fall on a single root-to-leaf path, by Lemma 3.2, Invariant 1, the transaction `lowest(writers`(ℓ)`)` belongs to `writers`(ℓ) and is a descendant of all transactions in `writers`(ℓ). Similarly, let $Q =$ `lastReaders`$(\ell) \subseteq$ `desc(lowest(writers`(ℓ)`))` denote the set of readers implied by Invariant 2. If a memory operation u tries to read ℓ, abstractly, there is no conflict exactly if and only if `lowest(writers`(ℓ)`)` is an ancestor of u. Similarly, when u tries to write to ℓ, by Invariant 2, there is no conflict if for all $Z \in$ `lastReaders`(ℓ), Z is an ancestor of u.

3.3 A naive TM

The CWSTM semantics described in Section 3.2 suggest a design for a TM system that supports transactions with nested parallelism. In particular, Lemma 3.2 suggests that for each object ℓ, the TM can maintain an active writing transaction `lowest(writers`(ℓ)`)` and some active reading transactions `lastReaders`(ℓ). This scheme allows transactions accessing ℓ to test for conflicts against these transactions. This section focuses on a straightforward data structure, called an "access stack," used to maintain these values. I show that an access stack yields a TM with poor worst-case performance, even assuming the rest of the TM system incurs no overhead. The CWSTM design uses a lazy variant of the access stack, described in Section 3.4, that has much better performance.

The ***access stack*** for an object ℓ is a stack containing the active transactions that have written to ℓ and sets of active transactions that have read from ℓ. The order of transactions on the stack is consistent with the ancestry of transactions in the computation tree. The writing transaction `lowest(writers`(ℓ)`)` is either on top (first item to pop) of the stack, or is the next element on the stack. If the writer is not on the top of the stack, then `lastReaders`(ℓ) is. No two consecutive elements are sets of readers.

The access stack is maintained as follows, locking the relevant stack on all memory access to guarantee atomicity. Consider (a memory operation whose transactional parent is) a transaction X that successfully reads ℓ. If the top of the stack contains a set of readers, then X is added to that set, assuming it is not already there. If the top of the stack is a writer other than X, then $\{X\}$ is added to the top of the stack. Similarly, if X successfully writes ℓ, then X is pushed onto the top of the stack if it not already there.

Whenever a transaction X commits, for each ℓ in X's readset, X is removed from the top of ℓ's access stack and replaced with `xparent`$[X]$ (in a fashion that ensures there are no duplicated transactions). This action mimics the commit semantics from Section 3.2: when a transaction X commits, the objects in its readset and writeset are moved to `xparent`$[X]$'s readset and writeset, respectively. If instead X aborts, then X is popped from each relevant object's access stack. To facilitate rollback on aborts, every access-stack entry corresponding to a write stores the old value before the write.[2]

Maintaining the access stack has poor worst-case performance because the work required on the commit of transaction X is proportional to the size X's readset. If the original program (without

[2]This value can either be stored in the stack itself, or in a log per transaction.

transactions) had work T_1, then this implementation might require work $\Omega(dT_1)$, where d is the maximum nesting depth of transactions. In particular, consider the following code snippet:

```
void f(int i) {
  if (i >= 1) {  atomic { x[i]++; f(i-1); }   }
}
```

A call of `f(d)` generates a serial chain of nested transactions, each incrementing a different place in the array `x`. When the transaction at nesting depth j commits, it updates $d - j$ access stacks for a total of $\Theta(d^2)$ access-stack updates. The work of the original program (without transactions), however, is only $\Theta(d)$.

In general, this asymptotic blowup can occur if a TM system with nested transactions must perform work proportional to the size of a transaction's readset or writeset on every commit. For example, a TM system that validates every transaction due to lazy conflict detection for reads exhibits this problem. Similarly, a TM system that copies data on commit due to lazy object updates also has this issue.

3.4 CWSTM overview

This section describes our CWSTM design for a transactional-memory system with nested parallel transactions and eager updates and eager conflict detection. We first describe how CWSTM updates the computation-tree-node statuses on commits and aborts. We then give an overview of the conflict-detection mechanism, deferring details of the XConflict data structure to later sections. The conflict-detection mechanism includes a "lazy access stack," improving on the shortcoming of the access stack from Section 3.3. Finally, we describe properties of the Cilk-like work-stealing scheduler that CWSTM uses. The XConflict data structure requires such a scheduler for its performance and correctness.

CWSTM explicitly builds the internal nodes of the computation tree (i.e., leaf nodes for memory operations are omitted). Each node maintains a status field which in most cases, explicitly represents the node's status (PENDING_ABORT, PENDING, COMMITTED, or ABORTED), and changes in a straightforward fashion. For example, when a transaction X commits, CWSTM atomically changes $\texttt{status}[X]$ from PENDING to COMMITTED.

Since a transaction may signal an abort of a transaction running on a (possibly different) processor whose descendants have not yet completed, aborting transactions is more involved. When an active transaction X aborts itself (possibly because of a conflict), it simply atomically updates $\texttt{status}[X] \leftarrow$ ABORTED. We refer to this type of update as an `xabort`. Alternatively, suppose a processor p_i wishes to abort X even though p_i is not currently executing X. First, p_i atomically changes $\texttt{status}[X]$ from PENDING to PENDING_ABORT. Then p_i walks X's active subtree, changing $\texttt{status}[Y] \leftarrow$ PENDING_ABORT atomically for each active $Y \in \texttt{desc}(X)$. Notice that p_i never changes any status to ABORTED— only the processor running a transaction Y is allowed to perform that update. When X "discovers" that its status has changed to PENDING_ABORT, it has no active descendants (otherwise, X cannot be ready, and hence X cannot be executing). Then, X simply performs an `xabort` on itself.

For reasons specific to XConflict, the data structure the CWSTM design uses for conflict detection, during an abort of X, some of X's *COMMITTED* descendants Y also have their status field changed to ABORTED. Our conflict-detection algorithm uses these updates to more quickly determine that a memory operation does not conflict with Y, since Y has an ABORTED ancestor X. Section 3.8 describes when these updates occur.

```
XCONFLICT-ORACLE(X, u)
   ▷ For any node X and active memory operation u
1  if ∃Z ∈ ances(X) such that status[Z] = ABORTED
2     then return "no conflict: X aborted"

3  Y ← closest active transactional ancestor of X
4  if Y ∈ ances(u)
5     then return "no conflict: X committed to u's ancestor"
6     else pick a transaction B in (ances(Y) − ances(lca(Y, u)))
7          return "conflict with B"
```

Figure 3-6: Pseudocode for a conflict-detection query suggested by Definition 3.3. Many subroutine (e.g., line 3) details are omitted (and in fact do not have efficient implementations). The lca function returns the least common ancestor of two nodes in the computation tree.

In CWSTM, the rollback of objects on abort occur lazily, and thus is decoupled from an `xabort` operation. Once the status of a transaction X changes to ABORTED, other transactions that try to access an object modified by X help with cleanup for that object.

Conflict detection and the lazy access stack

We now discuss conflict detection. The key observation that allows us to avoid explicit maintenance of active readers and writers (or transaction readsets and writesets) is the following alternate conflict definition.

Definition 3.3 *Consider a (possibly inactive) transaction X that has written to ℓ and a new memory operation v that reads from or writes to ℓ. Then v conflicts with X if*

1. *not transactional ancestor X has aborted, and*
2. *X's nearest active transactional ancestor is not an ancestor of v.*

The case when X has read from ℓ and v writes to ℓ is analogous.

This definition is equivalent to Definition 3.1 because X's nearest active transactional ancestor logically belongs to `writers`(ℓ) if X doesn't have an aborted ancestor.

Definition 3.3 suggests a conflict-detection algorithm that does not require explicitly maintaining `lowest`(`writers`(ℓ)) and the normal access stack. In particular, let X be the last node that has successfully written to ℓ. Then, when u accesses ℓ, test for conflict by finding X's nearest active transactional ancestor Y and determine whether Y is an ancestor of u. Figure 3-6 gives pseudocode for this test. Note that CWSTM does not actually implement this query as given — instead, it uses an equivalent, but more efficient query, described in Section 3.5.

To maintain the most recent successful write (and reads), facilitating the necessary conflict queries, CWSTM uses a ***lazy access stack***. The structure of the lazy access stack is somewhat different from the simple access stack given in Section 3.3. An object ℓ's lazy stack stores (possibly complete) transactions that have written to ℓ and sets of transactions that have read from ℓ, but now these stack entries are ordered chronologically by access. The top of the stack holds the last writer or the last readers. We have the invariant that if a transaction X on the stack has aborted, then all transactions located above X on the stack (later chronologically) also have aborted ancestors, and

thus represent deprecated values. The main difference in maintenance is that the lazy access stack is not updated on transactional commit (thus ignoring the merge of a transaction's readset and writeset into its parent's). On memory operations, new transactions are added to the access stack in the same way as described in Section 3.3.

Figure 3-7 gives pseudocode for an instrumentation of each memory access, assuming for simplicity that all memory accesses behave as `write` instructions.[3] Incorporating readers into the access stacks is more complicated, but conceptually similar. If a memory access u does not belong to an aborting transaction, then it is allowed to proceed. First, we test for conflict with the last writer in lines 4–5. If the last writer has aborted (or has an aborted ancestor), handled in lines 6–9, then the access stack should be cleaned up by calling CLEANUP. (This auxiliary procedure, given in Figure 3-8, rolls back the value of the topmost aborted transaction on ℓ's access stack.) Since there is no new conflict, after CLEANUP, the access should be retried. If, on the other hand, there is a conflict between u and an active transaction (lines 10–16), then either xparent[u] must abort or the conflicting transaction (B) must abort. Finally, if there are no conflicts, then the access is successful. The access stack is updated as necessary (lines 18–20), and the access is performed.

Note that while u is running the ACCESS method, concurrent transactions (that access ℓ) can continue to commit or abort. The commit or abort of such a transaction can eliminate a conflict with u, but never create a new conflict with u. Thus, concurrent changes may introduce spurious aborts, but do not affect correctness.

The CWSTM scheduler

As with SP-ordered-bags in Chapter 2, XConflict relies on an efficient Cilk-like work-stealing scheduler of Section 2.3 for efficiency and correctness.

3.5 CWSTM conflict detection

This section describes the high-level ***XConflict*** scheme for conflict detection in CWSTM. As the computation tree dynamically unfolds during an execution, our algorithm dynamically divides the computation tree into "traces," where each trace consists of memory operations (and internal nodes) that execute on the same processor. These traces are slightly simpler than those introduced in Section 2.5, but the main idea is the same. Our algorithm uses several data structures that organize either traces (analogous to the global tier for SP-ordered-bags in Section 2.5), or nodes and transactions contained in a single trace (analogous to the local tier of SP-ordered bags in Section 2.6). This section describes traces and gives a high-level algorithm for conflict detection.

By dividing the computation tree into traces, we reduce the cost of locking on shared data structures. Updates and queries on a data structure whose elements belong to a single trace are also performed without locks because these updates are performed by a single processor. Data structures whose elements are traces also support queries in constant time without locks. These data structures are, however, shared among all processors, and therefore require a global lock on updates. Since the traces are created only on steals, however, we can bound the number of traces by $O(PT_\infty)$ — the number of steals performed by the Cilk-like work-stealing runtime system. Therefore, the number of updates on these data structure can be bounded similarly.

The technique of splitting the computation into traces and having two types of data structures — global data structures whose elements are traces and local data structures whose elements belong to a single trace — is similar to the one used for SP-maintenance in Chapter 2. Although the trace

[3] It is possible to reduce locking on the access stack, but I do not describe that optimization in this chapter.

```
ACCESS(u, ℓ)
 1  Z ← xparent[u]
 2  if status[Z] = PENDING_ABORT return XABORT
    ▷ Otherwise Z is active
 3  accessStack(ℓ).LOCK()

    ▷ Set X to be the last writer.
 4  X ← accessStack(ℓ).TOP()
 5  result ← XCONFLICT-ORACLE(X, u)

 6  if result is "no conflict: X aborted"
 7     then accessStack(ℓ).UNLOCK()
 8          CLEANUP(ℓ)                          ▷ Rollback some values
 9          return RETRY                   ▷ The access should be retried

10  if result indicates a conflict with transaction B
11     then if choose to abort self
12             then accessStack(ℓ).UNLOCK()
13                  return XABORT
14             else accessStack(ℓ).UNLOCK()
15                  signal an abort of B
16                  return RETRY

    ▷ Otherwise, there is no conflict: X is an ancestor of Z
17  if Z ≠ X                                  ▷ Z's first access to ℓ
18     then ▷ Log the access
19          LOGVALUE(Z, ℓ)
20          accessStack(ℓ).PUSH(Z)

    ▷ Actually perform the write operation
21  Perform the write
22  accessStack(ℓ).UNLOCK()
23  return SUCCESS
```

Figure 3-7: Pseudocode instrumenting an access by u to an object ℓ, assuming that all accesses are writes. ACCESS(u, ℓ) returns XABORT if Z should abort, RETRY if the access should be retried, or SUCCESS if the memory operation succeeded.

```
CLEANUP(ℓ)
1  accessStack(ℓ).LOCK()
2  X ← accessStack(ℓ).TOP()
3  if ∃Z ∈ ances(X) such that status[Z] = ABORTED
4     then RESTOREVALUE(X, ℓ) ▷ Restore ℓ from X's log
5          accessStack(ℓ).POP()
6  accessStack(ℓ).UNLOCK()
```

Figure 3-8: Code for cleaning up an aborted transaction from the top of accessStack(ℓ), assuming all accesses are writes. If the last writer has an aborted ancestor, it should be rolled back.

structure for conflict detection is simpler, the other data structures used for conflict detection are more complicated. The analysis technique, however, is virtually identical, and hence much of the analysis is not repeated here.

Trace definition and properties

XConflict assigns computation-tree nodes to ***traces*** in the essentially the same fashion as the SP-ordered-bags data structure described in Chapter 2. I briefly describe the structure of traces here. Since the computation tree has a slightly different canonical form from the canonical Cilk parse tree used for SP-ordered-bags, XConflict simplifies the trace structure slightly by merging some traces together.

Formally, each trace U is a disjoint subset of nodes of the (*a posteriori*) computation tree. We let $\mathcal{Q}$ denote the set of all traces. $\mathcal{Q}$ partitions the nodes of the computation tree $\mathcal{C}$. Initially, the entire computation belongs to a single trace. As the program executes, traces dynamically split into multiple traces whenever steals occur.

A trace itself executes on a single processor in a depth-first manner. Whenever a steal occurs and a processor steals the right subtree of a P-node $P \in U$, the trace U splits into three traces U_0, U_1, and U_2 (i.e., $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{U_0, U_1, U_2\} - \{U\}$). Each of the left and right subtrees of P become traces U_1 and U_2, respectively. The trace U_0 consists of those nodes remaining after P's subtrees are removed from U. Notice that although the processor performing the steal begins work on only the right subtree of P, both subtrees become new traces. Figure 3-9 gives an example of traces resulting from a steal. The left and right children of the highest uncompleted P-node P_1 (both these nodes are nontransactional S-nodes in our canonical tree) are the roots of two new traces, U_1 and U_2.

Traces in CWSTM satisfy the following properties.

Property 3.4 *Every trace $U \in \mathcal{Q}$ has a well-defined **head** nontransactional S-node $S = \texttt{head}[U] \in U$ such that for all nodes $B \in U$, we have $S \in \texttt{ances}(B)$.*

For a trace $U \in \mathcal{Q}$, we use $\texttt{xparent}[U]$ as a shorthand for $\texttt{xparent}[\texttt{head}[U]]$. We similarly define $\texttt{nsParent}[U]$.

Property 3.5 *The computation-tree nodes of a trace $U \in \mathcal{Q}$ form a tree rooted at $S = \texttt{head}[U]$.*

Property 3.6 *Trace boundaries occur at P-nodes; either both children of the P-node and the node itself belong to different traces, or all three nodes belong to the same trace. All children of an S-node, however, belong to the same trace.*

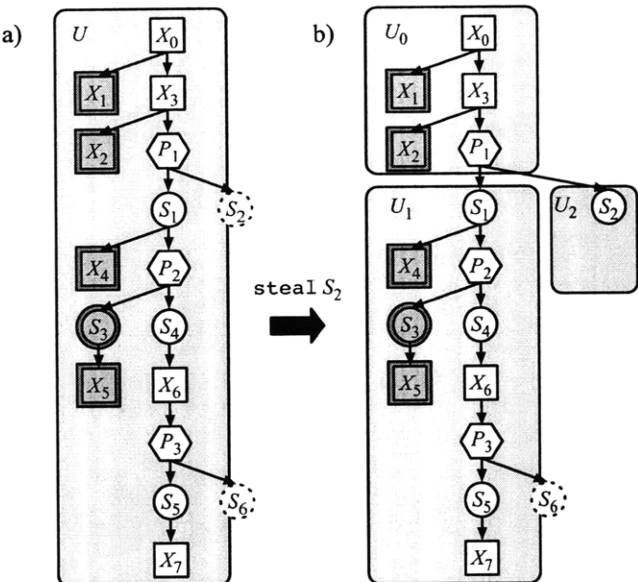


Figure 3-9: Traces of a computation tree (a) before and (b) after a `steal` action. Before the `steal`, only one processor is executing the subtree, but S_2 and S_6 are ready. After the steal, the subtree rooted at the highest ready S-node (S_2) is executed by the thief. The subtree rooted at S_1, on the other hand, is still owned and executed by the victim processor.

Property 3.7 *Trace boundaries occur at "highest" P-nodes. That is, suppose a P-node P has a stolen child (i.e., P and its children belong to different traces). Consider all ancestor P-nodes P' of P such that P is in the left subtree of P'. Then P' must have a stolen child (i.e., P and ancestor P' belong to different traces).*

The last property follows from the Cilk-like work stealing and Property 2.9.

The partition $\mathcal{Q}$ of nodes in the computation tree $\mathcal{C}$ induces a tree of traces $J(\mathcal{C})$ as follows. For any traces $U, U' \in \mathcal{Q}$, there is an edge $(U, U') \in J(\mathcal{C})$ if and only if $\texttt{parent}[\texttt{head}[U']] \in U$.[4] The properties of traces and the fact that traces partition $\mathcal{C}$ into disjoint subtrees together imply that $J(\mathcal{C})$ is also a tree.

We say that a trace U is ***active*** if and only if $\texttt{head}[U]$ is active. The following lemma states that if a descendant trace U' is active, then U' is a descendant of *all* active nodes in U. The proof relies on the fact that traces execute serially in a depth-first (or equivalently, left-to-right) manner.

Lemma 3.8 *Consider active traces $U, U' \in \mathcal{Q}$, with $U \neq U'$. Let $D \in U'$ be an active node, and suppose $D \in \texttt{desc}(\texttt{head}[U])$ (i.e., U' is a descendant trace of U). Then for any active node $B \in U$, we have $B \in \texttt{ances}(D)$.*

Proof. Since traces execute on a single processor in a depth-first manner, only a single head-to-leaf path of each trace can be active. Thus, if a descendant trace U' is active, it must be the descendant of some node along that path. In particular, we claim that U' is a descendant of the leaf, and hence it is a descendant of *all* active nodes as the lemma states. This claim follows from Property 3.6, because both children of the *active* P-node on the trace boundary must belong to different traces. □

[4]The function `parent[]` refers to the parent in the computation tree $\mathcal{C}$, not in the trace tree $J(\mathcal{C})$.

```
XCONFLICT(B, u)
    ▷ For any computation-tree node B and any
      active memory-operation u

    ▷ Test for simple base cases
 1  if trace(B) = trace(u)
 2     then return "no conflict"
 3  if some ancestor transaction of B is aborted
 4     then return "no conflict: B aborted"

 5  Let X be the nearest transactional ancestor of B
       belonging to an active trace.
 6  if X = null                                   ▷ committed at top level
 7     then return "no conflict: B committed to root"
 8  U_X ← trace(X)

 9  Let Y be the highest active transaction in U_X

10  if Y ≠ null and Y is an ancestor of X
11     then if U_X is an ancestor of u
12             then return "no conflict: B committed
                                  to u's ancestor"
13             else return "conflict with Y"
14     else Z ← xparent[U_X]
15          if Z = null or trace(Z) is an ancestor of u
16             then return "no conflict: B committed
                                  to u's ancestor"
17             else return "conflict with Z"
```

Figure 3-10: Pseudocode for the XConflict algorithm.

XConflict algorithm

Recall that CWSTM instruments memory accesses, testing for conflicts on each memory access by performing queries of XConflict data structures. In particular, XConflict must test whether a recorded access by node B conflicts with the current access by node u. Suppose that B does not have an aborted ancestor. Then recall Definition 3.3 states that a conflict occurs if only if the nearest uncommitted transactional ancestor of B is *not* an ancestor of u.

A straightforward algorithm (given in Figure 3-6) for conflict detection finds the nearest uncommitted transactional ancestor of B and determines whether this node is an ancestor of u. Maintaining such a data structure subject to parallel updates is costly (in terms of locking overheads).

XConflict performs a slightly simpler query that takes advantages of traces. XConflict does not explicitly find the nearest uncommitted transactional ancestor of B; it does, however, still determine whether that transaction is an ancestor of u. In particular, let Z be the nearest uncommitted transactional ancestor of B, and let U_Z be the trace that contains Z. Then XConflict finds U_Z (without necessarily finding Z). Testing whether U_Z is an ancestor of u is sufficient to determine whether

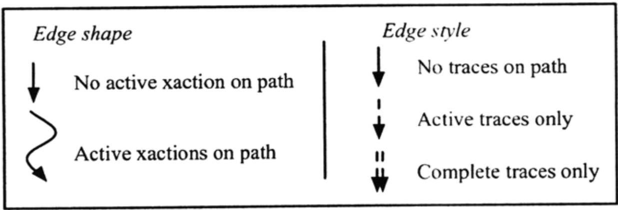


Figure 3-11: The definition of arrows used to represent paths in Figures 3-12, 3-13 and 3-14.

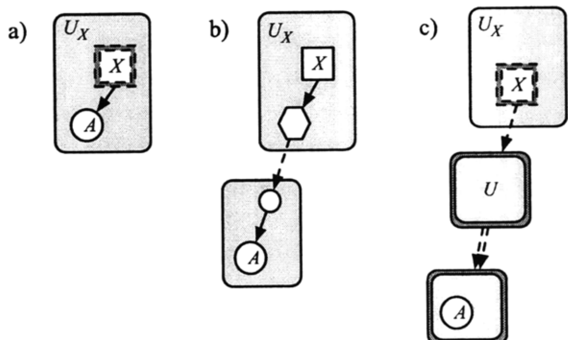


Figure 3-12: The three possible scenarios in which X is the nearest transactional ancestor of B that belongs to an active trace. Arrows represent paths between nodes (i.e., many nodes are omitted): see Figures 3-4 and 3-11 for definitions. In both (a) and (b), B belongs to an active trace. In (a), $\texttt{xparent}[B]$ belongs to the same active trace as B. In (b), $\texttt{xparent}[B]$ belongs to an ancestor trace of $\mathsf{trace}(B)$. In (c), B belongs to a complete trace, U is the highest completed ancestor trace of B, and X is the $\texttt{xparent}[U]$.

Z is an ancestor of u. Note that XConflict does not lock on any queries. Many of the subroutines (described in later sections) need only perform simple ABA tests to see if anything changed between the start and end of the query.

The XCONFLICT algorithm is given by pseudocode in Figure 3-10. Lines 1–4 handle the simple base cases. If B and u belong to the same trace, they are executed by a single processor, so there is no conflict. If B is aborted, there is also no conflict.

Suppose B is not aborted and that B and u belong to different traces. XCONFLICT first finds X, the nearest transactional ancestor of A that belongs to an active trace, in line 5. The possible locations of X in the computation tree are shown in Figure 3-12. Let $U_X = \mathsf{trace}(X)$. Notice that U_X is active, but X may be active or inactive. For cases (a) or (b), we find X with a simple lookup of $\texttt{xparent}[B]$. Case (c) involves first finding U, the highest completed ancestor trace of $\mathsf{trace}(B)$, then performing a simple lookup of $\texttt{xparent}[U]$. Section 3.8 describes how to find the highest completed ancestor trace.

Line 9 finds Y, the highest active transaction in U_X. If Y exists and is an ancestor of X, as shown in the left of Figure 3-13, then XCONFLICT is in the case given by lines 11–13. If U_X is an ancestor of u, we conclude that A has committed to an ancestor of u. Figure 3-13 (a) and (b) show the possible scenarios where U_X is an ancestor of u: either X is an ancestor u, or X has committed to some transaction Z that is an ancestor of u.

Suppose instead that Y is not an ancestor of X (or that Y does not exist), as shown in the left of Figure 3-14. Then XCONFLICT follows the case given in lines 15–17. Let Z be the transactional parent of U_X. Since X has no active transactional ancestor in U_X, it follows that X has committed

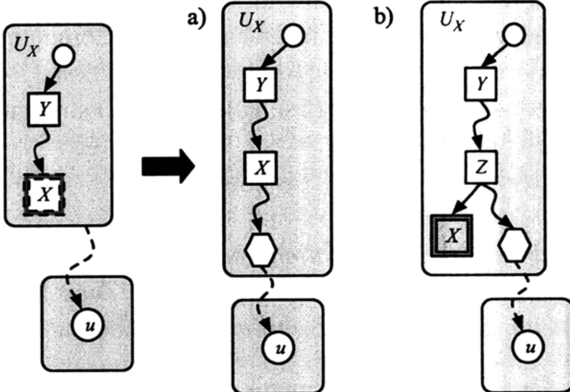


Figure 3-13: The possible scenarios in which the highest active transaction Y in U_X is an ancestor of X, and U_X is an ancestor of u (i.e., line 11 of Figure 3-10 returns true). Arrows represent paths between nodes (i.e., many nodes are omitted): see Figures 3-4 and 3-11 for definitions. The block arrow shows implication from the left side to either (a) or (b).

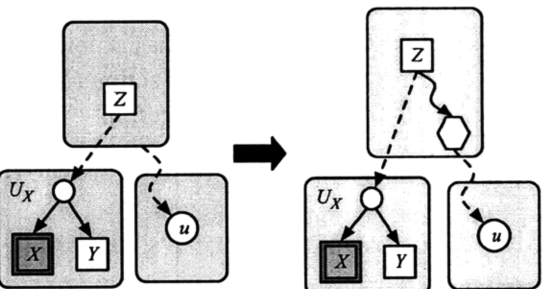


Figure 3-14: The scenario in which the highest active transaction Y in U_X is not an ancestor of X, and $Z = \mathtt{xparent}[U_X]$ is an ancestor of u (i.e., line 15 of Figure 3-10 returns true). The block arrow shows implication from the left side to the situation on the right.

to Z. Thus, if $\text{trace}(Z)$ is an ancestor of u, we conclude that A has committed to an ancestor of u, as shown in Figure 3-14.

Section 3.6 describes how to find the trace containing a particular computation-tree node (i.e., computing $\text{trace}(B)$). Section 3.7 describes how to maintain the highest active transaction of any trace (used in line 9). Section 3.8 describes how to find the highest completed ancestor trace of a trace (used for line 5), or find an aborted ancestor trace (line 3). Computing the transactional parent of any node in the computation tree (`xparent`$[B]$) is trivial. Section 3.9 describes a data structure for performing ancestor queries within a trace (line 10), and a data structure for performing ancestor queries between traces (lines 11 and 15).

The following theorem states that XConflict is correct.

Theorem 3.9 *Let B be a node in the computation tree, and let u be a currently executing memory access. Suppose that B does not have an aborted ancestor. Then* XCONFLICT(B, u) *reports a conflict if and only if the nearest (deepest) active transactional ancestor of B is an ancestor of u.*

Proof. If B has an aborted ancestor, then lines 1–2 of XCONFLICT properly returns no conflict.

Let Z be the nearest active transactional ancestor of B. Let U_Z be the trace containing Z; since Z is active, U_Z is active. Lemma 3.8 states that U_Z is an ancestor of u if and only if Z is an ancestor of u. It remains to show that XConflict finds U_Z.

XConflict first finds X, the nearest transactional ancestor of B belonging to an active trace (line 5). The nearest active ancestor of B must be X or an ancestor of X. Let U_X be the trace containing X, and let Y be the highest active transaction in U_X. If Y is an ancestor of X, then either $Z = X$, or Z is an ancestor of X and a descendant of Y (as shown in Figure 3-13). Thus, XConflict performs the correct test in lines 11–13.

Suppose instead that Y, the *highest* active transaction in U_X, is not an ancestor of X. Then no active transaction in U_X is an ancestor of X. Let Z be the transactional ancestor of U_X. Since U_X is active, Z must be active. Thus, Z is the nearest uncommitted transactional ancestor of B, and XConflict performs the correct test in lines 15–17.

In the above explanation, we assume that no XConflict data-structural changes occur concurrently with a query. The case of concurrent updates is a bit more complicated and omitted from this proof. The main idea for proving correctness subject to concurrent updates is as follows. Even when trace splits occur, if a conflict exists, XCONFLICT has pointers to traces that exhibit the conflict. Similarly, if XCONFLICT acquires pointers to a transaction (Y or Z) deemed to be active, that transaction was active at the start of the XCONFLICT execution. □

Note that XCONFLICT may return some spurious conflicts if transactions complete during the course of a query.

3.6 Trace maintenance

This section describes how to maintain trace membership for each node B in the computation tree, subject to queries $\text{trace}(B)$. The queries take $O(1)$ time in the worst case. I give the main idea of the scheme here for completeness, but I omit details as they are similar to the local-tier of SP-ordered-bags in Section 2.6.

To support trace membership queries, XConflict organizes computation-tree nodes belonging to a single trace as follows. Nodes are associated with their nearest nontransactional S-node ancestor. These S-nodes are grouped into sets, called "trace bags." Each bag b has a pointer to a trace, denoted `traceField`$[b]$, which must be maintained efficiently. A trace may contain many trace bags.

Bags are merged dynamically in a way similar to the SP-fast-bags (Section 2.2) in the local tier of SP-ordered-bags using a particular disjoint-sets data structure [93, 101]. Since traces execute on a single processor, we do not lock the data structure on update (UNION) operations. The difference in the conflict-detection setting is that we only need to use only one kind of bag (instead of two in SP-fast-bags).[5]

When steals occur, a global lock is acquired, and then a trace is split into multiple traces, as in SP-ordered-bags. The difference in the conflict-detection setting is that traces split into three traces (instead of five in SP-ordered-bags). As before, trace splits can be performed in $O(1)$ worst-case time by simply moving a constant number of bags. When the trace constant-time split completes (including the split work in Sections 3.7 and 3.9), the global lock is released.

The other details are similar to SP-ordered-bags. To query what trace a node B belongs to, we perform the operations `traceField`[FIND-BAG(`nsParent`[B])]. These queries (in particular, FIND-BAG) take $O(1)$ worst-case time as in SP-ordered-bags. Merging bags uses a UNION operation and takes $O(1)$ amortized time, but the same FAST-UNION optimization from Section 2.6 improves UNIONs to worst-case $O(1)$ time whenever the amortization might adversely increase the program's span.

Why do traces differ between SP-ordered-bags and XConflict? There are two primary differences: the number of subtraces resulting from a steal, and the use of S- and P-bags (in SP-ordered-bags) vs. just one kind of bag (for XConflict). SP-ordered-bags splits a trace into more pieces because of the use of the SP-order global tier and the type of query being performed. For XConflict, the query among traces can be answered without splitting a trace into as many pieces. Similarly, the use of S- and P-bags in SP-ordered-bags arises from the type of query being performed. In particular, the bags are serving two purposes — they are maintaining trace-membership information *and* answering local queries about series-parallel relationships. For conflict detection, the bags are used only for trace-membership information. In fact, this difference can be removed in two ways. We could modify XConflict to maintain S- and P-bags, but ignore the extra information provided by the S and P labels. Alternatively, we could modify SP-ordered-bags to use a single bag as with XConflict. The bags, however, would then only answer trace-membership queries; to query the series-parallel relationships among strands within the same trace, SP-ordered-bags would then need an additional local-tier data structure.

3.7 Highest active transaction

This section describes how XConflict finds the highest active transaction in a trace, used in line 9 of Figure 3-10 in $O(1)$ time.

For each nontransactional S-node S, we have a field $nextx[S]$ that stores a pointer to the nearest active descendant transaction of S. Maintaining this field for *all* S-nodes is expensive, so instead we maintain it only for some S-nodes as follows. Let $S \in U$ be an active nontransactional S-node such that either $S = \texttt{head}[U]$, or S is the left child of a P-node and S's nearest S-node ancestor (which is always a grandparent) is a transaction. Then $nextx[S]$ is defined to be the nearest, active descendant transaction of S in U. Otherwise, $nextx[S] = \texttt{null}$.

Finding the highest active transaction simply entails a call to $nextx[\texttt{head}[U]]$, which takes $O(1)$ time. The complication is in maintaining the $nextx$ values, especially when dynamic trace splits occur.

[5] The use of one bag here is merely a simplification. Using the SP-fast-bags algorithm as is would suffice for correctness. The only necessary modification is to make the obvious changes to deal with 3 subtraces instead of 5.

To maintain $nextx$, we keep a stack of those S-nodes in U for which $nextx$ is defined. Initially push $\texttt{head}[U]$ onto the stack. For each of the following scenarios, let S be the S-node on the top of the stack. Whenever encountering a transactional S-node X, check $nextx[S]$. If $nextx[S] = \texttt{null}$, then set $nextx[S] \leftarrow X$. Otherwise, do nothing. Whenever completing a transaction X, check $nextx[S]$. If $nextx[S] = X$, then set $nextx[S] \leftarrow \texttt{null}$. Otherwise, do nothing. Whenever encountering a nontransactional S-node S', if $nextx[S] = \texttt{null}$, do nothing. Otherwise, push S' onto the stack. Whenever completing a nontransactional S-node S', pop S' from the stack if it is on top of the stack.

Finally, XConflict maintains these $nextx$ values even subject to trace splits. Consider a split of trace U into three traces U_1, U_2, and U_3, rooted at S, S_1, and S_2, respectively. Since CWSTM steals from a highest P-node in the computation tree (Property 2.9), S_1 must be the highest, active, nontransactional S-node descendant of S that is the left child of a P-node. Recall that the structure of our computation tree is such that the only nontransactional S-nodes are children of P-nodes (or the root). Thus, either S_1 is the second S-node on U's stack, or S_1 is not on U's stack.

If S_1 is on U's stack, then $nextx[S]$ is defined to be an ancestor of S, and we leave it as such. Moreover, since S_1 is on the stack, $nextx[S_1]$ is defined appropriately. Simply split the stack into two just below S to adjust the data structure to the new traces. Suppose instead that S_1 is not on U's stack. Then the $nextx[S]$ may be a descendant of S_1 (or it is undefined). Set $nextx[S_1] \leftarrow nextx[S]$ and $nextx[S] \leftarrow \texttt{null}$. Then split the stack below S, and prepend S_1 at the top of its stack. The necessary stack splitting takes $O(1)$ worst-case time. This splitting occurs while holding the global lock acquired during the steal (as in Section 3.6).

3.8 Supertraces

This section describes XConflict's data structure to find the highest completed ancestor trace of a given trace (used as a subprocedure for line 5 in Figure 3-10, illustrated by U in Figure 3-12 (c)). To facilitate these queries, XConflict groups traces together into "supertraces." Grouping traces into supertraces also facilitates faster aborts — when aborting a transaction in trace U, we need only abort some of the supertrace children of U, not the entire subtree in $\mathcal{C}$. This section also provides some details on performing the abort.

All update operations on supertraces take place while holding the same global lock acquired during the steal (as in Sections 3.6, 3.7, and 3.9). Unlike the data structures in Sections 3.6, 3.7, and 3.9, the updates to supertraces do not occur when steals occur. To prove good performance (in Section 3.10), we use the fact that the number of supertrace-update operations is asymptotically identical to the number of steals. This amortization is similar to the global tier of SP-ordered-bags.

At any point during program execution, a completed trace $U \in \mathcal{Q}$ belongs to a ***supertrace*** $K = \mathsf{strace}(U) \subseteq \mathcal{Q}$. In particular, the traces in K form a tree rooted at some ***representative trace*** $rep[K]$, which is an ancestor of all traces in K. Our structure of supertraces is such that either $rep[\mathsf{strace}(U)]$ is the highest completed ancestor trace of U (i.e., as used by line 5 in Figure 3-10), or U has an aborted ancestor. We prove this claim in Lemma 3.10 after describing how to maintain supertraces.

Supertraces are implemented using a disjoint-sets data structure [71, Chapter 21]. In particular, we use Gabow and Tarjan's data structure that supports MAKE-SET, FIND (implementing $\mathsf{strace}(U)$), and UNION operations, all in $O(1)$ amortized time when unions are restricted to a tree structure (as they are in our case).

When a trace U is created, we create an empty supertrace for U (so $\mathsf{strace}(U) = \emptyset$). When the trace completes (i.e., at a $\texttt{join}$ operation), we acquire the global lock. We then add U to U's

supertrace (giving $\text{strace}(U) = \{U\}$). Next, we consider all child traces U' of U (in the tree of traces $J(C)$).[6] If $\texttt{head}[U']$ is ABORTED, then we skip U'. If $\texttt{head}[U']$ is COMMITTED, we merge the two supertraces with $\textsc{Union}(\text{strace}(U), \text{strace}(U'))$. Thus, for U' (and all relevant descendants), $rep[\text{strace}(U')] = rep[\text{strace}(U)] = U$. Once these updates complete, the global lock is released. Later, U's supertrace may be merged with its parents, thereby updating $rep[\text{strace}(U)]$.

A naive algorithm to abort a transaction X must walk the entire computation subtree rooted at X, changing all of X's COMMITTED descendants to ABORTED. Instead, we only walk the subtree rooted at X in U, not $\mathcal{C}$. Whenever hitting a trace boundary (i.e., $B \in U$, $D \in \texttt{children}(B)$, $D \in U' \neq U$), we set that root of the child trace (D) to be aborted and do not continue into its descendants. Thus, we enforce all descendants of B have a supertrace with an aborted representative.

The following lemma states that either the representative of U's supertrace is the highest completed ancestor trace of U, or U has an aborted ancestor.

Lemma 3.10 *For any completed trace $U \in \mathcal{Q}$, let $K = \text{strace}(U)$, and let $U' = rep[K]$. Exactly one of the following cases holds.*

1. *Either* $\texttt{head}[U']$ *is ABORTED, or*
2. $\texttt{head}[U']$ *is COMMITTED,* $\text{trace}(\texttt{parent}[\texttt{head}[U']])$ *is active, and there is no ABORTED transaction between* $\texttt{head}[U]$ *and* $\texttt{head}[U']$.

Proof. We claim also that U' is an ancestor of all traces in K. We prove this claim and the lemma inductively on the (tree of traces) height of the highest committed ancestor of U.

For a base case, consider the moment when U completes. Then K contains only descendants of U, and $rep[K] = U$ satisfying our claims.

Suppose that U's highest committed ancestor occurs at height h in the tree of traces, and assume that $U' = rep[K]$ satisfies the lemma. Let V be height-$(h+1)$ ancestor of U. Then we show that the lemma is maintained when V commits. We divide the proof into the two cases.

Suppose $\texttt{head}[U']$ is ABORTED. Since U' is an ancestor of all traces in K, the only time U' can be merged with another trace is when its parent completes. ABORTED children, however, are not unioned. Thus, once the representative U' of a supertrace is aborted, the supertrace does not change.

Suppose instead that $\texttt{head}[U']$ is COMMITTED. Then, when V commits, XConflict unions the two supertraces for U' and V, and then updates that supertrace's representative to V. Thus, $rep[\text{strace}(U)]$ is the highest completed ancestor trace of U. This trace may be COMMITTED or ABORTED, but either satisfies the claim. □

3.9 Ancestor queries

This section describes how XConflict performs ancestor queries. XConflict performs a "local" ancestor query of two nodes belonging to the same trace (line 10 of Figure 3-10) and a "global" ancestor query of two different traces (lines 11 and 15 of Figure 3-10). Both of these queries can be performed in $O(1)$ worst-case time. The global lock is acquired only on updates to the global data structure, which occurs on trace splits (i.e., steals).

[6] Maintaining a list of all child traces is not difficult. We keep a linked list for each node in the trace tree and add to it whenever a trace splits.

Local ancestor queries

CWSTM executes a trace on a single processor, and each trace is executed in depth-first, left-to-right order (as stated in Property 2.9). We thus view a trace execution as a depth-first execution of a computation (sub)tree (or a depth-first tree walk).

To perform ancestor queries on a depth-first walk of a tree, we associate with each tree node u the ***discovery time*** $d[u]$, indicating when u is first visited (i.e., before visiting any of u's children), and the ***finish time*** $f[u]$, indicating when u is last visited (i.e., when all of u's descendants have finished). (This same labeling appears in depth-first search in [71, Section 22.3].) These timestamps are sufficient to perform ancestor queries in constant time.

In the context of XConflict, we simply need associate a "time" counter with each trace. Whenever a trace splits, this counter's value is copied to the new traces.

Global ancestor queries

Since the computation tree does not execute in a depth-first manner, the same discovery/finish time approach does not work for ancestor queries between traces. Instead, we keep two total orders on the traces dynamically using order-maintenance data structures [33, 81]. These two orders give us enough information to query the ancestor-descendant relationship between two nodes in the tree of traces. These total orders are updated while holding the global lock acquired during the steal, as in Sections 3.6 and 3.7.

This global ancestor-query data structure resembles the SP-order data structure for the SP-maintenance problem (Section 2.1) and the global tier of SP-ordered-bags (Section 2.5). In particular, we maintain a left-to-right order and a right-to-left order of the tree of traces. In our left-to-right order, a node U precedes (all the nodes in) the left subtree of U, which precedes (all the nodes in) the right subtree of U. In other words, we order the nodes (traces) in the order they are output in a preorder tree walk that visits the children of a node in left-to-right order. In our right-to-left order, a node U precedes its descendants, but now U's *right* subtree precedes U's *left* subtree. This ordering corresponds to a preorder tree walk that visits the children of a node in right-to-left order. These orderings are analogous to English and Hebrew orderings, respectively, except that there is notion of P- or S-nodes here.

Let $<_L$ denote precedence in the left-to-right order. That is, $u <_L v$ means that u precedes v in the left-to-right order. Similarly, $>_R$ denotes precedence in the right-to-left order. Then the following lemma states how to perform ancestor queries.

Lemma 3.11 *Let U and V be two nodes in a tree. Then, U is an ancestor of V if and only $U <_L V$ and $U <_R V$.* □

XConflict uses order-maintenance data structures O_L and O_R to maintain the left-to-right and right-to-left orderings, respectively, subject to dynamically created traces. Since these traces can be created in parallel by multiple processors during steals, we obtain a global lock during steals. Whenever a trace U splits into three traces U_0, U_1, and U_2, we obtain the lock and perform insert operations into each ordering. (The traces U_0 and U_2 are added, whereas U_1 holds everything else that used to be part of U.) Notice that the traces U_0 and U_2 are the parent and right sibling, respectively, of the existing trace $U = U_1$. The ordering of these nodes is U_0, U_1, U_2 in O_L and U_0, U_2, U_1 in O_R.

One added difficulty is that at the time of the split, the resulting U_1 may already have several

descendant traces.[7] At the time a trace is U is created, we insert both U and a placeholder $U^{(r)}$ in both lists such that U and $U^{(r)}$ surround all of U's descendants.[8] The placeholders increase the number of inserted elements by a constant factor (2), and hence have no affect on the asymptotic analysis. Consider a split of U. At the time of the split, O_L and O_R each contain U followed immediately by all of U's descendants followed by $U^{(r)}$. These existing nodes serve the purpose of the new U_1 resulting from the split, and we need to insert the nodes for the parent U_0 and the right sibling U_2. In the left-to-right ordering O_L, insert U_0 before U and $U_2, U_2^{(r)}, U_0^{(r)}$ after $U^{(r)}$, so the order reads $U_0, U, \ldots, U^{(r)}, U_2, U_2^{(r)}, U_0^{(r)}$. In the right-to-left ordering O_R, insert $U_0, U_2, U_2^{(r)}$ before U and $U_0^{(r)}$ after $U^{(r)}$, so the ordering reads $U_0, U_2, U_2^{(r)}, U, \ldots, U^{(r)}, U_0^{(r)}$.

Since the global ancestor-query data structure resembles the global tier of SP-ordered-bags (Section 2.5), I omit further details of the data structure here. As in SP-ordered-bags, each query has a worst-case cost of $O(1)$, and trace splits have an amortized cost of $O(1)$.

3.10 Performance claims

The section bounds the running time of an CWSTM program in the absence of conflicts. The bound includes the time to check for conflicts *assuming that all accesses are writes* and to maintain the relevant data structures. Checking for conflicts with multiple readers, however, increases the runtime. Additionally, aborts add more work to the computation. Those slowdowns are not included in the analysis.

The following theorem states the running time of an CWSTM program under nice conditions. We give bounds for both Cilk's normal randomized work-stealing scheduler, and for a round-robin work-stealing scheduler (as in Section 2.8). The proof technique here is similar to the proof of performance of SP-ordered-bags in Section 2.8, and hence we omit the details of the proof.

Theorem 3.12 *Consider an CWSTM program with T_1 work and span T_∞ in which all memory accesses are writes. Suppose the program, augmented with XConflict, is executed on P processors and that no transaction aborts occur.*

1. *When using a randomized Cilk-like work-stealing scheduler, the program runs in $O(T_1/P + P(T_\infty + \lg(1/\varepsilon)))$ time with probability at least $1 - \varepsilon$, for any $\varepsilon > 0$,*
2. *When using a round-robin Cilk-like work-stealing scheduler, the program runs in $O(T_1/P + PT_\infty)$ worst-case time.*

One way of viewing these bounds is as the overhead of XConflict algorithm itself. These bounds nearly match those of a Cilk program without XConflict's conflict detection. The only difference is that the T_∞ term is multiplied by a factor of P. In most cases, we expect $PT_\infty \ll T_1/P$, so these bound represents only constant-factor overheads beyond optimal. We would also expect the first bound (using the randomized scheduler) to have better constants hidden in the big-O.

These XConflict bounds translate to bounds on completion time of an CWSTM program under optimistic conditions. For illustration, consider a program where all concurrent paths access disjoint sets of memory. The overhead of maintaining the XConflict data structures is $O(T_1/P + PT_\infty)$.

[7] In contrast, the SP-ordered-bags algorithm essentially splits traces only at leaves in the tree of traces, so there are no descendant traces to worry about. The main reason for this difference is our different definition of traces.

[8] The use of an explicit placeholder describes an Euler tour of the tree. A single Euler tour is sufficient for performing the desired ancestor queries — we do not need a left-to-right *and* right-to-left Euler tour. Thus, there are two equivalent approaches here. At first glance, using a single Euler tour appears more space efficient; the $U^{(r)}$ nodes, however, need not be explicitly maintained when maintaining two orderings, so both approaches are identical in terms of space. We consider the two-ordering approach here as it is more similar to SP-order earlier in this thesis.

Each memory access queries the XConflict data structure at most once. Since each query requires only $O(1)$ time, the entire program runs in $O(T_1/P + PT_\infty)$ time.

The CWSTM design we describe does not provide any reasonable performance guarantees when we allow multiple readers. There are two reasons for this problem. First, concurrent reads to an object may contend on the access stack to that object. Second, even in the case where concurrent read operations never wait to acquire an access stack lock, it appears that write operation may need to check for conflicts against potentially many readers in a reader list (some of which may have already committed). Therefore, a write operation is no longer a constant time operation, and it seems the work of the computation might increase proportional to the number of parallel readers to an object. It is part of future work to improve the CWSTM design and analysis in the presence of multiple readers.

3.11 Concluding remarks

The CWSTM model describes one approach for implementing a software transactional memory system that supports transactions with nested fork-join parallelism. CWSTM design was guided by a few major goals.

- Supporting nested transactions of unbounded depth.
- Small overhead when there are no aborts.
- Avoid asymptotically increasing the work or the critical path of the computation too much.

I believe that we have achieved these goals to some extent, since the CWSTM guarantees provably good completion time in the case when there are no aborts, and there are no concurrent readers (or all accesses are treated as writes).

I believe that supporting unbounded nesting depth in transactions is important for composability of programs. If a function is called from inside a transaction, the caller should not have to worry about how many transactions are nested inside the function call. It may be true, however, that the common case is transactions of small depth. In this case, a simpler design like the one described in Section 3.3 might be sufficient in the common case, at the expense of a slowdown in the case when the nesting depth is large. It is difficult, however, to conclude what the common case is, since there are currently few examples of programs with nested parallel transactions. An important part of future research would be to write series-parallel programs with nested transactions to understand what the common case is, and what one should optimize for.

CWSTM is a TM system design and has not been implemented yet. As described in this chapter, each memory access may potentially require multiple data structure queries. CWSTM also may have a large memory footprint. Due to lazy cleanup on aborts, and fast commits, the access stack for an object may grow and require space proportional to the number of accesses to that object. Also, access stacks may contain pointers to transaction logs that persist long after the transactions are committed or aborted. Thus, a computation's memory footprint can become quite large. In practice, implementing a separate, concurrent thread for "garbage-collection" of metadata may help. As part of future work, we would like to implement the system in the Cilk runtime system to evaluate its practical performance and explore ways to optimize the implementation.

It would be interesting to see if CWSTM-like mechanisms are useful for high-performance languages like Fortress [12] and X10 [88]. Both these languages support transactions and fork-join parallelism. The language specification for Fortress also permits nested parallel transactions. These are richer languages than Cilk, however, and may require more complicated mechanisms to support nested parallel transactions.

Chapters 2–3 considered SP-ordered-bags and XConflict as algorithms that take an input fork-join program (in a canonical form such as the parse tree or computation tree). When implementing these algorithms in practice, one would instead modify the runtime system to incorporate SP-ordered-bag or XConflict. Fortunately, only a handful of places in the runtime system need to call into the SP-ordered-bag (or XConflict) data structures. In particular, the runtime system should be modified on memory accesses, steals, forks, joins, transaction begin and ends, etc. The bulk of the remainder of the runtime system remains unchanged.

Chapter 4

Scheduling Under Uncertainty

This chapter concerns approximation algorithms for multiprocessor scheduling under uncertainty, first introduced in [146]. This chapter represents joint work with Christopher Y. Crutchfield, Zoran Dzunic, David R. Karger, and Jacob H. Scott, previously appearing in [73].

The model for multiprocessor scheduling under uncertainty extends the classical construction of machine scheduling to handle cases where machines run jobs for discrete timesteps and succeed in processing them only probabilistically. The motivation for studying this problem stems from the increasing use of distributed computing to handle large, computationally intensive tasks. Projects like Seti@Home [19] divide computations into smaller jobs of relatively uniform length, which are then executed on unreliable machines (e.g., of volunteers).

Scheduling multiple machines to process the same job at once can help overcome the problem of unreliable machines, but too many machines processing a single job can also slow down overall throughput. The situation is exacerbated when precedence constraints among jobs are present, which is often the case for sophisticated computations. Here, a single job failing may delay the start of many others. Note that the special case of having no precedence constraints retains practical significance. Google's MapReduce architecture [78], for example, generates jobs whose dependencies form a complete bipartite graph, which is equivalent to two phases of independent jobs.

Motivated by these examples, this chapter studies the ***multiprocessor scheduling under uncertainty*** (SUU) problem. An SUU instance is comprised of a set of n unit-time jobs and a set of m machines. For each machine i and job j, we are given a failure probability q_{ij}, which is the chance that job j does not complete when run on machine i for a single timestep. Any precedence constraints are modeled as a directed acyclic graph (dag). The objective is to construct a schedule assigning machines to eligible jobs at each timestep, minimizing the expected time until all jobs have successfully completed. In contrast to many other scheduling problems, SUU allows multiple machines to execute the same job in a single timestep.

When SUU is properly reformulated (in Section 4.1), it is strikingly similar to the problem of minimizing makespan on unrelated machines in a stochastic setting, where job lengths set according to random variables. Indeed, the algorithms presented in this chapter can apply to this family of problems, attaining similar asymptotic approximation ratios.

Related work

Malewicz's initial presentation of SUU [146] includes a polynomial-time dynamic-programming solution for instances where both the number of machines and the width of the precedence dag are constant. If either of these constraints is relaxed, he proves that the problem becomes NP-hard. Furthermore, when both constraints are removed, there is no polynomial-time approximation

algorithm for the problem achieving an approximation ratio lower than 5/4, unless $P = NP$. This work does not include approximation algorithms for the general (NP-hard) problem.

Lin and Rajaraman present the first (and, to date, only other) approximation algorithms for SUU [141]. Using a greedy algorithm to maximize the chance of success across all jobs, they give an $O(\log n)$-approximation when all jobs are independent. More sophisticated techniques, including LP-rounding and random delay [139, 180], yield a variety of $O(\text{poly}\log(n+m))$ approximations when precedence graphs are constrained to form only disjoint chains, collections of in- or out-trees, and directed forests. For these settings, our algorithms improve their approximation ratios by a $(\log n / \log\log n)^2$ factor, when $n = m^{\Theta(1)}$. See Figure 4-1 for a complete comparison.

The wider field of machine scheduling is an established and well-studied area of research with a large number of variations on its core theme (see [127] for a survey). There are three main differences between SUU and problems studied in the literature. First, in SUU, jobs may run on multiple machines in the same timestep. Second, each job has a chance of failing to complete on any machine that processes it. Third, jobs must be scheduled at unit granularity.

I am not aware of many scheduling problems in which jobs may run on multiple machines at the same time. Interestingly, Serafini, motivated by scheduling looms in the textile industry, provides a polynomial-time algorithm [178] for scheduling independent jobs on unrelated machines, given that jobs can be split arbitrarily and run independently on different machines. This problem is quite close to a deterministic analog of our model (with independent jobs), but allows arbitrarily small splits (as opposed to unit-step allocations), yielding a simpler linear-programming solution that is not applicable to our problem.

Of deterministic scheduling problems, SUU most closely resembles the problem of preemptively scheduling jobs with precedence constraints on unrelated parallel machines so as to minimize makespan. In Graham's notation [109], this problem is written as $R|\,prec, pmtn\,|C_{\max}$. Instead of failure probabilities q_{ij}, there is a deterministic processing time p_{ij}, denoting how long it takes for machine i to complete job j. In contrast to SUU, however, machines never fail, and jobs may only run on one machine at a time. As in [141], techniques for this problem [136, 139, 180] play an important role in the approximations this chapter. Solutions here also borrow techniques from "job-shop scheduling" [96] for our SUU algorithms for precedence constraints, but the particulars of that setting are not very similar to the ones considered here. In the case of $R|\,prec, pmtn\,|C_{\max}$, I am only aware of approximations for precedence constraints forming disjoint chains or trees.

There is also a large body of work on stochastic scheduling (see [168, Part 2] for a representative sample). The majority of the work in this area considers how to schedule jobs whose input lengths are not known, but instead given as random variables distributed according to some probability distribution. In this problem, there are no failure probabilities. Particular attention has been paid to the case when these distributions are restricted to exponential families. Section 4.1 shows that SUU is closely related to the problem of preemptively scheduling precedence-constrained jobs whose processing times are given by exponential distributions, on unrelated parallel machines so as to minimize their expected makespan. In stochastic scheduling, this problem is known as $R|\,pmtn, prec, p_j \sim stoch\,|\mathrm{E}[C_{\max}]$. I know of no previous nontrivial approximation algorithms for this problem, although an optimal algorithm exists when the machines are related and jobs are independent [196].

Contributions

This chapter describes improved approximation algorithms for SUU when jobs are independent (there are no precedence constraints) and when the precedence constraints form disjoint chains. All of these algorithm represent joint work with Christopher Y. Crutchfield, Zoran Dzunic, David

Precedence Constraints	Lin and Rajaraman [141]	This work
Independent	$O(\log(n))$	$O(\log\log(\min\{m,n\}))$
Disjoint Chains	$O\left(\frac{\log(m)\log(n)\log(n+m)}{\log\log(n+m)}\right)$	$O(\log(n+m)\log\log(\min\{m,n\}))$
Directed Forests	$O\left(\frac{\log(m)\log^2(n)\log(n+m)}{\log\log(n+m)}\right)$	$O(\log(n+m)\log(n)\log\log(\min\{m,n\}))$

Figure 4-1: Improved approximation ratios

R. Karger, and Jacob H. Scott, previously appearing in [73]. For independent jobs, I describe an $O(\log\log(\min\{m,n\}))$-approximation algorithm. One component of this algorithm is based on an LP relaxation.

The analysis for independent jobs relies on a competitive analysis as defined in [181]. Essentially, I show that our algorithm is $O(\log(p_{\max}/p_{\min}))$-competitive for a deterministic scheduling problem (similar to $R|\,pmtn\,|C_{\max}$) in which each machine has a deterministic speed, but processing times for jobs are chosen arbitrarily by an adversary, with minimum and maximum values $p_{\min}$ and $p_{\max}$. This competitive result is interesting in its own right.

When the precedence constraints on jobs form a collection of disjoint chains, Section 4.3 gives an $O(\log(n+m)\log\log(\min\{m,n\}))$-approximation algorithm. This disjoint-chains algorithm uses an LP relaxation similar to the one used for independent jobs and also utilizes techniques from network-flow theory and prior work on the SUU problem for chains [141]. The $\log\log(\min\{m,n\})$ factor arises from the independent-jobs algorithm — improving that independent-jobs algorithm, therefore, immediately yields a better algorithm for chains.

The algorithm for disjoint chains can be extended to handle directed forests using the chain-decomposition techniques of [136, 141], yielding an $O(\log(n+m)\log(n)\log\log(\min\{m,n\}))$-approximation.

I also show how to apply these algorithms to similar variants in the problem of stochastic scheduling, where jobs have stochastic processing times. To the best of my knowledge, these are the first approximation algorithms for stochastic scheduling with the expected-completion-time objective and *unrelated* machines.

Chapter outline

Section 4.1 gives formal definitions of the SUU problem, scheduling algorithms, and an equivalent formulation of SUU that plays an important role in the approximation algorithms provided later in the chapter. Section 4.2 presents an algorithms for independent jobs, and Section 4.3 shows how to extend the algorithm to handle precedence constraints forming chains. I address tree-like precedence constraints and stochastic scheduling in Sections 4.4 and 4.5, respectively.

4.1 Preliminaries

This section gives of a formal statement of the scheduling problem and defines a schedule. Most of the notation is consistent with that of Malewicz [146] or Lin and Rajaraman [141]. One minor difference is that I use "failure probabilities" q_{ij} in lieu of "success probabilities" p_{ij}, for ease of notation. This section next includes a reformulation of the SUU, which is used in subsequent sections to simplify both the algorithms and the analyses involved.

The SUU problem

An instance $I = (J, M, \{q_{ij}\}, G)$ of the SUU problem includes a set J of unit-step jobs and a set M of machines. Throughout this chapter, let $n = |J|$ be the number of jobs and $m = |M|$ be the number of machines. For each machine i and job j, we are given a ***failure probability*** q_{ij}, which is the probability that job j *does not* complete when run on machine i for one unit step; these probabilities are independent. In addition, without loss of generality, assume that for each job j, there exists a machine i such that $q_{ij} < 1$.

An SUU instance also includes a set of precedence constraints comprising a directed acyclic graph (dag) G with jobs as vertices. A job j is ***eligible*** for execution at time t if all jobs preceding j (i.e, jobs having a directed path to j) in the dag have successfully completed before time t. If a job j is eligible at time t, a schedule may assign multiple machines $M_{jt} \subseteq M$ to execute j in parallel. As all machine/job failures are independent, the probability that j *does not* complete in that timestep is $\prod_{i \in M_{jt}} q_{ij}$.

Failure probabilities are difficult to work with because they multiply. Instead, the ***log failure*** of job j on machine i, denoted by ℓ_{ij}, is defined by $\ell_{ij} = -\lg q_{ij}$. By definition, $q_{ij} = 1/2^{\ell_{ij}}$, and hence $\prod_{i \in M_{jt}} q_{ij} = 1/2^{\sum_{i \in M_{jt}} \ell_{ij}}$.

This chapter focuses on finding scheduling algorithms that minimize expected makespans for restricted classes of precedence constraints. If there are no precedence constraints, then the jobs are ***independent***, and we refer to the problem as ***SUU-I***. When the precedence constraints form a collection of disjoint chains, we call the problem ***SUU-C***. When the constraints form a collection of disjoint trees, we call the problem ***SUU-T***. A sans-serif fonts to refer to problem variants, whereas serif fonts refer to algorithms/schedules for the problem.

Schedules

A ***schedule*** Σ is a policy for assigning machines to (uncompleted) jobs. Jobs must be scheduled at a unit granularity, but the schedule may assign multiple machines to the same job. A schedule may base its decisions on any of its history, but this chapter considers only schedules that can be computed in polynomial time. More formally, a schedule is a function $\Sigma : (H \times \mathbb{N}) \to (M \to J \cup \{\perp\})$ that, given a history[1] $h \in H$ and time $t \in \mathbb{N}$, returns a function assigning machines to jobs. The symbol $\perp$ to indicate that the machine remains idle. To allow for more concise schedules, the assignment function returned by $\Sigma(h, t)$ may map a machine to a job that has already completed.

An ***execution*** of Σ is defined as follows. Suppose that $h \in H$ is the history of the execution up to time t. Then Σ assigns machine i to job $j = \Sigma(h, t)(i)$ at step t. If j has been completed when it is scheduled to run, i is assigned to $\perp$. Since J, M, $\{q_{ij}\}$, and G are invariant over a problem instance, we allow Σ to reference those implicitly.

Whenever the schedule Σ is such that it assigns machines to jobs depending only on the current time and the initial set of jobs, not the jobs that have completed (i.e., for all t, $\Sigma(h, t) = \Sigma(h', t)$ for all $h, h' \in H$), then the schedule is said to be ***oblivious***. An oblivious schedule has finite length if it is only defined for $t \leq t_o$, for some t_o.

A schedule is ***semioblivious*** if it can be decomposed into "rounds" such that the assignments within each round are characterized by finite oblivious schedules. Thus, while executing a step contained in a particular round, the assignment of machines to jobs depends only on the initial set

[1] A full history for a deterministic schedule can be captured by the sets of remaining jobs at each timestep prior to the current timestep t. More formally, let $H_t = \{\langle S_1, S_2, \ldots, S_t \rangle | J = S_1 \supseteq S_2 \supseteq \cdots \supseteq S_t\}$ denote the set of all feasible ordered sets of remaining jobs at timesteps $1, 2, \ldots, t$. Then valid histories are given by the set $H = \bigcup_{t=1}^{\infty} H_t$. Note that compact representations of the history exist, so a polynomially computable schedule may consider the entire history.

of jobs when the round began and the number of steps the round has been running. All oblivious schedules are naturally semioblivious, whereas the converse is not necessarily true. The schedule given later in this chapter for SUU-I is semioblivious.

Malewicz [146] also defines ***regimens*** as schedules having assignment of machines to jobs dependent only on the subset of jobs remaining. Although regimens appear to be the most intuitive form of schedules for the SUU problem, none of the schedules given in this chapter are regimens.

Let T_Σ be a random variable denoting the length of the execution of schedule Σ, which is the number of steps before all jobs have completed. The objective is to minimize $\mathrm{E}\left[T_\Sigma\right]$ (denoted by $\mathrm{E}\left[C_{\max}\right]$ in much of the scheduling literature). Throughout this chapter, Σ_{OPT} denotes a schedule that has minimum expected makespan, and its expected makespan, which is finite [146], is denoted by $\mathrm{E}\left[T_{\mathrm{OPT}}\right]$. For any SUU instance, Σ_{OPT} exists, and can be computed (inefficiently) by selecting the assignment of jobs to machines on a particular timestep that minimizes the expected makespan of the remaining jobs.

This chapter considers schedules that are polynomial-time computable (in the variables n, m, and $\log \mathrm{E}\left[T_{\mathrm{OPT}}\right]$) and whose expected makespans approximate $\mathrm{E}\left[T_{\mathrm{OPT}}\right]$. We say that Σ is an ***α-approximation*** if $\mathrm{E}\left[T_\Sigma\right] \leq \alpha \mathrm{E}\left[T_{\mathrm{OPT}}\right]$ for all choices of probabilities $\{q_{ij}\}$.

Throughout the remainder of this chapter, the terms algorithm and schedule are interchangeable. Moreover, I generally do not give the schedule explicitly as a function assigning machines to jobs. Instead, I describe it algorithmically.

Problem reformulation

I now describe a new, and equivalent formulation of the SUU problem, which I refer to as SUU*. Because of their equivalence, I refer to both problems as SUU later in the chapter.

An SUU* instance $I = (J, M, \{q_{ij}\}, G)$ has the same structure as an SUU instance. The difference is that rather than considering the success or failure of a job as it runs on machines in each timestep, use the Principle of Deferred Decisions [158] to view the problem as one of deterministically scheduling jobs with randomly distributed lengths.

Instead of failure probability, in SUU*, view $\ell_{ij} = -\log q_{ij}$ as an amount of ***work*** that a machine does towards a job completion in each unit timestep. As in SUU, machines must be scheduled at a unit granularity. At the start of a schedule's execution, draw for each job j a single random variable p_j chosen from an exponential distribution with rate parameter $\lambda_j = \ln 2$; specifically, $\Pr\{p_j \leq c\} = 1 - 2^{-c}$. A job j completes when the total work accrued by j exceeds p_j. In other words, j completes at the first step t in which $\sum_{k=1}^{t} \sum_{i \in M_{jk}} \ell_{ij} \geq p_j$.

A schedule is oblivious to the random values p_j. Instead, it is only aware of whether a job completes in each timestep. Thus, a schedule must make its decisions for assignments in step t based only on the surviving set of jobs at each previous timestep. Hence, the same schedule may be applied to both SUU and SUU*. In fact, a particular schedule has the same distribution of remaining jobs at each timestep in both SUU and SUU*, and hence both problem formulations are equivalent.

Proof of equivalence

Here, I prove that SUU and SUU* are equivalent. First, we define a history of an execution. Then, I prove the main theorem, which shows that SUU and SUU* have the same distribution over histories.

Consider an execution of a schedule Σ on SUU or SUU*. Define the ***history*** of the execution after $t-1$ steps by a sequence of job subsets $\langle S_1, S_2, \ldots, S_t \rangle$, with $J = S_1 \supseteq S_2 \supseteq \cdots \supseteq S_t$, where S_k is the set of jobs remaining at the start of step t.

Theorem 4.1 *Consider the $(t-1)$-step execution histories h_t and h_t^* of schedule Σ on SUU-instance $I = (J, M, \{q_{ij}\}, G)$ and corresponding SUU*-instance $I^* = (J, M, \{q_{ij}\}, G)$, respectively. For any particular $(t-1)$-step history x_t, we have* $\Pr\{h_t = x_t\} = \Pr\{h_t^* = x_t\}$.

Proof. By induction on time t. Initially, $\Pr\{h_1 = \langle J \rangle\} = \Pr\{h_1^* = \langle J \rangle\} = 1$.

Suppose that both executions have the same $(t-1)$-step history x_t, and let M_{jt} be the set of machines assigned to job j at step t by the schedule Σ given the history x_t. We need only show that both SUU and SUU* have the same distribution over remaining jobs after step t.

Let S_{t+1} and S_{t+1}^* be the random variables denoting the set of remaining jobs at the start of step t, for SUU and SUU* executions, respectively. For the SUU execution, the probability that job j fails to complete in step t is given by $\Pr\{j \in S_{t+1} | x_t\} = \prod_{i \in M_{jt}} q_{ij}$.

Consider now the probability that the SUU* execution fails to complete a remaining job j in step t. For job j, let $z_{j,k} = \sum_{i \in M_{jk}} \ell_{ij}$ be the amount of work j receives at step k. By definition, j fails to complete if $\sum_{k=1}^{t} z_{j,k} < p_j$. By assumption, since j remains at step t, we have $\sum_{k=1}^{t-1} z_{j,k} < p_j$. Moreover, since the exponential distribution of p_j is memoryless, we have

$$\Pr\left\{ \sum_{k=1}^{t} z_{k,j} < p_j \middle| \sum_{k=1}^{t-1} z_{k,j} < p_j \right\} = \Pr\left\{ \sum_{i \in M_{jt}} \ell_{ij} < p_j \right\}$$

Thus, we have

$$\begin{aligned} \Pr\{j \in S_{t+1}^* | x_t\} &= \Pr\left\{ \sum_{i \in M_{jt}} \ell_{ij} < p_j \right\} \\ &= 2^{-\sum_{i \in M_{jt}} \ell_{ij}} \\ &= \prod_{i \in M_{jt}} q_{ij}, \end{aligned}$$

and hence $\Pr\{j \in S_{t+1} | x_t\} = \Pr\{j \in S_{t+1}^* | x_t\}$.

Since all jobs have the same distribution given the history, we conclude $\Pr\{S_{t+1} = S | x_t\} = \Pr\{S_{t+1}^* = S | x_t\}$. Applying the inductive hypothesis (i.e., $\Pr\{h_t = x_t\} = \Pr\{h_t^* = x_t\}$) completes the proof. □

4.2 Independent jobs

This section describes an $O(\log\log n)$-approximation algorithm for SUU-I, the SUU problem with independent jobs. I first give an oblivious $O(\log n)$-approximation algorithm for SUU-I, based on scheduling an (approximation of an) integer linear program. I then modify this algorithm into a semioblivious solution consisting of $O(\log\log n)$ nearly optimal rounds.

An oblivious $O(\log n)$-approximation

I now describe an $O(\log n)$-approximation for SUU-I, called SUU-I-OBL. The main approach is to construct a schedule of length $O(\mathrm{E}[T_{\mathrm{OPT}}])$, based on an integer linear program, such that each job has no more than a constant probability of failure upon completion. This finite oblivious schedule is repeated until all jobs have completed. Using Chernoff bounds, we conclude that the expected number of repetitions is $O(\log n)$, yielding an $O(\log n)$-approximation.

Use the following integer linear program for SUU-I-OBL. Let x_{ij} denote the number of steps during which machine i is assigned to job j. Recall $\ell_{ij} = -\log q_{ij}$ is the log failure of job j on machine i. Let L_j be a nonnegative real, representing a target work for each job. To understand the integer program, think of L_j as being fixed at $L_j = 1$ for all jobs j. The better algorithm, SUU-I, will assign $L_j = 0$ to jobs that do not need to be scheduled, and increasing values to L_j for the semioblivious $O(\log \log n)$-approximation.

$$
\begin{aligned}
\textbf{(LP1)} \qquad & \min t & & \\
\text{s.t.} \sum_{i \in M} \ell_{ij} x_{ij} &\geq L_j \quad \forall j \in J & \qquad (4.1)\\
\sum_{j \in J} x_{ij} &\leq t \quad \forall i \in M & \qquad (4.2)\\
x_{ij} &\in \mathbb{N} \cup \{0\} \quad \forall i \in M, j \in J\,. & \qquad (4.3)
\end{aligned}
$$

Here Equation (4.1) enforces that every job in J has a failure probability no greater than 2^{-L_j} (or $1/2$ when $L_j = 1$), and Equation (4.3) guarantees that all jobs are scheduled for an integral number of steps on each machine. The notation (LP1) refers to this integer linear program generically, and $LP1(J', L)$ refers to it with $L_j = L$ if $j \in J'$, and $L_j = 0$ otherwise. The optimal value for $LP1(J', L)$ is denoted by $t_{LP1(J',L)}$.

A solution for $LP1(J', L)$ naturally generalizes to a finite oblivious schedule, denoted by $\Sigma_{LP1(J',L)}$, with length $t_{LP1(J',L)}$ as follows. Consider a machine i, and consider each job j in arbitrary order. Assign machine i to job j for x_{ij} timesteps. To finish my description of this schedule, I first claim that $t_{LP1(J,1)}$ approximates $\mathrm{E}[T_{\mathrm{OPT}}]$. Then, I show how to approximate (LP1) in polynomial time.

Lemma 4.2 *Let $t_{LP1(J,1)}$ denote the optimal value for $LP1(J, 1)$. Then, we have $t_{LP1(J,1)} = O(\mathrm{E}[T_{\mathrm{OPT}}])$.*

Proof. Consider any subset $U \subseteq J$, and its complement $\overline{U}$. Then $t_{LP1(U,1)} + t_{LP1(\overline{U},1)} \geq t_{LP1(J,1)}$, since we can construct a solution to $LP1(J, 1)$ by adding a solution to $LP1(U, 1)$ and a solution to $LP1(\overline{U}, 1)$.

Recall our view of the problem in terms of SUU*: there is a p_j chosen from an exponential distribution for each job j such that job j completes only if $\sum_{i \in M} \ell_{ij} x_{ij} \geq p_j$. For any sample from the event space, let U be the set of jobs j for which $p_j > 1$, and let $\overline{U}$ be the complement set of jobs j for which $p_j < 1$ (note $p_j = 1$ with probability 0 so can be ignored). By definition, each job belongs to U independently with probability $1/2$. Next, observe that whatever Σ_{OPT} is, it must allocate at least 1 unit of work to each job in U; in other words, Equation (4.1) of (LP1) must hold for every $j \in U$. Thus, the optimum schedule contains a feasible solution to $LP1(U, 1)$.

By construction, U is a uniformly random subset of J, meaning all subsets are equally likely.

Thus,

$$\begin{aligned} E[T_{\text{OPT}}] &= 2^{-n}\sum_U E[T_{\text{OPT}} \mid U] \\ &= 2^{-n}\cdot\frac{1}{2}(\sum_U E[T_{\text{OPT}} \mid U] + \sum_U E[T_{\text{OPT}} \mid \overline{U}]) \\ &\geq 2^{-n}\frac{1}{2}\sum_U (t_{LP1(U,1)} + t_{LP1(\overline{U},1)}) \\ &\geq 2^{-n}\frac{1}{2}\sum_U t_{LP1(J,1)} \\ &= \frac{1}{2}t_{LP1(J,1)}\ , \end{aligned}$$

where the second line of this derivation follows from the first paragraph of this proof. □

The following lemma states that, in polynomial time, we can find an integral assignment that approximates (LP1) to within a constant factor. Some aspects of the proof are similar to [141, Theorem 4.1], which solves a slightly different problem useful for precedence constraints that form disjoint chains, but the proof here adds several steps that improve the approximation ratio. The tighter approximation does apply to the more general disjoint-chains variant, which we revisit in Section 4.3.

Lemma 4.3 *There exists a polynomial-time algorithm that computes a feasible solution to the linear program $LP1(J', L)$, having value $O(t_{LP1(J',L)})$.*

Proof. We relax the integer linear program to a linear program, and then show that the relaxed LP can be rounded to yield an integral $\{\widehat{x_{ij}}\}$ solution with value $O(T_{LP1(J',L)})$.

First, let $\ell'_{ij} = \min\{\ell_{ij}, L\}$. Then, replace each ℓ_{ij} in Equation (4.1) with ℓ'_{ij}, yielding the constraint $\sum_{i\in M}\ell'_{ij}x_{ij} \geq L, \forall j \in J'$. Since assignments are restricted to be integral, this change has no effect on either the feasibility or the value of an assignment. Next, remove Equation (4.3) and solve the relaxed linear program. Letting $\left\{x^*_{ij}, t^*\right\}$ be an optimal solution, we have that $t^* \leq t_{LP1(J',L)}$, because integral solutions are feasible.

Our goal now is to round the LP solution to an integral solution, while not increasing its value by very much. This rounding proceeds in three steps, outlined as follows. First, group machines having similar ℓ'_{ij} for a job j, yielding a single assignment for the whole group. Then, round those assignments to integers. Finally, show that the rounded assignments satisfy (LP1), using an integral flow network.

For each job j, group machines having ℓ'_{ij} values within a factor of 2, and determine the total assignment to that group. More formally, for each j and integer k, let

$$D^*_{jk} = \sum_{i:\lfloor \log \ell'_{ij}\rfloor = k} x^*_{id}$$

be the total assignment of machines with $\ell'_{ij} \in [2^k, 2^{k+1})$ to job j. It follows that for all $j \in J'$, we have

$$\sum_{i\in M}\ell'_{ij}x^*_{ij} \geq \sum_k D^*_{jk}2^k \geq L/2.$$

We next round the value of D^*_{jk} up to $\lfloor 6D^*_{jk} \rfloor$. We claim that for all $j \in J'$, we have

$$\sum_k \lfloor 6D^*_{jk} \rfloor 2^k \geq L.$$

Since $\ell'_{ij} \leq L$, the maximum value of k having nonzero D^*_{jk} is $\lfloor \log L \rfloor$. Thus, the claim follows because

$$\begin{aligned}\sum_k \lfloor 6D^*_{jk} \rfloor 2^k &\geq 3\left(2\sum_k D^*_{jk}2^k\right) - \left(\sum_{k\leq \log L} 2^k\right) \\ &\geq 3(L) - \left(\sum_{k=0}^{\infty} L/2^k\right) \\ &\geq 3L - 2L = L.\end{aligned}$$

In other words, rounding the group assignments down to integers only causes a loss of at most $2L$ work. We therefore need an assignment giving $3L$ work to the job.

To complete the integral assignment, construct a network-flow instance as follows. For each job j and integer k, create a node u_{jk}. For each machine i, create a node v_i. Also add a source-node s and a sink-node w. For each u_{jk}, add a directed edge (s, u_{jk}) with capacity $\lfloor 6D^*_{jk} \rfloor$. For each v_i, add a directed edge (v_i, w) with capacity $\lceil 6t^* \rceil$. Finally, add a directed edge (u_{jk}, v_i) with infinite capacity, for any j, k, i such that $\lfloor \log \ell'_{ij} \rfloor = k$. Note that for a given j and i, there is exactly one k such that (u_{jk}, v_i) exists. Refer to this edge as edge (j, i).

If we set the capacity of edges (s, u_{jk}) to be $6D^*_{jk}$ instead, then a flow of demand $\sum_{jk} 6D^*_{jk}$ exists in this network, and it can be constructed by setting the flow along edge (j, i) to be $6x^*_{ij}$ (and flow along edge (v_i, w) to be $6\sum_{j\in J} x^*_{ij}$). Thus, a flow of capacity $\sum_{jk} \lfloor 6D^*_{jk} \rfloor$ exists when we lower the capacity of edge (s, u_{jk}) to $\lfloor 6D^*_{jk} \rfloor$.

Ford-Fulkerson's theorem [71,95] states that an integral max flow exists whenever the capacities are integral, as they are here. We therefore take the flow across the edges (j, i) as our integral assignments $\widehat{x_{ij}}$. Moreover, by construction, $\widehat{x_{ij}}$ satisfy

$$\begin{aligned}\forall i \in M, \quad &\sum_{j\in J} \widehat{x_{ij}} \leq \lceil 6t^* \rceil, \\ \forall j \in J, \quad &\sum_{i\in M} \ell'_{ij}\widehat{x_{ij}} \geq \sum_k \lfloor 6D^*_{jk} \rfloor 2^k \geq L.\end{aligned}$$

We thus have an integral feasible solution $\{\widehat{x_{ij}}, 6t^*\}$. Noting that $6t^* \leq 6T_{LP1(J',L)}$ completes the proof. □

Since Lemma 4.2 shows that $t_{LP1(J,1)} = O(\mathrm{E}[T_{\mathrm{OPT}}])$, Lemmas 4.2 and 4.3 in concert state that in polynomial time, we can find a schedule Σ of length $O(\mathrm{E}[T_{\mathrm{OPT}}])$ such that every job fails with at most constant probability. Repeating Σ until all jobs complete produces the oblivious schedule SUU-I-OBL.

Theorem 4.4 *Let $T_{\text{SUU-I-OBL}}$ denote the random variable corresponding to the time it takes for an execution of SUU-I-OBL to complete all jobs. Then, we have $\mathrm{E}[T_{\text{SUU-I-OBL}}] = O(\mathrm{E}[T_{\mathrm{OPT}}] \log n)$.*

Proof. From Lemmas 4.2 and 4.3, we have a schedule Σ of length $O(\mathrm{E}[T_{\mathrm{OPT}}])$ that gives each job a constant probability of success. Applying a Chernoff bound gives us that a particular job completes in $O(\log n)$ repetitions of Σ with probability at least $1 - 1/n^{\Theta(1)}$, where the constant exponent appears as a constant factor in the number of repetitions. Taking a union bound over all jobs gives that with probability at least $1 - 1/n^{\Theta(1)}$, all jobs complete in $O(\log n)$ repetitions. Since this probability drops off dramatically as the number of repetitions increases, we have $\mathrm{E}[T_{\text{SUU-I-OBL}}] = O(\mathrm{E}[T_{\mathrm{OPT}}] \log n)$. □

An $O(\log\log(\min\{m, n\}))$-approximation

Construct the semioblivious schedule SUU-I as follows. The schedule is divided into "rounds." The first round corresponds to an execution of the schedule suggested by the (rounded) solution to $LP1(J, 1)$. In each following round, (LP1) is applied to all remaining jobs with doubling target work. If $J_k \subseteq J_{k-1} \subseteq J$ are the set of jobs left at the start of round k, then for that round find an approximate solution to $LP1(J_k, 2^{k-1})$, and schedule obliviously according to it.

SUU-I runs at most $K = \log\log(\min\{m, n\}) + O(1)$ of these rounds. If uncompleted jobs remain after the Kth round, one of two things is done. If $n \leq m$, SUU-I runs each job one at a time on all machines, until all jobs are completed. If $m < n$, SUU-I simply repeats the schedule $\Sigma_{LP1(J_k, 2^{k-1})}$ given by the Kth round until all jobs complete.

I will prove that SUU-I achieves an approximation factor of $O(\log\log\min\{m, n\})$. A key aspect of the analysis is viewing SUU-I as an "online algorithm" to solve the SUU* problem over the hidden input $\{p_j\}$. Essentially, SUU-I "discovers" the values of p_j as jobs complete.

In the proof of the following lemma, I compare the length of rounds $2, 3, \ldots, K$ against the makespan of an optimal offline algorithm, called OFF, which knows the values $\{p_j\}$. In particular, I show that if OFF takes total time t on input $\{p_j\}$, then each round of SUU-I takes time $O(t)$ on the same input. This part of our proof is essentially a competitive analysis [181]. In subsequent lemmas, I bound the time of SUU-I for the later rounds.

Lemma 4.5 *The total expected time of rounds* $2, 3, \ldots, K$ *of SUU-I is* $O(K \cdot E[T_{\mathrm{OPT}}])$.

Proof. First, I show that for any fixed set of random values $\{p_j\}$, each round of SUU-I takes time proportional to that of the optimal offline strategy OFF. I then combine all these rounds, taking an expectation over $\{p_j\}$, to get that rounds $2, 3, \ldots, K$ take total expected time $O(K \cdot \mathrm{E}[T_{\mathrm{OPT}}])$.

Consider an optimal offline strategy OFF, which knows the random values of $\{p_j\}$, and let $T_{\mathrm{OFF}}(\{p_j\})$ denote OFF's makespan given the values $\{p_j\}$ (i.e., $T_{\mathrm{OFF}}(\{p_j\})$ is the minimum over all strategies for a fixed $\{p_j\}$). For each $k \in \{2, 3, \ldots, K\}$, let $J_k \subseteq J$ be the subset of jobs such that $p_j > 2^{k-2}$. Thus, OFF must assign machines to job $j \in J_k$ such that $\sum_{i \in M} x_{ij}\ell_{ij} \geq 2^{k-2}$. And hence we have $T_{\mathrm{OFF}}(\{p_j\}) \geq T_{LP1(J_k, 2^{k-2})}$.

Now consider an execution of SUU-I for the same $\{p_j\}$. If a job j remains uncompleted at the start of the kth round, for $k \in \{2, 3, \ldots K\}$, then $p_j > 2^{k-2}$. This inequality follows from the fact that in the $(k-1)$th round, every job receives work that exceeds 2^{k-2}. Hence, the jobs executed in the kth round are a subset of those defined by J_k above. By Lemma 4.3, the kth round takes time $O(T_{LP1(J_k, 2^{k-1})})$. Observing that $T_{LP1(J_k, 2^{k-1})} \leq 2T_{LP1(J_k, 2^{k-2})}$, we conclude that SUU-I's kth round takes time $O(T_{\mathrm{OFF}}(\{p_j\}))$.

Thus, for any particular $\{p_j\}$, rounds $2, 3, \ldots, K$ of SUU-I take total time $O(K \cdot T_{\mathrm{OFF}}(\{p_j\}))$. Since OFF is the optimal offline strategy, it follows that $T_{\mathrm{OPT}}(\{p_j\}) \geq T_{\mathrm{OFF}}(\{p_j\})$ on the same $\{p_j\}$. Thus, SUU-I takes time at most $O(k \cdot T_{\mathrm{OPT}}(\{p_j\}))$. Taking the expectation over all $\{p_j\}$, we

conclude that these rounds take in expectation total time

$$O\Big(K \cdot E_{\{p_j\}}[T_{\text{OPT}}(\{p_j\})]\Big) = O(K \cdot \mathrm{E}\,[T_{\text{OPT}}])\,. \qquad \square$$

In fact, Lemma 4.5 shows that this doubling technique yields a deterministic $O(\log(p_{\max}/p_{\min}))$-competitive algorithm for the problem of hidden but arbitrarily chosen $p_j \in [p_{\min}, p_{\max}]$. SUU-I, however, changes strategies after K rounds. Since $p_{\max}$ is chosen from an exponential distribution, the probability that $p_{\max}$ exceeds $\Theta(\log n)$ is small.

The following two lemmas bound the expected time of the late rounds of SUU-I when $m > n$ or $n > m$, respectively.

Lemma 4.6 *Suppose $m \geq n$. Then rounds $K+1, K+2, \ldots$ of SUU-I take total expected time $O(E[T_{\text{OPT}}])$.*

Proof. Recall that after the Kth round, SUU-I runs jobs one after the other. Running jobs one at a time on all machines is trivially an $O(n)$-approximation. It remains to bound the probability that SUU-I proceeds to the Kth round.

Recall also that jobs remaining after round K must have $p_j \geq 2^{K-1} \geq 2^{\log\log n + O(1)} \geq 2\log n$ (for appropriate choice of constant in $O(1)$), which occurs with probability at most $2^{-2\log n} = 1/n^2$. Applying a union bound over all j, we get probability at most $1/n$ that any job survives to round $K+1$. Hence, the expected time for rounds $K+1, K+2, \ldots$ is at most $(1/n)O(n\mathrm{E}\,[T_{\text{OPT}}]) = O(\mathrm{E}\,[T_{\text{OPT}}])$. $\square$

Lemma 4.7 *Suppose $n > m$. Then rounds $K+1, K+2, \ldots$ of SUU-I take total expected time $O(E[T_{\text{OPT}}])$.*

Proof. Let $\Sigma_K = \Sigma_{LP1(J_K, 2^{k-1})}$ be the schedule computed for the the Kth round. Here, SUU-I repeats Σ_K until all jobs complete. Define the ***load*** H of a finite schedule to be the maximum number of timesteps during which any machine is assigned to an uncompleted job (i.e., $H = \max_i \sum_j x_{ij}$); the schedule Σ_K can be easily "compressed" to run in exactly a number of a timesteps equal to its load. We will analyze how long it takes for the load of our instance to drop from its initial expected value $O(\mathrm{E}\,[T_{\text{OPT}}])$ (at the end of the Kth round) to 0 (when all jobs have completed).

Let T_K be the random variable denoting the length of Σ_K. Let X_i be the random variable representing a number of repetitions of Σ_k necessary to drop the load from $T_K/2^i$ to $T_K/2^{i+1}$. Define X to be the random variable denoting the number of timesteps until the compressed load of Σ_K drops below 1 (and hence reaches 0). Then, we have

$$X \leq \sum_{i=0}^{\log T_K} \frac{X_i T_K}{2^i} = T_K \sum_{i=0}^{\log T_K} \frac{X_i}{2^i}.$$

By construction, a single execution of Σ_K ensures that each job remains with probability at most $1/m^2$, and thus the expected load of each machine shrinks by at least a (multiplicative) factor of m^2. Markov's inequality implies that each machines load decreases by a factor of $m/2$ with probability at least $1 - 1/(2m)$. Taking a union bound over all machines gives us that the load of all machines decreases by a factor of $m/2$ with probability at least $1/2$.

For $m > 4$, we now have that each repetition of Σ_K decreases the remaining load by a factor of 2 with probability at least $1/2$, and hence $\mathrm{E}\,[X_i] \leq 2$ for all i—requiring only an expected constant

number of repetitions to reduce the load by a constant factor. We now have that

$$E\left[\sum_{i=0}^{\log T_K} \frac{X_i}{2^i}\right] \leq \sum_{i=0}^{\infty} \frac{\mathrm{E}[X_i]}{2^i}$$
$$\leq O(1).$$

Since T_k and each X_i are independent,[2] it follows that

$$\mathrm{E}[X] \leq \mathrm{E}[T_K] \sum_{i=0}^{\infty} \frac{\mathrm{E}[X_i]}{2^i}$$
$$= O(\mathrm{E}[T_K]).$$

Having $\mathrm{E}[T_K] \leq O(\mathrm{E}[T_{\mathrm{OPT}}])$ (as exhibited by Lemma 4.5) completes the proof. □

Theorem 4.8 *Let* $K = \log\log(\min\{m, n\}) + O(1)$ *and let the random variable* $T_{\text{SUU-I}}$ *be defined as the time it takes for an execution of SUU-I to complete all of the jobs. Then, we have* $\mathrm{E}[T_{\text{SUU-I}}] = O(K \cdot \mathrm{E}[T_{\mathrm{OPT}}])$.

Proof. Apply Lemma 4.2 to bound the expected length of the first round, Lemma 4.5 to bound the expected length of rounds $2, 3, \ldots, K$, and Lemma 4.6 or Lemma 4.7 as appropriate to bound the expected length of rounds $K+1, K+2, \ldots$. It follows that makespan is bounded by $\mathrm{E}[T_{\text{SUU-I}}] = O(\mathrm{E}[T_{\mathrm{OPT}}] + K \cdot \mathrm{E}[T_{\mathrm{OPT}}] + \mathrm{E}[T_{\mathrm{OPT}}]) = O(K \cdot \mathrm{E}[T_{\mathrm{OPT}}])$. □

4.3 Jobs with chain-like precedence constraints

This section describes an $O(\log(n+m)\log\log(\min\{m, n\}))$-approximation for SUU-C, the case when precedence constraints form a collection of disjoint chains, called SUU-C. SUU-C may be used as a subroutine for SUU-T, the more general case where precedence constraints form disjoint trees (see Section 4.4).

In SUU-C, the dependency graph G is a collection disjoint chains $G = \{C_1, C_2, \ldots, C_z\}$, where each C_k gives a total order on a subset of jobs.

SUU-C is similar to Lin and Rajaraman's algorithm [141], but SUU-C achieves a better approximation ration through various improvements. I first give an overview of the algorithm. I then provide more details later in the section.

To construct the schedule, first find an assignment $\{x_{ij}\}$ of machines to jobs, where x_{ij} is an integral number of steps for which machine i is assigned to job j), giving each job one unit of work such that the "length" and "load" of the assignment are bounded by $O(\mathrm{E}[T_{\mathrm{OPT}}])$. The ***load*** of a machine is the number of timesteps for which any job is assigned to it (i.e, $\sum_j x_{ij}$), and the load of the assignment is the maximum across all machines. The ***length*** of a chain is the sum of the length of the jobs in the chain. The length of a job j, denoted by d_j, is the maximum number of steps for which j is assigned to a single machines (i.e., $d_j = \max_i x_{ij}$). Clearly, a schedule taking time T must have a length and load no more than T.

[2]Independence follows from the fact that the exponential distribution is memoryless. The variable T_k is only a function of which jobs remain at the start of the Kth round, i.e., those jobs j for which $p_j \geq 2^{K-1}$. In contrast, given a set of remaining jobs, X_i depends only on the degree to which the p_js exceed 2^{K-1}. Since the exponential distribution is memoryless, we have independence.

SUU-C uses an LP relaxation (similar to (LP1) in Section 4.2) to generate our assignment. Details appear later in the section. As in Section 4.2, the LP relaxation achieves an $O(1)$-approximation. This assignment does not immediately yield a schedule, because it does not take into account the precedence constraints.

To transform the assignment into an adaptive schedule, SUU-C treats long and short jobs differently. A job is ***short*** if the length of its assignment is at most some value γ, to be defined later, and the job is ***long*** otherwise. To simplify presentation, suppose for now that all jobs are short. I later describe how to deal with long jobs.

Next, transform the assignment into an adaptive schedule Σ_k for each chain C_k. The schedule Σ_k considers the next eligible (uncompleted) job j in C_k, and (obliviously) schedules the next d_j timesteps according to the assignment $\{x_{ij}\}$. Specifically, if Σ_k begins executing job j at time t, then it schedules j from time t to $t + x_{ij}$ on machine i. (Machine i remains idle from time $t + x_{ij}$ to $t + d_j$.) After the d_j timesteps, Σ_k again considers the next eligible job in the chain (which may be the same job if it failed). Since each time job j is obliviously scheduled, it has a constant probability of success.

All the Σ_ks can be combined in a straightforward manner, yielding a "pseudoschedule" for the SUU-C instance, denoted by $\{\Sigma_k\}$. In particular, a ***pseudoschedule*** runs all Σ_k "in parallel," possibly assigning multiple jobs to the same machine in each timestep. To avoid confusion, we call each of the timesteps of a pseudoschedule a ***superstep***, and we call the number of jobs assigned to a single machine during a superstep t the ***congestion*** at that superstep, denoted by $c(t)$. We "flatten" each superstep to $c(t)$ timesteps by arbitrarily ordering the jobs assigned to each machine, thus yielding a schedule called SUU-C. If $c_{\max}$ is the maximum congestion over all supersteps, and Z is the maximum length of any chain, then SUU-C comprises $O(c_{\max} Z)$ timesteps.

To reduce congestion, apply a random-delay technique [139, 180], also used by Lin and Rajaraman [141]. To prove the main bound, I will utilize (and prove) the fact that when chains consist of sufficiently many (short) jobs, the number of supersteps spanned by Σ_k is near the expected length of Σ_k, with high probability. To deal with long jobs, run SUU-I $O(\log(n+m))$ times, which dominates the runtime, yielding the $O(\log(n+m)\log\log(\min\{m,n\}))$-approximation.

Finding an assignment with low load and length.

As in Section 4.2, SUU-C uses an integer linear program to optimize for the constraints. This integer linear program for chains matches that used in [141, LP1].

$$
\begin{array}{rrcll}
\textbf{(LP2)} & \min t & & & \\
\text{s.t.} & \sum_{i \in M} \ell_{ij} x_{ij} & \geq & 1 \quad \forall j \in J & (4.4) \\
& \sum_{j \in J} x_{ij} & \leq & t \quad \forall i \in M & (4.5) \\
& \sum_{j \in C_k} d_j & \leq & t \quad \forall C_k \in G & (4.6) \\
& 0 \leq x_{ij} & \leq & d_j \quad \forall i \in M, j \in J & (4.7) \\
& d_j & \geq & 1 \quad \forall j \in J & (4.8) \\
& x_{ij} & \in & \mathbb{N} \cup \{0\} \quad \forall i \in M, j \in J\,. & (4.9)
\end{array}
$$

Equations (4.4), (4.5), and (4.9) correspond to Equations (4.1), (4.2), and (4.3), respectively, in (LP1). Equation (4.5) bounds the load of each machine. Equation (4.6) bounds the length of each

chain, and Equations (4.7) and (4.8) determines the length of each job.

The following lemma, proven in [141, Lemma 4.2], states that the optimal value for (LP2) is a lower bound on $\mathrm{E}[T_{\mathrm{OPT}}]$.

Lemma 4.9 *Let $t_{(LP2)}$ be the optimal value for (LP2). Then $t_{(LP2)} = O(\mathrm{E}[T_{\mathrm{OPT}}])$.* □

The next lemma exhibits an $O(1)$-approximation to (LP2). Lemmas 4.9 and 4.10 together imply a polynomial-time algorithm giving an integral assignment $\{x_{ij}\}$ of machines to jobs such that the load and length are both $O(\mathrm{E}[T_{\mathrm{OPT}}])$.

Lemma 4.10 *Let $t_{(LP2)}$ be the optimal value for (LP2). There exists a polynomial-time algorithm that computes a feasible solution to (LP2) having value $O(t_{(LP2)})$.*

Proof. The rounding proceeds as in Lemma 4.3, starting by removing Equation (4.9) and replacing Equation (4.4) by $\sum_{i\in M} \ell'_{ij} x_{ij} \geq 1$ for $\ell'_{ij} = \min\{\ell_{ij}, 1\}$. The only major difference is in the capacity of some edges in the flow network. Instead of giving edge (j,i) an infinite capacity, restrict the capacity of edge (j,i) to $\lceil 6d_j^* \rceil$, where d_j^* is the assignment given by the optimal solution to the relaxed linear program. Here, the length of a chain C_k may increase by at most $6\sum_{j\in C_k} d_j^* + |C_k| \leq 7\sum_{j\in C_k} d_j^*$. Since the capacity of edges (v_i, w) from machine nodes to the sink node remains $\lceil 6t^* \rceil$, machine loads are also bounded. □

Reducing congestion of SUU-C

As described thus far, SUU-C may have $\Theta(n)$ congestion. We take advantage of a random-delay technique [139,180] to reduce congestion to $O(\log(n+m)/\log\log(n+m))$, with high probability. Essentially, modify SUU-C to simply delay the start time of each chain by a value chosen uniformly at random from $\{0, 1, \ldots, H\}$, where H is the load of SUU-C.

The delay technique is summed up by the following theorem, proof omitted (as similar theorems appear elsewhere). It originates in [139], and Lin and Rajaraman [141, Section 4.1] outline the necessary proof as applied to SUU-C.

Theorem 4.11 *Consider a pseudoschedule $\{\Sigma_k\}$ with total load H, where H is polynomially bounded in n and m. Consider the pseudoschedule $\{\Sigma_{k'}\}$ generated by randomly shifting, or "delaying," the start time of each chain schedule Σ_k by a value chosen uniformly at random from $\{0, 1, \ldots, H\}$. Then, $\{\Sigma_{k'}\}$ has congestion at most $O(\log(n+m)/\log\log(n+m))$, with high probability with respect to n and m.* □

Notice that whenever the load of and length of $\{\Sigma_k\}$ are bounded by $O(\mathrm{E}[T_{\mathrm{OPT}}])$, it follows that the length of $\{\Sigma_{k'}\}$ is at most $O(\mathrm{E}[T_{\mathrm{OPT}}])$ supersteps, with high probability.

Since Σ_k repeats the assignment for some jobs, the load and length of the pseudoschedules $\{\Sigma_{k'}\}$ are random variables. The random successes and failures of jobs (and hence load and length), however, are independent of the initial random delay selected. Suppose that a random execution yields a load and length of at most $O(\mathrm{E}[T_{\mathrm{OPT}}])$. Suppose also that T_{OPT} is polynomially bounded in n and m. Then, apply Theorem 4.11 to conclude that each superstep has low congestion, and thus the makespan is $O(c_{\max}\mathrm{E}[T_{\mathrm{OPT}}]) = O((\log(n+m)/\log\log(n+m))\mathrm{E}[T_{\mathrm{OPT}}])$ steps. It remains only to bound the random load and length of the pseudoschedule.

The following lemma implies that most executions of SUU-C result in low load and length. (I address the fact that T_{OPT} may not be polynomially bounded later in the section.) In this lemma, y_j

is the random variable indicating the number of repetitions of job j's assignment used to complete j, and d_j denotes the length of job j's assignment. In the SUU-C context, we have $\eta = n + m$. The value W here represents the load or the length of the assignment. The lemma states that whenever a job has length (or causes load) that is logarithmically smaller than the total, then the length of the chain or (load on a machine) is close to the expectation, with high probability. Union bounding over all $O(n)$ chains (or m machines) implies that the total length (and load) of SUU-C schedule is close to the expectation, with high probability.

Lemma 4.12 *For each $j \in \{1, 2, \ldots, n\}$, let y_j be a positive integer drawn from the geometric distribution* $\Pr\{y_j = k\} = (1/2)^k$ *(for k a positive integer), and let $d_j \geq 1$ be a weight associated with each j. Let W and η be chosen such that $W/\log\eta \geq d_j$ for all j, $W \geq \sum_j 2d_j$, and $\log\eta \leq W$. Then, we have $\sum_j d_j y_j \leq O(cW)$ with probability at least $1 - 1/\eta^c$, for any positive constant c.*

Proof. First, round all the d_j up to the next power of 2, grouping the d_j by value. Specifically, let Z_k be the set of j such that d_j is bounded in the following way

$$W/(2^k \log\eta) < d_j \leq W/(2^{k-1}\log\eta),$$

for $k \in \{1, 2, \ldots, \lceil\log(W/\log\eta)\rceil\}$. Since

$$\sum_j d_j y_j \leq \sum_k \frac{W}{2^k \log\eta} \sum_{j \in Z_k} y_j,$$

our goal is first to bound $\sum_{j \in Z_k} y_j$ near its expectation.

To bound $\sum_{j \in Z_k} y_j$, view each y_j as the length of a sequence of fair-coin flips that terminates when flipping the first "head." Thus, their sum exceeds some value b_k (to be assigned later) when b_k fair-coin flips do not yield $|Z_k|$ heads. Let B_k be the sum of b_k Bernoulli random variables with expectation $1/2$ (i.e., B_k is the number of "heads" in a size-b_k set of fair-coin flips). Then, we have $\Pr\{B_k < |Z_k|\} = \Pr\left\{\sum_{j \in Z_k} y_j > b_k\right\}$.

We will bound $\Pr\{\exists k \text{ s.t. } B_k < |Z_k|\} \leq \eta^{-c}$ by taking a union bound over all k. Since k ranges over roughly $\log W$ values and W is not a function of η, we need a stronger bound than $\Pr\{B_k < |Z_k|\} \leq \eta^{-c}$. Instead, set b_k such that $\Pr\{B_k < |Z_k|\} \leq \eta^{-c}2^{-k}$. Then taking a union bound over all k gives probability at most

$$\begin{aligned} \sum_{k=1}^{\log(W/\log\eta)} \eta^{-c}2^{-k} &\leq \eta^{-c}\sum_{k=1}^{\infty} 2^{-k} \\ &\leq \eta^{-c} \end{aligned}$$

that there exists a k such that $\sum_{j \in Z_k} y_j > b_k$. Thus, setting $b_k = \Theta(c(|Z_k| + \log\eta + k))$ and applying a Chernoff bound for $\Pr\{B_k < |Z_k|\}$ achieves the desired probability bound.

Thus, with probability at least $1 - 1/\eta^c$, we are left with

$$\begin{aligned}
\sum_j d_j y_j &\le \sum_k \frac{W}{2^k \log \eta} \Theta(c(|Z_k| + \log \eta + k)) \\
&= O\left(c \sum_k \frac{W\,|Z_k|}{2^k \log \eta}\right) + O\left(cW \sum_k \frac{\log \eta + k}{2^k \log \eta}\right) \\
&\le O\left(c \sum_j d_j\right) + O\left(cW \sum_{k=1}^{\infty} \frac{1+k}{2^k}\right) \\
&= O(cW)\ ,
\end{aligned}$$

where the third line of the derivation follows from the restriction that $W \ge \sum_j 2d_j$. □

To conclude, if jobs are short, where short jobs have length at most $\gamma = t_{LP2}/\log(n{+}m)$, and if t_{LP2} is polynomial in n and m, then SUU-C takes time $O((\log(n+m)/\log\log(n+m))\mathrm{E}[T_{\mathrm{OPT}}])$ with high probability. To get this bound in expectation, simply modify SUU-C to run the $O(n)$-approximation (as for SUU-I) whenever congestion, load, or length exceed the desired bounds, which occurs with probability at most $1/n$.

Handling long jobs

I now extend SUU-C to handle jobs having length greater than $t_{LP2}/\log(n+m)$. In the chain schedule Σ_k, replace each longer job by a "pause" of length $t_{LP2}/\log(n+m)$. Specifically, no job from the chain is scheduled until $t_{LP2}/\log(n+m)$ supersteps later. Next, divide the schedule for SUU-C into $O(\log(n+m))$ segments of length $t_{LP2}/\log(n+m)$ supersteps. By construction, there is at most one pause per chain per segment. After executing each segment, SUU-C executes SUU-I on the jobs corresponding to the pauses starting in that segment (suspending the rest of the chains until completion). Once those long jobs complete, SUU-I continues to the next segment.

All of the previous analyses (that assume jobs are short) still hold. In particular, this construction satisfies the requirement that all relevant long jobs complete before the short jobs are scheduled again. Since there are $O(\log(n+m))$ executions of SUU-I, it follows that the total expected time increases to

$$O(\log(n+m)\log\log(\min\{m,n\}) \cdot \mathrm{E}[T_{\mathrm{OPT}}])\,,$$

giving an $O(\log(n+m)\log\log(\min\{m,n\}))$-approximation.

Extending to nonpolynomial t_{LP2}

I now address the requirement in Theorem 4.11 that load and length be polynomially bounded in n and m.

We make use of a trick from [180, Section 3.1], also used in [141]. Consider the chain schedule Σ_k (having length $O(t_{LP2})$, with high probability) before the random delay is applied. Round each assignment x_{ij} down to the nearest multiple of t_{LP2}/nm. We thus treat the assignments as integers in the range $\{0, 1, \ldots, O(nm)\}$. We can then apply the random-delay technique (from Theorem 4.11) to these rounded assignments.

The issue now is that the rounding may have decreased many assignments, and so we reinsert steps into the schedule. In particular, whenever executing job j, we reinsert the steps (*not* supersteps) removed from rounding into the execution, executing only job j during those steps.

Theorem 4.13 *Let $T_{\text{SUU-C}}$ be the random variable denoting the time at which an execution of SUU-C completes all jobs. Then* $\mathrm{E}[T_{\text{SUU-C}}] = O(\log(n+m)\log\log(\min\{m,n\})\mathrm{E}[T_{\text{OPT}}])$.

Proof. Ignoring the reinserted steps, Theorem 4.11 along with Lemma 4.12 scale appropriately to imply that all short jobs are accounted for by $O((\log(n+m)/\log\log(n+m))\mathrm{E}[T_{\text{OPT}}])$ makespan. Similarly, since there are $\log(n+m)$ segments, and long jobs are executed using SUU-I between segments, the long jobs add $O(\log(n+m)\cdot\mathrm{E}[T_{\text{SUU-I}}])$ to the makespan, which is bounded by $O(\log(n+m)\log\log(\min\{m,n\})\mathrm{E}[T\mathit{opt}])$ in Theorem 4.8.

It remains to consider the steps that must be reinserted after rounding down each x_{ij} to the nearest multiple of t_{LP2}/nm. The execution of job j may result in reinserting at most an expected $2t_{LP2}/nm$ steps for each machine, and hence $2t_{LP2}/n$ steps in total. Summing across all n jobs gives an expected $2t_{LP2}$ steps, thereby increasing the total length of SUU-C by $O(\mathrm{E}[T_{\text{OPT}}])$. □

4.4 Jobs with tree-like precedence constraints

Algorithms for tree-like precedence constraints can be obtained by trivially applying techniques from [136], as done in [141]. I state the bound here without proof.

When precedence constraints form a directed forest, the technique from [136] decomposes the graph into $O(\log n)$ blocks, each consisting of disjoint chains. Then, apply SUU-C $O(\log n)$ times.

Theorem 4.14 *If precedence constraints form a directed forest, there exists a polynomially computable schedule with expected makespan* $O(\log(n)\log(n+m)\log\log(\min\{m,n\})\mathrm{E}[T_{\text{OPT}}])$. □

4.5 Stochastic scheduling

This section shows how the algorithms from Sections 4.2 and 4.3 apply to the problem of preemptively scheduling, on unrelated parallel machines, jobs whose lengths are given by exponentially distributed random variables. In the now commonplace Graham's notation [109], these problems are of the form $R|\,\mathit{pmtn}, \mathit{prec}, p_j{\sim}\mathit{stoch}\,|\mathrm{E}[C_{\max}]$, and we refer to the group as STOCH. In general, Graham's notation takes the form $\alpha|\beta|\gamma$, where α denotes the machine environment (here, R indicates unrelated machines), β denotes various constraints on jobs and the environment (*pmtn* is often used as a shorthand for preemption and *prec* as a shorthand for precedence constraints), and γ denotes the objective function ($C_{\max}$ denotes makespan). I show an $O(\log\log n)$-approximation for the special case of STOCH-I, when all jobs are independent. Then, I discuss briefly how the rest of our algorithms for SUU generalize to the STOCH context.

Preliminaries

An instance $I_{\text{stoch}} = (J, M, \{\lambda_j\}, \{v_{ij}\}, G)$ of STOCH contains a set of jobs J, a set of machines M, and a dependency graph G just as in SUU. The length p_j of each job j, instead of identically 1, is set randomly according to the exponential distribution with rate parameter λ_j, that is, $\Pr\{p_j \le c\} = 1 - e^{-c\lambda_j}$. The actual value of p_j is not revealed until the job completes. Finally, for each machine i and job j, v_{ij} specifies the speed with which machine i processes job j. Thus, if x_{ij} is the amount of time during which machine i processes job j, then j completes once $\sum_i x_{ij}v_{ij} \ge p_j$. For STOCH, x_{ij} need not be integral, but we do require that every job be processed by at most one machine at any point in time.

An $O(\log\log n)$-approximation for STOCH-I

I now show how to provide a $O(\log\log n)$-approximation for STOCH-I. The techniques here are analogous to those in Section 4.2, and the algorithm STC-I operates similarly to SUU-I.

STC-I runs in $K = \log\log n + O(1)$ rounds, each corresponding to an oblivious schedule Σ_k. Construct the oblivious Σ_k such that any job having $p_j \leq 2^{k-2}/\lambda_j$ completes by the end of the round. Specifically, Σ_k corresponds to solving the deterministic problem $R|\,pmtn\,|C_{\max}$, with the length each job j fixed as $2^{k-2}/\lambda_j$. Jobs remaining after the end of the Kth round are run one at a time on the fastest possible machine. Use the algorithm from Lawler and Labetoulle [138] to compute a schedule for $R|\,pmtn\,|C_{\max}$ in polynomial time, giving us each of our Σ_k.

The following theorem states that STC-I approximates STOCH-I. The proof sketch is similar to Theorem 4.4 and Lemma 4.2

Theorem 4.15 *Let $T_{\text{STC-I}}$ be the random variable denoting the time it takes for an execution of STC-I to complete all jobs. Then, we have* $\mathrm{E}[T_{\text{STC-I}}] = O(\log\log n \cdot \mathrm{E}[T_{\text{OPT}}])$.

Proof. We show that the length of the first round approximates $\mathrm{E}[T_{\text{OPT}}]$ using a slight modification to Lemma 4.2, where L_j are set according to $1/(2\lambda_j)$. Then, we use a competitive-analysis argument to prove that the $K-1$ subsequent rounds take expected time $O(\mathrm{E}[T_{\text{OPT}}] \cdot K)$, along the lines of Lemma 4.5.

To complete the proof, we note that when $\max_j p_j$ is bounded above by $2\log n/\lambda_j$, all jobs complete by the end of round K. Since the p_j are exponentially distributed, we have

$$\Pr\{\exists j \text{ s.t. } p_j > 2\log n/\lambda_j\} \leq 1/n$$

Thus, running jobs sequentially trivially induces an n-approximation, but we see that it occurs only with probability at most $1/n$. The expected completion time is thus dominated by the first K rounds, which take $O(\mathrm{E}[T_{\text{OPT}}] \cdot K) = O(\log\log n \cdot \mathrm{E}[T_{\text{OPT}}])$. □

A slight modification to this algorithm gives an $O(\log\log(n))$-approximation to the slightly weaker setting of STOCH-R-I. Here, ***restarts*** are premitted instead of preemption. In this setting, a job may be moved from one machine to another, but it loses all previous progress when this migration occurs. In contrast, recall that with preemption a job accrues work from all machines that process it. The only necessary change to Theorem 4.15 is setting Σ_k according to $R||C_{\max}$ instead of $R|\,pmtn\,|C_{\max}$.

Other results

One can apply the remaining algorithms from Sections 4.2 and 4.3 to STOCH, with identical approximation ratios. Thus, we have algorithms achieving an $O(\log\log(\min\{m,n\}))$-approximation for STOCH-I, an $O(\log(n+m)\log\log(\min\{m,n\}))$-approximation for STOCH-C, and also an $O(\log(n)\log(n+m)\log\log(\min\{m,n\}))$-approximation for STOCH-T. Here problems have been named according to their SUU analogs.

4.6 Concluding remarks

In this chapter, I have presented improved approximation algorithms for multiprocessor scheduling under uncertainty. I believe that these bounds are not tight. In particular, I believe that a fully adaptive schedule should be able to trim an $O(\log\log(\min\{m,n\}))$ factor from the bounds. It would also be interesting if a greedy heuristic could achieve the same bounds. Finally, I would be

interested in developing nontrivial approximations for more general precedence constraints. At first glance, however, it seems like any technique for SUU and arbitrary precedence constraints may generalize to $R|\,pmtn, prec\,|C_{\max}$, which remains unsolved.

Chapter 5

Contention Resolution with Heterogeneous Job Sizes

Randomized backoff is a common mechanism for reducing contention on a shared resource. Processes/jobs make competing attempts to access the resource, but only one can gain control of the resource at a time. If an access attempt fails due to contention, then that process waits for a random amount of time before trying again. On subsequent failed attempts, the waiting time increases, thereby reducing the probability of a collision and increasing the chance of successful resource acquisition.

Backoff is used in many contexts, for example, network access (e.g., an Ethernet bus [154]), wireless communication [2], transactional memory [119], and speculative-lock elision [171]. In these and other applications of randomized backoff, the lengths of jobs fluctuate substantially. Most theoretical analyses, however, assume unit-length jobs. In a transactional shared-memory system, for example, jobs (transactions) can vary by four to five orders of magnitude [18]. In a wireless network, jobs (packet transmissions) can vary by over three orders of magnitude. The job length is proportional to both transmission length (in bits) and the transmission speed; the speed of the transmitters alone varies considerably (e.g., from roughly 10Kb/s to 10Mb/s).

This chapter, representing joint work with Michael A. Bender and Seth Gilbert previously appearing in [45], gives the first theoretical analysis of randomized backoff when jobs have variable sizes. This chapter analyzes a system consisting of jobs $1, \ldots, n$. Job i has size $t_i \geq 1$, which indicates that i must run for t_i consecutive units of time in order to complete. Define the ***volume*** of the jobs as $V = \sum_{i=1}^{n} t_i$. Each job knows its own size, but does not know any other job size or the number of other jobs. For simplicity, assume that t_i is integral and that time is divided into unit-sized time slots, but the analyses extend to the case of nonintegral sizes.

The jobs compete for access to a ***simple channel*** and have no other means of communication. Whenever a job of size t_i makes a run attempt, it must execute for the full t_i consecutive timeslots. If a job's run is uncontested, then the job completes successfully. If multiple jobs make overlapping run attempts, then all attempts fail and the jobs must retry. A job i learns whether its run attempt is successful only after the full t_i time slots, not instantly when the collision occurs. A job gains information only by making run attempts—there is no "listening" on the channel. No other information (e.g., the number of jobs that made attempts in a time slot) is available to the job. (These assumptions are roughly the worst case, in terms of information learned when a collision occurs.)

This chapter considers the ***batch problem*** (also called the ***control-tower problem*** [145] or ***shopping-cart problem*** [94, 125]), where all jobs arrive at time 0. I present analyses of the worst-case ***makespan***, or maximum completion time among all jobs, of the protocols considered.

This chapter discusses ***windowed backoff protocols*** in which time is divided into a sequence of windows $\langle W_1, W_2, W_3, \ldots \rangle$. A job makes at most one run attempt in any window. A job can make a run attempt only if the window is larger than the job size. Even if a job does fit in a window, it may choose not to make a run attempt. If the job does choose to execute, it randomly chooses a position in the window such that there is sufficient time left for the job to execute fully within the window.

Contributions

Binary exponential backoff and generalizations. I begin by presenting results on binary exponential backoff. Since a single large job can slow down many small jobs, the performance for heterogeneous job sizes is significantly worse than for unit-sized jobs. I show that it achieves a makespan of $V2^{\Theta(\sqrt{\log n})}$ with error probability polynomially small in n. I next give a variant of exponential backoff that backs off more slowly and yields a makespan of $\Theta(V \log V)$ also with error probability small in n. A key tool is a tight analysis of "fixed-window backoff," where all windows have size $\Theta(V)$. These protocols achieve the specified makespan with error probability polynomially small in the number of jobs.

Size-hashed backoff. The principal result in this chapter is a backoff protocol that achieves makespan $O(V \log^3 \log V)$. The main technique is to group jobs by size. Roughly speaking, size-hashed backoff "hashes" jobs based on their size to specific windows in which they can make run attempts. An explicit construction, based on the modulo hash function results in a makespan of $O(V\sqrt{\log V} \log^2 \log V)$. Grouping the job sizes using specially designed "good" hash functions yields a makespan of $O(V \log^3 \log V)$. I use the probabilistic method to show that such hash functions exist. These protocols achieve the specified makespans with error probability polynomially small in $\log V$.

These results represent joint work with Michael A. Bender and Seth Gilbert, previously appearing in [45].

Related work

The most closely related work is that of Gereb-Graus and Tsantilas [104] (see also [112]) and Bender et al. [43]. Gereb-Graus and Tsantilas [104] show that for unit-size jobs in the batch setting, there is a backoff-backon protocol (which is sometimes called "sawtooth") that achieves an optimal makespan of $O(n)$; a similar backoff-backon approach also appears in Greenberg and Leiserson [112] in the context of routing. Bender et al. [43] analyze fixed backoff, exponential backoff, polynomial backoff, and optimal monotone backoff in the batch setting; they analyze exponential backoff in an adversarial queuing-theory setting. For binary exponential backoff ($W_i = 2^i$) with unit-size jobs, they prove a makespan of $\Theta(n \log n)$. (With variable-length jobs the situation is quite different; see Theorems 5.3 and 5.4.) Batch arrivals have been considered by several other authors [107, 108, 110, 111] with the goal of routing h-relations, involving multiple channels.

In the wireless-networking literature, this batch problem is known as the ***shopping-cart problem*** [94, 125] and models a shopping cart full of items with RFID tags passing through a sensor all at the same time. Currently implemented protocols are far from achieving the linear makespan described in [104, 112].

5.1 Traditional backoff with variable-sized jobs

This section analyzes classic backoff protocols. I first consider ***fixed backoff***, where the window size is fixed at $\Theta(V)$. (This models the case where an estimate of the volume is known in advance.) I then turn to ***binary exponential backoff***, where the window size repeatedly doubles, i.e., $W_{i+1} = 2W_i$. If a job fits in window, it makes a random run attempt. In both cases the makespan of these strategies is worse for variable-size jobs than for unit-size jobs, with binary exponential backoff significantly worse. I end by giving a faster monotone backoff strategy, whose performance matches fixed backoff to within constant factors, even when the volume V is not known in advance.

Fixed-volume backoff

I first analyze the protocol FIXED-BACKOFF$_W$, where the volume V of the jobs is known in advance. FIXED-BACKOFF$_W$ is the windowed protocol in which $W_i = W = \Theta(V)$, where W is the (unchanging) window size throughout the protocol. I first show that FIXED-BACKOFF$_{\Theta(V)}$ has the following lower bound:

Theorem 5.1 *Let $W = (1+\varepsilon)V$ for any constant $\varepsilon \in \mathbb{R}^+$. There exists n sufficiently large such that the makespan of* FIXED-BACKOFF$_W$ *is $\Omega(W \log n)$ with error probability polynomially small in n.*

Proof. Suppose there is one large job of size $n+1$ and $n-1$ small jobs of size 1. Hence, the total volume is $V = 2n$. This proof proceeds by calculating a lower bound on the number of rounds it takes for all but $\Theta(\log^6 n)$ of the small jobs to complete, which is clearly a lower bound on the number of rounds for all jobs to complete.

First, observe that as long as the big job is still around, the probability of a small job colliding with the big job is $(n+1)/W = (n+1)/(1+\varepsilon)V > 1/2(1+\varepsilon)$, which is constant. Let $b = 2(1+\varepsilon)$, simplifying this expression to $1/b$. As long as the number of small jobs is larger than $\Omega(\log n)$, we can apply a Chernoff bound to conclude that the number of small jobs does not decrease by more than a constant factor in a round, with high probability.

Specifically, suppose that a round begins with X jobs and that X' jobs survive that round. Then, we have $\mathrm{E}[X'] \geq X/b$. Using Chernoff bounds, we have that $Pr[X' < (1-\delta)X/b]$ is superpolynomially small. Specifically, let $\delta = 1/\ln^2 n$. The error probability is largest when X is smallest, which occurs at $X = \ln^6 n$. Thus, we obtain $Pr[X' < (1-\delta)X/b] \leq e^{-X\delta^2/2b} = e^{-\ln^6 n(1/\ln^2 n)^2/2b} = n^{-\ln n/2b}$.

We want a lower bound on the number of rounds before we have fewer than $\ln^6 n$ jobs remaining. Let $d = b/(1-\delta)$. The number of rounds is therefore at least $\log_d n - \log_d(\ln^6 n)$. Since $\log_d n \geq \log_b n - 1$, the number of rounds is at least $\log_b n - \log_b(\ln^6 n) - 1$, as promised. □

I now give a matching upper bound for FIXED-BACKOFF$_W$, when $W \geq 3V$.

Theorem 5.2 *Let $W = \alpha V$, for $\alpha \geq 3$. Then the makespan of* FIXED-BACKOFF$_W$ *is $O(W \log n)$ with error probability polynomially small in n.*

Proof. This proof shows that in each round, a constant fraction of the jobs complete. Consider a given round r. Let n' be the number and V' be the volume of the jobs remaining in round r. Divide these jobs into two classes: "large jobs" that are larger than twice the average remaining volume, $2V'/n'$, and "small jobs" that are at most size $2V'/n'$.

By a simple counting argument at least half of the jobs are small. The goal now is to show that a constant fraction of these small jobs complete in round r, and hence a constant fraction of all jobs.

Without loss of generality, assume that each of these small jobs has size exactly $2V'/n'$, increasing the total volume by at most a factor of 3. This modification can only increase the makespan.

There are at most $n'/2$ big jobs. Each big job can cause conflicts while it is executing and for a duration $2V'/n'$ prior to starting. Therefore, the total time during which big jobs can cause conflicts is $V' + (n'/2)(2V'/n') = 2V'$. The probability, then, that a particular small job does not collide with any big job is at least $V/W \geq 1/3$. The probability that a small job does not collide with another small job is at least $2V'/W \geq 2/3$. Therefore, each small jobs completes with probability at least $2/9$. By a Chernoff bound, in each round a constant fraction of jobs complete with high probability. Therefore, after $O(\log n)$ rounds, all jobs have completed. □

There is a vast difference between fixed backoff in the variable-size and the unit-size case. If all jobs are unit size, then the makespan is $n \lg n \pm O(n)$ with high probability [43]. Moreover, the makespan *improves* when the window size dips slightly below the volume $V = n$, say to $W = 3n/\lg\lg\lg n$. At this point the makespan attains its optimal value of $\Theta(n \log\log n/\log\log\log n)$ [43]. With variable-length jobs, the makespan grows arbitrarily large if $W = V$.

Exponential backoff

I next analyze binary exponential backoff with variable-size jobs. In binary exponential backoff, $W_i = 2^i$, for $i = 1, 2, \ldots$, and for any job j in the system, j must make a run attempt in window W_i, as long as $t_j \leq W_i$.

Theorem 5.3 *Consider n jobs having total volume V running binary exponential backoff. The makespan is $V2^{O(\sqrt{\log n})}$ with error probability polynomially small in n (for sufficiently large n).*

Proof. The proof will proceed as follows. First, divide the jobs into classes of "small" and "large" jobs. Next, argue that as the rounds increase, an increasingly larger fraction of the small jobs complete with high probability. As the rounds increase, the small jobs will also make up an increasingly large fraction of the total number of jobs, meaning that as the rounds increase an increasing large fraction of the total jobs complete with high probability. Next, show that $O(\sqrt{n})$ rounds after the window size is W, there are only $O(\log n)$ remaining jobs. Then, switch to an alternative argument to show that all these $O(\log n)$ stragglers complete in the next $O(\sqrt{n})$ rounds with high probability.

Start considering windows when they are larger than the total volume V. Consider the ith window of size $W_i \geq 2^i V$. Let n_i and V_i be the number of and total volume of the remaining jobs, respectively. Consider all the jobs of size at most $2^{i/2}V_i/n_i$, that is, jobs of size at most $2^{i/2}$ times the average. By a simple counting argument, there are at least $n_i(1 - 1/2^{i/2})$ such jobs. The total volume of the larger jobs is at most $V_i \leq V$.

Consider first the probability of a given small job colliding with a big job. Each big job can cause a conflict with a small job while it is executing and for a duration of $2^{i/2}V_i/n_i$ prior to starting. Thus, the total time that the big jobs can cause conflicts is at most $V_i + n_i(1/2^{i/2})(2^{i/2}V_i/n_i) \leq 2V$. Consequently, the probability that a small job collides with a large job is $1/2^{i-1}$.

Consider next the probability of a given small job colliding with other small jobs. Each small job has size at most $2^{i/2}V_i/n_i$, so a given small job can collide with a second small job if the first small job begins during a time interval of at most $2^{i/2+1}V_i/n_i$ positions relative to the first. Thus, the probability of two particular small jobs colliding is less than $(2^{i/2+1}V_i/n_i)/W_i \leq 1/n_i 2^{i/2-1}$. Consequently, a particular small job has probability at most $1/2^{i/2-1}$ of colliding with a small job.

Thus, the total probably that a small job collides is at most $1/2^{i-1} + 1/2^{i/2-1} \leq 1/2^{i/2-2}$. Hence, in expectation, at most a $1/2^{i/2-2}$ fraction of small jobs do not complete in the ith window.

Since at most a $1/2^{i/2}$ fraction of jobs can be large, we can bound $n_{i+1} \leq 5 \cdot 2^{-i/2} n_i$ in expectation. As long as $\mathrm{E}[n_{i+1}] = \Omega(\log n)$, applying Chernoff bounds implies that n_{i+1} is at most a constant factor larger than its expectation, say $n_{i+1} \leq n_i/2^{i/2-3}$ with high probability, which means that $n_i \leq n2^{\sum_{j=1}^{i-1} -j/2+3}$.

Once $j = \Theta(\sqrt{\log n})$, there are at most $O(\log n)$ jobs remaining, and we switch to a different strategy. At this point the window has size $V2^{\Theta(\sqrt{\log n})}$. Now, the probability that any job survives a round is at most $V \lg n / (V2^{\Theta(\sqrt{\lg n})}) = \lg n 2^{-\Theta(\sqrt{\lg n})}$. (This is the case even if we blow up each remaining job to size V, which is an upper bound.) The probability that a packet survives an additional $\Theta(\sqrt{\lg n})$ rounds, is $O\left(2^{-\Theta(\lg n)}\right)$, which is polynomially small in n. Therefore, the probability that any remaining job survives that next $\Theta(\sqrt{\lg n})$ rounds is also polynomially small in n. The bounds on the makespan follow. □

I now give a lower bound on the performance of binary exponential backoff.

Theorem 5.4 *There exists an instance of* $(c+3)m \ln m + 1$ *jobs for which the makespan of exponential backoff is* $\Omega(V2^{\sqrt{\lg V}/2})$ *rounds with probability* $(1 - 1/m^c)$*, for any* $c > 1$.

Proof. Consider an instance in which there is one large job of size n and $(c+3)n \ln n$ small jobs of size 1, resulting in a total volume of $V = (c+3)n \ln n + n$. There are two regimes, which we analyze separately. While $W_i < n$, the ***small-window regime***, only small jobs attempt to execute. When $W_i \geq n$, the ***large-window regime***, the large job also attempts to execute. We first show that no job completes in the small-window regime with high probability, and then show that no job completes during the first $\Omega(\sqrt{\log n})$ windows of the large-window regime with high probability.

First, consider some window W_i in the small-window regime. For some job j, the probability of collision with some other job k is at least $1/n$, since the windows are of size $\leq n$. Therefore, the probability that j runs successfully is at most $(1-1/n)^{(c+3)n \ln n} \leq 1/n^{c+3}$. Hence, the probability of any job completing in window W_i is $1/n^{c+2}$. By a union bound over the first $\lg n$ windows, the probability of even one job completing is at most $\lg n / n^{c+2} \leq 1/n^{c+1}$.

Next, consider the first window $W_i \geq n$ in the large-window regime, and assume that no small jobs completed during the small-window regime. The goal now is to calculate the probability that a small job j does not complete prior to window W_{i+r}, for some $r \geq 0$. When $n \leq W_i \leq 2n$, the big job occupies at least half of the window. Therefore, the probability that job j fails to complete in W_i is at least $1/2$. Similarly, the probability that job j fails to complete in window W_{i+r}, given that job j has not completed in any smaller window, is at least $1/2^{r+1}$. We thus conclude that the probability that job j does not complete prior to window W_{i+r} is at least $\prod_{k=1}^{r-1} 1/2^k = 2^{-r(r-1)/2} \geq 2^{-r^2}$.

Choose $r = \sqrt{\lg n}$. Then the probability that job j does not survive until window W_{i+r} is at least $1/n$. Finally, let us calculate the probability that at least one of the $(c+3)n \ln n$ small jobs survives for r rounds in the large-window regime: $1 - (1-1/n)^{(c+3)n \ln n} \geq 1 - (1/e)^{c \ln n} \geq (1 - 1/n^c)$. Thus, with high probability, the makespan is at least $n2^{\sqrt{\lg n}}$.

Finally, express $n2^{\sqrt{\lg n}}$ in terms of V. We have $V = (c+3)n \ln n + n \leq (c+4)n \ln n \leq (c+4)n \ln V$, implying that $n \geq V/(c+4) \ln V$. With approximations, we obtain that $n2^{\sqrt{\lg n}} \leq V2^{\sqrt{\lg V}/2}$. Thus, in the small-job regime, the probability that even one small job completes is smaller than $1/n^{c+1}$; in the large job regime, the probability that all jobs complete before $V2^{\sqrt{\lg V}/2}$ is smaller than $1/n^{c+1}$. Hence, with probability at least $1-1/n^c$, the makespan is at least $V2^{\sqrt{\lg V}/2}$, with probability at least $1 - 1/n^c$, as promised. □

Optimized exponential backoff

The following theorem states that a variant of exponential backoff chieves better performance by backing off more slowly, nearly matching the performance of fixed backoff.

Theorem 5.5 *There exists a parameter choice for exponential backoff achieving a makespan of* $O(V \log V)$ *with error probability polynomially small in* n.

Proof. The idea is to double window sizes (as in exponential backoff) but only after repeating a window of size W $\Theta(\log W)$ times, allowing all jobs to complete when W is an accurate guess of V. Thus, we effectively backoff by a factor of only $1 + O(1/\log V)$ (instead of a factor of 2 as with binary exponential backoff). This algorithm matches the asymptotic performance of FIXED-BACKOFF$_{\Theta(V)}$. □

5.2 Size-hashed backoff

This section describes more efficient backoff protocols that improve on the traditional ones analyzed in Section 5.1. The main difficulty in dealing with different-sized jobs is that larger jobs are not likely to succeed until enough of the smaller jobs complete. This fact is exploited in Theorem 5.1's proof, where just one large job interferes with all the other jobs. The approach in this section groups jobs by size so that jobs with different sizes cannot interfere with each other for too long. In particular, we divide jobs into $\lceil \lg V \rceil$ ***job classes*** based on size. Jobs of size t_i belong to the $(\lceil \lg t_i \rceil + 1)$th job class.

I first review a "backon" protocol for constant-sized jobs, which forms a subcomponent of our new strategy. I then overview the general strategy for size-hashed backoff. Next, I discuss the mapping "hash" functions (for which the protocol is named). I present the detailed protocol, called ***size-hashed backoff***, and two specific mapping functions resulting in specific instantiations of size-hashed backoff. The first mapping yields a protocol with makespan $O(V\sqrt{\log V}\log^2 \log V)$. I then show the existence of a mapping that achieves $O(V \log^3 \log V)$ makespan. Both of these versions achieve the specified makespan with probability $1 - 1/\log^c V$ for any constant $c > 1$ with a linear dependence on c in the makespan.

Backon protocol for constant-sized jobs

A key component of size-hashed backoff is the DESCEND "backon" subprotocol. This DESCEND protocol (i.e., the participating jobs) takes three parameters: (1) *jclass*, the job class, (2) W, the window size, and (3) r, a number of repetitions. The DESCEND$(jclass, W, r)$ subprotocol consists of roughly $O(r(\log\log W + \log W))$ windows, having total size $O(rW)$. Specifically, it begins with a window of size W. DESCEND then "backs on," reducing the size of each subsequent window by a constant fraction, until reaching a window of size $W/\log W$. The size-$W/\log W$ window is then repeated $\Theta(\log W)$ times. Thus, the total size of all windows is $\Theta(W)$ while decreasing by a constant fraction, and $\Theta(\log W \cdot W/\log W) = \Theta(W)$ when repeating the smallest window, for total size $\Theta(W)$. To achieve higher probability of success, this entire process is repeated r times.

Suppose there are m jobs, all having the same size, with total volume $V' < \alpha W$,for some constant α, and that these jobs all begin DESCEND at the same time with window size W. Then, all jobs finish during the DESCEND subprotocol with probability at least $1 - 1/2^r$. The main idea of the analysis here is that once the window has size $\Omega(V')$, then a constant fraction of the jobs should complete. At this point, the protocol backs on, using windows that have smaller and smaller sizes.

A constant fraction of the jobs should complete in each of these smaller windows until reaching the last window of size $W/\log W$. At this point, the remaining jobs should have total volume $O(W/\log W)$. The DESCEND protocol then essentially performs a FIXED-BACKOFF$_{W/\log W}$ for $\log W$ windows, giving all the remaining jobs a good chance of completing. In particular, there is a constant probability that all jobs complete. Repeating r times drives the probability up to $1 - 1/2^r$.

Since a close variant of DESCEND has been previously analyzed by Gereb-Graus and Tsantilas [104], I omit the proof here.

Overview of size-hashed backoff

As in exponential backoff, size-hashed backoff proceeds by repeated doubling on the estimated volume. We refer to each iteration as a ***round***. The algorithm completes in (or before) the first round in which the estimated volume is sufficiently large ($V' > V$). Each round of the protocol proceeds in ***phases***. When the estimated volume is sufficiently large, in each phase the number of ***nonempty*** job classes — those with jobs remaining — is reduced by a constant fraction.

In the first phase, each job class runs separately. That is, take a time interval of size $\Theta(rV)$, where $r = \Theta(\log\log V)$ is a number of repetitions for the DESCEND protocol, and divide it into $\lg V$ size-$\Theta(rV/\log V)$ "buckets," one for each job class. Specifically, bucket i is designated for the jobs in job class i (i.e., those jobs j with size $2^{i-2} < t_j \leq 2^{i-1}$). During the ith bucket, each job in the ith job class runs the DESCEND subprotocol for time $\Theta(rV/\log V)$. If the volume in the job class is small enough — specifically, $O(V/\log V)$ — then that job class ***completes***, i.e., becomes empty. Since the volume is distributed among various job classes, a constant fraction of the job classes have small enough volume to complete. In particular, a simple counting argument shows that at least $1/2$ of the $\lg V$ job classes have volume at most $2V/\lg V$. We conclude that after $O(rV) = O(V\log\log V)$ time, at least half the job classes are empty.

It would be ideal if, during a second phase, we could allocate buckets for only the nonempty job classes. Since at least half the job classes are empty, we can, in principle, allocate half as many buckets of twice the size and run DESCEND for each bucket. Once again, at least half of the job classes have a small enough volume to complete. After $\lg\lg V$ phases following this process, there is a single nonempty job class in a $\Theta(rV)$-size bucket, and hence this last job class completes. Since each of the phases takes time $\Theta(V\log\log V)$, the resulting makespan is $O(V\log^2\log V)$.

The problem with this approach is that jobs have no *a priori* knowledge as to which job classes become empty during a given phase, and they cannot observe this information. Surprisingly, we can still resurrect the spirit of this idea.

Consider again the situation after the first phase, when an unknown half of the $\lg V$ job classes are empty. If we randomly choose two job classes from the entire set of $\lg V$, there is a $1/2$ probability that exactly one of the two job classes is empty. Thus, if we assign two randomly chosen job classes to each of the buckets, we expect half the buckets to contain exactly one non-empty job class. Moreover, some fraction of these classes (in expectation) contain a volume that is now small enough to complete (by a counting argument). Hence, in $O(\log\log V)$ phases, all the job classes complete.

This approach is also not feasible, as it involves coordination across entire job classes to choose the same random bucket. The approach is, however, closer to the size-hashed protocol that we describe later in the section.

To generalize, size-hashed backoff creates a deterministic mapping from $\lg V$ job classes to a set of fewer than $\lg V$ buckets. Given any small set of nonempty job classes, the mapping has the property that a constant fraction are assigned to their own bucket,[1] thus allowing them to complete

[1]This property is similar to the collision property of a hash function. It also appears to have close connections to

using DESCEND. To ensure this property, size-hashed backoff uses extra buckets, resulting in a makespan of $O(V \log^3 \log V)$ for the fastest variant.

Mapping job classes to buckets

Here, I define the mapping problem more formally. We are given η objects $X = \{x_1, x_2, \ldots, x_\eta\}$ and some integer $m < \eta$. Consider a mapping $F_{m\eta} : X \to \wp(B)$ of objects to subsets of buckets $B = \{B_1, B_2, \ldots\}$. For example, $F_{m\eta}(x_1) = \{B_1, B_7, B_{10}\}$ indicates that object x_1 maps to buckets B_1, B_7, and B_{10}.

A mapping is an ***α-good mapping***, with $0 < \alpha \leq 1$, if for each size-m subsets $Y \subseteq X$, there exists a size-$\lceil \alpha m \rceil$ subset of Y in which each object is assigned its own bucket. More formally, a mapping is α-good if for any size-m subset Y, there exists a size-$\lceil \alpha m \rceil$ subset $Z \subseteq Y$ such that for all elements $z \in Z$, there exists some bucket $b \in F_{m\eta}(z)$ such that no other object $y \in Y - \{z\}$ maps to the same bucket b.

This "good mapping" property is exactly what we need for size-hashed backoff. In the backoff setting, $\eta = \lceil \lg V \rceil$ is the number of job classes. In any phase, we have an estimate (specifically, an upper bound) of the number m of nonempty job classes; we do not know, however, which classes are nonempty. We want at least a constant fraction of them to end up assigned to their own buckets. We can then ensure that a constant fraction of job classes complete. For example, if a phase has buckets of size $2rV/m$ (i.e., at most $m/2$ job classes are "too big" to complete) and the mapping is a $3/4$-good mapping, then at least $m/4$ of the nonempty job classes must be small enough to complete in a bucket *and* be mapped to a unique bucket.

The size-hashed backoff algorithm considers good mappings of a simplified form, making the functions (slightly) easier to think about. Rather than having arbitrary functions from objects to bucket sets, split the buckets into "collections" of consecutive buckets. Each object is mapped to exactly one bucket in each collection. Construct the mapping $\mathcal{F}_{m\eta}$ as a sequence of functions $\mathcal{F}_{m\eta} = \{f_{m\eta 1}, f_{m\eta 2}, \ldots, f_{m\eta s_{m\eta}}\}$ such that $f_{m\eta i} : X \to B$ maps an object to a single bucket in the ith collection. Define $s_{m\eta} = |\mathcal{F}_{m\eta}|$ to be the cardinality of the set of functions in the mapping $\mathcal{F}_{m\eta}$. Adding more functions to $\mathcal{F}_{m\eta}$ increases the chance of achieving α-goodness. Define $r_{m\eta i}$ to be the ***range*** of, or the number of buckets used by, the function $f_{m\eta i}$.

Size-hashed protocol

I now give the protocol for size-hashed backoff in more detail, assuming an α-good mapping $\mathcal{F}_{m\eta} = \{f_{m\eta i}\}$. Figure 5-1 gives pseudocode for the size-hashed protocol. I will argue that all jobs eventually make successful run attempts with probability at least $1 - 1/\lg^c V$ for any constant $c > 1$. Since the makespan depends on the size and range of the mapping, however, I defer that discussion to the particular variants later in the section.

Recall that size-hashed backoff executes in rounds (lines 3–18), and we repeatedly double the estimated volume in each round (line 4). Each round is divided into phases (lines 7–17), and phases are constructed so that a constant fraction of job classes should complete using the α-good mapping $\mathcal{F}_{m\eta}$. Each phase is subdivided into ***subphases*** (lines 10–15) which correspond to each function $f_{m\eta i}$ in the mapping $\mathcal{F}_{m\eta}$, so each job class maps to exactly one bucket in each subphase. The α-goodness property guarantees that at least αm of the m nonempty job classes are assigned to unique buckets. The buckets use the geometrically-decreasing DESCEND protocol to ensure that jobs complete when (1) the buckets are large enough, and (2) $\mathcal{F}_{m\eta}$ assigns a unique bucket (line 14).

---

expanders, specifically lossless, bipartite expanders.

```
SIZE-HASHED-BACKOFF(t_i) ▷ t_i is the job size of process i.
1  V' ← 1
2  jclass ← ⌈lg t_i⌉ + 1
3  repeat ▷ Each iteration is a round.
4          V' ← 2V'
5          η ← lg V'
6          m ← η ▷ m bounds the number of nonempty job classes.
7          repeat ▷ Each iteration is a phase.
8                  wsize ← c_1 V'/m ▷ Window size for DESCEND.
9                  bsize ← c_3 wsize lg lg V' ▷ Bucket for DESCEND iteration.
                   ▷ s_mη is the number of functions in the mapping.
                   ▷ Iterate over subphases/functions.
10                 for i ← 1 to s_mη
11                     do ▷ r_mηi is the number of buckets used by f_mηi.
                          ▷ Iterate over buckets.
12                        for bucket ← 1 to r_mηi
13                            do if f_mηi(jclass) = bucket
14                                  then DESCEND(jclass, wsize, c_3 lg lg V')
15                                  else Wait bsize time.
16                 m ← ⌊m/c_2⌋
17            until m = 0 ▷ End loop over phases.
18    until job i executes ▷ Ends the loop over rounds.
```

Figure 5-1: Pseudocode for the size-hashed backoff protocol, described form the perspective of a single job with size t_i.

Consider the ith phase, during which there should be (at most) m nonempty job classes remaining. During this phase, the protocol creates $s_{m\eta}$ subphases, where subphase j uses $r_{m\eta j}$ buckets of size $bsize = \Theta(rV/m)$ (lines 8–9). Thus, the total length of the ith phase is $\sum_{j=1}^{s_{m\eta}} r_{m\eta j}\Theta(rV/m)$. To understand what these numbers mean, consider the "ideal" mapping in which each job knows exactly which job classes are empty; in this case $s_{m\eta} = 1$ and $r_{m\eta 1} = m$, giving a total phase length of $\Theta(rV) = \Theta(V \log\log V)$.

The following theorem states that SIZE-HASHED-BACKOFF completes all the jobs in $\lg V + O(1)$ rounds (i.e., when the window size is $\Theta(V)$). I later analyze the length of each round — and hence the makespan — in the context of the specific family of mappings $\mathcal{F}$, which determines the number of buckets.

Theorem 5.6 *Suppose n jobs with volume V execute* SIZE-HASHED-BACKOFF*, beginning at the same time. Suppose also that $\mathcal{F}$ is an α-good mapping for some constant α. If we set $c_1 = 2/\alpha$, $c_2 = 2/(2-\alpha)$, and $c_3 = c+2$, where c_1, c_2, c_3 are the constants from the pseudocode, then all n jobs complete before the $(\lg V + O(1))$th round with probability at least $1 - 1/\lg^c V$, for any $c \geq 1$.*

Proof. This body of this proof proves the invariant that during the round with $V < V' \leq 2V$, m is an upper bound on the number of nonempty job classes. There are initially at most η job classes (since $V' > V$). Thus, setting $m \leftarrow \lg V'$ in line 6 satisfies the invariant at the start of the round. We use induction over the phases to show that the invariant holds.

Since the total volume of jobs is $V < V'$, there can be at most $\lfloor m/c_1 \rfloor$ job classes with volume exceeding $c_1 V'/m$. These "large" job classes do not (necessarily) complete during the phase. Since $\mathcal{F}$ is an α-good mapping, at most $m - \lceil \alpha m \rceil$ of the nonempty job classes *do not* end up in their own bucket in any subphase in lines 10–15. These "colliding" job classes also do not (necessarily) complete during the phase.

There are at most $m - \lceil \alpha m \rceil + \lfloor m/c_1 \rfloor \leq \lfloor m(1 - \alpha + 1/c_1) \rfloor$ large or colliding job classes. As long as $\alpha > 1/c_1$, then $0 < 1 - \alpha + 1/c_1 < 1$. If we set $1/c_1 = 1 - 1/c_2 = \alpha/2$, then we have $1/c_2 = 1 - \alpha + 1/c_1$, with $c_2 > 1$. Thus, at most $\lfloor m/c_2 \rfloor$ job classes are large or colliding.

Consider a noncolliding, nonlarge job class. The DESCEND protocol guarantees that all jobs in this job class complete during line 14 with probability at least $1 - 1/\lg^{c_3} V' > 1 - 1/\lg^{c_3} V$. Taking a union bound across all $\lceil \lg V \rceil$ job classes gives probability at least $1 - 1/\lg^{c_3 - 1} V$. Thus, with this probability, at most $\lfloor m/c_2 \rfloor$ job classes survive the phase.

Taking a union bound across all $\Theta(\lg \lg V)$ phases gives a success probability that is at least $1 - 1/\lg^{c_3 - 2} V$. We conclude by setting $c_3 = c + 2$. □

We now provide two α-good mappings and analyze the resulting performance.

Analysis of a 1-good mapping

This section presents a 1-good mapping based on a simple modulo function, which results in a makespan of $\Theta(V\sqrt{\log V}\log^2 \log V)$ for size-hashed backoff.

Let $g_{m\eta i}$ be the identity function: $g_{m\eta i}(x_j) = j$. (That is, the jth object maps to bucket j.) Recall that each function $g_{m\eta i}$ maps objects to exactly one bucket in collection i. By construction, each collection contains η buckets. Let $f_{m\eta i}(x_j) = j \pmod i$. Here, the ith collection contains i buckets. I now define a 1-good mapping, parameterized by a variable t (defined later), as follows:

$$\mathcal{F}_{m\eta} = \begin{cases} \{g_{m\eta 1}\} & : \text{ if } m > \eta/t \\ \{f_{m\eta 1}, f_{m\eta 2}, \ldots, f_{m\eta \Theta(m \log \eta)}\} & : \text{ if } m \leq \eta/t\,. \end{cases}$$

Lemma 5.7 *The functions $\mathcal{F}_{m\eta}$ form a 1-good mapping.*

Proof. If $m > \eta/t$, then $\mathcal{F}$ is the identity mapping, which is 1-good. Assume that $m \leq \eta/t$. Consider any two objects $x_j \neq x_k \in X$. Consider C prime numbers $p_1, p_2, \ldots, p_C$, each of which is at least $m \lg \eta$, and suppose by contradiction they collide everywhere, i.e., $f_{m\eta p_\ell}(x_j) = f_{m\eta p_\ell}(x_k)$ for all $\ell \in 1, 2, \ldots, C$. Then, the difference between j and k must be divisble by each of these prime numbers, and hence at least $(m \lg \eta)^C$. Choosing $C > \lg \eta / \lg(m \lg \eta)$, which implies that $(m \lg \eta)^C > \eta$, yields a contradiction, since $|j - k|$ can be at most η. Thus, x_j and x_k can collide in at most $C - 1$ of the functions $\{f_{m\eta p_\ell}\}$.

Recall that there are $\Theta(m \log \eta)$ functions $f_{m,\eta,i}$. Thus for a sufficiently large constant in the Θ-notation, there are at least $mC = m \lg \eta / \lg(m \lg \eta)$ functions $f_{m\eta i}$ in which i is prime and $i \geq m \lg \eta$. For a given set Y of size $\leq m$ and a given object $x_j \in Y$, there must be one of the mC functions in which x_j does not collide with any of the $\leq m$ objects in Y, implying that $\mathcal{F}$ is 1-good. □

The following lemma calculates the running time of each round.

Lemma 5.8 *The running time for a single round of size-hashed backoff, with 1-good mapping $\mathcal{F}$ is $O(V'\sqrt{\log V'}\log^2 \log V')$.*

Proof. First, consider a phase corresponding to $m > \eta/t$ nonempty job classes. Recall that in this case, the number $s_{m\eta}$ of collections 1 and the number $r_{m\eta 1}$ of buckets in a collection is η. Thus, the

running time of the phase is $r_{m,\eta,1}$ *bsize* $= \Theta(\eta V' \log\log V'/m)$ (lines 8–9). Since m decreases geometrically (by c_2 in each phase), the sum of running times of all phases with $m > \eta/t$ can be bounded by the phase with minimum m, which is thus $O(tV' \log\log V')$.

Consider next a phase where $m \leq \eta/t$. Then, the number $s_{m\eta}$ of collections is $\Theta(m \log \eta)$ and the number $r_{m\eta i}$ of buckets per collection is i. Thus, a phase completes in *bsize* $\sum_{i=1}^{s_{m\eta}} r_{m\eta} =$ *bsize* $O(m^2 \log^2 \eta)$ time. Substituting for *bsize* and η, we have $O(V' \log\log V' m^2 \log^2 \eta/m) = O(mV' \log^3 \log V')$. Since m decreases geometrically, the sum of running times of all phases with $m \leq \eta/t$ can be bounded by the phase with maximum m, which is thus $O(V' \log V' \log^3 V'/t)$.

Thus, the total duration of a round is $\Theta(tV' \log\log V' + V' \log V' \log^3 V'/t)$. Setting $t = \sqrt{\log V'} \log\log V'$ yields a time of $\Theta(V'\sqrt{\log V'} \log^2 \log V')$. □

Theorem 5.6 shows that size-hashed backoff, when using this 1-good function, terminates in $\lg V + O(1)$ rounds. Together with Lemma 5.8, we can conclude:

Corollary 5.9 *Suppose that n jobs having total volume V begin executing size-hashed backoff, using the 1-good mapping $\mathcal{F}$, at the same time. Then, all n jobs make a successful run attempt in time $O(V\sqrt{\log V} \log^2 \log V)$ with probability at least $1 - 1/\lg^c V$, for any $c \geq 1$ and sufficiently large V.* □

Analysis of a $1/2$-good mapping

The final version of size-hashed backoff achieves a makespan of $O(V \log^3 \log V)$. This algorithm relies on a more efficient α-good mapping, which I show exists using the probabilistic method. The goal of this section is to prove the existence of a $1/2$-good mapping where $s_{m\eta} = \Theta(\log\log V)$ and $r_{m\eta i} = \Theta(m)$, yielding as corollary of Theorem 5.6 a makespan of $O(V \log^3 \log V)$.

There are three $\log\log V$ factors in the makespan. Two of these come from the general structure of size-hashed backoff: there are $\Theta(\log\log V)$ phases reducing the number of nonempty job classes, and DESCEND runs for $\Theta(\log\log V)$ windows. The third $\log\log V$ factor arises from the number $s_{m\eta}$ of functions.

I first present a preliminary "balls and bins" lemma, which will be used later to existence of an appropriate $1/2$-good mapping.

Lemma 5.10 *Suppose there are m balls thrown uniformly at random into cm bins, for $c > 15$. Then for some $0 < \delta < 1$, the probability that fewer than $m/2$ bins have exactly one ball is $\leq \delta^{cm}$.*

Proof. Let ρ be the number of bins with at least one ball. We first show that $\Pr[\rho \leq 3m/4] \leq \delta^{cm}$. First, we calculate the probability that $\rho \leq i$, for some particular $0 \leq i \leq m$. Specifically, for each subset of i bins, we calculate the probability that all m balls land in one of those i bins:

$$\Pr[\rho \leq i] \leq \left(\frac{i}{cm}\right)^m \binom{cm}{i} \leq e^i \left(\frac{i}{cm}\right)^{m-i} .$$

In particular, for $i = 3m/4$, we conclude that:

$$\Pr[\rho \leq 3m/4] \leq e^{3m/4} \left(\frac{3m/4}{cm}\right)^{m/4} \leq \left(\left(\frac{3e^3}{4c}\right)^{1/4c}\right)^{cm} .$$

Choose $\delta = \left(\frac{3e^3}{4c}\right)^{1/4c}$, and notice that for $c > 15$, $\delta < 1$, as required.

Next, when ρ bins have at least one ball, it is clear that at most $m - \rho$ bins have at least two balls. From this it follows that at least $\rho - (m - \rho) = 2\rho - m$ bins have exactly one ball. Therefore, with probability at least $1 - \delta^{cm}$, $\rho > 3m/4$, implying that $m/2$ bins have exactly one ball. □

I next show that there exist appropriate $1/2$-good functions. This proof is merely existential, not constructive.

Theorem 5.11 *There exists a set of* $1/2$*-good functions* $\mathcal{F}_{m\eta} = \{f_{m\eta i}\}$ *with range* $r_{m\eta i} = \Theta(m)$ *and a number* $s_{m\eta}$ *of subphases of* $\Theta(\log \eta)$.

Proof. Show the existence of the functions $\{f_{m\eta i}\}$ using the probabilistic method. For each η and $m \le \eta$, and for each $i \in [1, s_{m\eta}]$, choose $f_{m\eta i}$ at random: choose $f_{m\eta i}(j)$, $j \le \eta$, uniformly at random from the range $[1, r_{m\eta i}]$. The goal is to show that with probability strictly larger than 0, the resulting family of functions is $1/2$-good.

First, let us calculate for a fixed set Y of size at most m and a fixed $i \in [1 \ldots s_{m,\eta}]$, the probability that $f_{m\eta i}$ is $1/2$-good. Think of each of the m values $f_{m\eta i}(j)$, $j \in Y$, as a ball that is thrown uniformly at random into $r_{m\eta i} = \Theta(m)$ bins. By Lemma 5.10, there exists some $\delta < 1$ such that the probability that fewer than 1/2 the "balls" are in their own bin is at most $\delta^{\Theta(n)}$.

The probability that for all $i \in [1, s_{m\eta}]$ $1/2$-goodness is violated is, therefore, at most $\delta^{\Theta(m)s_{m\eta}} \le \delta^{\Theta(m \lg \eta)}$, since each function i is selected independently.

Finally, let us compute the number of possible sets Y of size m. In particular, there are $\binom{m+\eta}{m} \le \binom{2\eta}{m}$ ways to distribute m possible jobs over η job classes. This expression reduces to:

$$\begin{aligned}\binom{2\eta}{m} &\le \left(\frac{2e\eta}{m}\right)^m \\ &\le e^m 2^{m \lg \eta} .\end{aligned}$$

Applying a union bound over the possible sets Y gives

$$\begin{aligned}\Pr[\mathcal{F}_{m\eta} \text{ is not } 1/2\text{-good}] &\le \delta^{\Theta(m \lg \eta)} e^m 2^{m \lg \eta} \\ &\le 2^{-\Theta(m \lg \eta \lg 1/\delta) + m \lg e + m \lg \eta} \\ &\le 2^{-\Theta(m \lg \eta) + m \lg e} \\ &< 1\end{aligned}$$

We conclude that with strictly positive probability, the randomly chosen $\mathcal{F}_{m\eta}$ is $1/2$-good, and thus a $1/2$-good mapping with the desired properties exists. □

I conclude this section by calculating the makespan as a corollary of Theorem 5.6:

Corollary 5.12 *Suppose that n jobs having total volume V begin executing size-hashed backoff at the same time. Suppose also that size-hashed backoff uses a* $1/2$*-good mapping* $\mathcal{F}$ *with sizes* $s_{m\eta} = \Theta(\log \eta)$ *and ranges* $r_{m\eta i} = \Theta(m)$*. Then, all n jobs make successful run attempts in time* $O(V \log^3 \log V)$ *with probability at least* $1 - 1/\lg^c V$ *for any constant* $c \ge 1$ *and sufficiently large* V.

Proof. By Theorem 5.6, all jobs complete by round $\lg V + O(1)$ with appropriate probability. Each phase in the round takes time $bsize \sum_{i=1}^{s_{m\eta}} r_{m\eta i} = bsize\, \Theta(m \log \eta)$. Substituting for $bsize$ and η yields phase length $\Theta(V \log^2 \log V)$. Summing over $\Theta(\log \log V)$ phases completes the proof. □

5.3 Concluding remarks

This chapter studied randomized backoff protocols when jobs differ in size. I analyzed binary exponential backoff and showed that it performs poorly, yielding makespan $V2^{\Theta(\sqrt{\log n})}$. A slower rate of backoff achieves makespan $\Theta(V \log V)$. The main result is the size-hashed backoff protocols, where the backoff strategy depends on the job lengths. Size-hashed backoff (potentially) reduces the makespan to only $O(V \log^3 \log V)$.

These results raise many questions. First, what are the lower bounds? Is a linear makespan possible? Next, on a simple channel jobs learn about contention only by making run attempts; what if jobs can listen on the channel without running? Also, a job i learns that a run attempt has failed only after the full t_i time steps. What if jobs learn of failure as soon as a collision occurs, enabling them to abort early? Can exponential backoff and its variants perform better?

This chapter considers the batch problem, where jobs arrive at time 0. Ultimately I hope to understand the online problem, where jobs arrive over time. What can be proved for queuing-theory arrivals? Are there reasonable worst-case models, similar to those in [43], that apply to backoff with different-size jobs?

Chapter 6

Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks

When multiplying a large $n \times n$ sparse matrix A having nnz nonzeros by a dense n-vector x, the memory bandwidth for reading A can limit overall performance. Consequently, most algorithms to compute Ax store A in a compressed format. One simple "tuple" representation stores each nonzero of A as a triple consisting of its row index, its column index, and the nonzero value itself. This representation, however, requires storing $2\,nnz$ row and column indices, in addition to the nonzeros. The current standard storage format for sparse matrices in scientific computing, ***compressed sparse rows (CSR)*** [175], is more efficient, because it stores only $n + nnz$ indices or pointers. This reduction in storage of CSR compared with the tuple representation tends to result in faster serial algorithms.

In the domain of parallel algorithms, however, CSR has its limitations. Although CSR lends itself to a simple parallel algorithm for computing the matrix-vector product Ax, this storage format does not admit an efficient parallel algorithm for computing the product $A^{\mathsf{T}}x$, where A^{T} denotes the transpose of the matrix A — or equivalently, for computing the product $x^{\mathsf{T}}A$ of a row vector x^{T} by A. Although one could use ***compressed sparse columns (CSC)*** to compute $A^{\mathsf{T}}x$, many applications, including iterative linear system solvers such as biconjugate gradients and quasi-minimal residual [175], require both Ax and $A^{\mathsf{T}}x$. One could transpose A explicitly, but computing the transpose for either CSR or CSC formats is expensive. Moreover, since matrix-vector multiplication for sparse matrices is generally limited by memory bandwidth, it is desirable to find a storage format for which both Ax and $A^{\mathsf{T}}x$ can be computed in parallel without performing more than nnz fetches of nonzeros from the memory to compute either product.

This chapter presents a new storage format called ***compressed sparse blocks (CSB)*** for representing sparse matrices. Like CSR and CSC, the CSB format requires only $n+nnz$ words of storage for indices. Because CSB does not favor rows over columns or vice versa, it admits efficient parallel algorithms for computing either Ax or $A^{\mathsf{T}}x$, as well as for computing Ax when A is symmetric and only half the matrix is actually stored. This chapter represents joint work with Aydın Buluç, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson, previously appearing in [160].

Previous work on parallel sparse matrix-vector multiplication has focused on reducing communication volume in a distributed-memory setting, often by using graph or hypergraph partitioning techniques to find good data distributions for particular matrices ([63, 193], for example). Good partitions generally exist for matrices whose structures arise from numerical discretizations of par-

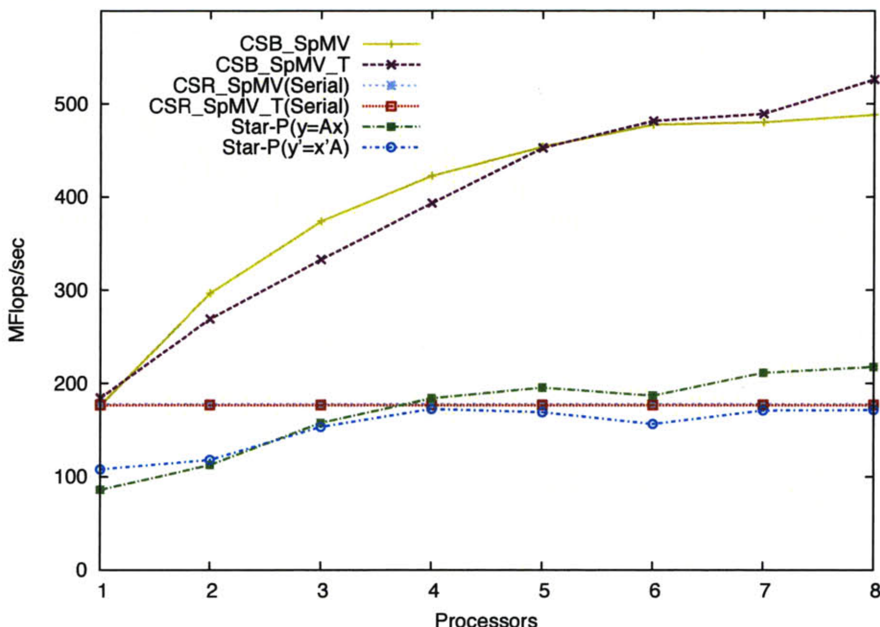


Figure 6-1: Average performance of Ax and $A^{\mathsf{T}}x$ operations on 13 different matrices from our benchmark test suite. CSB_SpMV and CSB_SpMV_T use compressed sparse blocks to perform Ax and $A^{\mathsf{T}}x$, respectively. CSR_SpMV (Serial) and CSR_SpMV_T (Serial) use OSKI [195] and compressed sparse rows without any matrix-specific optimizations. Star-P (y=Ax) and Star-P (y'=x'A) use Star-P [179], a parallel code based on CSR. The experiments were run on a ccNUMA architecture powered by AMD Opteron 8214 (Santa Rosa) processors.

tial differential equations in two or three spatial dimensions. Our work, by contrast, is motivated by multicore and manycore architectures, in which parallelism and memory bandwidth are key resources. Our algorithms are efficient in these measures for matrices with arbitrary nonzero structure.

Figure 6-1 presents an overall summary of achieved performance. The serial CSR implementation uses plain OSKI [195] without any matrix-specific optimizations. The graph shows the average performance over all our test matrices except for the largest, which failed to run on Star-P [179] due to memory constraints. The performance is measured in Mflops (Millions of FLoating-point OPerationS) per second. Both Ax and $A^{\mathsf{T}}x$ take 2 nnz flops. To measure performance, we divide this value by the time it takes for the computation to complete. Section 6.6 provides detailed performance results.

The remainder of this chapter is organized as follows. Section 6.1 discusses the limitations of the CSR/CSC formats for parallelizing Ax and $A^{\mathsf{T}}x$ calculations. Section 6.2 describes the CSB format for sparse matrices. Section 6.3 presents the algorithms for computing Ax and $A^{\mathsf{T}}x$ using the CSB format, and Section 6.4 provides a theoretical analysis of their parallel performance. Section 6.5 describes the experimental setup we used, and Section 6.6 presents the results. Section 6.7 offers some concluding remarks.

```
CSR_SPMV(A, x, y)
1  n ← A.rows
2  for i ← 0 to n − 1 in parallel
3      do y[i] ← 0
4         for k ← A.row_ptr[i] to A.row_ptr[i + 1] − 1
5             do y[i] ← y[i] + A.val[k] · x[A.col_ind[k]]
```

Figure 6-2: Parallel procedure for computing $y \leftarrow Ax$, where the $n \times n$ matrix A is stored in CSR format.

6.1 Conventional storage formats

This section describes the CSR and CSC sparse-matrix storage formats and explores their limitations when it comes to computing both Ax and $A^{\mathsf{T}}x$ in parallel. We review the work/span formulation of parallelism and show that performing Ax with CSR (or equivalently $A^{\mathsf{T}}x$ with CSC) yields ample parallelism. We consider various strategies for performing $A^{\mathsf{T}}x$ in parallel with CSR (or equivalently Ax with CSC) and why they are problematic.

The compressed sparse row (CSR) format stores the nonzeros (and ideally only the nonzeros) of each matrix row in consecutive memory locations, and it stores an index to the first stored element of each row. In one popular variant [85], CSR maintains one floating-point array $val[nnz]$ and two integer arrays, $col_ind[nnz]$ and $row_ptr[n]$ to store the matrix $A = (a_{ij})$. The row_ptr array stores the index of each row in val. That is, if $val[k]$ stores matrix element a_{ij}, then $row_ptr[i] \leq k < row_ptr[i+1]$. The col_ind array stores the column indices of the elements in the val array. That is, if $val[k]$ stores matrix element a_{ij}, then $col_ind[k] = j$.

The compressed sparse column (CSC) format is analogous to CSR, except that the nonzeros of each column, instead of row, are stored in contiguous memory locations. In other words, the CSC format for A is obtained by storing A^{T} in CSR format.

The earliest written description of CSR that we have been able to divine from the literature is an unnamed "scheme" presented in Table 1 of the 1967 article [190] by Tinney and Walker, although in 1963 Sato and Tinney [176] allude to what is probably CSR. Markowitz's seminal paper [150] on sparse Gaussian elimination does not discuss data structures, but it is likely that Markowitz used such a format as well. CSR and CSC have since become ubiquitous in sparse matrix computation [77, 87, 89, 103, 106, 175].

The following lemma states the well-known bound on space used by the index data in the CSR format (and hence the CSC format as well). By index data, we mean all data other than the nonzeros — that is, the row_ptr and col_ind arrays.

Lemma 6.1 *The CSR format uses $n \lg nnz + nnz \lg n$ bits of index data for an $n \times n$ matrix.*

For a CSR matrix A, computing $y \leftarrow Ax$ in parallel is straightforward, as shown in Figure 6-2. Procedure CSR_SPMV in the figure computes each element of the output array in parallel, and it does not suffer from race conditions, because each parallel iteration i writes to a single location $y[i]$ which is not updated by any other iteration.

As in Chapters 2–3, we measure the complexity of this code, and other codes in this chapter, in terms of work and span [72, Ch. 27]. The definitions are repeated again here for clarity:

- The ***work***, denoted by T_1, is the running time on 1 processor.
- The ***span***, denoted by T_∞, is running time on an infinite number of processors.

```
CSR_SPMV_T(A, x, y)
1  n ← A.cols
2  for i ← 0 to n − 1
3       do y[i] ← 0
4  for i ← 0 to n − 1
5       do for k ← A.row_ptr[i] to A.row_ptr[i + 1] − 1
6               do y[A.col_ind[k]] ← y[A.col_ind[k]] + A.val[k] · x[i]
```

Figure 6-3: Serial procedure for computing $y \leftarrow A^{\mathsf{T}}x$, where the $n \times n$ matrix A is stored in CSR format.

The ***parallelism*** of the algorithm is T_1/T_∞, which corresponds to the maximum possible speedup on any number of processors. Generally, if a machine has somewhat fewer processors than the parallelism of an application, a good scheduler should be able to achieve linear speedup. Thus, for a fixed amount of work, our goal is to achieve a sufficiently small span so that the parallelism exceeds the number of processors by a reasonable margin.

The work of CSR_SPMV is $\Theta(nnz)$, assuming, as we shall, that $nnz \geq n$, because the body of the outer loop starting in line 2 executes for n iterations, and the body of the inner loop starting in line 4 executes for the number of nonzeros in the ith row, for a total of nnz times.

The span of CSR_SPMV depends on the maximum number nr of nonzeros in any row of the matrix A, since that number determines the worst-case time of any iteration of the loop in line 4. The n iterations of the parallel loop in line 2 contribute $\Theta(\lg n)$ to the span, assuming that loops are implemented as binary recursion. Thus, the total span is $\Theta(nr + \lg n)$.

The parallelism is therefore $\Theta(nnz\,/(nr + \lg n))$. In many common situations, we have $nnz = \Theta(n)$, which we will assume for estimation purposes. The maximum number nr of nonzeros in any row can vary considerably, however, from a constant, if all rows have an average number of nonzeros, to n, if the matrix has a dense row. If $nr = O(1)$, then the parallelism is $\Theta(nnz\,/\lg n)$, which is quite high for a matrix with a billion nonzeros. In particular, if we ignore constants for the purpose of making a ballpark estimate, we have $nnz\,/\lg n \approx 10^9/(\lg 10^9) > 3 \times 10^7$, which is much larger than any number of processors one is likely to encounter in the near future. If $nr = \Theta(n)$, however, as is the case when there is even a single dense row, we have parallelism $\Theta(nnz\,/n) = \Theta(1)$, which limits scalability dramatically. Fortunately, we can parallelize the inner loop (line 4) using divide-and-conquer recursion to compute the sparse inner product in $\lg(nr)$ span without affecting the asymptotic work, thereby achieving parallelism $\Theta(nnz\,/\lg n)$ in all cases.

Computing $A^{\mathsf{T}}x$ serially can be accomplished by simply interchanging the row and column indices [86], yielding the pseudocode shown in Figure 6-3. The work of procedure CSR_SPMV_T is $\Theta(nnz)$, the same as CSR_SPMV.

Parallelizing CSR_SPMV_T is not straightforward, however. We shall review several strategies to see why it is problematic.

One idea is to parallelize the loops in lines 2 and 5, but this strategy yields minimal scalability. First, the span of the procedure is $\Theta(n)$, due to the loop in line 4. Thus, the parallelism can be at most $O(nnz\,/n)$, which is a small constant in most common situations. Second, in any practical system, the communication and synchronization overhead for executing a small loop in parallel is much larger than the execution time of the few operations executed in line 6.

Another idea is to execute the loop in line 4 in parallel. Unfortunately, this strategy introduces

race conditions in the read/modify/write to $y[A.\ col_ind[k]]$ in line 6.[1] These races can be addressed in two ways, neither of which is satisfactory.

The first solution involves locking column $col_ind[k]$ or using some other form of atomic update.[2] This solution is unsatisfactory because of the high overhead of the lock compared to the cost of the update. Moreover, if A contains a dense column, then the contention on the lock is $\Theta(n)$, which completely destroys any parallelism in the common case where $nnz = \Theta(n)$.

The second solution involves splitting the output array y into multiple arrays y_p in a way that avoids races, and then accumulating $y \leftarrow \Sigma_p y_p$ at the end of the computation. For example, in a system with P processors (or threads), one could postulate that processor p only operates on array y_p, thereby avoiding any races. This solution is unsatisfactory because the work becomes $\Theta(nnz + Pn)$, where the last term comes from the need to initialize and accumulate P (dense) length-n arrays. Thus, the parallel execution time is $\Theta((nnz + Pn)/P) = \Omega(n)$ no matter how many processors are available.

A third idea for parallelizing $A^{\mathsf{T}}x$ is to explicitly transpose the matrix and then CSR_SPMV. Unfortunately, parallel transposition of a sparse matrix in CSR format is costly and encounters exactly the same problems we are trying to avoid. Moreover, every element is accessed at least twice: once for the transpose, and once for the multiplication. Since the calculation of a matrix-vector product tends to be memory-bandwidth limited, this strategy is generally inferior to any strategy that accesses each element only once.

Finally, of course, we could store the matrix A^{T} in CSR format, that is, storing A in CSC format, but then computing Ax becomes difficult.

To close this section, we should mention that if the matrix A is symmetric, so that only about half the nonzeros need be stored — for example, those on or above the diagonal — then computing Ax in parallel for CSR is also problematic. For this example, the elements below the diagonal are visited in an inconvenient order, as if they were stored in CSC format.

6.2 The CSB storage format

This section describes the CSB storage format for sparse matrices and shows that it uses the same amount of storage space as the CSR and CSC formats. We also compare CSB to other blocking schemes.

For a given ***block-size parameter*** β, CSB partitions the $n \times n$ matrix A into n^2/β^2 equal-sized $\beta \times \beta$ square ***blocks***[3]

$$A = \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,n/\beta-1} \\ A_{10} & A_{11} & \cdots & A_{1,n/\beta-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n/\beta-1,0} & A_{n/\beta-1,1} & \cdots & A_{n/\beta-1,n/\beta-1} \end{pmatrix},$$

where the block A_{ij} is the $\beta \times \beta$ submatrix of A containing elements falling in rows $i\beta, i\beta + 1, \ldots, (i+1)\beta - 1$ and columns $j\beta, j\beta + 1, \ldots, (j+1)\beta - 1$ of A. For simplicity of presentation, we shall assume that β is an exact power of 2 and that it divides n; relaxing these assumptions is straightforward.

[1] In fact, if $nnz > n$, then the "pigeonhole principle" guarantees that the program has at least one race condition.

[2] No mainstream hardware supports atomic update of floating-point quantities, however.

[3] The CSB format may be easily extended to nonsquare $n \times m$ matrices. In this case, the blocks remain as square $\beta \times \beta$ matrices, and there are nm/β^2 blocks.

Many or most of the individual blocks A_{ij} are ***hypersparse*** [62], meaning that the ratio of nonzeros to matrix dimension is asymptotically 0. For example, if $\beta = \sqrt{n}$ and $nnz = cn$, the average block has dimension $\sqrt{n}$ and only c nonzeros. The space to store a block should therefore depend only on its nonzero count, not on its dimension.

CSB represents a block A_{ij} by compactly storing a triple for each nonzero, associating with the nonzero data element a row and column index. In contrast to the column index stored for each nonzero in CSR, the row and column indices lie within the submatrix A_{ij}, and hence require fewer bits. In particular, if $\beta = \sqrt{n}$, then each index into A_{ij} requires only half the bits of an index into A. Since these blocks are stored contiguously in memory, CSB uses an auxiliary array of pointers to locate the beginning of each block.

Specifically, CSB maintains a floating-point array $val[nnz]$ and additionally three integer arrays $row_ind[nnz]$, $col_ind[nnz]$, and $blk_ptr[n^2/\beta^2]$. We describe each of these arrays in turn.

The val array stores all the nonzeros of the matrix and is analogous to CSR's array of the same name. The difference is that CSR stores *rows* contiguously, whereas CSB stores *blocks* contiguously. Although each block must be contiguous, the ordering among blocks is flexible. Let $f(i,j)$ be the bijection from pairs of block indices to integers in the range $0, 1, \ldots, n^2/\beta^2 - 1$ that describes the ordering among blocks. That is, $f(i,j) < f(i',j')$ if and only if A_{ij} appears before $A_{i'j'}$ in val. We discuss choices of ordering later in this section.

The row_ind and col_ind arrays store the row and column indices, respectively, of the elements in the val array. These indices are relative to the block containing the particular element, not the entire matrix, and hence they range from 0 to $\beta - 1$. That is, if $val[k]$ stores the matrix element $a_{i\beta+r, j\beta+c}$, which is located in the rth row and cth column of the block A_{ij}, then $row_ind = r$ and $col_ind = c$. As a practical matter, we can pack a corresponding pair of elements of row_ind and col_ind into a single integer word of $2\lg\beta$ bits so that they make a single array of length nnz, which is comparable to the storage needed by CSR for the col_ind array.

The blk_ptr array stores the index of each block in the val array, which is analogous to the row_ptr array for CSR. If $val[k]$ stores a matrix element falling in block A_{ij}, then $blk_ptr[f(i,j)] \le k < blk_ptr[f(i,j)+1]$.

The following lemma states the storage used for indices in the CSB format.

Lemma 6.2 *The CSB format uses* $(n^2/\beta^2)\lg nnz + 2\, nnz \lg\beta$ *bits of index data.*

Proof. Since the val array contains nnz elements, referencing an element requires $\lg nnz$ bits, and hence the blk_ptr array uses $(n^2/\beta^2)\lg nnz$ bits of storage.

For each element in val, we use $\lg\beta$ bits to represent the row index and $\lg\beta$ bits to represent the column index, requiring a total of $nnz\lg\beta$ bits for each of row_ind and col_ind. Adding the space used by all three indexing arrays completes the proof. □

To better understand the storage requirements of CSB, we present the following corollary for $\beta = \sqrt{n}$. In this case, both CSR (Lemma 6.1) and CSB use the same storage.

Corollary 6.3 *The CSB format uses* $n\lg nnz + nnz\lg n$ *bits of index data when* $\beta = \sqrt{n}$. □

Thus far, we have not addressed the ordering of elements within each block or the ordering of blocks. Within a block, we use a Z-Morton ordering [157], storing first all those elements in the top-left quadrant, then the top-right, bottom-left, and finally bottom-right quadrants, using the same layout recursively within each quadrant. In fact, these quadrants may be stored in any order, but the recursive ordering is necessary for our algorithm to achieve good parallelism within a block.

The choice of storing the nonzeros within blocks in a recursive layout is opposite to the common wisdom for storing dense matrices [90]. Although most compilers and architectures favor conventional row/column ordering for optimal prefetching, the choice of layout within the block becomes

less significant for sparse blocks as they already do not take full advantage of such features. More importantly, a recursive ordering allows us to efficiently determine the four quadrants of a block using binary search, which is crucial for parallelizing individual blocks.

Our algorithm and analysis do not, however, require any particular ordering among blocks. A Z-Morton ordering (or any recursive ordering) seems desirable as it should get better performance in practice by providing spatial locality, and it matches the ordering within a block. Computing the function $f(i, j)$, however, is simpler for a row-major or column-major ordering among blocks.

Comparison with other blocking methods

A blocked variant of CSR, called ***BCSR***, has been used for improving register reuse [121]. In BCSR, the sparse matrix is divided into small dense blocks that are stored in consecutive memory locations. The pointers are maintained to the first block on each row of blocks. BCSR storage is converse to CSB storage, because BCSR stores a sparse collection of dense blocks, whereas CSB stores a dense collection of sparse blocks. We conjecture that it would be advantageous to apply BCSR-style register blocking to each individual sparse block of CSB.

Nishtala *et al.* [162] have proposed a data structure similar to CSB in the context of cache blocking. Our work differs from theirs in two ways. First, CSB is symmetric without favoring rows over columns. Second, our algorithms and analysis for CSB are designed for parallelism instead of cache performance. As shown in Section 6.4, CSB supports ample parallelism for algorithms computing Ax and $A^{\mathsf{T}}x$, even on sparse and irregular matrices.

Blocking is also used in dense matrices. The Morton-hybrid layout [4, 143], for example, uses a parameter equivalent to our parameter β for selecting the block size. Whereas in CSB we store elements in a Morton ordering within blocks and an arbitrary ordering among blocks, the Morton-hybrid layout stores elements in row-major order within blocks and a Morton ordering among blocks. The Morton-hybrid layout is designed to take advantage of hardware and compiler optimizations (within a block) while still exploiting the cache benefits of a recursive layout. Typically the block size is chosen to be 32×32, which is significantly smaller than the $\Theta(\sqrt{n})$ block size we propose for CSB. The Morton-hybrid layout, however, considers only dense matrices, for which designing a matrix-vector multiplication algorithm with good parallelism is significantly easier.

6.3 Matrix-vector multiplication using CSB

This section describes a parallel algorithm for computing the sparse-matrix dense-vector product $y \leftarrow Ax$, where A is stored in CSB format. This algorithm can be used equally well for computing $y \leftarrow A^{\mathsf{T}}x$ by switching the roles of row and column. We first give an overview of the algorithm and then describe it in detail.

At a high level, the CSB_SPMV multiplication algorithm simply multiplies each "blockrow" by the vector x in parallel, where the ith ***blockrow*** is the row of blocks $(A_{i0}A_{i1} \cdots A_{i,n/\beta-1})$. Since each blockrow multiplication writes to a different portion of the output vector, this part of the algorithm contains no races due to write conflicts.

If the nonzeros were guaranteed to be distributed evenly among block rows, then the simple blockrow parallelism would yield an efficient algorithm with n/β-way parallelism by simply performing a serial multiplication for each blockrow. One cannot, in general, guarantee that distribution of nonzeros will be so nice, however. In fact, sparse matrices in practice often include at least one dense row containing roughly n nonzeros, whereas the number of nonzeros is only $nnz \approx cn$ for some small constant c. Thus, performing a serial multiplication for each blockrow yields no better

```
CSB_SPMV(A, x, y)
 1  for i ← 0 to n/β − 1 in parallel                  ▷ For each blockrow.
 2      do Initialize a dynamic array R_i
 3         R_i[0] ← 0
 4         count ← 0                              ▷ Count nonzeroes in chunk.
 5         for j ← 0 to n/β − 2
 6             do count ← count + nnz(A_ij)
 7                if count + nnz(A_{i,j+1}) > Θ(β)
 8                  then ▷ End the chunk, since the next block
                           ▷ makes it too large.
 9                         append j to R_i               ▷ Last block in chunk.
10                         count ← 0
11         append n/β − 1 to R_i
12         CSB_BLOCKROWV(A, i, R_i, x, y[iβ .. (i+1)β − 1])
```

Figure 6-4: Pseudocode for the matrix-vector multiplication $y \leftarrow Ax$. The procedure CSB_BLOCKROWV (pseudocode for which can be found in Figure 6-5) as called here multiplies the blockrow by the vector x and writes the output into the appropriate region of the output vector y. The notation $x[a \,..\, b]$ means the subarray of x starting at index a and ending at index b. The function $nnz(A_{ij})$ is a shorthand for $A.\,blk_ptr[f(i,j)+1] - A.\,blk_ptr[f(i,j)]$, which calculates the number of nonzeros in the block A_{ij}. For conciseness, we have overloaded the $\Theta(\beta)$ notation (in line 7) to mean "a constant times β"; any constant suffices for the analysis, and we use the constant 3 in our implementation.

than c-way parallelism.

To make the algorithm robust to matrices of arbitrary nonzero structure, we must parallelize the blockrow multiplication when a blockrow contains "too many" nonzeros. This level of parallelization requires care to avoid races, however, because two blocks in the same blockrow write to the same region within the output vector. Specifically, when a blockrow contains $\Omega(\beta)$ nonzeros, we recursively divide it "in half," yielding two subblockrows, each containing roughly half the nonzeros. Although each of these subblockrows can be multiplied in parallel, they may need to write to the same region of the output vector. To avoid the races that might arise due to write conflicts between the subblockrows, we allocate a temporary vector to store the result of one of the subblockrows and allow the other subblockrow to use the output vector. After both subblockrow multiplications complete, we serially add the temporary vector into the output vector.

To facilitate fast subblockrow divisions, we first partition the blockrow into "chunks" of consecutive blocks, each containing at most $O(\beta)$ nonzeros (when possible) and $\Omega(\beta)$ nonzeros on average. The lower bound of $\Omega(\beta)$ will allow us to amortize the cost of writing to the length-β temporary vector against the nonzeros in the chunk. By dividing a blockrow "in half," we mean assigning to each subblockrow roughly half the chunks.

Figure 6-4 gives the top-level algorithm, performing each blockrow vector multiplication in parallel. The "**for** ... **in parallel do**" construct means that each iteration of the **for** loop may be executed in parallel with the others. For each loop iteration, we partition the blockrow into chunks in lines 2–11 and then call the blockrow multiplication in line 12. The array R_i stores the indices of the last block in each chunk; specifically, the kth chunk, for $k > 0$, includes blocks $(A_{i,R_i[k-1]+1} A_{i,R_i[k-1]+2} \cdots A_{i,R_i[k]})$. A chunk consists of either a single block containing $\Omega(\beta)$ nonzeros, or it consists of many blocks containing $O(\beta)$ nonzeros in total. To compute chunk

```
CSB_BLOCKROWV(A, i, R, x, y)
13  if R.length = 2                          ▷ The subblockrow is a single chunk.
14     then ℓ ← R[0] + 1                               ▷ Leftmost block in chunk.
15          r ← R[1]                                  ▷ Rightmost block in chunk.
16          if ℓ = r
17             then ▷ The chunk is a single (dense) block.
18                  start ← A.blk_ptr[f(i, ℓ)]
19                  end ← A.blk_ptr[f(i, ℓ) + 1] − 1
20                  CSB_BLOCKV(A, start, end, β, x, y)
21             else ▷ The chunk is sparse.
22                  multiply y ← (A_{iℓ} A_{i,ℓ+1} ··· A_{ir})x serially
23          return
    ▷ Since the block row is "dense," split it in half.
24  mid ← ⌈R.length/2⌉ − 1                          ▷ Divide chunks in half.
    ▷ Calculate the dividing point in the input vector x.
25  xmid ← β · (R[mid] − R[0])
26  allocate a length-β temporary vector z, initialized to 0
27  in parallel
28      do CSB_BLOCKROWV(A, i, R[0 .. mid], x[0 .. xmid −1], y)
29      do CSB_BLOCKROWV(A, i, R[mid .. R.length −1],
                                 x[xmid .. x.length −1], z)
30  for k ← 0 to β − 1
31      do y[k] ← y[k] + z[k]
```

Figure 6-5: Pseudocode for the subblockrow vector product $y \leftarrow (A_{i\ell}A_{i,\ell+1}\cdots A_{ir})x$. The **in parallel do** ... **do** ... construct indicates that all of the **do** code blocks may execute in parallel. The procedure CSB_BLOCKV (pseudocode for which can be found in Figure 6-6) calculates the product of the block and the vector in parallel.

boundaries, just iterate over blocks (in lines 5–10) until enough nonzeros are accrued.

Figure 6-5 gives the parallel algorithm CSB_BLOCKROWV for multiplying a blockrow by a vector, writing the result into the length-β vector y. In lines 24–31, the algorithm recursively divides the blockrow such that each half receives roughly the same number of chunks. We find the appropriate middles of the chunk array R and the input vector x in lines 24 and 25, respectively. We then allocate a length-β temporary vector z (line 26) and perform the recursive multiplications on each subblockrow in parallel (lines 27–29), having one of the recursive multiplications write its output to z. When these recursive multiplications complete, we merge the outputs into the vector y (lines 30–31).

The recursion bottoms out when the blockrow consists of a single chunk (lines 14–23). If this chunk contains many blocks, it is guaranteed to contain at most $\Theta(\beta)$ nonzeros, which is sufficiently sparse to perform the serial multiplication in line 22. If, on the other hand, the chunk is a single block, it may contain as many as $\beta^2 \approx n$ nonzeros. A serial multiplication here, therefore, would be the bottleneck in the algorithm. Instead, we perform the parallel block-vector multiplication CSB_BLOCKV in line 20.

If the blockrow recursion reaches a single block, we perform a parallel multiplication of the block by the vector, given in Figure 6-6. The block-vector multiplication proceeds by recursively

```
CSB_BLOCKV(A, start, end, dim, x, y)
      ▷ A.val[start..end] is a dim × dim matrix M.
32  if end − start ≤ Θ(dim)
33     then ▷ Perform the serial computation y ← y + Mx.
34          for k ← start to end
35              do y[A.row_ind[k]] ← y[A.row_ind[k]]
                                   +A.val[k] · x[A.col_ind[k]]
36          return
37  ▷ Recurse. Find the indices of the quadrants.
38  binary search start, start+1, ..., end for the smallest s2
          such that (A.row_ind[s2] & dim/2) ≠ 0
39  binary search start, start+1, ..., s2 − 1 for the smallest s1
          such that (A.col_ind[s1] & dim/2) ≠ 0
40  binary search s2, s2 + 1, ..., end for the smallest s3
          such that (A.col_ind[s3] & dim/2) ≠ 0
41  in parallel
42      do CSB_BLOCKV(A, start, s1 − 1, dim/2, x, y)        ▷ M00.
43      do CSB_BLOCKV(A, s3, end, dim/2, x, y)              ▷ M11.
44  in parallel
45      do CSB_BLOCKV(A, s1, s2 − 1, dim/2, x, y)           ▷ M01.
46      do CSB_BLOCKV(A, s2, s3 − 1, dim/2, x, y)           ▷ M10.
```

Figure 6-6: Pseudocode for the subblock-vector product $y \leftarrow Mx$, where M is the list of tuples stored in $A.\,val[start\,..\,end]$, $A.\,row_ind[start\,..\,end]$, and $A.\,col_ind[start\,..\,end]$, in recursive Z-Morton order. The & operator is a bitwise AND of the two operands.

dividing the (sub)block M into quadrants M_{00}, M_{01}, M_{10}, and M_{11}, each of which is conveniently stored contiguously in the Z-Morton-ordered *val*, *row_ind*, and *col_ind* arrays between indices *start* and *end*. We perform binary searches to find the appropriate dividing points in the array in lines 38–40.

To understand the pseudocode, consider the search for the dividing point s_2 between $M_{00}M_{01}$ and $M_{10}M_{11}$. For any recursively chosen $dim \times dim$ matrix M, the column indices and row indices of all elements have the same leading $\lg \beta - \lg dim$ bits. Moreover, for those elements in $M_{00}M_{01}$, the next bit in the row index is a 0, whereas for those in elements in $M_{10}M_{11}$, the next bit in the row index is 1. The algorithm does a binary search for the point at which this bit flips. The cases for the dividing point between M_{00} and M_{01} or M_{10} and M_{11} are similar, except that we focus on the column index instead of the row index.

After dividing the matrix into quadrants, we execute the matrix products involving matrices M_{00} and M_{11} in parallel (lines 41–43), as they do not conflict on any outputs. After completing these products, we execute the other two matrix products in parallel (lines 44–46).[4] This procedure resembles a standard parallel divide-and-conquer matrix multiplication, except that our base case of serial multiplication starts at a matrix containing $\Theta(dim)$ nonzeros (lines 33–36). Note that although we pass the full length-β arrays x and y to each recursive call, the effective length of each

[4] The algorithm may instead do M_{00} and M_{10} in parallel followed by M_{01} and M_{11} in parallel without affecting the performance analysis. Presenting the algorithm with two choices may yield better load balance.

array is halved implicitly by partitioning M into quadrants. Passing the full arrays is a technical detail required to properly compute array indices, as the indices $A.row_ind$ and $A.col_ind$ store offsets within the block.

The CSB_SpMV_T algorithm is identical to CSB_SpMV, except that we operate over blockcolumns rather than blockrows.

6.4 Analysis

In this section, we prove that for an $n \times n$ matrix with nnz nonzeros, CSB_SpMV operates with work $\Theta(nnz)$ and span $O(\sqrt{n}\lg n)$ when $\beta = \sqrt{n}$, yielding a parallelism of $\Omega(nnz/\sqrt{n}\lg n)$. We also provide bounds in terms of β and analyze the space usage.

We begin by analyzing block-vector multiplication.

Lemma 6.4 *On a $\beta \times \beta$ block containing r nonzeros,* CSB_BlockV *runs with work $\Theta(r)$ and span $O(\beta)$.*

Proof. The span for multiplying a $dim \times dim$ matrix can be described by the recurrence $S(dim) = 2S(dim/2) + O(\lg dim) = O(dim)$. The $\lg dim$ term represents a loose upper bound on the cost of the binary searches. In particular, the binary-search cost is $O(\lg z)$ for a submatrix containing z nonzeros, and we have $z \le dim^2$, and hence $O(\lg z) = O(\lg dim)$, for a $dim \times dim$ matrix.

To calculate the work, consider the degree-4 tree of recursive procedure calls, and associate with each node the work done by that procedure call. We say that a node in the tree has height h if it corresponds to a $2^h \times 2^h$ subblock, i.e., if $dim = 2^h$ is the parameter passed into the corresponding CSB_BlockV call. Node heights are integers ranging from 0 to $\lg\beta$. Observe that each height-h node corresponds to a distinct $2^h \times 2^h$ subblock (although subblocks may overlap for nodes having different heights). A height-h leaf node (serial base case) corresponds to a subblock containing at most $z = O(2^h)$ nonzeros and has work linear in this number z of nonzeros. Summing across all leaves, therefore, gives $\Theta(r)$ work. A height-h internal node, on the other hand, corresponds to a subblock containing *at least* $z' = \Omega(2^h)$ nonzeros (or else it would not recurse further and be a leaf) and has work $O(\lg 2^h) = O(h)$ arising from the binary searches. There can thus be at most $O(r/2^h)$ height-h internal nodes having total work $O((r/2^h)h)$. Summing across all heights gives total work of $\sum_{h=0}^{\lg\beta} O((r/2^h)h) = r\sum_{h=0}^{\lg\beta} O(h/2^h) = O(r)$ for internal nodes. Combining the work at internal nodes and leaf nodes gives total work $\Theta(r)$. □

The next lemma analyzes blockrow-vector multiplication.

Lemma 6.5 *On a blockrow containing n/β blocks and r nonzeros,* CSB_BlockRowV *runs with work $\Theta(r)$ and span $O(\beta\lg(n/\beta))$.*

Proof. Consider a call to CSB_BlockRowV on a row that is partitioned into C chunks, and let $W(C)$ denote the work. The work per recursive call on a multichunk subblockrow is dominated by the $\Theta(\beta)$ work of initializing a temporary vector z and adding the vector z into the output vector y. The work for a CSB_BlockRowV on a single-chunk subblockrow is linear in the number of nonzeros in the chunk. (We perform linear work either in line 22 or in line 20 — see Lemma 6.4 for the work of line 20.) We can thus describe the work by the recurrence $W(C) \le 2W(\lceil C/2\rceil) + \Theta(\beta)$ with a base case of work linear in the nonzeros, which solves to $W(C) = \Theta(C\beta + r)$ for $C > 1$. When $C = 1$, we have $W(C) = \Theta(r)$, as we do not operate on the temporary vector z.

To bound work, it remains to bound the maximum number of chunks in a row. Notice that any two consecutive chunks contain at least $\Omega(\beta)$ nonzeros. This fact follows from the way chunks are chosen in lines 2–11: a chunk is terminated only if adding the next block to the chunk would

increase the number of nonzeros to more than $\Theta(\beta)$. Thus, a blockrow consists of a single chunk whenever $r = O(\beta)$ and at most $O(r/\beta)$ chunks whenever $r = \Omega(\beta)$. Hence, the total work is $\Theta(r)$.

We can describe the span of CSB_BLOCKROWV by the recurrence $S(C) = S(\lceil C/2 \rceil) + O(\beta) = O(\beta \lg C) + S(1)$. The base case involves either serially multiplying a single chunk containing at most $O(\beta)$ nonzeros in line 22, which has span $O(\beta)$, or multiplying a single block in parallel in line 20, which also has span $O(\beta)$ from Lemma 6.4. We have, therefore, a span of $O(\beta \lg C) = O(\beta \lg(n/\beta))$, since $C \leq n/\beta$. □

We are now ready to analyze matrix-vector multiplication itself.

Theorem 6.6 *On an $n \times n$ matrix containing nnz nonzeros,* CSB_SPMV *runs with work $\Theta(n^2/\beta^2 + nnz)$ and span $O(\beta \lg(n/\beta) + n/\beta)$.*

Proof. For each blockrow, we add $\Theta(n/\beta)$ work and span for computing the chunks, which arise from a serial scan of the n/β blocks in the blockrow. Thus, the total work is $O(n^2/\beta^2)$ in addition to the work for multiplying the blockrows, which is linear in the number of nonzeros from Lemma 6.5.

The total span is $O(\lg(n/\beta))$ to parallelize all the rows, plus $O(n/\beta)$ per row to partition the row into chunks, plus the $O(\beta \lg(n/\beta))$ span per blockrow from Lemma 6.5. □

The following corollary gives the work and span bounds when we choose β to yield the same space for the CSB storage format as for the CSR or CSC formats.

Corollary 6.7 *On an $n \times n$ matrix containing $nnz \geq n$ nonzeros, when choosing $\beta = \Theta(\sqrt{n})$,* CSB_SPMV *runs with work $\Theta(nnz)$ and span $O(\sqrt{n} \lg n)$.* CSB_SPMV *thus achieves a parallelism of $\Omega(nnz / \sqrt{n} \lg n)$.* □

Since CSB_SPMV_T is isomorphic to CSB_SPMV, we obtain the following corollary.

Corollary 6.8 *On an $n \times n$ matrix containing $nnz \geq n$ nonzeros, when choosing $\beta = \Theta(\sqrt{n})$,* CSB_SPMV_T *runs with work $\Theta(nnz)$ and span $O(\sqrt{n} \lg n)$.* CSB_SPMV_T *thus achieves a parallelism of $\Omega(nnz / \sqrt{n} \lg n)$.* □

The work of our algorithm is dominated by the space of the temporary vectors z, and thus the space usage on an infinite number of processors matches the work bound. When run on fewer processors however, the space usage reduces drastically. We can analyze the space in terms of the ***serialization*** of the program, which corresponds to the program obtained by removing all **parallel** keywords.

Lemma 6.9 *On an $n \times n$ matrix, by choosing $\beta = \Theta(\sqrt{n})$, the serialization of* CSB_SPMV *requires $O(\sqrt{n} \lg n)$ space (not counting the storage for the matrix itself).*

Proof. The serialization executes one blockrow multiplication at a time. There are two space overheads. First, we use $O(n/\beta) = O(\sqrt{n})$ space for the chunk array. Second, we use β space to store the temporary vector z for each outstanding recursive call to CSB_BLOCKROWV. Since the recursion depth is $O(\lg n)$, the total space becomes $O(\beta \lg n) = O(\sqrt{n} \lg n)$. □

A typical work-stealing scheduler executes the program in a depth-first (serial) manner on each processor. When a processor completes all its work, it "steals" work from a different processor, beginning a depth-first execution from some unexecuted parallel branch. Although not all work-stealing schedulers are space efficient, those maintaining the ***busy-leaves property*** [55] (e.g., as used in the Cilk work-stealing scheduler [54]) are space efficient. The "busy-leaves" property roughly says that if a procedure has begun (but not completed) executing, then there exists a processor currently working on that procedure or one of its descendants procedures.

Corollary 6.10 *Suppose that a work-stealing scheduler with the busy-leaves property schedules an execution of* CSB_SPMV *on an* $n \times n$ *matrix with the choice* $\beta = \sqrt{n}$. *Then, the execution requires* $O(P\sqrt{n}\lg n)$ *space.*

Proof. Combine Lemma 6.9 and Theorem 1 from [54]. □

The work overhead of our algorithm may be reduced by increasing the constants in the $\Theta(\beta)$ threshold in line 7. Specifically, increasing this threshold by a constant factor reduces the number of reads and writes to temporaries by the same constant factor. As these temporaries constitute the majority of the work overhead of the algorithm, doubling the threshold nearly halves the overhead. Increasing the threshold, however, also increases the span by a constant factor, and so there is a trade-off.

6.5 Experimental design

This section describes our implementation of the CSB_SPMV and CSB_SPMV_T algorithms, the benchmark matrices we used to test the algorithms, the machines on which we ran our tests, and the other codes with which we compared our algorithms.

Implementation

We parallelized our code using Cilk++ [69], which is a faithful extension of C++ for multicore and shared-memory parallel programming. Cilk++ is based on the earlier MIT Cilk system [99], and it employs dynamic load balancing and provably optimal task scheduling. The CSB code used for the experiments is freely available for academic use at `http://gauss.cs.ucsb.edu/~aydin/software.html`.

The *row_ind* and *col_ind* arrays of CSB, which store the row and column indices of each nonzero within a block (i.e., the lower-order bits of the row and column indices within the matrix A), are implemented as a single ***index array*** by concatenating the two values together. The higher-order bits of *row_ind* and *col_ind* are stored only implicitly, and are retrieved by referencing the *blk_ptr* array.

The CSB blocks themselves are stored in row-major order, while the nonzeros within blocks are in Z-Morton order. The row-major ordering among blocks may seem to break the overall symmetry of CSB, but in practice it yields efficient handling of block indices for look-up in A. *blk_ptr* by permitting an easily computed look-up function $f(i, j)$. The row-major ordering also allowed us to count the nonzeros in a subblockrow more easily when computing $y \leftarrow Ax$. This optimization is not symmetric, but interestingly, we achieved similar performance when computing $y \leftarrow A^{\mathsf{T}}x$, where we must still aggregate the nonzeros in each block. In fact, in almost half the cases, computing $A^{\mathsf{T}}x$ was faster than Ax, depending on the matrix structure.

The Z-Morton ordering on nonzeros in each block is equivalent to first interleaving the bits of *row_ind* and *col_ind*, and then sorting the nonzeros using these bit-interleaved values as the keys. Thus, it is tempting to store the index array in a bit-interleaved fashion, thereby simplifying the binary searches in lines 38–40. Converting to and from bit-interleaved integers, however, is expensive with current hardware support,[5] which would be necessary for the serial base case in lines 33–36. Instead, the kth element of the index array is the concatenation of *row_ind*$[k]$ and *col_ind*$[k]$, as indicated earlier. This design choice of storing concatenated, instead of bit-interleaved, indices requires either some care when performing the binary search (as presented in Figure 6-6) or implicitly

[5]Recent research [173] addresses these conversions.

converting from the concatenated to interleaved format when making a binary-search comparison. Our preliminary implementation does the latter, using a C++ function object for comparisons [183]. In practice, the overhead of performing these conversions is small, since the number of binary-search steps is small.

Performing the actual address calculation and determining the pointers to x and y vectors are done by masking and bit-shifting. The bitmasks are determined dynamically by the CSB constructor depending on the input matrix and the data type used for storing matrix indices. Our library allows any data type to be used for matrix indices and handles any type of matrix dynamically. For the results presented in Section 6.6, nonzero values are represented as double-precision floating-point numbers, and indices are represented as 32-bit unsigned integers. Finally, as our library aims to be general instead of matrix specific, we did not employ speculative low-level optimizations such as software prefetching, pipelining, or matrix-specific optimizations such as index and/or value compression [134, 201], but we believe that CSB and our algorithms should not adversely affect incorporation of these approaches.

Choosing the block size β

We investigated different strategies to choose the block size that achieves the best performance. For the types of loads we ran, we found that a block size slightly larger than $\sqrt{n}$ delivers reasonable performance. Figure 6-7 shows the effect of different blocksizes on the performance of the $y \leftarrow Ax$ operation with the representative matrix Kkt_power. The closest exact power of 2 to $\sqrt{n}$ is 1024, which turns out to be slightly suboptimal. In our experiments, the overall best performance was achieved when β satisfies the equation $\lceil \lg \sqrt{n} \rceil \leq \lg \beta \leq 3 + \lceil \lg \sqrt{n} \rceil$.

Merely setting β to a hard-coded value, however, is not robust for various reasons. First, the elements stored in the index array should use the same data type as that used for matrix indices. Specifically, the integer $\beta - 1$ should fit in 2 bytes so that a concatenated *row_ind* and *col_ind* fit into 4 bytes. Second, the length-β regions of the input vector x and output vector y (which are accessed when multiplying a single block) should comfortably fit into L2 cache. Finally, to ensure speedup on matrices with evenly distributed nonzeros, there should be enough parallel slackness for the parallelization across blockrows (i.e., the highest level parallelism). Specifically, when β grows large, the parallelism is roughly bounded by $O(nnz/(\beta \lg(n/\beta)))$ (by dividing the work and span from Theorem 6.6). Thus, we want $nnz/(\beta \lg(n/\beta))$ to be "large enough," which means limiting the maximum magnitude of β.

We adjusted our CSB constructor, therefore, to automatically select a reasonable block-size parameter β. It starts with $\beta = 3 + \lceil \lg \sqrt{n} \rceil$ and keeps decreasing it until the aforementioned constraints are satisfied. Although a research opportunity may exist to autotune the optimal block size with respect to a specific matrix and architecture, in most test matrices, choosing $\beta = \sqrt{n}$ degraded performance by at most 10%–15%. The optimal β value barely shifts along the x-axis when running on different numbers of processors and is quite stable overall.

An optimization heuristic for structured matrices

Even though CSB_SpMV and CSB_SpMV_T are robust and exhibit plenty of parallelism on most matrices, their practical performance can be improved on some sparse matrices having regular structure. In particular, a block diagonal matrix with equally sized blocks has nonzeros that are evenly distributed across blockrows. In this case, a simple algorithm based on blockrow parallelism would suffice in place of the more complicated recursive method from CSB_BlockV. This divide-and-conquer within blockrows incurs overhead that might unnecessarily degrade per-

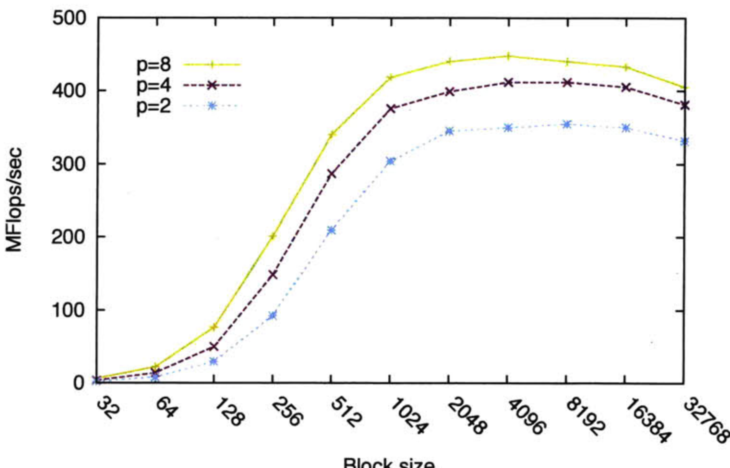


Figure 6-7: The effect of block size parameter β on SpMV performance using the Kkt_power matrix. For values $\beta > 32768$ and $\beta < 32$, the experiment failed to finish due to memory limitations. The experiment was conducted on the AMD Opteron.

formance. Thus, when the nonzeros are evenly distributed across the blockrows, our implementation of the top-level algorithm (given in Figure 6-4) calls the serial multiplication in line 12 instead of the CSB_BLOCKROWV procedure.

To see whether a given matrix is amenable to the optimization, we apply the following "balance" heuristic. We calculate the imbalance among blockrows (or blockcolumns in the case of $y \leftarrow A^{\mathsf{T}}x$) and apply the optimization only when no blocks have more than twice the average number of nonzeros per blockrow. In other words, if $\max(nnz(A_i)) < 2 \cdot \mathrm{mean}(nnz(A_i))$, then the matrix is considered to have balanced blockrows and the optimization is applied. Of course, this optimization is not the only way to achieve a performance boost on structured matrices.

Optimization of temporary vectors

One of the most significant overheads of our algorithm is the use of temporary vectors to store intermediate results when parallelizing a blockrow multiplication in CSB_BLOCKROWV. The "balance" heuristic above is one way of reducing this overhead when the nonzeros in the matrix are evenly distributed. For arbitrary matrices, however, we can still reduce the overhead in practice. In particular, we only need to allocate the temporary vector z (in line 26) if both of the subsequent multiplications (lines 27–29) are scheduled in parallel. If the first recursive call completes before the second recursive call begins, then we instead write directly into the output vector for both recursive calls. In other words, when a blockrow multiplication is scheduled serially, the multiplication procedure detects this fact and mimics a normal serial execution, without the use of temporary vectors. Our implementation exploits an undocumented feature of Cilk++ to test whether the first call has completed before making the second recursive call, and we allocate the temporary as appropriate. This test may also be implemented using Cilk++ reducers [97].

Sparse-matrix test suite

We conducted experiments on a diverse set of sparse matrices from real applications including circuit simulation, finite-element computations, linear programming, and web-connectivity analysis.

Name Description	Spy Plot	Dimensions Nonzeros	CSC (mean/max) CSB (mean/max)	Name Description	Spy Plot	Dimensions Nonzeros	CSC (mean/max) CSB (mean/max)
Asic_320k circuit simulation		321K × 321K 1,931K	6.0 / 157K 4.9 / 2.3K	**Rajat31** circuit simulation		4.69M × 4.69M 20.31M	4.3 / 1.2K 3.9 / 8.7K
Sme3Dc 3D structural mechanics		42K × 42K 3,148K	73.3 / 405 111.6 / 1368	**Ldoor** structural prob.		952K × 952K 42.49M	44.6 / 77 49.1 / 43,872
Parabolic_fem diff-convection reaction		525K × 525K 3,674K	7.0 / 7 3.5 / 1,534	**Bone010** 3D trabecular bone		986K × 986K 47.85M	48.5 / 63 51.5 / 18,670
Mittelmann LP problem		1,468K × 1,961K 5,382K	2.7 / 7 2.0 / 3,713	**Grid3D200** 3D 7-point finite-diff mesh		8M × 8M 55.7M	6.97 / 7 3.7 / 9,818
Rucci Ill-conditioned least-squares		1,977K × 109K 7,791K	70.9 / 108 9.4 / 36	**RMat23** Real-world graph model		8.4M × 8.4M 78.7M	9.4 / 70.3K 4.7 / 222.1K
Torso Finite diff, 2D model of torso		116K × 116K 8,516K	73.3 / 1.2K 41.3 / 36.6K	**Cage15** DNA electrophoresis		5.15M × 5.15M 99.2M	19.2 / 47 15.6 / 39,712
Kkt_power optimal power flow, nonlinear opt.		2.06M × 2.06M 12.77M	6.2 / 90 3.1 / 1,840	**Webbase2001** Web connectivity		118M × 118M 1,019M	8.6 / 816K 4.9 / 2,375K

Figure 6-8: Structural information on the sparse matrices used in our experiments, ordered by increasing number of nonzeros. The first ten matrices and Cage15 are from the University of Florida sparse matrix collection [76]. Grid3D200 is a 7-point finite difference mesh generated using the Matlab Mesh Partitioning and Graph Separator Toolbox [105]. The RMat23 matrix [140], which models scale-free graphs, is generated by using repeated Kronecker products [29]. We chose parameters $A = 0.7, B = C = D = 0.1$ for RMat23 in order to generate skewed matrices. Webbase2001 is a crawl of the World Wide Web from the year 2001 [66].

These matrices not only cover a wide range of applications, but they also greatly vary in size, density, and structure. The test suite contains both rectangular and square matrices. Almost half of the square matrices are asymmetric. Figure 6-8 summarizes the 14 test matrices.

Included in Figure 6-8 is the load imbalance that is likely to occur for an SpMV algorithm parallelized with respect to columns (CSC) and blocks (CSB). In the last column, the average (mean) and the maximum number of nonzeros among columns (first line) and blocks (second line) are shown for each matrix. The sparsity of matrices can be quantified by the average number of nonzeros per column, which is equivalent to the mean of CSC. The sparsest matrix (Rajat31) has 4.3 nonzeros per column on the average while the densest matrix has about 73 nonzeros per column (Sme3Dc and Torso). For CSB, the reported mean/max values are obtained by setting the block dimension β to be approximately $\sqrt{n}$, so that they are comparable with statistics from CSC.

Architectures and comparisons

We ran our experiments on three multicore superscalar architectures. Opteron is a ccNUMA architecture powered by AMD Opteron 8214 (Santa Rosa) processors clocked at 2.2 GHz. Each core of Opteron has a private 1 MB L2 cache, and each socket has its own integrated memory controller. Although it is an 8-socket dual-core system, we only experimented with up to 8 processors. Harpertown is a dual-socket quad-core system running two Intel Xeon X5460's, each clocked at 3.16 GHz. Each socket has 12 MB of L2 cache, shared among four cores, and a front-side bus (FSB) running at 1333 MHz. Nehalem is a single-socket quad-core Intel Core i7 920 processor. Like Opteron, Nehalem has an integrated memory controller. Each core is clocked at 2.66 GHz and has a private 256 KB L2 cache. The four cores share an 8 MB L3 cache.

While Opteron has 64 GB of RAM, Harpertown and Nehalem have only 8 GB and 6 GB, respectively, which forced us to exclude our biggest test matrix (Webbase2001) from our runs on Intel architectures. We compiled our code using `gcc 4.1` on Opteron and Harpertown and with `gcc 4.3` on Nehalem, all with optimization flags `-O2 -fno-rtti -fno-exceptions`.

To evaluate our code on a single core, we compared its performance with "pure" OSKI matrix-vector multiplication [195] running on one processor of Opteron. We did not enable OSKI's preprocessing step, which chooses blockings for cache and register usage that are tuned to a specific matrix. We conjecture that such matrix-specific tuning techniques can be combined advantageously with our CSB data structure and parallel algorithms.

To compare with a parallel code, we used the matrix-vector multiplication of Star-P [179] running on Opteron. Star-P is a distributed-memory code that uses CSR to represent sparse matrices and distributes matrices to processor memories by equal-sized blocks of rows.

6.6 Experimental results

Figures 6-9 and 6-10 show how CSB_SPMV and CSB_SPMV_T, respectively, scale for the seven smaller matrices on Opteron, and Figures 6-11 and 6-12 show similar results for the seven larger matrices. In most cases, the two codes show virtually identical performance, confirming that the CSB data structure and algorithms are equally suitable for both operations. In all the parallel scaling graphs, only the values $p = 1, 2, 4, 8$ are reported. They should be interpreted as performance achievable by doubling the number of cores instead of as the exact performance on p threads (e.g., $p = 8$ is the best performance achieved for $5 \leq p \leq 8$).

In general, we observed better speedups for larger problems. For example, the average speedup of CSB_SPMV for the first seven matrices was 2.75 on 8 processors, whereas it was 3.03 for the sec-

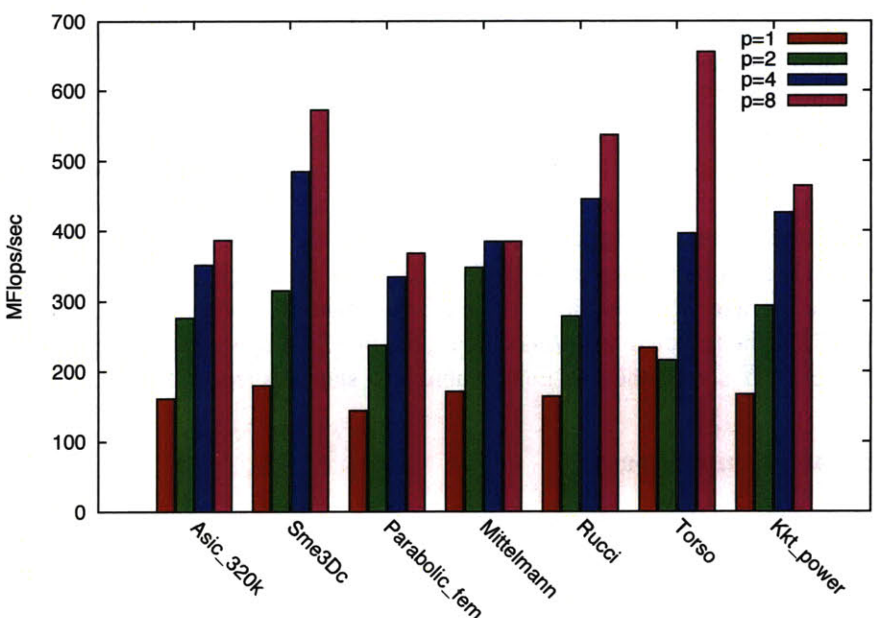


Figure 6-9: CSB_SPMV performance on Opteron (smaller matrices).

ond set of seven matrices with more nonzeros. Figure 6-13 summarizes these results. The speedups are relative to the CSB code running on a single processor, which Figure 6-1 shows is competitive with serial CSR codes. In another study [202] on the same Opteron architecture, multicore-specific parallelization of the CSR code for 4 cores achieved comparable speedup to what we report here, albeit on a slightly different sparse-matrix test suite. That study does not consider the $y \leftarrow A^{\mathsf{T}}x$ operation, however, which is difficult to parallelize with CSR but which achieves the same performance as $y \leftarrow Ax$ when using CSB.

For CSB_SPMV on 4 processors, CSB reached its highest speedup of 2.80 on the RMat23 matrix, showing that this algorithm is robust even on a matrix with highly irregular nonzero structure. On 8 processors, CSB_SPMV reached its maximum speedup of 3.93 on the Webbase2001 matrix, indicating the code's ability to handle very large matrices without sacrificing parallel scalability.

Sublinear speedup occurs only after the memory-system bandwidth becomes the bottleneck. This bottleneck occurs at different numbers of cores for different matrices. In most cases, we observed nearly linear speedup up to 4 cores. Although the speedup is sublinear beyond 4 cores, in every case (except CSB_SPMV on Mittelmann), we see some performance improvement going from 4 to 8 cores on Opteron. Sublinear speedup of SpMV on superscalar multicore architectures has been noted by others as well [202].

We conducted an additional experiment to verify that performance was limited by the memory-system bandwidth, not by lack of parallelism. We repeated each scalar multiply-add operation of the form $y_i \leftarrow y_i + A_{ij}x_j$ a fixed number t of times. Although the resulting code computes $y \leftarrow tAx$, we ensured that the compiler did not optimize away any multiply-add operations. Setting $t = 10$ did not affect the timings significantly—flops are indeed essentially free—but, for $t = 100$, we saw almost perfect linear speedup up to 16 cores, as shown in Figure 6-14. We performed this experiment with Asic_320k, the smallest matrix in the test suite, which should exhibit the least parallelism. Asic_320k is also irregular in structure, which means that our balance heuristic does not apply. Nevertheless, CSB_SPMV scaled almost perfectly given enough flops per byte.

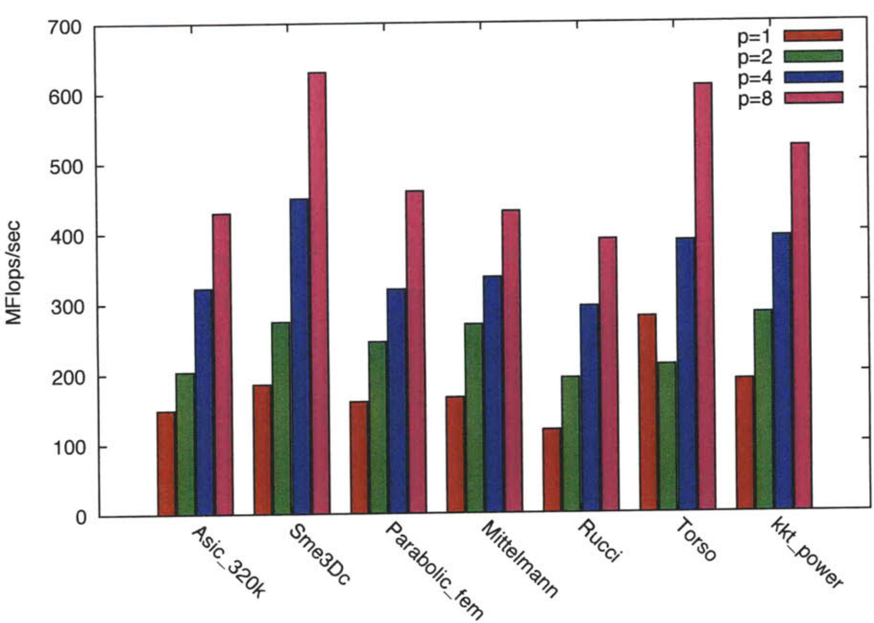


Figure 6-10: CSB_SPMV_T performance on Opteron (smaller matrices).

The parallel performance of CSB_SPMV and CSB_SPMV_T is generally not affected by highly uneven row and column nonzero counts. The highly skewed matrices RMat23 and Webbase2001 achieved speedups as good as for matrices with flat row and column counts. An unusual case is the Torso matrix, where both CSB_SPMV and CSB_SPMV_T were actually slower on 2 processors than serially. This slowdown does not, however, mark a plateau in performance, since Torso speeds up as we add more than 2 processors. We believe this behavior occurs because the overhead of intrablock parallelization is not amortized for 2 processors. Torso requires a large number of intrablock parallelization calls, because it is unusually irregular and dense.

Figure 6-15 shows the performance of CSB_SPMV on Harpertown for a subset of test matrices. We do not report performance for CSB_SPMV_T, as it was consistently close to that of CSB_SPMV. The performance on this platform levels off beyond 4 processors for most matrices. Indeed, the average Mflops/sec on 8 processors is only 3.5% higher than on 4 processors. We believe this plateau results from insufficient memory bandwidth. The continued speedup on Opteron is due to its higher ratio of memory bandwidth (bytes) to peak performance (flops) per second.

Figure 6-16 summarizes the performance results of CSB_SPMV for the same subset of test matrices on Nehalem. Despite having only 4 physical cores, for most matrices, Nehalem achieved scaling up to 8 threads thanks to hyperthreading. Running 8 threads was necessary to utilize the processor fully, because hyperthreading fills the pipeline more effectively. We observed that the improvement from oversubscribing is not monotonic, however, because running more threads reduces the effective cache size available to each thread. Nehalem's point-to-point interconnect is faster than Opteron's (a generation old Hypertransport 1.0), which explains its better speedup values when comparing the 4-core performance of both architectures. Its raw performance is also impressive, beating both Opteron and Harpertown by large margins.

To determine CSB's competitiveness with a conventional CSR code, we compared the performance of the CSB serial code with plain OSKI using no matrix-specific optimizations such as register or cache blocking. Figures 6-17 and 6-18 present the results of the comparison. As can be

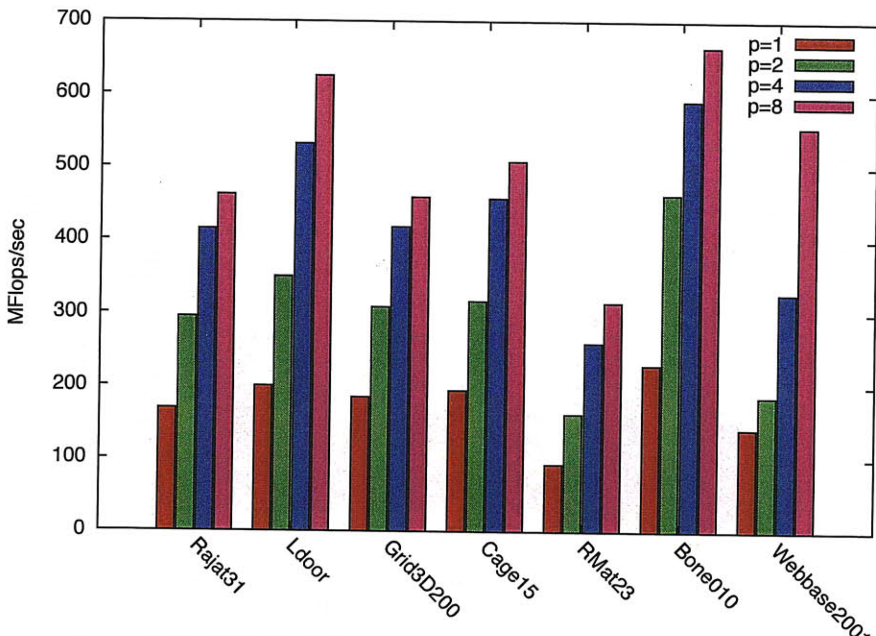


Figure 6-11: CSB_SPMV performance on Opteron (larger matrices).

seen from the figures, CSB achieves similar serial performance to CSR.

In general, CSR seems to perform best on ***banded matrices***, all of whose nonzeros are located near the main diagonal. (The maximum distance of any nonzero from the diagonal is called the matrix's ***bandwidth***, not to be confused with memory bandwidth.) If the matrix is banded, memory accesses to the input vector x tend to be regular and thus favorable to cacheline reuse and automatic prefetching. Strategies for reducing the bandwidth of a sparse matrix by permuting its rows and columns have been studied extensively (see [74,191], for example). Many matrices, however, cannot be permuted to have low bandwidth. For matrices with scattered nonzeros, CSB outperforms CSR, because CSR incurs many cache misses when accessing the x vector. An example of this effect occurs for the RMat23 matrix, where the CSB implementation is almost twice as fast as CSR.

Figure 6-19 compares the parallel performance of the CSB algorithms with Star-P. Star-P's blockrow data distribution does not afford any flexibility for load-balancing across processors. Load balance is not an issue for matrices with nearly flat row counts, including finite-element and finite-difference matrices, such as Grid3D200. Load balance does become an issue for skewed matrices such as RMat23, however. Our performance results confirm this effect. CSB_SPMV is about 500% faster than Star-P's SpMV routine for RMat23 on 8 cores. Moreover, for any number of processors, CSB_SPMV runs faster for all the matrices we tested, including the structured ones.

6.7 CSB Discussion

Compressed sparse blocks allow parallel operations on sparse matrices to proceed either row-wise or column-wise with equal facility. We have demonstrated the efficacy of the CSB storage format for SpMV calculations on a sparse matrix or its transpose. It remains to be seen, however, whether the CSB format is limited to SpMV calculations or if it can also be effective in enabling parallel algorithms for multiplying two sparse matrices, performing LU-, LUP-, and related decompositions, linear programming, and a host of other problems for which serial sparse-matrix algorithms

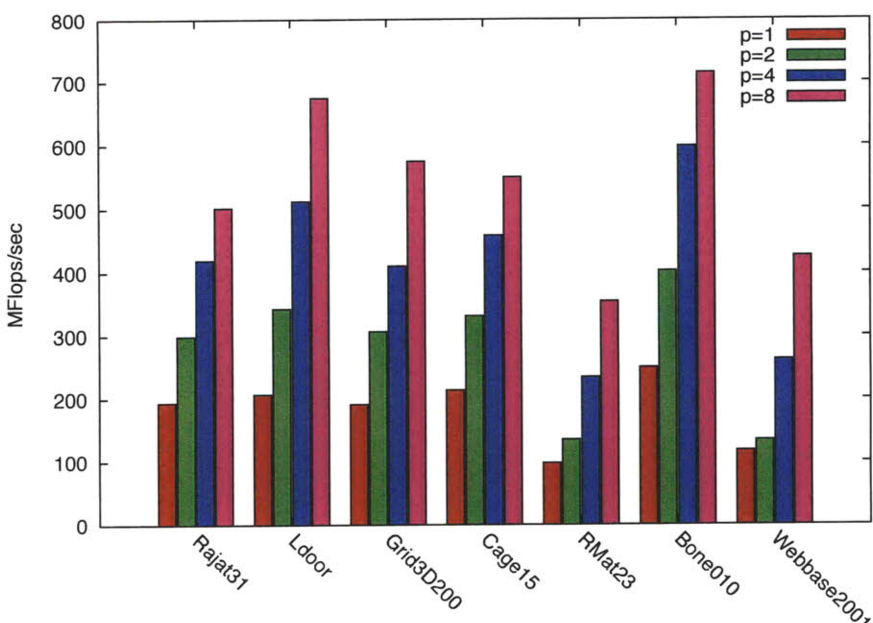


Figure 6-12: CSB_SPMV_T performance on Opteron (larger matrices).

Processors	CSB_SPMV		CSB_SPMV_T	
	1–7	8–14	1–7	8–14
$P = 2$	1.65	1.70	1.44	1.49
$P = 4$	2.34	2.49	2.07	2.30
$P = 8$	2.75	3.03	2.81	3.16

Figure 6-13: Average speedup results for relatively smaller (1–7) and larger (8–14) matrices. These experiments were conducted on Opteron.

currently use the CSC and CSR storage formats.

The CSB format readily enables parallel SpMV calculations on a symmetric matrix where only half the matrix is stored, but we were unable to attain one optimization that serial codes exploit in this situation. In a typical serial code that computes $y \leftarrow Ax$, where $A = (a_{ij})$ is a symmetric matrix, when a processor fetches $a_{ij} = a_{ji}$ out of memory to perform the update $y_i \leftarrow y_i + a_{ij}x_j$, it can also perform the update $y_j \leftarrow y_j + a_{ij}x_i$ at the same time. This strategy halves the memory bandwidth compared to executing CSB_SPMV on the matrix, where $a_{ij} = a_{ji}$ is fetched twice. It remains an open problem whether the 50% savings in storage for sparse matrices can be coupled with a 50% savings in memory bandwidth, which is an important factor of 2, since it appears that the bandwidth between multicore chips and DRAM will scale more slowly than core count.

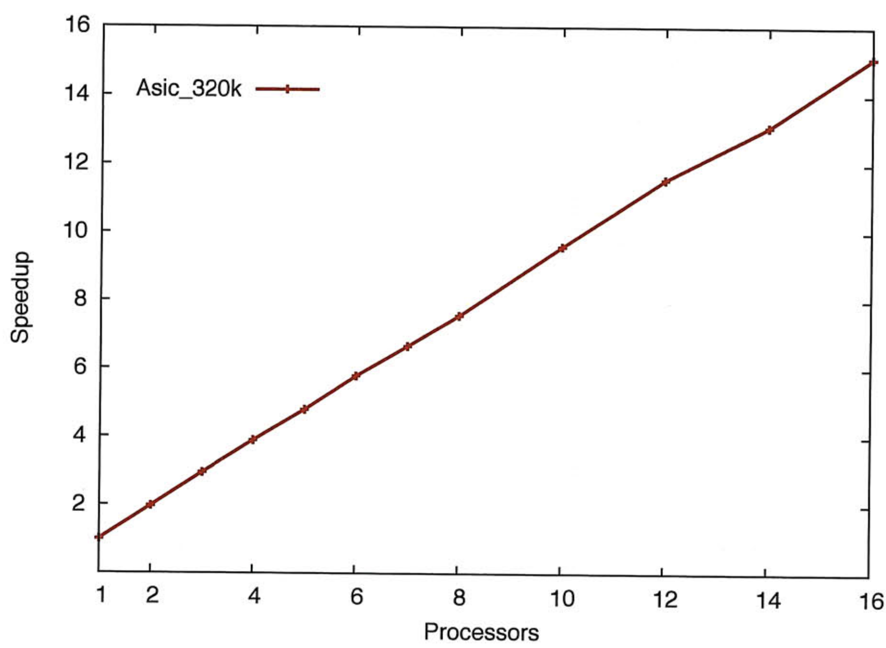


Figure 6-14: Parallelism test for CSB_SPMV on Asic_320k obtained by artificially increasing the flops per byte. The test shows that the algorithm exhibits substantial parallelism and scales almost perfectly given sufficient memory bandwidth.

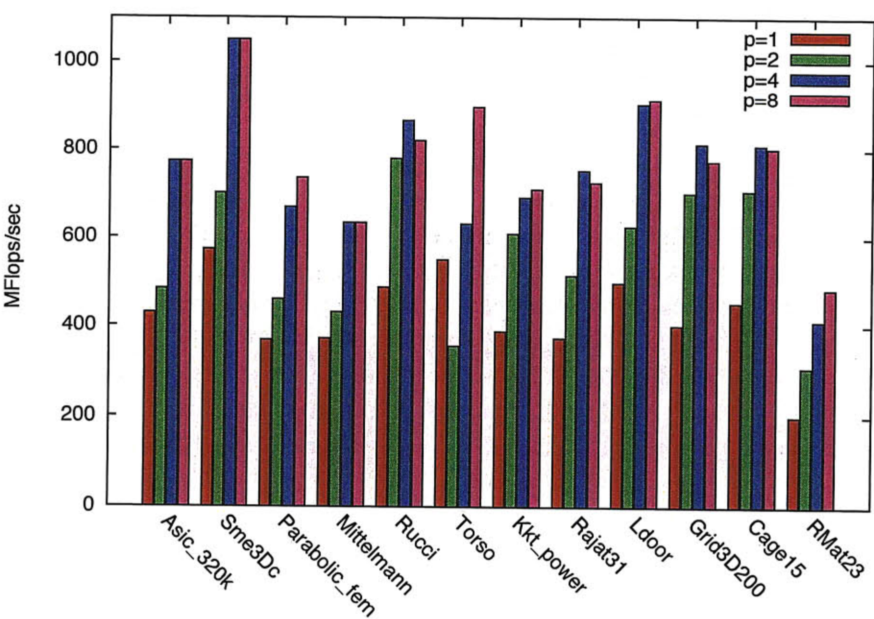


Figure 6-15: CSB_SPMV performance on Harpertown.

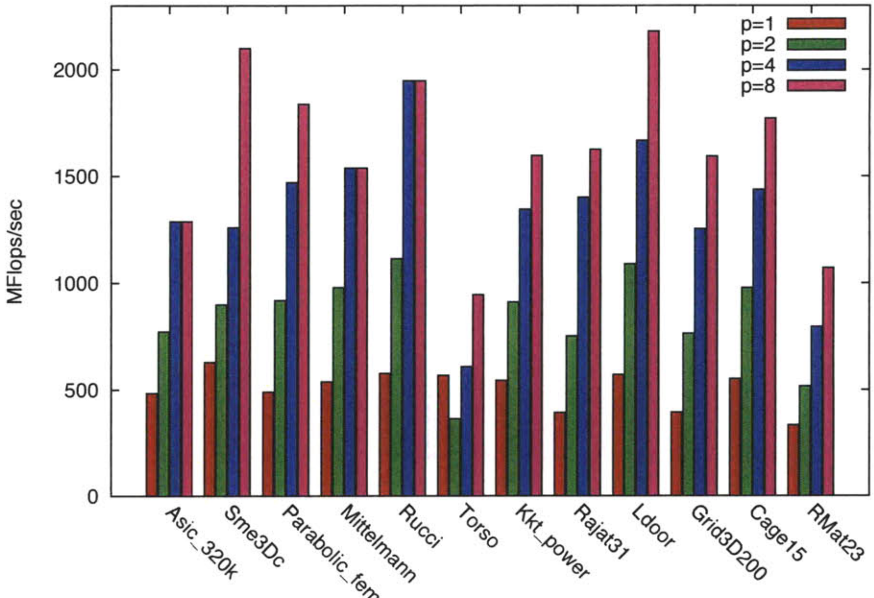


Figure 6-16: CSB_SPMV performance on Nehalem.

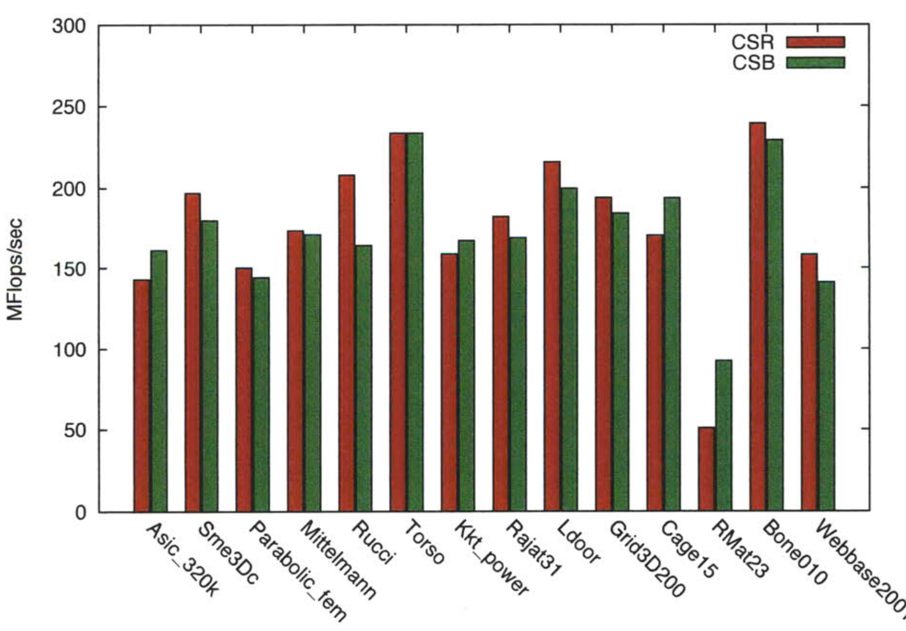


Figure 6-17: Serial performance comparison of SpMV for CSB and CSR.

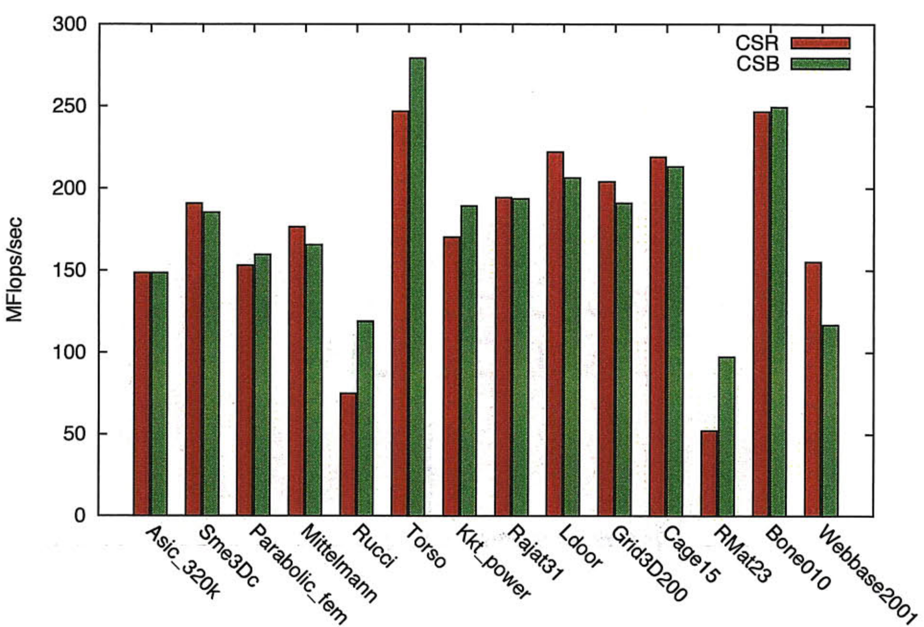


Figure 6-18: Serial performance comparison of SpMV_T for CSB and CSR.

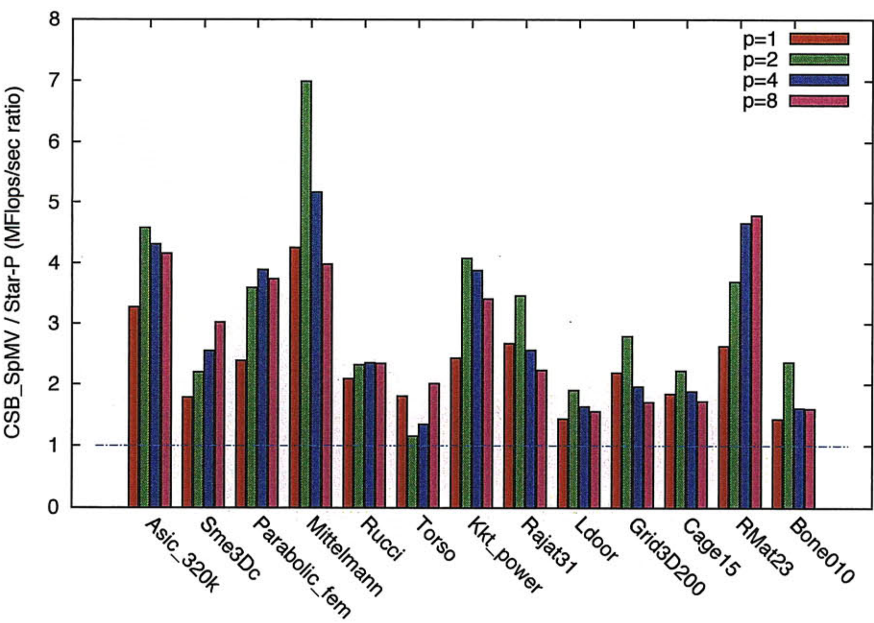


Figure 6-19: Performance comparison of parallel CSB_SPMV with Star-P, which is a parallel-dialect of Matlab. The vertical axis shows the performance ratio of CSB_SPMV to Star-P. A direct comparison of CSB_SPMV_T with Star-P was not possible, because Star-P does not natively support multiplying the transpose of a sparse matrix by a vector.

Chapter 7

Introduction to Cache-Oblivious Dictionaries

For over three decades, the B-tree [30, 70] has been the data structure of choice for maintaining searchable, ordered data on disk. Traditional B-trees are effective in a large part because they minimize the number of blocks accessed during a search. Specifically, for block size B, an N-node B-tree supports searches, insertions, and deletions in $O(\log_B N)$ block transfers and scans of L consecutive elements in $O(1 + L/B)$ transfers. B-trees are designed to achieve good data locality at only one level of the memory hierarchy and for one fixed block size.

In contrast, cache-oblivious (CO) B-trees attain near-optimal memory performance at all levels of the memory hierarchy and for all block sizes (e.g., [34, 35, 37, 41, 60]). A CO B-tree performs a search operation with $O(\log_B N)$ cache misses for all possible block sizes simultaneously, even when the block size is unknown. In a complex memory hierarchy consisting of many levels of cache, the CO B-tree minimizes the number of memory transfers between each adjacent cache level. Thus CO B-trees perform near optimally in theory, and experiments have shown promise of outperforming traditional B-trees [38, 128].

Chapters 8–10 describe various cache-oblivious dynamic dictionaries. A ***dynamic dictionary*** is a data structure that supports insertions, deletions, and predecessor queries. When the cache-oblivious dictionary achieves bounds resembling those of the B-tree, we call it a ***CO B-tree***. Chapter 8 investigates adding concurrency to cache-oblivious B-trees and Chapters 9 and 10 introduce cache-oblivious dictionaries that have better insertion/deletion cost than CO B-trees.

Model

Most modern computers have a multilevel ***memory hierarchy***, consisting of many levels, each acting as a cache for the next. For example, a typical architecture may have registers, level 1 cache, level 2 cache, memory, and disk. Each of these levels has an access cost that may be one or more orders of magnitude less than that of the next. When algorithms operate on large data sets, the cost of memory accesses may dominate the cost incurred by processor operations, and hence a standard RAM model [71, Section 2.2] may not accurately model the performance of the algorithm.

To model the cost of memory access more accurately, Aggarwal and Vitter [6] proposed the I/O model, which assumes a two-level memory hierarchy consisting of a fast memory of size M, a slow memory of infinite size, and transfers from slow to fast memory in chunks or ***blocks*** of size B. A memory access has cost 0 if the location currently resides in the fast memory. If the location being accessed is not currently stored in fast memory, the relevant block must be brought into fast memory

(called a ***block transfer*** or I/O), incurring a cost of 1, and another block may need to be evicted from fast memory to make space. In this model, the algorithm designer has full control over which blocks are evicted from the fast memory when a new block is brought in.

Frigo et al. [98] introduced the notion ***cache-oblivious*** algorithms. These algorithms, analyzed in the ***ideal-cache model***, are oblivious to the block size B and memory size M. The ideal-cache model is roughly equivalent to the I/O model. The significant difference is that the ideal-cache model assumes an optimal cache-replacement policy. Thus, although the cache-oblivious algorithms themselves are not aware of and have no control over evictions, upper-bound analyses may posit any eviction strategy.

Cache-oblivious algorithms are appealing for several reasons. Since performance analyses hold for any B and M, an analysis in a two-level model holds for *all* levels of the memory hierarchy. Moreover, an algorithm need not be aware of the particular architecture of the machine on which it is run, allowing for more portable code programs.

To simplify bounds, we assume without loss of generality that at least on block of each data structure is always in memory. Otherwise, all bounds should include a $+1$ (e.g., $O(\log_B N + 1)$ instead of $O(\log_B N)$). Moreover, we assume that $B \geq 2$. (To accommodate $B = 1$, many bounds should have B replaced by $B + 1$, e.g., $O(\log_{B+1} N)$, which is unnecessarily difficult to read.)

The query/insert tradeoff

In fact, there is a tradeoff between the cost of searching and inserting in external-memory dictionaries [59], and B-trees achieve only one point on this tradeoff. Another point is achieved by the ***buffered-repository tree (BRT)*** [61]. The BRT is a cache-aware dictionary with searches and inserts costing $O(\log N)$ and $O((\log N)/B)$ transfers, respectively. Thus, searches are slower in the BRT than in the B-tree, whereas insertions are significantly faster.

More generally, Brodal and Fagerberg's cache-aware data structure from [59], which we call the ***B^ε-tree*** , spans a large range of this tradeoff: For $0 \leq \varepsilon \leq 1$, the B$^{\varepsilon}$-tree supports insertions in amortized $O((\log_{B^{\varepsilon}+1}(N/M))/B^{1-\varepsilon})$ transfers and searches in $O(\log_{B^{\varepsilon}+1}(N/M))$ transfers. Thus, when $\varepsilon = 1$ it matches the performance of a B-tree, and when $\varepsilon = 0$, it matches the performance of a BRT. An interesting intermediate point is when $\varepsilon = 1/2$, in which case searches are slower by a factor of roughly 2, but insertions are faster by a factor of roughly $\sqrt{B}/2$ when compared with a B-tree.

When $\varepsilon = 1$, the B$^{\varepsilon}$-tree has performance matching a B-tree. When $\varepsilon = 0$, it has searches costing $O(\log_2(N/M))$ and inserts costing $O((1/B)\log_2(N/M))$. When $\varepsilon = 1/2$, searches are slower than B-trees by a factor of roughly 2, but insertions are faster than B-trees by a factor of roughly $\sqrt{B}/2$.

Contributions

The next three chapters describe two main types of improvements to CO dictionaries: adding concurrency to CO B-trees and designing dictionaries with lower update costs.

One shortcoming of the previously described CO B-trees is that they do not support concurrent access by different processes, whereas typical applications, such as databases and file systems, need to access and modify their data structures concurrently. Chapter 8 describes three variants of a CO B-tree that supports concurrent accesses, using either lock-based and lock-free techniques. When accessed serially, these data structures all have performance bounds comparable with serial CO B-trees.

Data Structure	Search	Insert/Delete
static CO search [169]	$O(\log_B N)$	not supported
CO B-trees [37,41,60]	$O(\log_B N)$	$O(\log_B N)$
lower bound [59]	$O(\frac{1}{\varepsilon}\log_B \frac{N}{M})$	$\Longrightarrow$ $\Omega\left(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B \frac{N}{M}\right)$
COLA [Chapter 9]	$O(\log_2 \frac{N}{M})$	$O(\frac{1}{B}\log_2 \frac{N}{M})$
shuttle tree [42]	$O(\log_B N)$	$O\left(\frac{1}{B^{\Theta(1/(\log\log B)^2)}}\log_B N + \frac{1}{B}\log^2 N\right)$
xDict [Chapter 10]	$O(\frac{1}{\varepsilon}\log_B \frac{N}{M})$	$O\left(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B \frac{N}{M}\right)$

Figure 7-1: Summary of known cache-oblivious dictionaries. Our xDict uses the tall-cache assumption that $M = \Omega(B^2)$. For previous data structures, the $\log_B N$ may well reduce to $\log_B \frac{N}{M}$, but only $\log_B N$ was explicitly proven.

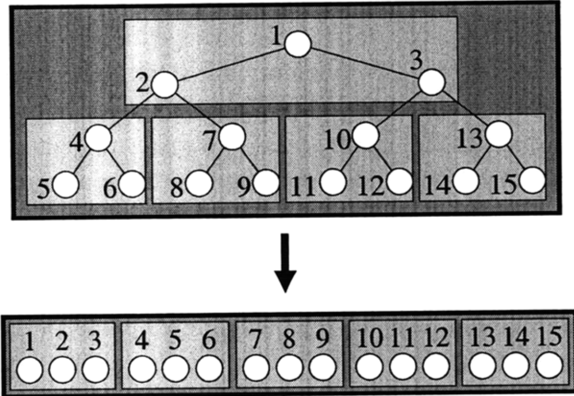


Figure 7-2: The van Emde Boas layout from [169]. A tree of height h is cut into subtrees at height near $h/2$, so each subtree is $\Theta(\sqrt{N})$ in size. This example shows the layout of a tree with 8 leaves.

Chapters 9 and 10 explore the query/insert tradeoff for *cache-oblivious* dictionaries, summarized by the Figure 7-1. Chapter 9 describes a simple cache-oblivious data structure, called the COLA, that achieves the optimal bounds at one point on the tradeoff curve, namely matching the B$^\varepsilon$-tree for $\varepsilon = 0$. Chapter 10 gives a more complicated cache-oblivious data structure, called the xDict, that achieves the tradeoff for any $\varepsilon > 0$. The shuttle tree mentioned in Figure 7-1 is also my work, but the data structure is more complicated and provides less general bounds than the xDict, and hence it is not included in this thesis.

7.1 Previous cache-oblivious dictionaries

There are two main approaches to implementing serial CO B-trees. One approach is based on packed-memory arrays and the other on exponential search trees. Both approaches employ a static CO search tree [169] as a building block.

A static CO search tree [169] contains a set of N ordered elements in a complete binary tree. It executes searches using $O(\log_B N)$ memory transfers. The elements are laid out in an array using the ***van Emde Boas layout*** (see Figure 7-2). Each node in the binary tree is assigned to a position in a length-N array. To perform the layout, split the tree at roughly half its height to obtain $\Theta(\sqrt{N})$ subtrees each with $\Theta(\sqrt{N})$ nodes. Each subtree is assigned to a contiguous portion of the array, within which the subtree is recursively laid out. The array is stored in memory or disk contiguously.

A static CO search tree executes searches in $O(\log_B N)$ memory transfers [169]. To understand

this bound, consider the decomposition of the tree into subtrees of size between $\sqrt{B}$ and B. The depth of each subtree is at least $\lg \sqrt{B}$, so any root-to-leaf path encounters at most $\lg N / \lg \sqrt{B} = 2 \log_B N$ subtrees. Each of these subtrees can cross at most one block boundary, leading to $4 \log_B N$ memory transfers in the worst case. This analysis is not tight. In particular, [39] proves a bound of $\left(2 + 6/\sqrt{B}\right) \log_B N + O(1)$ expected memory transfers, where the expectation is taken over the random placement of the tree in memory.

The packed-memory array [41, 123] appeared in the earliest serial dynamic CO B-tree [35] and subsequent simplifications [37, 60]. The packed-memory array stores the keys in order while permitting dynamic insertions and deletions, and uses a static CO search tree to search the array efficiently. The idea is to store all of the leaves of the tree in order in a single large array. If all the elements were packed into adjacent slots of the array with no spaces, then each insertion might require $\Omega(N)$ elements to be displaced in the worst case, as in insertion sort. The fix to this problem is to leave some gaps. In particular,one can arrange the gaps in the array so that insertions and deletions require amortized $O(\log_B N)$ memory transfers.[1] The amortized analysis allows that every once in a while the array can be cleaned up, e.g., when space is running out.

An exponential search tree [21, 23] can be used to transform a static CO search tree into a dynamic CO B-tree [34, 170]. An exponential search tree is similar in structure to a B-tree except that nodes vary dramatically in size, there are only $O(\log \log N)$ nodes on any root-to-leaf path, and the rebalancing scheme (where nodes are split and merged) is based on some weight-balance property, such as ***strong weight balance*** [25, 41, 152, 159]. Typically, if a node contains M elements, then its parent node contains something near M^2 elements. Each node in the tree is laid out in memory using a van Emde Boas layout, and the balancing scheme allows updates to the tree sufficient flexibility to maintain the efficient layout despite changes in the tree.

Most cache-oblivious dictionaries fall into the class of cache-oblivious B-trees, having bounds matching the B-tree. The notable exception is a cache-oblivious alternative to the BRT, which we call here the ***lazy-search BRT*** [24]. The lazy-search BRT supports searches and insertions in $O(\log N)$ and $O((\log N)/B)$ block transfers, respectively. Although it is useful in some contexts (such as cache-oblivious graph traversal) the lazy-search BRT is unsatisfactory in two crucial ways: keys are assumed to be in the range $[1, N]$, and searches are heavily amortized, so that the whole cost of searching is charged to the cost of previous insertions. Indeed, any given search might involve scanning the entire data structure.

[1] Sometimes the data structures are described with time bounds $O(\log_B N + (\log^2 N)/B)$. It usually depends whether the data structure is intended to support range queries efficiently, returning all elements having keys between two values. In fact, by using scanning structures such as [40] and amortizing the cost of range queries, the $(\log^2 N)/B$-term can be reduced or removed. However, the details and resulting structure are not directly pertinent to this thesis.

Chapter 8

Concurrent Cache-Oblivious B-Trees

This chapter explores adding concurrency to cache-oblivious B-trees. This chapter represents joint work with Michael A. Bender, Seth Gilbert, and Bradley C. Kuszmaul, previously appearing in [47].

Concurrency introduces a number of challenges. A naïve approach would lock segments of the data structure during updates. For example, in a traditional B-tree, each block is locked before being updated. Unfortunately, in the CO model it is difficult to determine the correct granularity at which to acquire locks because the block size is unknown. Locking too small a region yields complicated locking protocols. If the locking is not done carefully, deadlocks may be possible. Although locking larger regions may simplify the locking, locking too large a region may result in poor concurrency.

Second, since searches may be more common than inserts or deletes in database and file-system applications, it is desirable that searches be ***nonblocking***, meaning that each search can continue to make progress, even if other operations are stalled. Our solutions in this section do not require search operations to acquire locks.

Another problem arises with maintaining the "balance" of the data structure. Both main approaches to designing serial CO B-trees require careful weight balance of the trees to ensure efficient operations. With concurrent updates, the balance can be difficult to maintain. For example, the amortized analysis of a packed-memory array allows a process to rewrite the data structure completely once in a while. But if the rewrite acquires locks that prevent other processors from accessing the data structure, then the average concurrency drops: on average only $O(1)$ processes can access the data structure concurrently.

Finally, there is a significant asymmetry between reading and writing the memory. We analyze our data structures under the concurrent read, exclusive write (CREW) model. Two processes can read a location in memory concurrently, since two caches can simultaneously hold the same block in a read-only state. In fact, it may be preferable if parts of the data structure, e.g., the root, are accessed frequently and maintained in cache by all processes. On the other hand, when multiple processes attempt to write to a block, the memory accesses to the block are effectively serialized, and all parallelism is lost.

We respond to these challenges by developing new serial data structures that are more amenable to parallel accesses, and then we apply concurrency techniques to develop concurrent CO B-trees.

Contributions

This section presents three concurrent CO B-tree data structures: (1) an exponential CO B-tree (lock-based), (2) a packed-memory CO B-tree (lock-based), and (3) a packed-memory CO B-tree (nonblocking). I show that each data structure is ***linearizable***, meaning that each completed operation (and a subset of the incomplete operations) can be assigned a ***serialization point***; each operation

appears, from an external viewer's perspective, as if it occurs exactly at the serialization point (see, e.g., [120, 144]). I also show that the lock-based data structures are ***deadlock free***, i.e., that some operation always eventually completes. I show that the nonblocking B-tree is *lock-free*; that is, even if some processes fail, some operation always completes. When only one process is executing, the trees gain all the performance advantages of optimal cache-obliviousness; for example, they execute operations with the same asymptotic performance as the nonconcurrent versions and behave well on a multi-level cache hierarchy without any explicit coding for the cache parameters. When more than one process is executing, the B-trees still operate correctly and with little interference between disparate operations.

The remainder of this chapter is organized as follows. Section 8.1 describes the concurrent CO model. Section 8.2 presents a lock-based exponential CO B-tree. Section 8.3 presents a lock-based packed-memory CO B-tree, whereas Section 8.4 presents a lock-free packed-memory CO B-tree. Section 8.5 concludes with a discussion of other ways to build concurrent CO B-trees and of open problems.

8.1 The concurrent-cache model

This section describes the concurrent-cache model and discusses the concurrency mechanisms used throughout the remainder of this chapter.

Recall that the ideal-cache model, as introduced in [98, 169], models a single level of the memory hierarchy. The model consists of two components: the main memory and a cache of size M. Both are divided into blocks of size B. The values of M and B are unknown to the algorithm, hence the term "cache oblivious."

This section extends the model to a parallel (or distributed) setting consisting of P processors, which we call the ***concurrent-cache model***. We consider the case where each processor has its own cache of size M/P. As multiple processors are accessing the same memory locations concurrently, we also consider how the caches interact. In particular, we assume an interaction among caches that is consistent with the MSI cache coherency or more complicated variants like MESI or MOESI [116, Section 4.3]. A block may reside in multiple caches, in which case it is marked in each cache as *shared.* A block that is marked *exclusive* can reside in only a single cache. A processor can perform write operations only on blocks for which it has obtained exclusive access.

If multiple processors concurrently request shared access to a block, the block is placed in each of the caches and marked shared. If a processor requests exclusive access to a block, the block is evicted from all other caches and is marked exclusive; all other concurrent requests for shared or exclusive access fail. If a cache is full, each successful request results in an old block being evicted; we assume that the least recently used (LRU) block is evicted[1].

Each request for a block costs one memory transfer, regardless of whether the block is requested shared or exclusive and regardless of whether the request is successful. There is no fairness guarantee that a processor is eventually successful. A single read or write operation may, in fact, be quite expensive, resulting in a large number of unsuccessful block requests if there are many concurrent write requests.

This chapter makes use of two different types of support for concurrency. For much of the work, the algorithms use locks to synchronize access to pieces of memory. The nonblocking algorithms, on the other hand, use ***load-linked/store-conditional*** (LL/SC) operations. An LL operation reads

[1]The results presented later in this section hold for any reasonable replacement strategy. Unlike in the ideal-cache model, we do not assume an optimal replacement strategy. One difficulty in the concurrent setting is that it is unclear what "optimal" means because changes to the replacement policy affect the scheduling of the program.

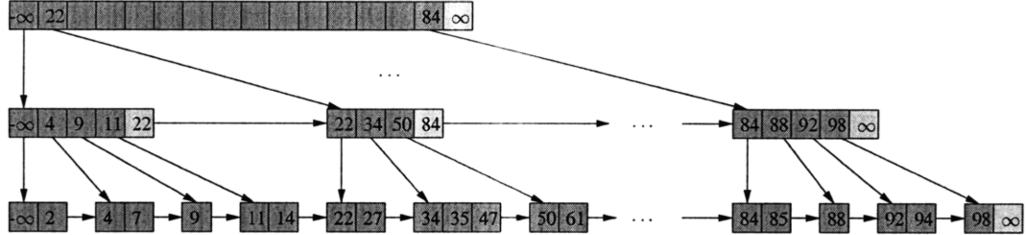


Figure 8-1: An exponential CO B-tree. Nodes grow doubly exponentially in the height of the tree. The dark-gray keys of internal nodes point to the smallest key in the corresponding subtree. Data is stored in the leaves. The light-gray keys of internal nodes indicate the *right-key*. Each node has a *right-link* pointer, forming a linked list at each level in the tree.

the memory and sets a link bit. If any other operation modifies the memory, the link bit is cleared. An SC operation writes the memory only if the link bit is still set; otherwise the memory remains unchanged. The return value of an SC indicates whether it succeeded. This approach avoids the well-known ABA problem that arises with ***compare and swap*** (CAS). There has been much research showing how to implement LL/SC with CAS and vice versa, implying that in some senses they are equivalent (e.g., [124, 155]), and we believe it is not difficult to modify our construction for the somewhat more common CAS operation.

8.2 Exponential CO B-tree

This section presents a concurrent cache-oblivious B-tree called an exponential cache-oblivious B-tree. We first describe the data structure. We then prove correctness and give a performance analysis. We conclude by discussing some interesting aspects of this data structure.

Data-structure description

The data structure uses a strongly weight-balanced exponential tree [21, 23, 34, 170] to support searches and insertions. Each node in the tree contains child pointers and a pointer to the node's right sibling $\mathit{right\text{-}link}$ (see Figure 8-1). A node maintains the set of keys that partition its children and the key $\mathit{right\text{-}key}$ of the minimum element in the $\mathit{right\text{-}link}$'s subtree.

I now describe the protocol for searches and insertions. The tree is parameterized by a constant α, for $1 < \alpha < 2$, which affects the ***height*** of the tree. A leaf has height 0, and a node has height 1 more than its children.

To search for a key κ, begin at the root of the tree and follow child pointers or sibling pointers until reaching the leaf containing the target element. At each intermediate node, examine $\mathit{right\text{-}key}$. If $\mathit{right\text{-}key}$ is smaller than κ, then follow the $\mathit{right\text{-}link}$ pointer. Otherwise, proceed to the appropriate child. On reaching a leaf, the search continues to follow $\mathit{right\text{-}link}$ pointers until either κ is found, in which case the search returns the value, or a key larger than κ is found, in which case the search fails. No locks are acquired during the search.

To insert key κ, first search for the leaf where κ should be inserted, acquire a lock on the leaf, perform the insertion at the leaf, and release the lock. Next, determine whether the key κ should be ***promoted***, inserting κ into the parent node, or whether the insertion is complete. Promote the key κ in the leaf to height 1 with probability $1/2$. When promoting κ, reacquire the lock and split the leaf u containing κ. Node u keeps all of the keys less than κ, and the new leaf acquires all of the keys greater than or equal to κ. We then release the lock and insert the promoted key κ into the parent of

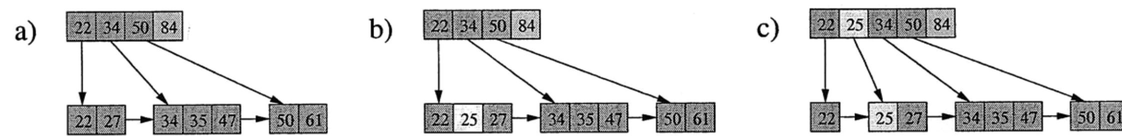


Figure 8-2: An example of an insert of the key 25 into an exponential CO B-tree. a) The original state of the tree. b) The resulting structure if 25 is not promoted. c) The resulting structure if 25 is promoted one level.

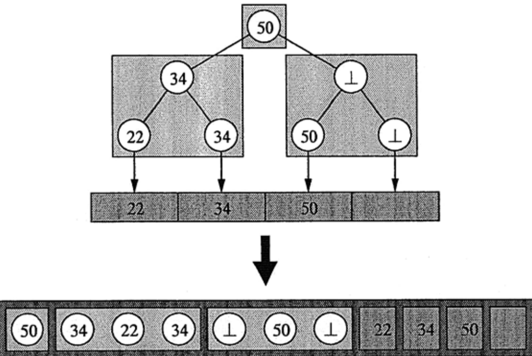


Figure 8-3: The modified van Emde Boas layout of a node in the exponential CO B-tree. This figure shows the layout of the node given as the root of (a) in Figure 8-2

u. More generally, when inserting a key κ into a node u' of height h, promote κ from height h to height $h+1$ with probability $1/2^{\alpha^h}$, which entails reacquiring the lock, splitting u', releasing the lock, and inserting κ into the parent of u'. Figure 8-2 gives an example of an insert and promotion.

In order for nodes to be searched efficiently, the keys in a node are laid out using a ***modified van Emde Boas layout*** so that a node of size k can be traversed with $\Theta(\log_B k + 1)$ memory transfers. An example of the modified layout is depicted in Figure 8-3. The node is divided into two pieces: a size-$\lceil\!\lceil k \rceil\!\rceil$ array[2] filled from the left holding all the keys in a node, and a complete $\lceil\!\lceil k \rceil\!\rceil$-leaf static CO search tree used to efficiently search for an array slot. The ith leaf of the search tree points to the i-th array slot. In particular, a leaf containing κ in the search tree points to κ in the array. This approach is reminiscent of [36] and reappears in Section 8.3. The static CO search tree consists of keys laid out in a van Emde Boas layout [169]. This layout consumes (roughly) the first $2/3$ of the memory used by a node. The final $1/3$ of the memory contains the keys stored in order as a vector. A search within a node ends at the location of the key in the vector.

When rewriting a node (either because the key is inserted into the node or because the node is split), update the modified van Emde Boas layout as follows. First, rewrite the vector of keys (the right half of the node), proceeding from largest to smallest. Then, update the van Emde Boas layout in the left half. This layout ensures that we can perform concurrent searches even while the node is being updated. If an insert is performed when the vector of keys is already full, allocate a new node of twice the size.

Correctness

I first argue that the data structure is correct, even under concurrent operations. The most common way of showing that an algorithm implements a linearizable object is to show that in every execution there exists a total ordering of the operations with the following properties: (1) the ordering is consistent with the desired insert/search semantics, and (2) if one operation completes before another

[2]The ***hyperceiling*** of x, denoted $\lceil\!\lceil x \rceil\!\rceil$, is defined to be $2^{\lceil \log x \rceil}$, i.e., the smallest power of 2 greater than x.

begins, then the first operation precedes the second in the ordering. Linearizability follows due to a straightforward extension of Lemmas 13.10 and 13.16 in [144].

Theorem 8.1 *The exponential CO B-tree guarantees linearizable insertions and searches and is deadlock-free.*

Proof. In order to show that an appropriate total ordering of the operations exists, we need to show that if an operation completes, all later operations are consistent with it. If an insert operation finishes inserting a key κ at level 0, or a search finds a key κ at level 0, then any later search also finds key κ.

First, notice that at every level of the tree, the keys are stored in order. That is, if the largest key in node u is κ, then the smallest key in node $u.\,right\text{-}link$ is larger than κ. This fact follows from the two ways in which a node is modified. First, a node may be split, say at key κ. In this case, the split operation, protected by a lock, preserves this invariant, moving κ and all larger elements into a new node that can be reached by *right-link*. Second, a key κ may be inserted into a node. In this case, the node is locked during the insertion, and some of the elements in the vector half of the node move one position to the right to make room for κ.

This ordering of keys implies that if key κ is in the leaf node of the tree when a search begins, then throughout the search, a leaf containing key κ is reachable by the search. The search begins at the root, and every leaf is reachable from the root. We proceed by induction: at each step of the search, a child pointer is chosen with a minimum key no greater than κ. The only interesting case is when the child node is concurrently split. Even in this case, however, the child node always contains a key no greater than κ, since ever when a node is split it always maintains its minimal key. We conclude that κ is still reachable.

Finally, deadlock-freedom follows immediately, since each process holds only one lock at a time. □

Cache-oblivious performance

I first consider the cost of individual operations when the data structure is accessed sequentially. I then analyze the concurrent performance for the special case when search and insert operations are performed uniformly at random. In both cases, we use the cache-oblivious cost model, counting only cache misses.

Theorem 8.2 *Assume that operations occur sequentially. A search in an N-node tree takes $O(\log_B N + \log_\alpha \lg B)$ block transfers, with high probability; an insert takes $O(\log_B N + \log_\alpha \lg B)$ block transfers in expectation.*

Proof. If a tree contains N keys, then with high probability every node in the tree has height less than $\log_\alpha \lg N + O(1)$. Specifically, the probability that any key is promoted to height $\log_\alpha \lg N + c$ is $O(n^{-\alpha^c+1})$. Moreover,if u is a node of height h, then the number of keys in u is $O(2^{\alpha^h})$ in expectation and $O(2^{\alpha^h} \lg N)$ with high probability. For $h = \Omega(\log_\alpha \lg\lg N)$, that is, when the expected node size is at least $\Omega(\lg N)$, the number of keys is $O(2^{\alpha^h})$ with high probability.

Next, let us calculate the cost of a search operation. Searching a node of height h, laid out in the modified van Emde Boas layout, for the correct child pointer takes $O(\log_B(2^{\alpha^h}) + \log_B \lg N + 1)$ memory transfers, with high probability. For nodes with expected size at least $\Omega(\lg N)$, the cost is $O(\log_B(2^{\alpha^h}) + 1)$ with high probability. We sum the costs for all levels, ranging from $i = 0$ to

$\log_\alpha \lg N + c$ (with high probability):

$$\Theta\left(\sum_{0}^{\log_\alpha \lg N + c} \log_B(2^{\alpha^h}) + \sum_{0}^{\log_\alpha \lg \lg N} \log_B \lg N\right).$$

The right term sums to $O(\log_B N)$. For the left term, we consider two cases, depending on whether or not the size of the node is at least B. For all levels where the nodes are of size at least B, the cost is dominated by the root node (or possibly a node just below the root). For nodes of size less than B, the block transfer cost is at most 2. Therefore, with high probability, the total cost of a search is $O(\alpha^c \log_B N + \log_\alpha \lg B)$.

Next, let us calculate the cost of an insert operation. Consider the expected cost of inserting a key at height h of the tree. Recall that we promote a key from level $h-1$ to level h with probability $2^{-\alpha^{h-1}}$. Thus, the probability of promoting a given key to height h is $2^{-(\alpha^h-1)/(\alpha-1)}$. The cost of inserting a key into a node at height h, given that the key reaches height h, is the cost of rebuilding the entire node, which is $O(1 + 2^{\alpha^h} \lg N/B)$ with high probability.

Therefore, the expected cost at height h is the probability that the insertion reaches height h times the cost of rewriting a node at height h. The expected cost is therefore $O(2^{-c'(\alpha^h-1)} \log N/B)$, where $0 < c' < 1$ depends on the value of α. The summation across all $\log_\alpha \lg N + O(1)$ levels, then, is $O(\lg N/B) \leq O(\log_B N)$. Hence, the total expected cost of an insert is the cost of searching for the right place to do the insertion plus the cost of doing the insertion. That is, the cost is $O(\log_B N + \log_\alpha \lg B)$. □

If there are a large number of concurrent search operations, but no insert operations, each search operation incurs a cost of $O(\log_B N + \log_\alpha \lg B)$, the same as in the sequential case.

If there are many concurrent searches and insertions, the performance may deteriorate. The performance of concurrent B-trees is difficult to analyze since it depends significantly on the underlying parallelism of the algorithm using the data structure. For example, if a large number of processes all try to modify a single key, then the operations are inherently serialized, and there is no possible parallelism. As a result, for most concurrent B-tree constructions (and in fact, most concurrent data structures), little is stated about their performance. Here, I analyze the "optimally parallel" case, where each search and insertion operation targets a randomly chosen key, and I show that the data structure still yields good performance.

Theorem 8.3 *Assume that all processors are synchronous. If there are $O(N^{2/\alpha-1}/\text{polylog}N)$ search and insert operations performed uniformly at random in an N-node tree, then the expected cost for an operation is $O(\log_B N + \log_\alpha \lg N)$ block transfers.*

Proof. In the absence of concurrent insert operations, each search operation takes $O(\log_B N + \log_\alpha \lg B)$ block transfers. If an insert operation delays a search, we charge that cost to the insert operation.

An insert operation has four costs: finding the insert point, acquiring the lock, performing the insertion, and delaying other operations. As in the case of searches, finding the appropriate insertion point costs $O(\log_B N + \log_\alpha \lg B)$ plus any delays caused by other insert operations. Again, this delay is charged to the operation causing the delay.

The cost of performing the insertion itself is equivalent to the sequential case: $O(\log_B N + \log_\alpha \lg B)$ block transfers.

It remains to calculate the effects of concurrency: the cost of acquiring the lock and the cost of delaying other operations. Consider the expected cost incurred by a delaying one particular

concurrent operation at some level h. There are three components to the cost: (1) the probability that the insert reaches level h, (2) the probability that the insert modifies the same node, and (3) the actual cost incurred by delaying the concurrent operation.

First, as in Theorem 8.2, an insert operation reaches level h with probability $2^{-(\alpha^h-1)/(\alpha-1)}$.

Next, an insert chooses the same node at level h with probability $2^{(\alpha^{h+1}-1)/(\alpha-1)}/N$, since the expected number of nodes at level h is the number of elements promoted to level $h+1$. This probability can vary by at most a log factor, with high probability, which is enough for the proof.

Finally, we consider the real cost of delaying any one operation at level h. The expected size of a node at level h in the tree is $O(2^{\alpha^h})$. Therefore, the concurrent operation can be delayed by at most $O(2^{\alpha^h})$ in expecation, which is the cost of performing one memory transfer for each element in the node. This worst-case analysis makes no use of the cache: in the case of a search operation, the block may be repeatedly transferred back and forth between the search operation and the insert operation. Moreover, the search may need to read the entire node since the contents of the node are changing during the search. The size of the node also bounds how long the insert may need to wait to acquire the lock.

We now sum over the possible heights in the tree. Consider $h \leq h_{max} = \log_\alpha \lg N + \log_\alpha(\alpha - 1) - 1$, which is the point at which the second term (i.e., the probability that the insert occurs at the same node) reaches 1. Multiplying the three terms results in an expected cost of $2^{2\alpha^h}/N$. In the worst case, where $h = h_{max}$, this expression is equal to $N^{1-2/\alpha}$; for smaller h, the cost decreases. Since there are at most $N^{2/\alpha-1}$ concurrent operations, the expected cost per level is $O(1)$. Summing over all levels $\leq h_{max}$ leads to an expected cost of $O(\log_\alpha \lg N)$.

Next, consider the case where $h \geq h_{max}$. The second term never exceeds 1. The first and third terms decrease geometrically like $O(1/2^{\alpha^h})$, since the tree is unlikely to exceed height h_{max}, and so the sum is bounded by the term $h = h_{max}$, as before.

The polylog in the theorem arises since there can be an $O(\log N)$-factor variation in the size and number of nodes at a level, with high probability. □

In a concurrent setting, the expected cost is $O(\log_B N + \log_\alpha \lg N)$, instead of $O(\log_B N + \log_\alpha \lg B)$ as in the sequential case. The additional cost is incurred when two operations interfere.

Exponential CO B-tree discussion

One interesting aspect of this data structure is parameterizing of the tree by α. By skewing the tree wider (rather than deeper), we reduce the concurrency at the root, thus improving the overall performance. When α is closer to 1, the concurrent performance is better; when α is closer to 2, the sequential performance is better.

A second aspect is that the data structure requires relatively minimal memory management, since it does not support delete operations. The only case in which memory must be deallocated (or wasted) is when a node in the exponential tree grows too big (before splitting), and a new contiguous block of memory must be allocated for the node.

8.3 Packed-memory CO B-tree

This section presents a second concurrent, cache-oblivious B-tree, called a packed-memory CO B-tree. It supports insertions, deletions, searches, and range queries using lock-based concurrency control. We first describe the lock-based data structure. We then prove the data structure is correct and analyze its serial performance.

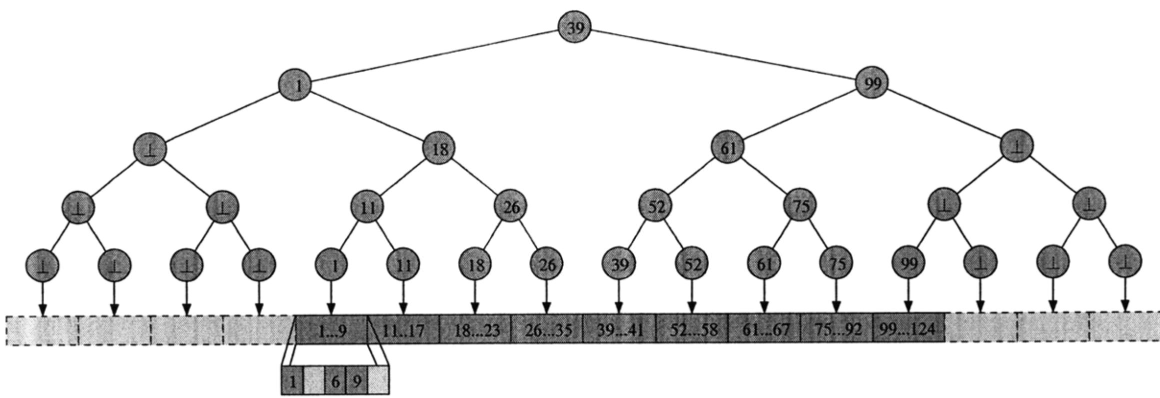


Figure 8-4: A packed-memory CO B-tree consists of a static cache-oblivious search tree used to index into the one-way packed-memory array. Each leaf in the tree points to a $\Theta(\log N)$-sized region of the array. The active region of the array is outlined with a solid border. The active region of the array contains gaps allowing for insertions, and the array can grow to the right or shrink from the left.

Data-structure description

Here, I describe the lock-based packed-memory CO B-tree. I present a lock-free version of this data structure in Section 8.4. The data structure is based on the cache-oblivious B-tree presented in [37] and consists of a static cache-oblivious search tree [169], which is used to index a packed-memory data structure. Instead of using the packed-memory array from [35], I introduce a new "one-way packed-memory structure," which is more amenable to concurrent operations. Each leaf in the static tree points to a $\Theta(\log N)$-size region in the packed-memory array.

The one-way packed-memory structure maintains N elements in order in an array of size $m = \Theta(N)$ (with m a power of 2) while permitting dynamic element insertion and deletion (see Figure 8-4). The array consists of three segments: the leftmost and rightmost segments contain extra empty space, and the middle segment — the ***active region*** — contains the N elements and some gaps between elements. On an insertion, the active region may grow to the right; on a deletion, the active region may shrink from the left. If the active region grows or shrinks too much, then we reallocate the array.

We maintain a near-constant "density" in the active region by ***rebalancing*** regions of the array—that is, evenly spreading out elements within a region—on insertions or deletions. In the one-way packed-memory structure, the rebalances ensure that elements move only in one direction: to the right (see Lemma 8.4).

The ***density*** of a subarray is the number of filled array positions divided by the size of the subarray. Consider a subarray of size k with $i = \lceil \lg k \rceil$. The size of a subarray determines its ***upper-bound density threshold***, τ_i, and its ***lower-bound density threshold***, ρ_i. For all i and j, $\rho_i < \rho_{i+1}$, $\tau_j > \tau_{j+1}$, and $\rho_i < \tau_j$.

The density thresholds follow an arithmetic progression defined as follows. Let $0 < \rho_{min} < \rho_{max} < \tau_{min} < \tau_{max} = 1$ be arbitrary constants. Let $\delta = \tau_{max} - \tau_{min}$ and $\delta' = \rho_{max} - \rho_{min}$. Then, define density thresholds τ_i and ρ_i to be

$$\tau_i = \tau_{max} - \frac{i-1}{\lg m - 1}\delta ,$$

and

$$\rho_i = \rho_{min} + \frac{i-1}{\lg m - 1}\delta' ,$$

for all i with $1 \leq i \leq \lg m$.

For example, suppose that $m = 16$ and fix the threshold $\tau_{min} = 1/2$. There are $\lg 16 = 4$ density thresholds $\tau_1 = \tau_{max}$, τ_2, τ_3, and $\tau_4 = \tau_{min}$. The thresholds increase linearly with $\tau_1 = 1$, $\tau_2 = 5/6$, $\tau_3 = 4/6$, and $\tau_4 = 1/2$. Similarly, if we fix $\rho_{min} = \rho_1 = 1/8$ and $\rho_{max} = \rho_4 = 1/4$, then $\rho_2 = 4/24 = 1/6$ and $\rho_3 = 5/24$.

Search. To search for a key κ, search down the static binary tree as normal, without any locks. When the search reaches the array, scan right until finding the appropriate array slot. Since the search proceeds without locks, we need to perform an ABA test (key, value, key) to make sure the element did not move during the read.

Insertion and deletion. To insert a new element y with key κ, first perform a search to find and then lock the $\Theta(\log N)$-sized leaf where key κ should be inserted. Next, scan the leaf from left to right to find the actual slot s for key κ. (Slot $s - 1$ should contain the largest key smaller than κ.) If the scan advances into the next $\Theta(\log N)$-size region (due to a concurrent operation), acquire a lock on the next tree leaf, give up the lock on the current, and continue.

If the slot s is free, place y in the available slot. Otherwise, we must rebalance a section of the packed-memory array. Explore right from s in the array, acquiring locks on regions along the way, until finding the smallest region of any size that is not too dense.[3] A region of size k, with $\lceil \lg k \rceil = i$, is not "too dense" when the density is no greater than τ_i. Then, rebalance the elements evenly within the window by moving elements to the right, starting with the rightmost element and working to the left. Move the elements such that no suffix of the rebalanced region has density exceeding the threshold τ_i.[4] Finally, release the locks. Unlike the packed-memory array from [35] rebalance windows do not have to have sizes that are powers of 2. Figure 8-5 gives an example of an insert into the one-way packed-memory array.

Deletions are analogous to insertions, except that when an element is deleted, always perform a rebalance. A deletion-triggered rebalance explores right until finding the first slot preceding an occupied slot, then explores left until finding a region that is "dense enough." This exploration proceeds without locks. Once the region is established, acquire locks on the region from left to right. As with the insertion-triggered rebalance, next spread elements evenly starting with the rightmost element and working left. Finally, release all the locks.

Once the insert/delete or rebalance is complete, update the keys in the static search tree. For each node in the subtree defined by the updated region, if the key at a node in the tree is too small, we lock the node, write the key, and release the lock. This strategy ensures that concurrent searches are not blocked.

Resizing the array When the right boundary of the active region hits the array boundary, or the active region becomes too small, ***resize*** the array, copying the elements into a new array, spreading the elements evenly. The "ideal" density of the active region is $(\tau_{min} - \rho_{max})/2$. If the old array has N elements, the new array has size $\lceil\lceil cN/(\tau_{min} - \rho_{max})/2)\rceil\rceil$, for a constant $c > 0$. The active region starts at the left portion of the new array.

[3]To attain these ***one-way rebalances***, select the rebalance regions differently than in [35]. Rebalance regions extend to the right on inserts and to the left on deletions. By not restricting region sizes to powers of 2, it is always possible to rebalance by moving elements only to the right. In contrast, previous algorithms (such as [33, 129]) employ packed-memory arrays in which the rebalance region extends entirely to the right of the inserted element, but elements may move in either direction. Unlike in [35], rebalances require only one pass to read the data and one pass to write the data.

[4]Thus, every prefix of the region not including the last slot has density at least the threshold τ_i. In particular, a prefix of size k has $\lceil k\tau_i \rceil$ elements.

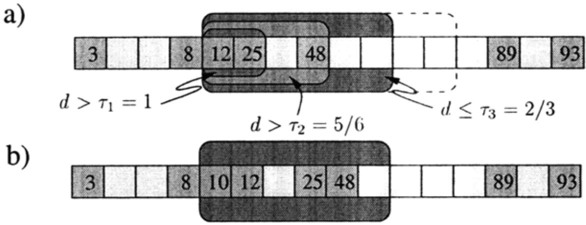


Figure 8-5: An insertion of the key 10 into the above packed-memory array of size $m = 16$ and thresholds $\tau_{max} = 1$ and $\tau_{min} = 1/2$. The new element 10 should go in the slot containing 12, so explore right until finding a region that is not too dense, including the new item to insert. Until exploring past a region of size 2 (indicated by the small rounded rectangle in (a)), the density d exceeds the corresponding threshold $\tau_1 = 1$. When exploring between a region of size between 3 and 4, indicated by the medium rounded rectangle, the density is 1 which exceeds the corresponding threshold $\tau_2 = 5/6$. When exploring regions of size between 5 and 8, we use the threshold $\tau_3 = 2/3$. The exploration stops at the sixth slot as there are 4 elements (including the new element 10) to place in 6 slots, giving a density $d = 4/6 = 2/3 = \tau_3$. (b) gives the state of the array after the insert and rebalance. Once the rebalance region is established, we begin at the right and move elements as far right as possible without exceeding the appropriate threshold (i.e., $\tau_3 = 2/3$).

The packed-memory CO B-tree uses a randomized strategy that allows the resizing to run quickly and concurrently. The resize proceeds in three phases: (1) Randomly choose leaves of size $\Theta(\log N)$ and count the elements in the leaf. Lock the node, write the count in the node, and release the lock. If the sibling has been counted, write the sum in the parent and proceed up the tree. When the root node is counted, the first phase ends. (2) Allocate a new array based on the count in the root. Note that the count in any node is exact — the randomization in the first phase randomizes only the order in which nodes are counted. (3) Randomly walk down the original tree to find a leaf, keeping a count of the number of elements to the left of this leaf. Then, copy the $\Theta(\log N)$ elements in the leaf to their correct location in the new array (spreading the elements evenly). For phases (1) and (3), mark nodes to ensure that leaves are not counted or copied multiple times and to discover when the phases complete.

Correctness

I now show that the packed-memory CO B-tree is correct under concurrent operations: the data structure guarantees linearizable operations and is deadlock-free.

The first lemma shows that the one-way rebalance does in fact move elements in only one direction. This lemma directly implies that searches return the correct elements.

Lemma 8.4 *The one-way rebalance moves each element only to the right. That is, the rebalance operation maintains the property that the key at a given memory location is nonincreasing.*

Proof. Consider a rebalance region r ranging from array slots s_1 to s_2. Let $i = \lceil \lg(s_2 - s_1 + 1) \rceil$. Then, the elements in r are spread evenly to a density of τ_i.

Assume for the sake of contradiction that an element in r moves left during the rebalance. Without loss of generality, let s_2' be the slot of the leftmost element that moves left. Consider the region r' ranging from s_1 to $s_2' - 1$. Let d' be the density of region r'. Since spreading the elements evenly in r moves an element into r', we have $d' < \tau_i$. Since r' is contained in r, however, we have $j = \lceil \lg(s_2' - s_1) \rceil$ for some $j \le i$. Thus, $\tau_j \ge \tau_i \ge d'$, generating a contradiction as the rebalance region would be r'.

The proof is similar for deletion-triggered rebalances. □

Theorem 8.5 *The packed-memory CO B-tree is deadlock-free and guarantees linearizable insertions, deletions, and searches.*

Proof. The deadlock-free proof follows directly from the fact that when an operation holds more than one lock, it always acquires these locks from left to right.

Next, we look at the claim of linearizability. The array of the packed-memory CO B-tree is similar to a single node in the exponential CO B-tree, except the array is slightly more complicated. The important property about updates to the array is given in Lemma 8.4. Thus, we can use a similar proof to Theorem 8.1. □

Cache-oblivious performance

I next analyze the cost of sequential operations in the packed-memory CO B-tree in terms of block transfers. I first bound the cost of the static-search-tree update and of resizing the array. I then apply an accounting argument to conclude that the packed-memory CO B-tree achieves the same serial costs as in [35, 36, 60].

First, let us bound the cost of the static tree update.

Lemma 8.6 *If a range of k memory locations are modified during an update in the packed-memory array, then updating the search tree costs* $O(\log_B N + k/B + 1)$ *memory transfers, assuming that operations occur sequentially.*

Proof. We can think of subtrees of the search tree as corresponding to ranges in the packed memory array. Updating the tree requires updating every node in a set of subtrees that constitute the range of k memory locations. The combined number of tree nodes in these subtrees is less than $2k$, but it remains to be shown that these nodes fit in $O(k/B + 1)$ blocks. Consider all the subtrees $\{T_1, \ldots, T_r\}$ to update from left to right. Then there exists an i such that the subtree rooted at $parent(T_i)$ contains the subtrees $T_1, \ldots, T_i$, and $parent(T_{i+1})$ contains $T_{i+1}, \ldots, T_r$. Thus, the tree nodes to update are contained in two subtrees, with total size at most $4k$. All of these subtrees, therefore, are laid out in $O(k/B + 1)$ blocks.

Additionally, some nodes, not in these subtrees, along the path to the root must be updated. Only nodes with a right child that changes are updated. Thus, we update only a single path to the root, which requires at most $O(\log_B N + 1)$ blocks. □

The insertion cost includes not only the cost of rebalancing the array and updating the static tree, but also the cost of resizing the array. The following lemma implies that the array is not resized frequently.

Lemma 8.7 *For a packed-memory array of size* m*, there must be* $\Omega(m)$ *insertions or deletions before the array is resized.*

Proof. Consider a resizing triggered by an insertion. We use an accounting argument to prove the lemma. We give 1 dollar to each filled slot in the array at the time of the last resizing. Whenever an item is inserted into some slot, we give that slot 1 dollar. Whenever an item is deleted, we leave the dollar in the slot. Whenever we rebalance a region of the array, we move 1 dollar with each item moved. All excess dollars (i.e., associated with items the have been removed) are moved to the first slot of the rebalanced region. Consequently, every nonempty slot has at least 1 dollar associated with it.

Let $d = (\tau_{min} - \rho_{max})/2$—the density to which the array is resized. We claim that any prefix of the array, starting at the left boundary (slot 0) and ending at a slot s preceding the right boundary

of the active region, contains at least sd dollars. This invariant holds at the time of the last resizing. It also trivially holds across insertions or deletions that do not trigger rebalances. It remains to be shown that the invariant holds across rebalances.

Consider an insertion-triggered rebalance that ranges from slot s_1 to s_2. The invariant is trivially unaffected for any slot s with $s < s_1$ or $s \geq s_2$ since no money moves into or out of the rebalanced region. It remains to show that the invariant holds for a slot s with $s_1 \leq s < s_2$. By assumption, we have at least $(s_1 - 1)d$ dollars preceding slot s_1. The rebalance algorithm guarantees that the density of every prefix of the rebalanced region is at least τ_{min}. Thus, the array contains at least $(s_1 - 1)d + (s - s_1 + 1)\tau_{min} \geq sd$ dollars by slot s.

For completeness, we also need to consider a deletion-triggered rebalance that ranges from slot s_1 to s_2. Consider a slot s with $s_1 \leq s < s_2$. The rebalance algorithm again guarantees that the density of every suffix of the rebalanced region is at most $\rho_{max} < d$. Thus, since the invariant holds at slot s_2 before the rebalance and all extra dollars are moved to slot s_1, we have that the invariant holds for all slots after the rebalance.

Now, we just apply the invariant to complete the proof. When an insertion triggers a rebalance, the active region includes the entire array. Thus, the array contains at least md dollars. At the last resizing, there were md/c dollars, where both $c > 1$ and $0 < d < 1$ are constants. Thus, there must have been $\Omega(m)$ insertions.

The proof for deletion-triggered resizings is similar. □

Now, let us bound the cost of resizing. The main idea is that $\Theta(N)$ random choices is enough to count $\Theta(N/\log N)$ leaves.

Lemma 8.8 *For a packed-memory CO B-tree containing N elements, the cost of resizing the packed-memory array is $O(N(\log_B N + 1))$ memory transfers, assuming that operations occur sequentially.*

Proof. There are $\Theta(m/\log m)$ leaves. We make only $O(m)$ random leaf selections, with high probability, before selecting every leaf in phases (1) and (3). In phase (3), finding a leaf follows a root-to-leaf path in the tree with a cost of $O(\log_B m + 1)$. The total cost of selecting all the leaves is, therefore, $O(m(\log_B m + 1))$.

Copying or counting a $\Theta(\log m)$-sized region (corresponding to a tree leaf) of the old array takes $\Theta(\log m/B + 1)$ block transfers. Since each leaf is counted and copied once in phases (1) and (3), respectively, the total cost of counting and copying the elements is $\Theta((m/\log m)(\log m/B + 1)) = O(m(\log_B m + 1))$.

Copying a $\log m$-sized region from the old array to the new array takes $\Theta(\log m/B)$ block transfers (since the array is kept near a constant density), for a total cost of $O(m/B)$ across the entire resize.

Updating the tree for the new array costs $O(\log_B m' + 1)$ for each element copied where m' is the size of the new array, for a total of $O(m(\log_B m' + 1))$ block transfers.

Since m and m' are $\Theta(N)$, the lemma follows. □

The following theorem states that we achieve the desired serial performance.

Theorem 8.9 *The amortized cost of insertions and deletions in a packed-memory CO B-tree is $O(\log_B N + \log^2 N/B + 1)$, assuming that operations occur sequentially.*

Proof. This proof is similar to the one in [129] that analyzes the cost of rebalance regions extending only one direction (but in which elements can move in both directions). This argument uses the accounting method, placing $\Theta(\log^2 m/B)$ dollars in an array slot on an insertion/deletion. In particular, $\Theta(\log m/B)$ dollars are associated with each of $\lg m$ accounts, corresponding to $\lg m$

density thresholds. Whenever a region of size $k > \lg m$ is rebalanced,[5] the region contains $\Omega(k/B)$ dollars in an appropriate account with which to pay for the rebalance. Our scenario is different because regions are not necessarily powers of 2, and our resizing algorithm is different. Whereas [129] uses an accounting argument that charges rebalances against the left half of the region of size k, we charge against the left subregion of size $[\![k]\!]/2$.

Moreover, we incur a cost for updating the static tree on top of the packed-memory data structure during a rebalance. Since we have $\Theta(k/B)$ potential saved up at the time of a rebalance of size k and a tree update costs $O(\log_B N + k/B + 1)$ (from Lemma 8.6) we can afford the tree update.

Finally, given that there must be $\Omega(N)$ insertions between resizings (see Lemma 8.7), and a resizing costs $O(N(\log_B N + 1))$ (see Lemma 8.8), we conclude that the cost of resizing the array is amortized to $O(\log_B N + 1)$ per insertion. □

Packed-memory CO B-tree discussion

Here, we discuss why the packed-memory CO B-tree is amenable to concurrency in contrast to the cache-oblivious packed-memory B-tree of [36].

- *Maintaining Nonblocking Searches* — When the packed-memory array from [35, 36] rebalances a region, it may move elements both to the right and the left in the array. A search for an element in a region that is being rebalanced may not find the element. Moreover, a scan of the array may miss the element if the element is concurrently moved in the opposite direction of the scan.

  We solved this problem by developing the one-way packed-memory array in which elements move only to the right. Thus, whenever a search reaches the array, the search is either at or to the left of the element (see Lemma 8.4 and a similar argument in Section 8.2).

- *Concurrent Resizing*— A naïve algorithm for copying data starts at the left of the active region and copy each element sequentially. This technique does not allow for concurrent work during a resize.

  We resized by randomly counting then randomly copying. Thus, many processors can work concurrently towards resizing the array.

8.4 Lock-free CO B-tree

I now discuss how to transform the lock-based data structure in Section 8.3 into a nonblocking, lock-free CO B-tree. Instead of using locks, I use load-linked/store-conditional (LL/SC) operations. For the sake of clarity, assume that keys and values are each a single word, and can be read and written by a single LL/SC operation; the data structure is easily extensible to multiword keys and values.

Recall that locks are used only in updating the packed-memory array; the static search tree is already nonblocking. Instead of acquiring locks before modifying the packed-memory array, Idescribe how to use four basic nonblocking primitives to update the data structure: (1) ***move***, which moves an element atomically from one slot in the packed-memory array to another, (2) ***cell-insertion***, which inserts a new key/value pair into a given cell in the packed-memory array, (3) ***cell-deletion***, which deletes an existing key/value pair from a cell in the packed-memory array, and (4) ***read***, which returns the key and value of a given cell in the packed-memory array. Each of these

[5] We can trivially pay for any small rebalances with the cost of inserting the new element.

primitives is nonblocking, and may fail when other operations interrupt it. For example, a move operation may fail if a cell-insertion is simultaneously performing an insertion at the target.

Markers

Each cell in the array is augmented with a ***marker*** which indicates whether an ongoing operation is attempting to modify the cell. The marker contains all the information necessary to complete the operation. For example, a move marker indicates the source and the destination of the move. Any processor that is performing an operation and discovers a marked cell helps to complete the operation indicated by the marker. For example, consider a move operation that is attempting to move an element from cell 14 to cell 15, while concurrently a cell-insert is attempting insert an element at cell 15. First, the cell-insert updates the marker at cell 15. Then, the move attempts to update the marker, and discovers the concurrent insertion. The move operation then performs the insertion, before proceeding with the move. In this way, the move can eventually complete, even if the processor performing the cell-insert has failed, or been swapped out of memory.

The primitive operations which modify the data structure (i.e., move, cell-insert, and cell-delete) are all initiated by marking an appropriate cell. Once a cell has been marked, any processor can complete the operation by simply processing the marker. In order to begin a move operation, the source of the move is updated. For a cell-delete operation, the cell containing the element to be deleted is marked. A cell-insert operation marks the cell immediately preceding the cell where the new element is being inserted. A requirement of a cell-insert is that this preceding cell not be empty. By marking the preceding cell, we prevent a concurrent move operation from moving an element "over" an ongoing cell-insertion. For example, if a cell-insert is happening at cell 15, we must prevent the element from cell 14 (or any smaller cell) from being moved to cell 16 (or any larger cell); otherwise, the new element might not be ordered correctly.

I conjecture that it is possible to implement a lock-free packed-memory CO B-tree using compare-and-swap, instead of LL/SC, by adding version tags to the markers.

Implementing the nonblocking primitives

For a move operation, once the source has been successfully marked, an LL operation is performed on the following items: (1) the marker, (2) the source key, (3) the source value, (4) the destination key, and (5) the destination value. Then, an SC is performed on the marker, rewriting the marker. This technique ensures that no concurrent process has modified the marker in an attempt to help complete the move. The move then completes by using SC to update the keys and values at the destination, and then at the source. If an SC fails during this final stage, it is ignored; some concurrent process has already helped to complete the move. Finally, the marker is cleared.

For a cell-insert, once the preceding cell has been marked, the insert performs a LL on the marker, and then on the keys and values at the new cell. If the key is already in the tree, then an error is returned. If the cell is not empty, an error is returned. The insert then rewrites the marker with an SC, ensuring that it has not changed in the interim. Finally, the key and value are updated, and the marker is cleared. A cell-delete is essentially identical to a cell-insert.

Finally, a read operation simply examines a cell, helps out if the cell is marked, and returns the appropriate values.

Insertions and deletions

An insertion proceeds as before, first using the static search tree to find and mark the appropriate cell in the array, that is, the cell containing the largest key in the tree that is less than or equal to the key being inserted. Once the cell is marked, a cell-insert begins. If the insertion succeeds, the element is successfully inserted. If the cell-insertion fails, however, then either the cell is not empty or a move caused interference. In this case, we need to rebalance the array.

A rebalance begins by exploring to the right, as before. In this case, however, it remembers which cells were filled and which were empty. It performs a load-link (LL) operation on each nonempty cell; each of these marked cells may need to be moved, and the move is initiated by performing an SC on the marker. In this way, the rebalance operation can detect when the array has changed during the rebalance.

When an appropriately sparse region is found, the operation can calculate the appropriate spacing of the elements in the array, as before. The rebalance then proceeds from right to left using move operations to spread elements evenly. (Elements are only moved from left to right, as before.) If any move fails, then the rebalance restarts. In particular, a move may fail if any of the markers has changed since they were initially linked during the scanning phase of the rebalance. If at any time during the rebalance, we otherwise detect that the array has changed, then the rebalance restarts.

Deletions are similar to insertions. We first search for the item to be deleted, and then perform a cell-deletion. Finally, we scan to the right until discovering a non-empty cell, and then perform a rebalance. Unlike the rebalance on insertions, however, this operation scans to the left, looking for a dense region (as is described in Section 8.3).

All other data structure operations proceed as before; after the array has been updated using the nonblocking primitives, the search tree is updated to reflect the changes. When resizing the array, load-link/store-conditional is used to atomically write the count of a leaf, instead of a lock.

Correctness

Theorem 8.10 *The packed-memory CO B-tree guarantees linearizable search, insert, and delete operations.*

Proof. If a key is in the data structure and the data structure remains in sorted order, then a search will find it: the static tree is updated after the packed-memory array, and elements are only moved to the right in the packed-memory array. Thus, a search always exits the static tree to the left of the key in question.

The key property, then, is that the elements in the array remain in sorted order. If they always remain in sorted order, then a search correctly finds any previously inserted element or any element returned by a prior search and it fails to find a previously deleted element, as is required.

In order to show that elements remain in order, we examine how elements are inserted. An important invariant is that if a cell is marked for a cell-insert, then the cell contains the largest key that is less than or equal to the key being inserted. Initially, on acquiring a marker, this invariant is ensured by the correctness of the search which locates the insertion cell, as well as the use of LL/SC to acquire the marker atomically.

Throughout the insertion, this invariant is maintained by the way in which a rebalance moves elements. In particular, a rebalance never moves an element from one side of a marked cell to another. In particular, whenever a rebalance moves an element, all the intervening cells are empty. This property is checked while the rebalance scans to the right, determining the region to rebalance and simultaneously performing a load-link (LL) on each non-empty cell in the region. The only way this property of a rebalance can be violated is if an element is inserted or moved between when the

region is scanned and the rebalance moves an element. In this case, however, the SC which acquires the move marker fails.

As a result, elements are always inserted in order, and rebalances never move elements out of order. Combined with the fact that searches terminate correctly, we have the main result. □

Theorem 8.11 *The packed-memory CO B-tree is a lock-free data structure.*

Proof. We need to show that if there is at least one ongoing operation, then eventually some operation completes. An operation can only be delayed by pausing to help a concurrent operation, or by working on a rebalance. If an operation is delayed by helping an insertion or deletion to complete, then some operation has completed, as desired.

Therefore the key requirement is to show that rebalances cannot prevent operations from making progress. If a rebalance is forced to restart because of a successful insertion or deletion, then some other operation has completed.

A rebalance may also restart because of interference by a concurrent rebalance. In this case, however, since elements are moved only to the right, when the rebalance restarts it has (strictly) less work to do than in the aborted rebalance. In particular, the concurrent rebalance that forced a restart must have moved at least one of the items in the region to the right, thus reducing the amount of rabalancing necessary. Since every time a rebalance restarts it has less work to do, eventually the rebalance completes. □

8.5 Concurrent CO B-tree discussion

This chapter has explored a range of issues for making cache-oblivious search structures concurrent. I consider both locking and lock-free solutions. Each of these approaches has practical and theoretical merits, and I make no judgement about which approach is best. A third approach, which I do not consider here, is to use transactional memory to support concurrency. This approach may lead to simpler coding and several new data-locality issues.

Like many previous concurrency studies, the analyses consider uniformly random insertions. These analyses do not reveal all the design principles that went into the data structures. In particular, in the exponential trees, the parameter α tunes the tradeoff between low-concurrency and high-concurrency performance of the tree. When α is larger, serial operations in the tree are faster; when α is smaller, the tree supports increased concurrency and reduced contention. Moreover, the randomized nature of the tree reduces contention at high levels in the tree, by temporally spacing out updates, even when insertions are adversarial.

As a whole, this chapter has focused on correctness issues more than performance issues. In particular, in order to get performance guarantees, there are different insertion patterns (such as adversarial), different process models (such as different speeds and changing speeds), and different models of memory (such as queuing on memory locations for both reads and writes). Some techniques from the design of overlay networks may to carry over to these models.

A natural extension of the one-way packed-memory structure from Section 8.3 is to implement the structure as circular array. Using the circular array may lead to less memory allocation. While a circular implementation is straightforward for the serial and locking cases, it is more complex in the lock-free setting.

Chapter 9

The Lookahead Array: A Cache-Oblivious Dictionary With Faster Insertions

This chapter, along with Chapter 10, explores the insert/search tradeoff for cache-oblivious dictionaries, shown in Figure 7-1. This chapter presents the ***lookahead array*** and, more significantly, the ***cache-oblivious lookahead array (COLA)***. The COLA achieves the optimal bound for $\varepsilon = 0$ (matching the BRT), with searches and insertions costing $O(\log(N/M))$ and $O((\log(N/M))/B)$, respectively. This chapter represents joint work with Michael A. Bender, Martin Farach-Colton, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson, previously appearing in [42, 161]. Chapter 10 presents a more complicated data structure that achieves the more general tradeoff for any $\varepsilon > 0$.

For disk-based storage systems, range queries are likely to be faster for a lookahead array than for a BRT because the data is stored contiguously in arrays, taking advantage of inter-block locality, rather than stored scattered on blocks across disk. This is the same reason why the cache-oblivious dictionary can support range queries nearly an order of magnitude faster than a traditional B-tree; see, e.g., [44].

The lookahead array is reminiscent of static-to-dynamic transformations [49] and fractional cascading [64]. This data structure is parametrized by a "growth factor." If the growth factor is chosen to be B^{ε}, then the lookahead array is cache aware and achieves the same amortized bounds as the B^{ε}-tree. If the growth factor is chosen to be a constant such as 2, then the lookahead array is cache-oblivious and matches the performance of the BRT. This version is called the ***cache-oblivious lookahead array (COLA)***. Unlike the BRT, the COLA is amortized, and any given insertion may trigger a rearrangement of the entire data structure.

I also show how to deamortize the lookahead array and (hence the COLA) when $M = \Omega(\log N)$. This deamortized COLA is the first cache-oblivious alternative to the BRT. There is no amortization on searches and the worst-case cost for an insert is no more than the cost of a search.

I next present experimental results, demonstrating how fast the COLA runs in comparison with a B-tree. I use the B-tree whose performance was described in [44]. For out-of-core data, the COLA was 790 times faster than the B-tree for random inserts, 3.1 times slower for sorted inserts, and 3.5 times slower for searches.

The remainder of this chapter is organizes as follows. Section 9.1 presents the lookahead array, the COLA, and their partial deamortizations. Section 9.2 gives results from our implementation

study of lookahead arrays.

9.1 Cache-oblivious lookahead array (COLA)

This section describes the lookahead array and the ***cache-oblivious lookahead array (COLA)***. I begin by describing the structure of the COLA with an amortized analysis. Then, I show how to deamortize the COLA to provide better worst-case bounds. I present the deamortization incrementally, showing first the basic idea, and then dealing with the full complexity of the data structure. I conclude this section with a discussion of the tradeoff between updates and queries.

Lookahead array description

The lookahead array is similar to the binomial list structure [49] of Bentley and Saxe, parameterized by integer ***growth factor*** g. It consists of $\lceil \log_g N \rceil$ arrays, or ***levels***. The kth array ($k \geq 1$) is of size g^k and the arrays are stored contiguously in memory. The kth array always contains some multiple of g^{k-1} elements (or is empty). The lookahead maintains the following invariants:

1. The kth array contains $n_k g^k$ elements if and only if the kth least significant bit of the base-g representation of N is n_k.
2. Each array contains its items in ascending order by key.

When a new item is inserted, we effectively perform the carries performed by adding a 1 to the base-g representation of N. That is to say, first put the new element into level 1, the size-g array. As the elements in each array are stored in sorted order, inserting an element into the first level requires a scan of the size-g array to make space for the new element. Whenever the kth array becomes completely full (i.e., adding a 1 to the corresponding bit would cause a carry), remove all g^k elements from the kth array and merge them into the $(k+1)$th array. This merge can be achieved with a single scan of both the kth and $(k+1)$th arrays by merging from largest key down to smallest key.

Thus far, I have not described searches. I will revisit those shortly.

The ***cache-oblivious lookahead array (COLA)*** is a lookahead array for which the growth factor is chosen to be a constant, like 2 or 4. In contrast, a general (i.e., cache-aware) lookahead array may choose a growth factor of B^{ε} to achieve the same bounds as a B$^{\varepsilon}$-tree. As the main goal here is to achieve a cache-oblivious structure, proofs of the general bounds are omitted.

For the remainder of Section 9.1, consider a COLA with growth factor 2 to simplify the discussion. All of the results generalize to other growth factors.

Lemma 9.1 *Insertion into the COLA incurs an amortized $O((\log(N/M)/B)$ block transfers.*

Proof. Once an item has been involved in $O(\log N)$ merges, it is in the last array. The smallest $O(\log_2 M)$ merges incur no cost because these arrays fit into a constant number of blocks, which we can assume are always kept in memory. For the larger merges, the arrays being merged have length at least B. Thus, the cost of merging two such arrays of length k is $O(k/B)$. The amortized cost of one merge is $O(1/B)$ per item, and so the total amortized merge cost is $O((\log(N/M)/B)$ block transfers. □

This proof assumes that portions of the data structure remain in memory between operations, and that we have already paid for loading certain portions of the data structure. If the COLA is not in memory, then the insert cost increases to $O((\log N)/B + 1)$.

Naïvely, searches can be implemented by binary searching separately in each of the $O(\log N)$ levels for a total complexity of $O(\log^2 N)$. We call this first data structure the ***basic COLA***.

Searches can be accelerated by fractional cascading [64]. Specifically, every eighth element in the $(k+1)$st array also appears in the kth array, with a ***real lookahead pointer*** to its location in the $(k+1)$st array. Each fourth cell in the kth array is reserved for a ***duplicate lookahead pointer***, which holds pointers to the nearest real lookahead pointer to its left and right. Thus, every level uses half its space for actual items and half for lookahead pointers, so we accommodate the additional lookahead overhead by doubling the size of all arrays.

Maintaining lookahead pointers does affects only the constant factors in the insertion bounds. In particular, all real lookahead pointers are added to the kth level whenever elements are moved from the kth level to the $(k+1)$th level. At this time, the kth level becomes empty, so adding lookahead pointers is easy and requires only a scan. Duplicate pointers are also easily maintained as elements move around the kth level. A simple implementation of a merge into the kth level would be to perform the normal insertion process first, except also leaving a vacancy for duplicate lookahead pointers. Then scan the kth level again from left to right, remembering the last lookahead pointer scanned, and writing this last pointer to each duplicate-pointer cell passed.

To search for element with key κ in the COLA, simply scan the smallest level for the element, stopping when the scan reaches an element larger than κ. If κ is not there, find the largest lookahead pointer smaller than κ. Commence the search in the next level from this lookahead pointer, again scanning from the target of the lookahead pointer until finding κ or a key larger than κ. If the key is not found, again follow the largest lookahead pointer not exceeding κ, and repeat in the next level. Note that because duplicate lookahead pointers are located in every fourth array cell, finding the largest lookahead pointer not exceeding κ entails considering at most 4 contiguous array locations and 1 additional location (if a duplicate pointer is followed).

Lemma 9.2 *COLA searches incur $O(\log(N/M))$ block transfers.*

Proof. To simplify the proof, on each level store $-\infty$ and $+\infty$ in the first and last cell and give them real lookahead pointers.

We prove inductively that a search for key κ looks at at most 9 items in each level (most of which are contiguous) and that κ is greater than or equal to the smallest of these and less than or equal to the largest of these. This induction will establish both the time bound and correctness of the search procedure.

The claim is true in the first level, because it has size at most 4. Suppose the claim is true at the kth level, where $k \geq 1$, and the search in this level looked at contiguous items with keys $\kappa_1 < \kappa_2 < \ldots < \kappa_8$ and that $\kappa_1 \leq \kappa \leq r_\kappa$.

Let ℓ be such that $\kappa_\ell \leq \kappa < \kappa_{\ell+1}$. If $\kappa_\ell = \kappa$ then we have found the target element or lookahead pointers that leads to the target element. In this case the induction goes through trivially for all remaining levels. Otherwise, $\kappa_\ell < \kappa$. In this case we restrict our search on the next level to those keys between the elements pointed to by two lookahead pointers, the two lookahead pointers whose key values are the maximum below κ and the minimum above κ. By construction, this range spans at most 8 elements in the next array. Moreover, this range spans a duplicate lookahead pointer, so finding the appropriate lookahead pointer to the next level requires only 1 additional array lookup, making 9 total.

Since we may assume that the $\Omega(\log_2 M)$ smallest levels remain in memory, the total number of levels is $O(\log_2(N/M))$ to complete the proof. □

Lookahead pointers can also be used to achieve $O(\log(N/M))$ block transfers for predecessor and successor queries and $O(\log(N/M) + L/B)$ block transfers for range queries, where L is the

number of items reported.

Lemma 9.3 *Suppose that $M > \log N$. Then a COLA can report, in sorted order, all items having keys between κ_1 and κ_2, $\kappa_1 \leq \kappa_2$, in $O(\log(N/M) + L/B)$ block transfers, where L is the number of items reported.*

Proof. The search procedure can easily be modified to return the predecessor of κ_1 in each array. (Do not terminate the search when discovering the element with key κ_1, and return the predecessor within each array.) Since elements are stored in sorted order within each array, we thus need only perform a $\log N$-way merge to consolidate the elements into a sorted list. If $M > \log N$, then a block from each level as well as an output block can be kept in memory at the same time, and this merge can be performed in $O(\log N + L/B)$ block transfers. The L/B term pays for all the full blocks scanned, and the $\log N$ term pays for partial scans of at most 2 blocks in each level. □

If $M \leq \log N$, but the elements are not required to be output in sorted order, then we can still output all the elements within the range in $O(\log(N/M) + L/B)$ block transfers.

Deamortization of basic COLA

As a first step, I show how to partially deamortize the basic COLA (which includes no lookahead pointers) when $M = \Omega(\log N)$. This deamortization improves the worst-case bound from $O(N/B)$ to $O(\log(N/M))$. I ignore the issue of deamortizing global rebuilding after the data structure doubles in size because this rebuilding can be handled using standard methods [165, Ch. 5].

The idea is as follows. In level k of the lookahead array maintain two arrays each of size 2^k. Whenever a level contains items in both of its arrays, begin merging those two arrays into an empty array in the next level. Each level is said to be either ***safe*** or ***unsafe***. Informally, a level is unsafe during merging. More formally, initially all levels are safe, level k become unsafe once it contains exactly 2^{k+1} items, and an unsafe level becomes safe when both of its arrays become empty.

Place newly inserted items into level 1, as before. Then, scan the levels from left to right, merging from unsafe levels to the next level, stopping when we have either run out of unsafe levels or moved a total of m items, whichever comes first (m to be chosen later). To facilitate these partial merges, maintain three arrays of size $\lceil \log_2 N \rceil$. One array keeps track of which levels are safe. The other two arrays' kth entries hold the indices we are at in the merge of the two arrays at level k.

The bigger m, the more aggressive the merge. We need to make sure merges are aggressive enough to guarantee that whenever merging from a level there is at least one free array in the next level, which would be implied by the next level being safe. The insertion worst-case bound is clearly $O(m)$, so we want m as small as possible while still keeping us aggressive enough. Lemma 9.4 shows that a fairly low setting of m suffices.

Lemma 9.4 *If a lookahead array contains k levels, setting $m = 2k+2$ guarantees that two adjacent levels are never simultaneously unsafe.*

Proof. By induction on levels we show that levels i and $i+1$ are never simultaneously unsafe. For $i = 0$ there are at least 2 levels, so $m \geq 6$ and we can afford to perform an entire merge from level 1 to 2 whenever level 1 becomes unsafe. Thus level 1 never remains unsafe, showing the base case.

We show that if level ℓ is unsafe, it becomes safe before level $\ell - 1$ becomes unsafe. When level ℓ becomes unsafe, level $\ell - 1$ is empty. Furthermore, for us to have moved items from level $\ell - 1$ to level ℓ, all previous levels had to have been safe since we process unsafe levels from left to right.

Therefore, levels 0 to $\ell - 2$ hold a total of at most $\sum_{j=0}^{\ell-2} 2^j = 2^{\ell-1} - 1$ items, and for level $\ell - 1$ to become unsafe again there must be at least $2^{\ell-1} + 1$ inserts. We show that level ℓ becomes safe after at most $2^{\ell-1}$ inserts. After $2^{\ell-1}$ inserts, we have accumulated at least $m\ell 2^{\ell-1}$ moves, and no move will go unused as long as level ℓ is unsafe.

After $2^{\ell-1}$ insertions there can be at most $2^{\ell} - 1$ items in levels $\ell - 1$ and below, and each one could have used at most $\ell - 2$ moves to get to its level. Thus, items below level ℓ could have used a total of at most $(\ell - 2)(2^{\ell} - 1)$ moves. We want at least $2^{\ell+1}$ moves available to move all the items from level ℓ to the next level, so we want $m2^{\ell-1} \geq (\ell - 2)(2^{\ell} - 1) + 2^{\ell+1}$, which holds for $m \geq 2\ell$. □

Theorem 9.5 *The basic COLA can be deamortized to perform $O(\log N)$ block transfers per insertion in the worst case and $O((\log(N/M)/B)$ amortized block transfers per insertion as long as $M = \Omega(\log N)$.*

Proof. The worst-case bound follows immediately from Lemma 9.4 since $k = O(\log N)$.

For the amortized bound, the argument is similar to that for the amortized COLA. The difference is that now when we incur a block transfer at a level beyond the $(\log_2 M)$th level, that block may be evicted due to previous levels becoming unsafe before we get a chance to move B items. We charge such an eviction to the block that is brought in. Since unsafe levels are processed from left to right, there can be no cycles in blocks charging one another for evictions. Furthermore, if a block C evicts a block D from a later level then is evicted itself before completing the merge it was brought in for, when C is later brought back into memory it must be because a block at an earlier level finished participation in its merge; C can take that block's space in memory. Thus, a block brought into memory during a merge is only charged for evicting at most one other block. This charging guarantees that the amortized cost of moving one item is still $1/B$.

Furthermore, since $M = \Omega(\log N)$, we can assume the three arrays keeping track of merge information are always in memory, so the insertion algorithm can determine which merges need to take place without incurring extra transfers. □

Deamortization of COLA

The introduction of lookahead pointers complicates COLA deamortization as we must maintain the pointers in the middle of merges to keep queries fast. Since the deamortized structure is only allowed to perform a small amount of work per update, it is not clear how the lookahead pointers can be updated. Here, I show how to extend the deamortization technique for the basic COLA to the COLA in such a way as to completely hide the details of the deamortization from the queries; from the viewpoint of a query, no level will appear to be in the middle of a merge. The precise description and proofs follow.

At level k, now maintain three arrays, each of size 2^k. At least one array is a ***shadow array*** and at least one is a ***visible array*** (unless level k is beyond the levels being used by the data structure yet, in which case all three arrays are considered shadow). The search algorithm ignore the shadow arrays. This labeling of shadow versus visible is not static throughout time; during updates a shadow array may become visible, and vice versa. Items in visible arrays never appear to be in the middle of a merge from a search's viewpoint, so the data structure induced by only looking at visible arrays appears almost exactly as the amortized COLA with lookahead pointers. The key difference from the deamortization method of the previous section is that when level k becomes unsafe, we do not move items from level k to $k + 1$; we instead *copy* them to a shadow array of level $k + 1$. Thus, from the point of view of a query, no arrays appear to be in the middle of a merge.

All levels start as being safe and all arrays start as being shadow arrays. Level k becomes unsafe once two of its arrays become full. Two full arrays at an unsafe level must be merged into any shadow array A of level $k+1$. If there is more than one shadow array in level $k+1$, preference is given to one already containing lookahead pointers. After the merge, lookahead pointers are copied from A into an empty shadow array at level k. We restrict ourselves to having a total combined number of copied lookahead pointers and merged items to being at most m during each insertion, m to be chosen later. Level k becomes safe once both the merge into A and the placement of lookahead pointers from A into level k are complete. At this point we say the array in level k that received lookahead pointers from A becomes ***linked*** to A.

I now discuss the transition from shadow to visible and vice versa. Level 0 only contains two arrays, both of which are always visible. A shadow array becomes visible when there is a sequence of linked arrays beginning at level 0 to that array. When a shadow array becomes visible there are two cases. If there are two other visible arrays in that level, their statuses are both changed to shadow and both arrays are then considered empty. Otherwise, if there are 0 or 1 other visible arrays, they remain visible.

Lemma 9.6 *Suppose level k becomes unsafe and merges into a shadow array A at level $k+1$. Then, A becomes visible before another merge into level $k+1$ occurs.*

Proof. Prove the claim by induction on level. If we merge from level 0 to an array A at level 1, the array A becomes immediately visible since an array at level 0 then becomes linked to A. Now, suppose we just merged two arrays from level k into array A at level $k+1$. At the end of the merge, level k has two arrays with items (the arrays that engaged in the merge) and one shadow array B that is linked to A. Since there are never two adjacent unsafe levels, B has received all its lookahead pointers from A before a merge into level k can occur. When a merge into level k does occur, it is into B, and the other two arrays at level k then become shadow. For another merge into level $k+1$ to occur, there must be another merge into level k after the merge into B in order to make level k unsafe. By our induction hypothesis, B becomes visible by this time, and thus so does A since B is linked to A. □

Theorem 9.7 *The COLA with lookahead pointers can be deamortized to perform $O(\log N)$ block transfers per insertion in the worst case and $O((\log(N/M))/B)$ amortized block transfers per insertion as long as $M = \Omega(\log N)$.*

Proof. Safeness of a level is similar to safeness in the basic COLA. In both scenarios, if level k is unsafe then we must scan $\Theta(2^k)$ items before level k becomes safe again (note that placing lookahead pointers in the previous level after a merge can be done by two simultaneous scans). Thus, we can use the proof idea of Lemma 9.4 to state that two adjacent levels are never unsafe if we choose to move $\Theta(\log N)$ items per insertion. By Lemma 9.6, three arrays per level suffice to guarantee that for any unsafe level there is at least one shadow array in the next level to merge into. The rest of the argument follows that of the proof of Theorem 9.5. □

Now let us discuss how to slightly modify the query algorithms to work when the COLA has two arrays at each level. As with the basic COLA deamortization, each level k has at most two arrays in use at any given time. One of these arrays is the ***main array***. Should there only be one array in use at level k, we consider that array to be the main array. Otherwise, the array merged into least recently is the main array and the other is ***secondary***.

The main array contains lookahead pointers into both the main and secondary arrays of the next level, and the secondary array, if it exists, contains no lookahead pointers. We only use half the

space of the secondary array to maintain that every array is exactly half-full with real items. When merging into what will become the main array of level $k + 1$, every eighth element in this main array appears as a lookahead pointer in an array of level k. When merging into what will become the secondary array of level $k+1$, every sixteenth element of each of the main and secondary arrays of level $k+1$ appears as a lookahead pointer in an array at level k. Thus, main arrays always contain exactly half real items and half lookahead pointers. Duplicate lookahead pointers at level k point to the nearest real lookahead pointers to their left and right in each of the main and secondary arrays of level $k + 1$. Queries then search both arrays at a level in parallel.

Cache-aware update/query tradeoff

With a few changes, it is possible to make the lookahead array cache-aware and achieve $O(\log_{B^\varepsilon+1}(N/M))$ block transfers per query and $O((\log_{B^\varepsilon+1} N)/B^{1-\varepsilon})$ block transfers per insertion for any $\varepsilon \in [0, 1]$, matching the bound of the B^ε-tree [59]. Instead of using a growth factor of 2, set a growth factor of $g = \Theta(B^\varepsilon)$. Here, instead of every eighth element of level k also appearing in level $k-1$ as lookahead pointers, every $\Theta(B^\varepsilon)$th element will appear as a lookahead pointer in the previous level. During queries we must look through $\Theta(B^\varepsilon)$ instead of $\Theta(1)$ cells of each array, but $\Theta(B^\varepsilon)$ cells still fit in at most 2 blocks implying a constant number of block transfers per level. When performing an insertion, the level being merged into may not be empty and we thus have to merge the pre-existing items with the ones from the previous level. Since the sum of the sizes of the first $k - 1$ levels is at least an $\Omega(1/B^\varepsilon)$ fraction of the size of the kth level, a level is merged into at most B^ε times before its items participate in a merge into a future level. This fact, together with there being at most $O(\log_{B^\varepsilon+1} N)$ levels, gives the amortized $O((\log_{B^\varepsilon+1}(N/M))/B^{1-\varepsilon})$ insertion bound.

The deamortization technique of the last section can be adjusted to reduce the worst-case insertion complexity of this cache-aware lookahead array to $O(\log_{B^\varepsilon+1} N)$. The main differences are that we need to move $\Theta(B^\varepsilon \log_{B^\varepsilon+1} N)$ items per insertion instead of $\Theta(\log N)$ items, and that we need an extra array per level to provide space for holding the result of merges into non-empty arrays.

9.2 COLA: experimental results

This section presents an experimental evaluation of the COLA. We compared COLAs with B-trees as well as the impact of insertion order on performance. First I describe our implementation, then the experiments, and present our results showing that the COLA can be orders of magnitude faster than traditional B-trees without sacrificing much performance on searches.

Implementation

Here I describe the implementation details of the amortized COLA. As discussed in Section 9.1, the growth factor is one parameter that can be varied for COLAs. In addition, one can also vary the density of lookahead pointers. There is some tradeoff between space used and performance, which I attempt to explore to some degree here.

One further optimization in our implementation is that we reduce the size of each array slightly. That is to say, my description in Section 9.1 suggests that the level-k array have size g^k (plus the space for lookahead pointers). Whenever the array reaches capacity, however, we immediately move its elements into the next level. Thus, we really only require $(g - 1)g^{k-1}$ space for the kth array.

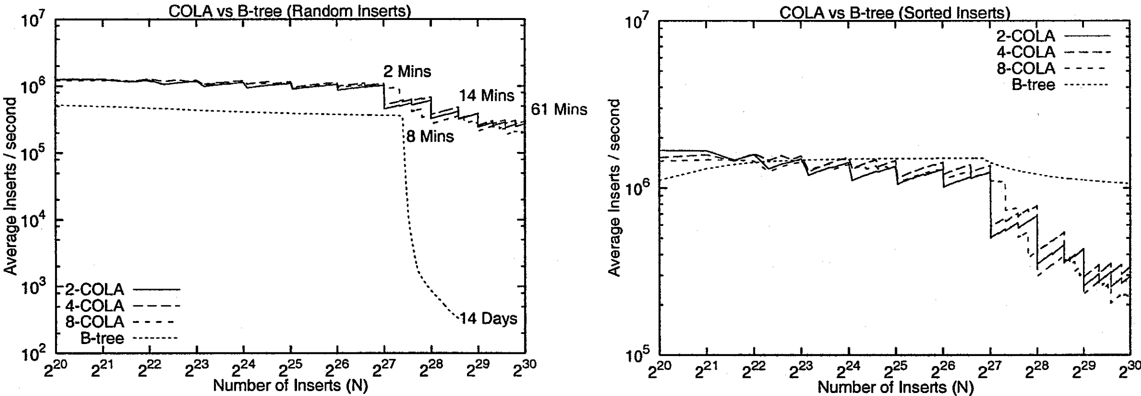


Figure 9-1: Data is inserted in random order. The 4-COLA is 790 times faster than the B-tree for $N = (256 \times 2^{20}) - 1$ (the largest N tested). Structures no longer fit in main memory when $N \approx 2^{27}$

Figure 9-2: Data is inserted in sorted order, which gives best-case performance for the B-tree. The 4-COLA is 3.1 times slower than the B-tree for $N = 2^{30} - 1$.

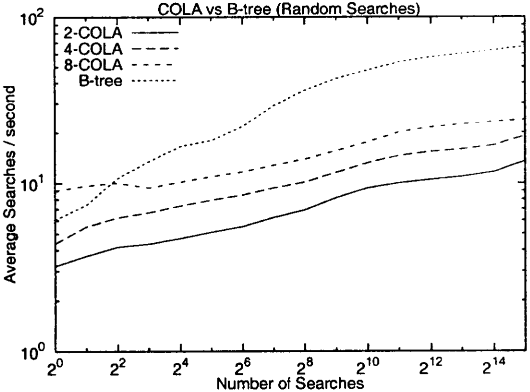
The ***pointer density*** controls the number of lookahead pointers in each level. For pointer density p, the kth level includes an additional $\lfloor 2p(g-1)g^{\ell-1} \rfloor$ array cells to accommodate real lookahead pointers. In Section 9.1, the pointer density was $p = 1/8$. For our experiments, we reduced the pointer density to $p = 0.1$ to reduce the space overhead of lookahead pointers.

When a level is not completely full (e.g., when it contains only redundant elements or has received fewer than $g - 1$ merges) we maintain the elements right justified in their array. Elements comprise key/value pairs, where keys and values each are of size 64 bits. We pad the elements to a total size of 32 bytes. Instead using of duplicate lookahead pointers, each real element uses 64 of its padding bits to hold a copy of the closest real lookahead pointer to its left. Redundant elements use 64 of their padding bits to hold the real lookahead pointer.

When performing merges, we merge the 2 smallest levels at a time. We alternate placing the result of the merge at the beginning of the target level and at the newly freed space at the beginning of the data structure, thus requiring space for only 1 additional element during merges. This merge pattern requires $O(k)$ CPU time and $O(k/B)$ memory transfers to merge a total of k items across any number of levels. When distributing lookahead pointers after a merge we proceed level by level. The target level is scanned to copy pointers down one level, the next largest level is scanned to copy pointers down to the next level, and so on. We perform searches as in Section 9.1, except that we compute right-hand lookahead pointers on the fly by scanning subsequent levels.

Our B-tree implementation employs blocks of size 4KiB. Key and value sizes were each 64 bits to match our COLA implementation. Both data structures access external memory via memory mapping. As a sanity check, we compared the performance of our traditional B-Tree to the Berkeley DB [182], a high-quality commercially available B-tree. Berkeley DB supports variable-sized keys, crash recovery, and very large databases, none of which our implementation supports. The Berkeley DB with the default buffer-pool allocation is much slower than our implementation, but is comparable once the parameters are tuned, and logging is turned off, suggesting that we did a reasonable job implementing our B-tree.

All experiments were performed on a dual Xeon 3.2GHz machine with 2MiB of L2 Cache, 4GiB RAM and two 250GB Maxtor $7L250S0$ SATA drives using software RAID-0 with 64KiB stripe width, running Linux 2.6.12-10-amd64-xeon in 64-bit mode.



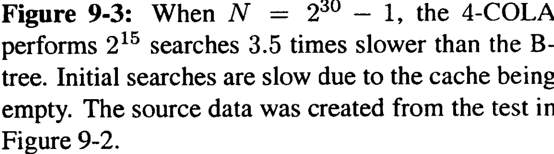


Figure 9-3: When $N = 2^{30} - 1$, the 4-COLA performs 2^{15} searches 3.5 times slower than the B-tree. Initial searches are slow due to the cache being empty. The source data was created from the test in Figure 9-2.

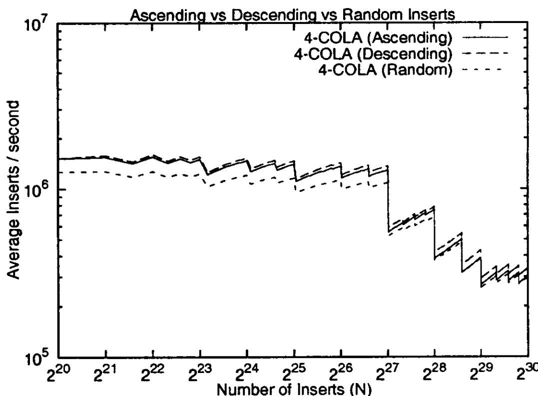


Figure 9-4: We measured the time to insert into COLAs for ascending, descending, and random keys. Inserting $2^{30} - 1$ keys sorted in descending order is 1.1 times faster than inserting in ascending order, and 1.1 times faster than inserting in random order.

Experiments

When performing the experiments we measured the time once every 2^{20} inserts. We measured user CPU time u, kernel CPU time k, and elapsed time since start of test w. We estimated disk time d as $d = w - u - k$. The RAID array contains only the memory-mapped file. Before measuring any times, we first created a large enough file to hold the entire experiment. We measured raw disk bandwidth of 120MiB/s by timing the write of 2^{37} bytes to the RAID array. We remounted the RAID array's file system before every insertion test to clear the file cache.

In all figures resulting from experiments, insert performance of the COLA appears to be periodic when plotted on a logarithmic scale because there is a merge of 2^k elements after every 2^k inserts.

We compared the B-tree to the COLA as follows. For $N = 2^{30} - 1$ we inserted keys $[N - 1, \ldots, 0]$ into a B-tree. We next attempted to insert $(256 \times 2^{20}) - 1$ random elements into a new B-tree, stopping after 87 hours at about 2^{28} insertions. We repeated the tests for a 2-, 4-, and 8-COLA, and completed the random insertion test of N elements. We were able to complete the insertions for the COLAs. After remounting the RAID array, we next searched in the g-COLAs and B-tree for 2^{15} random elements, measuring the time after search number 2^x, $0 \leq x \leq 15$. (To construct the complete B-tree we first sorted the N random elements then inserted them, since directly inserting the elements would have taken too long.)

As shown in Figure 9-1, the 4-COLA is 790 times faster than the B-tree for random inserts. Figure 9-2 shows that the 4-cola is 3.1 times slower for insertions where the insertions are performed in descending, rather than random, order. Figure 9-3 shows that the 4-cola is 3.5 times slower for searches.

We believe that the B-tree performs faster for sorted data because it only uses the leftmost root-to-leaf path, which can stay in memory. For random inserts, the B-tree loses the advantage of keeping its insertion path in memory. For random inserts, sequential prefetching of more than one block does not significantly help B-trees, but significantly helps COLAs. The 4-COLA is 1.1 times faster than the 2-COLA for random inserts, 1.1 times faster for sorted inserts, and 1.4 times faster for searches. The 4-COLA is 1.4 times faster than the 8-COLA for random inserts, 1.5 times faster for sorted inserts, and 1.2 times slower for searches. Given the superior tradeoff of the 4-COLAs, we use them for all subsequent experiments.

We next performed an experiment to measure COLA performance for different insertion patterns. For $N = 2^{30} - 1$ we inserted $[N-1, \ldots, 0]$ into a COLA, $[0, \ldots, N-1]$ into a second COLA, and N random elements into a third COLA. Figure 9-4 shows that inserting keys sorted in descending order is 1.1 times faster than inserting keys sorted in ascending order, and 1.1 times faster than inserting random keys. Inserting keys sorted in ascending order was 1.02 times faster than inserting random keys.

We believe inserting keys in descending order is faster due to the final merge. The number of elements we merge at each level grows geometrically. The last merge, into the target level, is the largest. When inserting in descending order, elements already in the target level do not move, whereas when inserting in ascending order, all elements in the target level move to make room for elements from smaller levels.

Chapter 10

The xDict: A CO B-Tree With Optimal Update/Query Tradeoff

This chapter continues the thread from Chapter 9, exploring the tradeoff between queries and updates in *cache-oblivious* dictionaries. In particular, this section presents a new cache-oblivious dictionary, called the ***xDict***, the implements searches in $O((1/\varepsilon)\log_B(N/M))$ worst-case memory transfers and insertions and deletions in $O\left((1/\varepsilon B^{1-\varepsilon})\log_B(N/M)\right)$ amortized memory transfers, for any constant ε with $0 < \varepsilon < 1$. For example, the xDict achieves subconstant amortized update cost when $N = M\,B^{o(B^{1-\varepsilon})}$, whereas the B-tree's $\Theta(\log_B(N/M))$ is always superconstant, and the previously best $\Theta\left((1/B^{\Theta(1/(\log\log B)^2)})\log_B N + (1/B)\log^2 N\right)$ [42][1] is subconstant only when $N = o(2^{\sqrt{B}})$. The xDict attains a provably optimal tradeoff between insertions and queries in the cache-oblivious model.

The xDict represents joint work with Gerth Stølting Brodal, Erik D. Demaine, John Iacono, Stefan Langerman, and J. Ian Munro.

10.1 Introducing the x-box

The xDict dynamic-dictionary data structure is built in terms of another structure called the x-box. For any positive integer x, an ***x-box*** supports a batched version of the dynamic-dictionary problem (defined precisely later) in which elements are inserted in batches of $\Theta(x)$. Each x-box will be defined recursively in terms of y-boxes for $y < x$, and later we build the overall xDict data structure in terms of x-boxes for x increasing doubly exponentially. Every x-box uses a global parameter, a real number $\alpha > 0$, affecting the insertion cost, with lower values of α yielding cheaper insertions. This parameter is chosen globally and remains fixed throughout.

As shown in Figure 10-1, an x-box is composed of three buffers (arrays) and many $\sqrt{x}$-boxes, called ***subboxes***. The three buffers of an x-box are the ***input buffer*** of size x, the ***middle buffer*** of size $x^{1+\alpha/2}$, and the ***output buffer*** of size $x^{1+\alpha}$. The $\sqrt{x}$-subboxes of an x-box are divided into two levels: the ***upper level*** consists of at most $(1/4)x^{1/2}$ subboxes, and the ***lower level*** consists of at most $(1/4)x^{1/2+\alpha/2}$ subboxes. Thus, in total, there are fewer than $(1/2)x^{1/2+\alpha/2}$ subboxes. See Table 10-2 for a table of buffer counts and sizes.

Logically, the upper-level subboxes are children of the input buffer and parents of the middle buffer. Similarly, the lower-level subboxes are children of the middle buffer and parents of the output buffer. However, the buffers and subboxes do not necessarily form a tree structure. Specifically, for

[1] This data structure, the shuttle tree, is also my work but is omitted from this thesis.

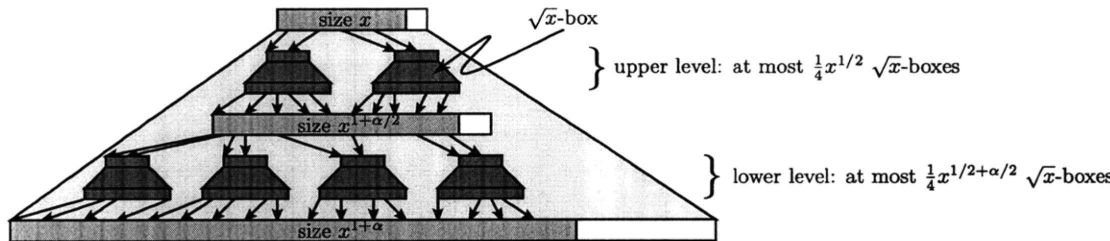


Figure 10-1: The recursive structure of an x-box. The arrows represent lookahead pointers. These lookahead pointers are evenly distributed in the target buffer, but not necessarily in the source buffer. Additional pointers (not shown) allow us to find the nearest lookahead pointer(s) in $O(1)$ memory transfers. The white space at the right of the buffers indicates empty space, allowing for future insertions.

an x-box D, there are pointers from D's input buffer to the input buffers of its upper-level subboxes. Moreover, there may be many pointers from D's input buffers to a single subbox. There are also pointers from the output buffers of the upper-level subboxes to D's middle buffer. Again, there may be many pointers originating from a single subbox. Similarly, there are pointers from D's middle buffer to its lower-level subboxes' input buffers, and from the lower-level subboxes' output buffers to D's output buffers.

The number of subboxes in each level has been chosen to match the buffer sizes. Specifically, the total size of the input buffers of all subboxes in the upper level is at most $(1/4)x^{1/2} \cdot x^{1/2} = (1/4)x$, which is a constant factor of the size of the x-box's input buffer. Similarly, the total size of the upper-level subboxes' output buffers is at most $(1/4)x^{1/2} \cdot (x^{1/2})^{1+\alpha} = (1/4)x^{1+\alpha/2}$, which matches the size of the x-box's middle buffer. The total size of the lower-level subboxes' input and output buffers are at most $(1/4)x^{1/2+\alpha/2} \cdot x^{1/2} = (1/4)x^{1+\alpha/2}$ and $(1/4)x^{1/2+\alpha/2} \cdot (x^{1/2})^{1+\alpha} = (1/4)x^{1+\alpha}$, which match the sizes of the x-box's middle and output buffers, respectively, to within a constant factor.

An x-box D organizes elements as follows. Suppose that the keys of elements contained in D range from $[\kappa_{\min}, \kappa_{\max})$. The elements located in the input buffer occur in sorted order, as do the elements located in the middle buffer and the elements located in the output buffer. All three of these buffers may contain elements having any keys between $\kappa_{\min}$ and $\kappa_{\max}$. The upper-level subboxes, however, partition the key space. More precisely, suppose that there are r upper-level subboxes. Then there exist keys $\kappa_{\min} = \kappa_0 < \kappa_1 < \cdots < \kappa_r = \kappa_{\max}$ such that each subbox contains elements in a distinct range $[\kappa_i, \kappa_{i+1})$. Similarly, the lower-level subboxes partition the key space.

Buffer	Size per buffer	Number of buffers	Total size
Top buffer	x	1	x
Top buffer	$x^{1/2}$	$\frac{1}{4}x^{1/2}$	$\frac{1}{4}x$
Middle buffer	$(x^{1/2})^{1+\alpha/2}$	$\frac{1}{4}x^{1/2}$	$\frac{1}{4}x^{1+\alpha/4}$
Bottom buffer	$(x^{1/2})^{1+\alpha}$	$\frac{1}{4}x^{1/2}$	$\frac{1}{4}x^{1+\alpha/2}$
Middle buffer	$x^{1+\alpha/2}$	1	$x^{1+\alpha/2}$
Top buffer	$x^{1/2}$	$\frac{1}{4}x^{1/2+\alpha/2}$	$\frac{1}{4}x^{1+\alpha/2}$
Middle buffer	$(x^{1/2})^{1+\alpha/2}$	$\frac{1}{4}x^{1/2+\alpha/2}$	$\frac{1}{4}x^{1+3\alpha/4}$
Bottom buffer	$(x^{1/2})^{1+\alpha}$	$\frac{1}{4}x^{1/2+\alpha/2}$	$\frac{1}{4}x^{1+\alpha}$
Bottom buffer	$x^{1+\alpha}$	1	$x^{1+\alpha}$

Figure 10-2: Sizes of buffers in an x-box. This table lists the sizes of the three buffers in an x-box and the sizes and number of buffers in its recursive $x^{1/2}$-boxes, expanding just one level of recursion.

There is, however, no relationship between the partition imposed by the upper-level subboxes and that of the lower-level subboxes; the subranges are entirely unrelated. What this setup means is that an element with a particular key may be located in the input buffer or the middle buffer or the output buffer or a particular upper-level subbox or a particular lower-level subbox. Our search procedure will thus look in all five of these locations to locate the element in question.

Before delving into more detail about the x-boxes, let us first give a rough sketch of insertions into the data structure. When an element is inserted into an x-box D, it is first inserted into D's input buffer. As the input buffer stores elements in sorted order, elements in the input buffer must move to the right to accommodate newly inserted elements. When D's input buffer becomes full (or nearly full), the elements are inserted recursively into D's upper-level subboxes. When an upper-level subbox becomes full enough, it is "split" into two subboxes, with one subbox taking the half of the elements with smaller keys and the other taking the elements with larger keys. When the maximum number of upper-level subboxes is reached, all elements are moved from the upper-level subboxes to D's middle buffer (which stores all elements in sorted order). When the middle buffer becomes full enough, elements are moved the lower-level subboxes in a similar fashion. When the maximum number of lower-level subboxes is reached, all elements are moved from the lower-level subboxes to D's output buffer.

To aid in pushing elements down to the appropriate subboxes, we embed in the input (and middle) buffers a pointer to each of the upper-level (and lower-level) subboxes. These ***subbox pointers*** are embedded into the buffers by associating with them the minimum key stored in the corresponding subbox, and then storing them along with the buffer's elements in sorted order by key.

To facilitate searches, we use lookahead pointers as in Chapter 9, employing the technique of *fractional cascading* [64]. That is to say, we give an x-box's input buffer (and middle buffer) a sample of the elements of the upper-level subboxes' (and lower-level subboxes') input buffers. Specifically, let U be an upper-level subbox of the x-box D. Then a constant fraction of the keys stored in U's input buffer are also stored in D's input buffer. This sampling is performed deterministically by placing every sixteenth element in U's input buffer into D's input buffer. The sampled element (in D's input buffer) has a pointer to the corresponding element in U's input buffer. This sampling also occurs on the output buffers. Specifically, the output buffers of the upper-level (and lower-level) subboxes contain a similar sample of the elements in D's middle buffer (and output buffer).[2]

The advantage of lookahead pointers is the same as in Chapter 9, and is roughly as follows. Suppose we are looking for a key κ in some buffer A. Let s be the multiple of 16 (i.e., a sampled element) such that $A[s] \leq \kappa < A[s + 16]$. Then our search procedure will scan slots s to $s + 16$ in the buffer A. If κ is not located in any of these slots, then it is not in the buffer. Ideally, this scan would also provide us with a good starting point to search within the next buffer.

As the lookahead pointers may be irregularly distributed in D's input (or middle) buffer, a stretch of sixteen elements in D's input (or middle) buffer may not contain a lookahead pointer to an upper (or lower)-level subboxes. To remedy this problem, we use duplicate lookahead pointers, again as in Chapter 9, associating with each element in D's input (or middle) buffer a pointer to the nearest lookahead or subbox pointers preceding and following it.

[2]Any sufficiently large sampling constant suffices here. We chose 16 as an example of one such constant for concreteness. The constant must be large enough that, when sampling D's output buffer, the resulting lookahead pointers occupy only a constant fraction of the lower-level subboxes' output buffers. To make the description more concise, the constant fraction we allow is $1/4$, but in fact any constant would work. As the lower-level subboxes' output buffers account for only $1/4$ of the space of D's output buffer, these two constants are multiplied to get that only $1/16$ of the elements in D's output buffer may be sampled.

Techniques. Not only does this dictionary employ the fractional cascading technique of Chapter 9, but it is also recursive. We are not aware of any previous cache-oblivious data structures that are both recursive and that use fractional cascading. Another subtler difference between the x-box and previous cache-oblivious structures involves the sizing of subboxes and our use of the middle buffer. If the x-box matched typical structures, then an x-box with an input buffer of size x and an output buffer of size $x^{1+\alpha}$ would have a middle buffer of size $x^{\sqrt{1+\alpha}}$, not the $x^{1+\alpha/2}$ that we use. That is to say, it natural natural to size the buffers such that a size-x input buffer is followed by a size-x^δ middle buffer for some δ, and a size-$y = x^\delta$ middle buffer is followed by a size-$y^\delta = x^{\delta^2}$ output buffer. Our choice of sizes causes the data structure to be more topheavy than usual, a feature which facilitates obtaining an optimal query/update tradeoff.

10.2 Sizing an x-box

An x-box stores the following fields, in order, in a fixed contiguous region of memory.

1. A counter of the number of elements stored in the x-box.
2. The top buffer.
3. Array of booleans indicating which upper-level subboxes are being used.
4. Array of upper-level subboxes, in an arbitrary order.
5. The middle buffer.
6. Array of booleans indicating which lower-level subboxes are being used.
7. Array of lower-level subboxes, in an arbitrary order.
8. The bottom buffer.

In particular, the entire contents of each $\sqrt{x}$-subbox are stored within the x-box itself. In order for an x-box to occupy a fixed contiguous region of memory, we need a upper bound on the maximum possible space usage of a box.

Lemma 10.1 *The total space usage of an x-box is at most $c\,x^{1+\alpha}$ for some constant $c > 0$.*

Proof. The proof is by induction. An x-box contains three buffers of total size $c'(x + x^{1+\alpha/2} + x^{1+\alpha}) \le 3c'x^{1+\alpha}$, where c' is the constant necessary for each array entry (including information about lookahead pointers, etc.). The boolean arrays use a total of at most $(1/4)x^{1/2+\alpha/2} \le c'x^{1+\alpha}$ space, giving us a running total of $4c'x^{1+\alpha}$ space. Finally, the subboxes by assumption use a total of at most $c(x^{1/2})^{1+\alpha} \cdot (1/2)x^{1/2+\alpha/2} = (c/2)x^{1+\alpha}$ space. Setting $c \ge 8c'$ yields a total space usage of at most $cx^{1+\alpha}$. □

10.3 Operating an x-box

An x-box D supports two operations:

1. BATCH-INSERT$(D, e_1, e_2, \ldots, e_{\Theta(x)})$: Insert $\Theta(x)$ keyed elements $e_1, e_2, \ldots, e_{\Theta(x)}$ given as a sorted array. BATCH-INSERT maintains lookahead pointers as previously described.
2. SEARCH(D, s, κ): Return a pointer to an element with key κ in the x-box D if such an element exists. If no such element exists, return a pointer to κ's predecessor in D's output buffer, that is, the element located in D's output buffer with the largest key smaller than κ. We assume that we are given the nearest lookahead pointer s preceding κ pointing into D's input buffer: that is, s points to the sampled element (i.e., having an index that is a multiple of 16) in D's input buffer that has the largest key not exceeding κ.

SEARCH

Searches are easiest. The operation SEARCH(D, s, κ) starts by scanning D's input buffer at slot s and continues until reaching an element with key κ or until reaching slot s' where the key at s' is larger than κ. By assumption on lookahead pointers, this scan considers $O(1)$ array slots. If the scan finds an element with key κ, then we are done. Otherwise, we consider the nearest lookahead or subbox pointer preceding slot $s' - 1$. We follow this pointer and search recursively in the corresponding subbox. That recursive search returns a pointer in the subbox's output buffer. We again reference the nearest lookahead pointer and jump to a point in the middle buffer. This process continues, scanning a constant-size region in middle buffer, searching recursively in the lower-level subbox, and scanning a constant-size region in the output buffer.

Lemma 10.2 *For $x > B$, a* SEARCH *in an x-box costs $O((1 + \alpha) \log_B x)$ memory transfers.*

Proof. The search considers a constant-size region in the three buffers, for a total of $O(1)$ memory transfers, and performs two recursive searches. Thus, the cost of a search can be described by the recurrence $S(x) = 2S(\sqrt{x}) + O(1)$. Once $x^{1+\alpha} = O(B)$, or equivalently $x = O(B^{1/(1+\alpha)})$, an entire x-box fits in a block, and no further recursions incur any other cost. Thus, we have a base case of $S(O(B^{1/(1+\alpha)})) = 0$.

The recursion therefore proceeds with a nonzero cost for $\lg \lg x - (1/1+\alpha) \lg \lg O(B)$ levels, for a total of $2^{\lg \lg x - (1/1+\alpha) \lg \lg O(B)} = (1 + \alpha) \lg x / \lg O(B) = O((1 + \alpha) \log_B x)$ memory transfers. □

BATCH-INSERT overview

For clarity, we decompose BATCH-INSERT into several operations, including two new auxiliary operations:

1. FLUSH(D): After this operation, all k elements in the x-box D are located in the first $\Theta(k)$ slots of D's output buffer. These elements occur in sorted order. The $\Theta()$ arises because of the presence of lookahead pointers directed from the output buffer. The FLUSH operation is an auxiliary operation used by the BATCH-INSERT.
2. SAMPLE-UP(D): This operation may only be invoked on an x-box that is entirely empty except for its output buffer (as with one that has just been FLUSHed). The sampling process is employed from the bottom up, creating subboxes as necessary, and placing the appropriate lookahead pointers. The SAMPLE-UP operation is an auxiliary operations used by BATCH-INSERT.

FLUSH

To FLUSH an x-box, first flush all the subboxes. Now consider the result. Elements can live in only five possible places: the input buffer, the middle buffer, the output buffer, the upper-level subboxes' output buffers, and the lower-level subboxes' output buffers. The elements are in sorted order in all of the buffers. Moreover, as the upper-level (and lower-level) subboxes partition the key space, the collection of upper-level subboxes' output buffers form a fragmented sorted list of elements. Thus, after flushing all subboxes, moving all elements to the output buffer requires just a 5-way merge into the output buffer. A constant-way merge can be performed in a linear number of memory transfers in general, but here we have to deal with fragmentation.

When merging all the elements into the output buffer, we merge only real elements, not lookahead pointers (except for the lookahead pointers that already occur in the output buffer). This step

breaks the sampling structure of the data structure, which we will later resolve with the SAMPLE-UP procedure.

When the flush completes, the input and middle buffers are entirely empty, and all subboxes are deleted.[3]

Lemma 10.3 *For $x^{1+\alpha} > B$, a* FLUSH *in an x-box costs $O(x^{1+\alpha}/B)$ memory transfers.*

Proof. We can describe the flush by the recurrence

$$F(x) = O(x^{1+\alpha}/B) + O(x^{1/2+\alpha/2}) + \frac{1}{2}x^{1/2+\alpha/2}F(\sqrt{x}) ,$$

where the first term arises due to scanning all the buffers (i.e., the 5-way merge), the second term arises from the random accesses both to load the first block of any of the subboxes and to jump when scanning the concatenated list of output buffers, and the third term arises due to the recursive flushing. When $x^{1+\alpha} = O(M)$, the second term disappears as the entire x-box fits in memory, and we thus need only pay for loading each block once. When applying a tall-cache assumption that $M = \Omega(B^2)$, it follows that the second term only occurs when $x^{1/2+\alpha/2} = \Omega(B)$, and hence when $x^{1/2+\alpha/2} = x^{1+\alpha}/x^{1/2+\alpha/2} = x^{1+\alpha}/\Omega(B) = O(x^{1+\alpha}/B)$. We thus have a total cost of

$$F(x) \le c_1x^{1+\alpha}/B + \frac{1}{2}x^{1/2+\alpha/2}F(\sqrt{x}) ,$$

where c_1 is a constant hidden by the order notation.

We next prove that $F(x) \le cx^{1+\alpha}/B$ by induction. As a base case, when $y^{1+\alpha}$ fits in memory, the cost is $F(y) = cy^{1+\alpha}/B$ as already noted, to load each block into memory once. Applying the inductive hypothesis that $F(y) \le cy^{1+\alpha}/B$ for some sufficiently large constant c and $y < x$, we have $F(x) \le c_1x^{1+\alpha}/B + (1/2)x^{1/2+\alpha/2}(cx^{1/2}/B) = c_1x^{1+\alpha}/B + (1/2)cx^{1+\alpha/2}/B$. Setting $c > 2c_1$ completes the proof. □

SAMPLE-UP

When invoking a SAMPLE-UP, we assume that the only elements in the x-box live in the output buffer. In this state, there are no lookahead pointers to facilitate searches. The SAMPLE-UP recomputes the lookahead pointers, allowing future searches to be performed efficiently.

The SAMPLE-UP operates as follows. Suppose that the output buffer contains $k < x^{1+\alpha}$ elements. Then we create $(k/16)/(x^{1/2+\alpha/2}/2) = k/8x^{1/2+\alpha/2} \le x^{1/2+\alpha/2}/8$ new lower-level subboxes. Recall that this is half the number of available lower-level subboxes. We then assign to each of these subboxes $x^{1/2+\alpha/2}/2$ of contiguous sampled pointers (thus filling half of their respective output buffers), and recursively call SAMPLE-UP in the subboxes. Then, we sample from the lower-level subboxes' input buffers to the middle buffer, and we sample the middle-buffer to upper-level subboxes in a similar fashion. Finally, we sample the upper-level subboxes up to the input buffer.

Lemma 10.4 *A* SAMPLE-UP *in an x-box, for $x^{1+\alpha} > B$, costs $O(x^{1+\alpha}/B)$.*

Proof. The proof is virtually identical to the proof for FLUSH. The recurrence is the same (in fact, it is better here because we can guarantee that the subboxes are, in fact, contiguous). □

[3] In fact, the subboxes use a fixed memory footprint, so they are simply marked as deleted.

BATCH-INSERT

The BATCH-INSERT operation takes as input a sorted array of elements to insert. In particular, when inserting into an x-box D, a BATCH-INSERT inserts $\Theta(x)$ elements. For conciseness, let us say that the constant hidden by the theta notation is $1/2$. First, merge the inserted elements into D's input buffer, and increment the counter of elements contained in D by $(1/2)x$. For simplicity, also remove the lookahead pointers during this step. We will readd them later.

Then, (implicitly) partition the input buffer according to the ranges owned by each of the upper-level subboxes. If any partition contains at least $(1/2)\sqrt{x}$ elements remove those elements from D's input buffer and insert them recursively into the appropriate subbox. If performing a recursive insert would cause the number of elements in the subbox to exceed $(1/2)\sqrt{x}^{1+\alpha}$, first "split" the subbox. A "split" entails first FLUSHing the subbox, creating a new subbox, moving the larger half of the elements from the old subbox's output buffer to the new subbox's output buffer (involving two scans), and calling SAMPLE-UP on both subboxes, and updating the counter recording the number of elements in each subbox to reflect the number of real elements in each.

Observe that after all the recursions occur, the number of elements in D's input buffer is at most $(1/2)\sqrt{x} \cdot (1/4)\sqrt{x} = (1/8)x$, as otherwise more elements would have moved down to a subbox.

After moving element from the input buffer to the upper-level subboxes, resample from the upper-level subboxes' input buffers into D's input buffer. Note that the number of lookahead pointers introduced into the input buffer is at most $(1/16)\sqrt{x} \cdot (1/4)\sqrt{x}$. When combining the number of lookahead pointers with the number of real elements, we see that D's input buffer is far less than half full, and hence it can accommodate the next insertion.

When a split causes the last available subbox to be allocated, we merge all input and upper-level elements into the middle buffer. We do so by first FLUSHing all the upper-level subboxes, and then merging into the middle buffer (similarly to how is done in the full FLUSH). We then perform a similar movement from the middle buffer to the lower-level subboxes as we did for the input buffer to upper-level subboxes. When the last lower-level subbox is allocated, we move elements to the output buffer and perform a similar sampling. After the lower-level recursive insertions complete, we sample into output buffers of the upper-level subboxes and call SAMPLE-UP on each of these subboxes. We then sample from the input buffers of the upper-level subboxes to D's input buffer.

Observe that the upper-level subboxes' output buffers collectively contain at most $(1/16)x^{1+\alpha/2}$ lookahead pointers. Moreover, after elements are moved into the middle buffer, these are the only elements in the upper-level subboxes. Because all subboxes remain at least half full, and these subboxes can accommodate $(1/4)x^{1+\alpha/2}$ elements in total, there must be at least $(1/8)x^{1+\alpha/2} - (1/16)x^{1+\alpha/2} = \Omega(x^{1+\alpha/2})$ insertions into D between moves into the middle buffer. A similar argument shows that there must be at least $\Omega(x^{1+\alpha})$ insertions into D between insertions into the output buffer.

Theorem 10.5 *A BATCH-INSERT into an x-box, with $x > B$, costs an amortized $O((1 + \alpha)\log_B(x)/B^{2/(2+\alpha)})$ memory transfers per element.*

Proof. An insert has several costs per element. First, there is the cost of merging into the input array, which is simply $O(1/B)$ per element. Next, each element is inserted recursively into a top-level subbox. These recursive insertions entail random accesses to load the first block of each of the subboxes. Then these subboxes must be sampled, which is dominated by the cost of the aforementioned random accesses and scans. An element may also contribute to a split of a subbox, but each split may be amortized against $\Omega(x^{1/2+\alpha/2})$ insertions. Then we must also consider the cost of moving elements from the upper-level subboxes to the middle buffer, but this movement may

be amortized against the $\Omega(x^{1+\alpha/2})$ elements being moved. Finally, there are similar costs among the lower-level subboxes.

Let us consider the cost of the random accesses more closely. If all of the upper-level subboxes fit into memory, then the cost of random accesses is actually the minimum of performing the random accesses or loading the entire upper-level into memory. For the upper-level, we denote this value by $\mathit{UpperRA}(x)$. We thus have $\mathit{UpperRA}(x) = O((1/4)x^{1/2})$ if $x^{1+\alpha/2} = \Omega(M)$, and $\mathit{UpperRA}(x) = O(\min\{(1/4)x^{1/2}, x^{1+\alpha/2}/B\})$ if $x^{1+\alpha/2} = O(M)$. In fact, we are really concerned with $\mathit{UpperRA}(x)/x$, as the random accesses can be amortized against the x elements inserted. We analyze the two cases separately, and we assume the tall-cache assumption that $M > B^2$.

1. Suppose that $x^{1+\alpha/2} = \Omega(M)$. Then we have $x^{1+\alpha/2} = \Omega(B^2)$ by the tall-cache assumption, and hence $x^{1/2} = \Omega(B^{2/(2+\alpha)})$. It follows that $\mathit{UpperRA}(x)/x = O(1/\sqrt{x}) = O(1/B^{2/(2+\alpha)})$.

2. Suppose that $x^{1+\alpha/2} = O(M)$. We have two subcases here. If $x > B^{2/(1+\alpha)}$, then we have a cost of at most $\mathit{UpperRA}(x)/x = O(x^{1/2}/x) = O(1/B^{1/(1+\alpha)})$. If, on the other hand, $x < B^{2/(1+\alpha)}$, then we have $\mathit{UpperRA}(x)/x = O(x^{1+\alpha/2}/Bx) = O(x^{\alpha/2}/B) = O(B^{\alpha/(1+\alpha)}/B) = O(1/B^{1/(1+\alpha)})$.

Because $1/B^{2/(2+\alpha)} > 1/B^{1/(1+\alpha)}$, we conclude that $\mathit{UpperRA}(x)/x = O(1/B^{2/(2+\alpha)})$.

We must also consider the cost of random accesses into the lower-level subboxes, which can be amortized against the $x^{1+\alpha/2}$ elements moved. A similar case analysis shows that $\mathit{LowerRA}(x)/x^{1+\alpha/2} = O(1/B^{1/(1+\alpha)})$.

We thus have a total insertion cost of

$$\begin{aligned}
I(x) &= O\left(\frac{x/B}{x}\right) + O\left(\frac{\mathit{UpperRA}(x)}{x}\right) + I(\sqrt{x}) + O\left(\frac{F(\sqrt{x})}{x^{1/2+\alpha/2}}\right) + O\left(\frac{\frac{1}{4}x^{1/2}F(\sqrt{x})}{x^{1+\alpha/2}}\right) \\
&\quad + O\left(\frac{x^{1+\alpha/2}/B}{x^{1+\alpha/2}}\right) + O\left(\frac{\mathit{LowerRA}(x)}{x^{1+\alpha/2}}\right) + I(\sqrt{x}) + O\left(\frac{F(\sqrt{x})}{x^{1/2+\alpha/2}}\right) \\
&\quad + O\left(\frac{\frac{1}{4}x^{1/2+\alpha/2}F(\sqrt{x})}{x^{1+\alpha}}\right) + O\left(\frac{x^{1+\alpha}/B}{x^{1+\alpha}}\right) \\
&= O(1/B) + O\left(\frac{\mathit{UpperRA}(x)}{x}\right) + O\left(\frac{\mathit{LowerRA}(x)}{x^{1+\alpha/2}}\right) + O\left(\frac{F(\sqrt{x})}{x^{1/2+\alpha/2}}\right) + 2\,I(\sqrt{x}) \\
&= O(1/B) + O(1/B^{2/(2+\alpha)}) + O(1/B^{1/(1+\alpha)}) + O\left(\frac{x^{1/2+\alpha/2}/B}{x^{1/2+\alpha/2}}\right) + 2\,I(\sqrt{x}) \\
&= O(1/B^{2/(2+\alpha)}) + 2\,I(\sqrt{x})
\end{aligned}$$

As we charge loading the first block of a subbox to an insert into the parent, we have a base case of $I(O(B^{1/(1+\alpha)})) = 0$, i.e., when the x-box fits into a single block. Solving the recurrence, we get a per element cost of $O(1/B^{2/(2+\alpha)})$ at $\lg\lg x - (1/1+\alpha)\lg\lg O(B)$ levels of recursion, and hence a total cost of $O(2^{\lg\lg x-(1/1+\alpha)\lg\lg O(B)}/B^{2/(2+\alpha)}) = O\left(((1+\alpha)\lg x)/(B^{2/(2+\alpha)}\lg O(B))\right) = O((1+\alpha)\log_B(x)/B^{2/(2+\alpha)})$. □

10.4 Building a dictionary out of x-boxes

The xDict data structure consists of $\log_{1+\alpha}\log_2 N + 1$ x-boxes of doubly increasing size. Specifically, for $0 \le i \le \log_{1+\alpha}\log_2 N$, the ith box has $x = 2^{(1+\alpha)^i}$. The x-boxes are linked together by incorporating into the ith box's output buffer the lookahead pointers corresponding to a sample from the $(i+1)$th box's input buffer.

We can define the operations on an xDict in terms of x-box operations. To insert an element into an xDict, we simply insert it into the smallest x-box ($i = 0$), which has $x = \Theta(1)$ so supports individual element insertions. When the ith box reaches capacity (containing $2^{(1+\alpha)^i}$ elements), we FLUSH it, insert all of its elements (contained in its output buffer) into the $(i+1)$th box, and empty the ith box's output buffer. This process terminates after performing a batch insert into the jth box if the jth box is the first box that can accommodate the elements (having not yet reached capacity). At this point, all boxes preceding the jth box are entirely empty. We next rebuild the lookahead starting from the $(j-1)$th box down to the 0th box by sampling from the $(i+1)$th box's input buffer into the ith box's output buffer and then calling SAMPLE-UP on the ith box. As elements are first inserted into the 0th box and eventually move through all boxes, our insertion analysis accounts for an insertion into each box for each element inserted.

To search in the xDict, we simply search in each of the x-boxes, and return the closest match. Specifically, we search the boxes in order from smallest to largest. If the element is not found in the ith box, we have a pointer into the ith box's output buffer. We use this pointer to find an appropriate lookahead pointer into the $(i+1)$th box's input buffer, and begin the search from that point.

The performance of the xDict is essentially a geometric series:

Theorem 10.6 *The xDict supports searches in $O(((1+\alpha)^2\alpha)\log_B(N/M))$ memory transfers and (single-element) inserts in $O(((1+\alpha)^2/\alpha)(\log_B(N/M))/B^{2/(2+\alpha)})$ amortized memory transfers.*

Proof. A simple upper bound on the search cost is

$$\begin{aligned}\sum_{i=0}^{\log_{1+\alpha}\log_2 N} O((1+\alpha)\log_B(2^{1+\alpha^i})) &= O\left(\frac{1+\alpha}{\lg B}\sum_{i=0}^{\log_{1+\alpha}\log_2 N}(1+\alpha)^i\right)\\ &= O\left(\frac{(1+\alpha)\lg N}{\lg B}\sum_{i=0}^{\infty}\frac{1}{(1+\alpha)^i}\right)\\ &= O(\frac{(1+\alpha)^2}{\alpha}\log_B N).\end{aligned}$$

The above analysis, however, exploits only a constant number of cache blocks. By preloading into cache all the smallest x-boxes that fit, the first $O(((1+\alpha)^2/\alpha)\log_B M)$ of the search cost is actually free, resulting in a search cost of $O(((1+\alpha)^2/\alpha)(\log_B(N/M))/B^{2/(2+\alpha)})$. The cache-oblivious model allows us to assume an optimal paging strategy, and caching the top of the structure is no better than optimal.

The analysis for insertions is identical except that all costs are multiplied by $O(1/B^{2/(2+\alpha)})$. □

Corollary 10.7 *For any ε with $0 < \varepsilon < 1$, there exists a setting of α such that the xDict supports searches in $O((1/\varepsilon)\log_B(N/M))$ memory transfers and supports insertions in $O((1/\varepsilon B^{1-\varepsilon})\log_B((N/M)))$ amortized memory transfers.*

Proof. First off, we consider only $\varepsilon < 1/2$ (rolling up a particular constant into the big-O notation), as larger ε only hurt the performance of inserts.

Choose $\alpha = 2\varepsilon/(1-\varepsilon)$, which gives $B^{2/(2+\alpha)} = B^{1-\varepsilon}$. Because $\varepsilon < 1/2$, we have $\alpha < 2$, so we can eliminate the $1+\alpha$ terms from Theorem 10.6. We are thus left with the $1/\alpha$ term, which solves to $1/\alpha = (1-\varepsilon)/(2\varepsilon) = O(1/\varepsilon)$. □

10.5 Final notes

I did not address deletion in detail, but claim that it can be handed using standard techniques. For example, to delete an element insert an anti-element with the same key value. In the course of an operation, should a key value and its antivalue be discovered, they annihilate each other while releasing potential which is used to remove them from the buffers they are in. Rebuilding the whole structure when the number of deletions since the last rebuild is half of the structure ensures that the total size does not get out of sync with the number of not-deleted items currently stored.

Another detail is that, to hold n items, an n-box may be created, which occupies up $\Theta(n^{1+\alpha})$ of address space. However, only a linear amount of space is occupied and the layout described ensures the unused space is at the end, so the amount of space is indeed linear.

Chapter 11

The Worst Page-Replacement Policy

Since the early days of computer science, thousands of papers have been written on how to optimize various components of the memory hierarchy. In these papers a recurrent question (at least four decades old) is the following: Which page-replacement strategies are the best possible?

The point of this chapter is to address the reverse question: *Which page-replacement strategies are the worst possible?* This chapter explores several ways to formulate this question. In some of these formulations, the worst strategy is a new algorithm that (luckily) has little chance of ever being implemented in software or silicon. In others, the worst strategy may be disturbingly familiar. This chapter represents join work with Kunal Agrawal and Michael A. Bender, previously appearing in [7]. Although this chapter is not entirely serious, the analyses may be interesting, and a theorem in Section 11.3 roughly states that a direct-mapped cache is provably bad.

Let us proceed by formalizing the paging problem. The machine model for the paging problem matches that of the I/O model [6] described in Chapter 7. That is, assume a two-level memory hierarchy consisting of a small fast memory, the ***cache***, and an arbitrarily large slow memory. Memory is divided into unit-size blocks or ***pages***. Exactly k pages fit in fast memory.[1] In order for a program to access a memory location, the page containing that memory location must reside in fast memory. Thus, as a program runs, it makes ***page requests***. If a requested page is already in fast memory, then the request is satisfied at no cost. Otherwise, the page must be transferred from slow to fast memory. When the fast memory already holds k pages, one page from fast memory must be evicted to make room for the new page. (The fast memory is initially empty, but once it fills up, it stays full.) The cost of a program is measured in terms of the number of transfers. Whereas the I/O model assumes that the eviction choices are optimal, the paging problem considers the page-eviction strategy.

The objective of the paging problem is to minimize the number of page transfers by optimizing which pages should be evicted on each page requests. When all page requests are known a priori (the offline problem), then the optimal strategy, proposed by Belady, is to replace the page whose next request occurs furthest in the future [31].

More recent work has focused on the online problem, in which the paging algorithm must continually decide which pages to evict without prior knowledge of future page requests. Sleator and Tarjan introduce competitive analysis [181] to analyze online strategies. Let $A(\sigma)$ represent the cost incurred by the algorithm A on the request sequence σ, and let $\mathrm{OPT}(\sigma)$ be the cost incurred by the optimal offline strategy on the same sequence. For a minimization problem, we say that an online

[1] I am using k here instead of the equivalent M/B in Chapter 7 to more closely match the notation in the literature for this topic area.

strategy A is ***c-competitive*** if there exists a constant β such that for every input sequence σ,

$$A(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + \beta\,.$$

Sleator and Tarjan prove that there is no online strategy for page replacement that is better than k-competitive, where k is the memory size. Moreover, they show that the ***least-recently-used (LRU)*** strategy, in which the page chosen for eviction is always the one requested least recently, is k-competitive. If the online strategy operates on a memory that is twice the size of that used by the offline strategy, they show that LRU is 2-competitive. Since this seminal result, many subsequent papers have analyzed paging algorithms using competitive analysis and its variations. Irani [122] gives a good study of many of these approaches.

Page-replacement strategies are used at multiple levels of the memory hierarchy. Between main memory and disk, memory transfers are called ***page faults***, and between cache and main memory, they are ***cache misses***. There are other differences in between levels besides mere terminology. In particular, because caches must be fast, a cache memory block, called a ***cache line***, can only be stored in one or a limited number x of cache locations. The cache is then called ***direct mapped*** or ***x-way-associative***, respectively. There have been several recent algorithmic papers showing how caches with limited associativity can use hashing techniques to acquire the power of caches with unlimited associativity [98, 177].

I now describe what I mean by the worst page-replacement stategy. First of all, this chapter considers only "reasonable" paging strategies. A strategy is ***reasonable*** if it is only allowed to evict a page from fast memory when

1. an eviction is necessary to service a new page request, i.e., when the fast memory is full, and
2. the evicted page is *replaced* by the currently requested page.

Without this reasonableness restriction a paging strategy could perform poorly by preemptively evicting all pages from fast memory. In contrast, this chapter explores strategies that somehow try to do the right thing, but just fail miserably.

Thus, the ***pessimal-cache problem*** is as follows: identify replacement strategies that maximize the number of memory transfers, no matter how efficiently code happens to be optimized for the memory system.

As with traditional paging problems, the pessimal-cache problem is analyzed using competitive analysis. Since we now have a maximization problem, the definition of competitive is slightly different: An online algorithm A is ***c-competitive*** if there exists a constant β such that for every input sequence σ,

$$A(\sigma) \geq \frac{1}{c} \cdot \mathrm{OPT}(\sigma) - \beta\,.$$

An input sequence σ consists of a sequence of page requests.[2] The objective of the online algorithm A is to maximize the number of page faults on the input sequence, and OPT is the offline strategy that maximizes the total number of page faults.

Since the pessimal-cache problem turns the traditional problem on its head, terminology may now seem backwards. Optimal now means optimally bad from a traditional point of view. The adversary is trying to give us an input sequence for which we do not have many more page faults than OPT. Thus, in some sense, the adversary is our friend, who is looking out for our own good, whereas we are trying to indulge in bad behavior.

[2] If a page is requested repeatedly without any other page being interleaved, all strategies have no choice, and there is no page fault on any but the first access. Thus, we consider only sequences in which the same page is repeated only when another page is accessed in between.

Note that the pessimal-cache problem still assigns cost in the same way, and thus counts competitiveness in terms of the number of *misses* and not the number of *hits*. We leave the problem in this form because OPT may have no hits, whereas even the best online strategies have infinitely many.

The remainder of this chapter is organized as follows. Section 11.1 provides lower bounds for the pessimal-cache problem. Section 11.1 shows that there is no deterministic, competitive, online algorithm for the pessimal-cache problem. Moreover, there is no (randomized) algorithm better than k-competitive. Section 11.2 gives an algorithm that is expected k-competitive, and hence optimal for the pessimal-cache problem. Since this strategy exhibits a $1/k$ fraction of the maximum number of page faults on every input sequence, this strategy is the worst page-replacement strategy. Finally, Section 11.3 examines page-replacement strategies for caches with limited associativity. This section roughly states that page-replacement strategy employed by direct-mapped caches is the worst possible, under the assumption that page locations are random.

11.1 Lower bounds

This section gives lower bounds on the competitiveness of the pessimal-cache problem. There are two main lemmas in this section. The first lemma states that no deterministic strategy is competitive. The second lemma, concerning randomized strategies, states that no strategy can be better than k-competitive, where k is the fast-memory size.

The following lemma states that there is no deterministic online strategy that is competitive with the offline strategy.

Lemma 11.1 *Consider any deterministic strategy A for the pessimal-cache problem with fast-memory size $k \geq 2$. For any $\varepsilon > 0$ and constant β, there exists an input sequence σ such that $A(\sigma) < \varepsilon \cdot \mathrm{OPT}(\sigma) - \beta$.*

Proof. Consider a sequence σ that begins by requesting pages $v_1, v_2, \ldots, v_{k+1}$. While the first k pages are requested, all strategies have no choice and have a fast-memory containing pages $v_1, \ldots, v_k$. At the time v_{k+1} is requested, one of the pages must be evicted from fast memory. Suppose that the deterministic strategy chooses to evict page v_i. Then for any j with $1 \leq j \leq k$ and $i \neq j$, consider the sequence $\sigma = v_1, v_2, \ldots, v_{k+1}, v_j, v_{k+1}, v_j, v_{k+1}, \ldots$ that alternates between v_{k+1} and v_j. Since the deterministic strategy has $v_1, \ldots, v_k$ in fast memory when v_{k+1} is requested, and $v_i \neq v_j$ is evicted, both v_j and v_{k+1} are in fast memory after the request. Thus, all future requests are to pages already in fast memory, and this strategy does not incur any more page faults after the first $k+1$. The offline strategy OPT, on the other hand, still incurs a page fault on every request by evicting page v_j when v_{k+1} is requested and vice versa. Extending the length of the sequence proves the lemma. □

Lemma 11.1 also holds even in the presence of ***resource deaugmentation***, where the online strategy runs with a *smaller* fast memory of size $k_{on} \geq 2$ and offline optimal strategy runs with a larger fast memory of size $k_{off} \geq k_{on}$.[3] Even so, there is still no competitive deterministic strategy. The same proof still applies with the same sequence–the proof just relies on the fact that there are two particular pages in the fast memory of the online algorithm.

Let us now turn our attention to randomized strategies with an ***oblivious adversary***, meaning that the adversary must choose the entire input sequence before seeing the result of any of the coin

[3]Note that resource deaugmentation in the pessimal-cache problem is the analog of resource augmentation in the classical problem, in which the online algorithm has a *larger* cache than the offline algorithm.

tosses used by the randomized algorithm. Note that in the presence of a nonoblivious adversary, randomization does not provide extra power for pessimal-cache problem.

The following lemma states that no randomized strategy is better than expected k-competitive when both the online and offline strategies have the same fast-memory size k. Moreover, when the offline strategy uses a fast memory of size k_{off} and the online strategy has a fast memory of size $k_{on} \le k_{off}$, no online strategy is better than $k_{off}/(k_{off} - k_{on} + 1)$.

Lemma 11.2 *Let k_{off} be the fast memory size of the offline strategy and k_{on} (with $1 \le k_{on} \le k_{off}$) be the fast memory size of the online strategy. Consider any (randomized) online strategy A. For any $c < k_{off}/(k_{off} - k_{on} + 1)$ and constant β, there exists an input σ such that $E[A(\sigma)] < \frac{1}{c} \cdot \mathrm{OPT}(\sigma) - \beta$.*

Proof. The proof is similar to that of Lemma 11.1. After the $(k_{off} + 1)$st page request $v_{k_{off}+1}$, the online algorithm A has k_{on} pages in fast memory. Page $v_{k_{off}+1}$ is definitely in fast memory. Of the remaining k_{off} pages requested so far, $k_{on} - 1$ are in A's fast memory.

Now let v_j be a randomly selected page from $v_1, \ldots, v_{k_{off}}$. Page v_j is in A's fast memory with probability $(k_{on} - 1)/k_{off}$. Now consider the sequence $v_1, \ldots, v_{k_{off}}, v_{k_{off}+1}, v_j, v_{k_{off}+1}, v_j, v_{k_{off}+1}, \ldots$. With probability $(k_{on} - 1)/k_{off}$, page v_j is still in fast memory after $v_{k_{off}+1}$ is requested. In this case, no future page requests cause page faults, giving the online strategy a total of $k_{off}+1$ page faults. With probability $(k_{off} - k_{on} + 1)/k_{off}$, v_j is not in memory, and the strategy may be able to attain the optimal ℓ page faults, where ℓ is the length of the sequence following the first request for $v_{k_{off}+1}$. Thus, the expected number of page faults is at most $(k_{off} + 1) + \ell(k_{off} - k_{on} + 1)/k_{off}$, whereas the offline strategy attains $(k + 1) + \ell$. Choosing a long enough sequence proves the lemma. □

11.2 Most-recently used

This section describes two k-competitive strategies for the pessimal-cache problem. The first strategy uses one step of randomization followed by the deterministic "most-recently-used" (MRU) heuristic. The second strategy uses more randomization to achieve the optimal result even when the offline and online strategies have different fast-memory sizes.

Since least-recently-used (LRU) is k-competitive and optimally competitive for traditional paging, it is natural to explore the reverse strategy for the pessimal-cache problem. The ***most-recently-used (MRU)*** heuristic always evicts the page in fast memory that was used most frequently. It might be reasonable to expect MRU to be k-competitive for the pessimal-cache problem. MRU, however, is deterministic, and Lemma 11.1 states that no deterministic strategy can be competitive.

Instead, consider a natural variation on MRU, which we call ***randomized MRU***. In randomized MRU, the first page evicted is chosen at random. (Recall that this first eviction happens when the $(k+1)$th distinct page is requested.) All subsequent evictions follow the MRU strategy. Randomized MRU gets around the alternating-page request sequence used to prove lower bounds in Lemmas 11.1 and 11.2. The following lemma shows that MRU keeps a (slightly) random set of pages in fast memory.

Lemma 11.3 *Let k be the size of fast memory (for both online and offline strategies), and consider any request sequence σ. After the $(k + 1)$st distinct page is requested, randomized MRU guarantees that there are k pages each having probability exactly $1 - 1/k$ of being in fast memory, and there is one page, the most-recently-used page, that has probability 1 of being in fast memory. All other pages are definitely not in fast memory.*

Proof. We prove the claim by induction on the requests over time.

The base case occurs when the $(k + 1)$st distinct page is requested, which causes the first eviction. Since there are k pages in fast memory at the time that the $(k + 1)$st distinct page is requested, and one is chosen to be evicted at random, the claim holds for the base case.

Suppose that the claim holds up until the tth request. Assume that the next request is for page v_i. There are several cases.

Case 1. Suppose that v_i is definitely not in fast memory. Then the most-recently-used page v_j is evicted, and hence v_i is definitely in fast memory and v_j is definitely not.

Case 2. Suppose that v_i is in fast memory with probability 1. Then none of the probabilities change.

Case 3. Suppose that v_i is in fast memory with probability $1 - 1/k$ and that v_j is the most-recently-used page. Then with probability $1/k$ we have v_i not in fast memory, and hence the request for v_i evicts v_j. Otherwise, v_j stays in fast memory. Thus, the probability that v_j is in fast memory is $1 - 1/k$, and the probability that the most recently used page v_i is in fast memory is 1.

The probability of any other page (other than the ones mentioned in the appropriate case) being in fast memory is unchanged across the request. □

The following theorem states that randomized MRU is k-competitive, where k is the size of fast memory.

Theorem 11.4 *Randomized MRU is expected k-competitive, where k is the size of fast memory.*

Proof. Consider any input sequence σ. If sequence σ contains requests to fewer than $k + 1$ distinct pages, then randomized MRU has at the same number of page faults as the offline strategy OPT (Both strategies have page faults only the first time each distinct page is requested.) Consider any request after the first $(k+1)$st distinct page is requested. If the request is for the most-recently-used page, then neither OPT nor randomized MRU have a page fault, since that page must be in fast memory. Otherwise, OPT causes a page fault. By Lemma 11.3, randomized MRU incurs a page fault with probability at least $1/k$. Specifically, MRU incurs a fault with exactly probability $1/k$ for any of k pages and probability 1 for any of the other pages. Thus, in expectation, randomized MRU incurs at least $1/k$ page faults for each page fault incurred by OPT. □

This result for randomized MRU is not quite analogous to the result of Sleator and Tarjan's [181] result for LRU. It is true that LRU is k-competitive for the traditional paging problem, and randomized MRU is k-competitive for the pessimal-cache problem. However, LRU also has good performance with resource augmentation. Specifically, if LRU has a fast memory of size k and the offline strategy has a fast memory size $(1 - 1/c)k$, then LRU is c-competitive. In particular, if the LRU has twice the fast memory of offline, then LRU is 2-competitive. The above result for the pessimal-cache problem does not generalize in the same way–the competitive ratio depends only on the size of randomized MRU's fast memory. If randomized MRU has a size-k fast memory and the offline strategy has a size $2k$ fast memory, then randomized MRU is still only k-competitive.

A more powerful MRU-based algorithm, which we call ***reservoir MRU***, achieves a better competitive ratio for the case of resource deaugmentation. As before, let k_{off} and $k_{on} \leq k_{off}$ be the sizes of the offline and online's fast memory, respectively.

The main idea of reservoir MRU is to keep a reservoir of $k_{on} - 1$ pages, where each previously-requested page resides in the reservoir with equal probability. (This technique is based on Vitter's reservoir sampling [194].) Reservoir MRU works as follows. For the first k_{on} distinct requests, the fast memory is not full, and thus there are no evictions. Subsequently, if there is a request for a previously-requested page v_i, and the page is not in memory, then the most-recently requested page is evicted. Otherwise, when the nth new page is requested, for any $n > k_{on}$, with probability

$1 - (k_{on} - 1)/(n - 1)$, the most recently requested page is evicted. Otherwise, the page to evict (other than the most-recently-used page) is chosen uniformly at random.

Reservoir MRU has an invariant that is a generalization of Lemma 11.3. After any request, the page that was requested most recently has probability 1 of being in fast memory. All other $n - 1$ pages have probability $(k_{on} - 1)/(n - 1)$ probability of being in fast memory.

Lemma 11.5 *Let k_{off} and $k_{on} \leq k_{off}$ be the fast memory sizes of the offline strategy and of reservoir MRU, respectively. Consider any page-request sequence σ to reservoir MRU. After the $n > k_{on}$th distinct page is requested, there is a single page, the most-recently-used page, that has probability 1 of being in fast memory. All other previously requested pages have probability $(k_{on} - 1)/(n - 1)$ of being in fast memory.*

Proof. The proof is by induction on the requests, and is reminiscent of the proof of Lemma 11.3.

After the $(k_{on} + 1)$th distinct request, the $(k_{on} + 1)$th page is definitely in fast memory, and one page randomly chosen has been evicted. Reservoir MRU evicts the most recently used page with probability $1 - (k_{on} - 1)/k_{on} = 1/k_{on}$ and all other page with the same probability. Thus, every page, except the last one has probability $1 - 1/k_{on}$ of being in fast memory, and the lemma holds for the base case.

Consider a request for page v_i after the nth distinct page has been requested. Assume by induction that the most-recently-used page v_j is definitely in fact memory and that all other $n - 1$ pages are in fast memory with probability $(k_{on} - 1)/(n - 1)$. There are several cases.

Case 1. Suppose that page $v_i = v_j$, i.e., v_j is in fast memory with probability 1. Then none of the probabilities change.

Case 2. Suppose that page v_i has been previously requested, but $v_i \neq v_j$. If v_i is already in fast memory then nothing is evicted. Otherwise, by the properties of reservoir MRU, page v_j is evicted. Since v_i was in fast memory with probability $(k_{on} - 1)/(n - 1)$, page v_j is evicted with probability $1 - (k_{on} - 1)/(n - 1)$ and remains in fast memory with probability $(k_{on} - 1)/(n - 1)$. None of the probabilities for pages other than v_i and v_j change.

Case 3. Suppose that page v_i has never been requested before, that is, v_i is the $(n + 1)$st distinct request. By the properties of reservoir MRU, the most-recently-used page v_j (which is definitely in fast memory) is evicted with probability $1 - (k_{on} - 1)/n$ and remains in fast memory with probability $(k_{on} - 1)/n$. Thus, the probability that v_j is in fast memory is at the desired value.

The probability that each additional page is in shared memory now also needs to decrease since the number of distinct pages has increased by one. Since with probability $(k_{on} - 1)/n$, a random page from the other $k_{on} - 1$ pages is evicted from fast memory, each page in fast memory is evicted with probability $1/n$. The probability that any page is in fast memory after this process is the probability that the page was in a fast memory before the $(n + 1)$st distinct page request times the probability that the page was not evicted by this request, which is $(k_{on} - 1)/(n - 1)(1 - 1/n) = (k_{on} - 1)/n$. Since the number of distinct pages requested is now $n+1$, this probability also matches the lemma statement. □

The previous lemma is employed by the following theorem to prove a better competitive ration for reservoir MRU in the case of resource deaugmentation.

Theorem 11.6 *Reservoir MRU is expected $k_{off}/(k_{off} - k_{on} + 1)$-competitive, where k_{off} is the size of fast memory of the offline strategy, and $k_{on} \leq k_{off}$ is the size of fast memory for reservoir MRU.*

Proof. Before k_{off} distinct requests, reservoir MRU has at least as many page faults as the offline strategy. And after this point, each time the offline strategy has a page fault, since $n > k_{off}$, reservoir

MRU incurs a page fault with probability at least $1-(k_{on}-1)/k_{off}$ from Lemma 11.5. □

This theorem means that when the offline strategy and reservoir MRU have the same fast-memory size k, reservoir MRU is k-competitive. When reservoir MRU has fast-memory size k_{on} and the offline strategy has fast-memory size $(1+1/c)k_{on}$, reservoir MRU is $(c+1)$-competitive.[4]

Reservoir MRU requires some additional state — in particular, it needs one bit per page to indicate whether the page has been requested before. Consequently, if the sequence requests n distinct pages, then reservoir MRU employs $O(n)$ extra bits of state. In contrast, Achlioptas et. al.'s [3] optimal randomized algorithm for the page-replacement problem requires only $O(k^2 \log k)$ extra bits of state. The extra state is unavoidable for reservoir MRU, however, because we must know when n, the number of distinct pages, increases. Fortunately, these extra bits can be stored in the slow memory, associated with each page—only the more reasonable $O(\log n)$ bits for the counter storing n need be remembered by the algorithm at any given time.

11.3 Direct mapping

This section considers the page-replacement strategy used in direct-mapped caches. The main result is that for the pessimal-cache problem, direct mapping is k-competitive under some assumptions about the mapping strategy or about the layout in slow memory.

In a direct-mapping strategy (see, e.g., [117]) each page v_i can be stored in only a single location $L(v_i)$ in fast memory. Thus, in a direct-mapped cache, once the function $L(v_i)$ is chosen, there are no algorithmic decisions to make: whenever a page v_i is requested, we must evict the page that is currently stored in location $L(v_i)$ and store v_i there instead.

This section shows that if $L(v_i)$ is randomly chosen for each v_i, then direct mapping is k-competitive with the optimal offline strategy (with no direct-mapped restrictions).

In fact, typically in real caches, the function $L(v_i)$ is determined by the low-order bits in the address of v_i in slow memory; it is not random. If each page v_i is stored in a random memory address in slow memory, however, then the following theorem still applies. While it is often unrealistic to assume that each page v_i is randomly stored, this approach was also used in [98, 177] to enable direct-mapped caches to simulate caches with no restrictions on associativity.

A direct mapping is not a reasonable strategy when compared with the optimal offline strategy with no mapping restrictions. In particular, a direct-mapped fast memory may evict a page before the rest of the fast memory is full. However, since caches with limited associativity are so common, it is of interest to explore this special case.

The following theorem states that direct mapping is competitive with the optimal offline strategy for the pessimal-cache problem.

Theorem 11.7 *Direct-mapping is k-competitive, where k is the fast-memory size of the both be the direct-mapping and offline strategies.*

Proof. We claim that if a particular page v_i is requested many times and the offline strategy incurs a page fault on ℓ of these requests, then direct mapping incurs at least ℓ/k page faults on v_i in expectation. We prove this claim by induction on the number of requests to v_i.

The first time that v_i is requested, there is a page fault. If v_i is requested again immediately (without any interleaving page requests), then both strategies have the page in fast memory. If v_i is requested again after another page is requested, then the offline strategy may have a page fault. The

[4] In fact, the offline strategy can have a slightly smaller memory–with size $\lceil(1+1/c)(k_{on}-1)\rceil$–and we still attain the $(c+1)$-competitiveness.

direct-mapping strategy incurs a page fault with probability at least $1/k$, because at least one page v_j is requested between v_i requests, and this page v_j has a $1/k$ probability of evicting v_i from fast memory. □

11.4 Concluding remarks

For the pessimal-cache problem, randomization is necessary to achieve any competitive ratio, and the best competitive ratio without resource deaugmentation is k. In contrast, for the original paging problem, deterministic strategies can be k-competitive [181], and upper [3, 92, 151] and lower [92] bounds of $\Theta(\log k)$ exist for randomized strategies against oblivious adversaries.

In this chapter, competitive ratios are k or larger; is there some model in which the competitive ratio is smaller? Essentially, I am trying to get a better definition of reasonable strategies giving the adversary just the right amount of power. This concept is similar to many approaches for the original page-replacement problem — for example, the graph-theoretic approach [58] tries to better model locality. Unfortunately, the traditional approaches seem to have little impact for the pessimal-cache problem. For example, looking at access patterns matching a graph, little can be said even if the graph is just a simple line. Adding power like lookahead to the online strategy, on the other hand, trivializes the problem since the optimal offline strategy can be implemented with a lookahead of 1. It would be nice to come up with a more accurate model that allows us to beat k-competitiveness.

It is interesting that direct-mapped cache is optimally bad when the program shows no locality (i.e., as in a multiprogrammed environment). In this model, however, we cannot show anything about the badness of a 2-way (or, more generally, c-way) set-associative cache using LRU. In particular, the LRU subcomponent forces the cache to make the "right" choice for eviction, and the sequence ping-ponging between two pages is sufficient to guarantee no future misses.

One way of weakening the adversary is to restrict the definition of "reasonable" strategies by disallowing the eviction of the most (or perhaps the c most) recently used pages. This restriction forces the cache to model some small amount of locality, since, after all, that is the purpose of the cache. This modification of the problem has the nice property that it allows us to analyze the pessimal-cache problem for a c-way set-associative cache. In particular, a 2-way set-associative cache is roughly k^2-competitive for the pessimal-cache problem. This result appears to generalize for c-way set-associative caches as well.

It would nice to see if anything from this chapter applies to other problems, or generalizations of the paging problem, like the k-servers on a line problem, for example.

Chapter 12

Incremental Topological Ordering

Let $G = (V, E)$ be a directed acyclic graph (dag) with $n = |V|$ and $m = |E|$. We say that a total ordering $\prec$ on vertices V is a ***topological ordering*** if for every edge $(u, v) \in E$, we have $u \prec v$. Given a specific dag G, there are two well-known approaches for finding a topological ordering in $O(n + m)$, either by depth-first search, or by repeated deletion of vertices with no incoming edges.

This chapter addresses an incremental variant of this problem, which arises in a variety of contexts, including compilers [148, 164], deadlock detection [32], pointer analysis [166, 167], and incremental circuit evaluation [13]. This chapter represents joint work with Michael A. Bender, Seth Gilbert, and Robert E. Tarjan, most of which previously appeared in [46].

In the problem of incremental topological ordering, the goal is to maintain a topological ordering even as edges are added to the graph. Initially, the graph G is unknown; edges are added one at a time. After each edge addition, we must recalculate a valid topological ordering. More specifically, the goal is to maintain a data structure that supports two operations: (1) edge insertions, in which a new edge is added to the graph G; and (2) queries of the form: "Does u come before v in the topological ordering?"[1] In this chapter, as well as in previous work, queries are answered in $O(1)$ time; the key question is how fast can edge insertions be processed?

Prior work

The simplest solution to the problem of incremental topological ordering is to recalculate a new ordering after each edge insertion, resulting in $O(n(m + n))$ time for m edge insertions. In recent years, there have been several significant improvements. Marchetti-Spaccamela et al. [149] gave the first nontrivial solution, handling m insertions in a total of $O(mn)$ time. Alpern et al. [13] gave an algorithm that performs well in a greedy sense: given a topological ordering and an edge insertion, their algorithm performs (almost) the minimum amount of work possible to find a new topological ordering. (This form of analysis, however, says little about the total time to perform m edge insertions.) Katriel and Bodlaender [131] gave a variant of the algorithm introduced by Alpern et al., which they show to run in $O(\min\{m^{3/2}\log n, m^{3/2} + n^2 \log n\})$ time. They also showed significantly better bounds for graphs of bounded treewidth. Liu and Chao [142] gave a tighter analysis of the Katriel-Bodlaender algorithm, showing that it runs in $O(m^{3/2} + mn^{1/2}\log n)$ time. Kavitha and Mathew [133] gave a slightly better variant taking $O(m^{3/2} + m^{1/2} n \log n)$. Most recently, Haeupler et al. [113, 114] gave a variant of the Alpern et al. / Katriel-Bodlaender algorithm that runs in time $O(m^{3/2})$.

[1] Although there may be many valid topological orderings, all answers given by the data structure after the kth edge insertion/deletion must be consistent with *the same* topological ordering.

The algorithms mentioned thus far do no better than $O(n^3)$ for dense graphs where $m = \Theta(n^2)$. Ajwani et al. [11] gave the first improvement for dense graphs, exhibiting an algorithm that runs in $O(n^{2.75})$ time for any number of edge insertions. Haeupler et al. [113, 133] improved this algorithm, resulting in a simpler algorithm requiring only $O(n^{2.5})$ time. This bound is not known to be tight, however, so this algorithm's true running time could potentially match our own.

Pearce and Kelly [166] gave an algorithm that they showed to be fast in practice on random sparse graphs, but that is provably worse than that Alpern et al. algorithm in the worst case. Ajwani and Friedrich [10] proved that the Alpern et al., Katriel and Bodlaender, and Pearce and Kelly algorithms all take expected time $O(n^2\text{polylog}(n))$ for edges forming a complete graph inserted in a random order.

The only nontrivial general lower bound that I am aware of is an $\Omega(n \log n)$ lower bound for $n-1$ edge insertions due to Ramalingam and Reps [172]. Katriel [130] gives an $\Omega(n^2)$ lower bound when $m = O(n)$ for algorithms that explicitly maintain the rank of each vertex in the topological order. Of known algorithms, this lower bound only applies to those algorithms that store the topological ordering in an array — those of Marchetti-Spaccamela et al., Pearce and Kelly, and Ajwani et al. It does not apply to the algorithms in this chapter, nor does it apply to any of the sparse-graph algorithms. Haeupler et al. [114] give an $\Omega(nm^{1/2})$ lower bound for algorithms that only update the "affected region" of the topological ordering on an edge insertion. This lower bound applies to all previous algorithms, but it does not apply to the algorithm given in this chapter.

Contributions

This chapter includes a new algorithm that takes $O(n^2 \log n)$ time to support any number of edge insertions. The analysis is tight in that there exist graphs and edge-insertion sequences causing our algorithm to run in $\Theta(n^2 \log n)$ time. This bound beats the $O(n^{2.5})$ bound of Haeupler et al. [113, 133] and beats the $O(m^{3/2})$ bound of Haeupler et al. [113, 114] whenever $m \geq n^{4/3} \log^{2/3} n$. I include an analysis of this algorithm in the RAM model (as is the case for all prior algorithms).

The approach here is quite different from previous algorithms. Typically, a topological ordering is maintained explicitly as either a linked list or an array. When adding an edge (u, v), the algorithm first checks whether u appears before or after v in the existing topological ordering; if u appears *after* v, then the array or linked list is updated so that u precedes v in the ordering, as is required by the insertion of edge (u, v). During the insertion, the algorithm modifies only vertices in the "affected region" of the list/array, i.e., those vertices that lie between v and u. The key to these algorithms is to efficiently discover which vertices in the affected region need to be moved.

This chapter's algorithm, by contrast, does not maintain an array or linked list, but instead assigns a *label* to each vertex in the graph, reassigning labels as edges are inserted.[2] The labels induce a topological ordering on the dag and are also used to assist in efficient updates during edge insertions.

Our data structure can be readily extended in various ways. First, it can be augmented to support additional operations, such as queries of the form: "What is u's successor (or predecessor) in the topological ordering?" (Such queries are supported by most previous algorithms.) Second, it can be augmented to detect cycles in the graph G; we assume throughout this paper that the graph G is acyclic; however, with a small amount of bookkeeping we can detect anomalous graphs. Finally, our data structure can readily support edge deletions as well as edge insertions; however, performance

[2] Previous linked-list-based algorithms implicitly use labels to perform queries as part of an order-maintenance data structure [33, 81]. By contrast, our labels have semantic meaning within the graph itself and play a key role in determining which vertices to update.

guarantees apply only to executions consisting only of insertions. We comment on these extensions in Section 12.2.

Chapter outline

The remainder of this chapter is organized as follows. First, Section 12.1 presents some preliminary definitions, along with an overview of our approach for maintaining a topological ordering. As a simple example of this approach, Section 12.1 provides an algorithm that achieves $O(mn)$ running time. Next, Section 12.2 presents a new algorithm in more detail that achieves the $O(n^2 \log n)$ bound for maintaining the data structure. Finally, Section 12.3 includes and analysis of the algorithm, and Section 12.4 contains some concluding remarks.

12.1 Basic strategy

This section describes a basic strategy for maintaining a topological ordering. This strategy can be applied in a simple fashion to achieve an $O(mn)$ algorithm for n vertices and m edge insertions. While this bound is not new, it demonstrates a quite different approach. Section 12.2 shows how this same basic approach can be more carefully applied to develop a more efficient algorithm that yields running time $O(n^2 \log n)$.

Terminology

Given a dag $G = (V, E)$, we say that a node u is a ***predecessor*** of a node v (and that v is a ***successor*** of u), if there is a directed path from u to v in G. According to this definition, a node u is a predecessor (and successor) of itself. When the edge (u, v) exists, we say that u is an ***immediate predecessor*** of v (and v is an immediate successor of u).

Labels and orders

The main idea of the approach described in this chapter is to associate with each vertex u an (integer) label $L(u)$. These labels imply a total order in the natural way: if $L(u) < L(v)$, then $u \prec v$. If $L(u) = L(v)$, then break the tie in an arbitrary but consistent fashion, say, using the unique identifiers of the vertices in question.

The ordering induced by the labels is a topological ordering as long as the labels are consistent with the underlying dag. That is, whenever (u, v) is an edge in E, we have $L(u) < L(v)$.

Labels are updated dynamically as edges are added to the graph. When adding edge (u, v), update the label at v, if necessary, along with some subset of v's successors. The first step involves determining whether the label of v needs to be increased. We refer to this process as ***visiting*** v. If v's label needs to be increased, a new value of the label is chosen. After visiting a node v, perform a truncated depth-first search on the dag, starting at v and visiting successors of v in turn. When traversing an edge in the dag, we say that we are ***following*** the edge. Choosing when to follow edges and how to update labels is at the heart of achieving an efficient algorithm.

A simple algorithm

As an example of how to apply this paradigm, consider a simple algorithm in which the label $L(v)$ represents the ***depth*** of v, where depth is defined as usual: if v has no immediate predecessors, then $L(v) = 0$; otherwise, $L(v) = \max_{u:(u,v)\in E} L(u) + 1$. If each node is labelled with its depth, it is

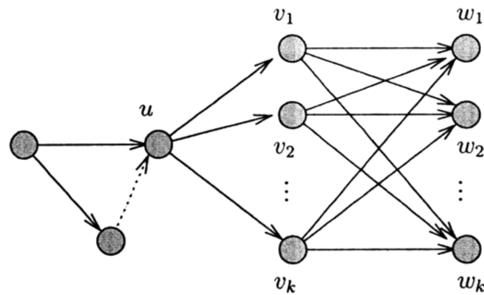


Figure 12-1: When the dotted edge is added, the depth of u increases by 1. The simple algorithm performs $\Theta(k^2)$ work, following edges (v_i, w_j) for all $1 \leq i, j \leq k$.

clear that the labelling induces a topological ordering: if u precedes v in the dag G and $u \neq v$, then $L(u) < L(v)$.

Initially, as there are no edges in the graph G, $L(u) = 0$ for every vertex u. Every time an edge is added to the dag, the labels on the vertices are updated to reflect the changes in depth caused by the new edge. Specifically, when adding edge (u, v), compare the labels at u and v: if $L(u) < L(v)$, then the depth of v remains unchanged. If, however, $L(u) \geq L(v)$, then update v's label: $L(v) \leftarrow L(u) + 1$. Then recursively follow all of v's outgoing edges, recursively performing the same "update label/follow edges" procedure at each of v's immediate successors. When this depth-first traversal terminates, each vertex is labelled with its depth in the graph.

There is a straightforward analysis of this algorithm. For each vertex u, the maximum depth is $n-1$, and hence $L(u) \leq n-1$. Since labels/depth are nondecreasing, it follows that $L(u)$ increases at most $n-1$ times, and thus the total cost of updating labels is $O(n^2)$. The remaining cost comes from following edges. (Recall that we may sometimes follow an edge (v, w) but not update the label at w, as it is already sufficiently large.) Notice that the algorithm follows the edge (v, w) only when (1) (v, w) is added, or (2) v's label increases. Since the label at v increases at most $n-1$ times, each edge can be followed at most n times. We conclude that the total cost to insert m edges is $O(mn)$.

12.2 Algorithm for dense graphs

This section describes an $O(n^2 \log n)$ algorithm for incremental topological sort, which is based on the basic strategy described in Section 12.1. The analysis appears in Section 12.3.

The example in Figure 12-1 demonstrates a shortcoming of the simple $O(mn)$ algorithm. Consider a bipartite clique with vertices $v_1, v_2, \ldots, v_k$ and $w_1, w_2, \ldots, w_k$, and an additional source vertex u that has a directed edge to every v_i. Consider an execution of the simple algorithm. When the depth of u increases, the algorithm follows each outgoing edge (u, v_i), and the label of each v_i also increases. Since v_i's label increases, the algorithm also follows each edge (v_i, w_j). Thus, whenever the depth of u increases, the algorithm follows *all of the* $\Theta(k^2)$ edges in the clique. Setting $k = \Theta(n)$ and increasing the depth of u by one $\Theta(n)$ times yields a total cost of $\Theta(n^3)$.

The main goal of this section is to reduce the number of times that any particular vertex w is visited, specifically, to bound the number of visits by $O(n \log n)$. (By contrast, the simple algorithm may visit each node up to $\Omega(n^2)$ times.)

Key modifications

In order to achieve better performance, we make two key modifications to the simple algorithm. First, do not blindly follow all outgoing edges from v whenever v's label increases. Instead, for each edge (v, w), cache the value of w's label at v and only follow the edge (v, w) if the cached information indicates that w's label needs to increase. Specifically, associate with each edge (v, w) the value $\mathit{cache}(v, w)$, which records the label of w as of the last time the algorithm followed edge (v, w). Since labels are nondecreasing, $\mathit{cache}(v, w) \leq L(w)$. Thus, do not follow (v, w) unless $L(v) \geq \mathit{cache}(v, w)$.

This first improvement alone does not improve the worst-case running time of the simple algorithm, as exhibited again by the bipartite-clique example in Figure 12-1: after each edge insertion completes, the cache at each edge leaving v_i correctly records the depth of each node w_j; however, when v_i increases by 1 on the next edge insertion, we must again follow all outgoing edges from v_i.

The second modification is to use a more aggressive label-update rule. If we could update the label of a node by a larger quantity (instead of incrementing it), then we could cache that larger value and avoid unnecessary visits. However, we want to avoid increasing the label too much, which may result in labels growing too big.

Thus, we no longer constrain the label of a node to represent its depth, as it is expensive to maintain a node's precise depth. Rather, consider the label $L(v)$ to approximate the total number of predecessors (not just immediate) of v. (Notice that the depth of a node is a very loose lower bound on the total number of predecessors.) That is, we assign a label $L(v)$ such that $0 \leq L(v) \leq \mathit{Pred}(v) < n$, where $\mathit{Pred}(v)$ is the total number of predecessors of v. (Of course, the labels also must induce a valid topological order.)

When increasing v's label, the magnitude of the increase depends on the number of immediate predecessors known to have large labels: the more predecessors of v that have large labels, the higher the increase in label of v. This strategy captures the intuition that if a vertex has many immediate predecessors with large labels, then it most likely has a very large number of predecessors (immediate or otherwise) in the dag. By increasing the label significantly, the number of updates is reduced.

The dense algorithm

As previously noted, the improved dense-graph algorithm maintains the value $L(v)$ (initially 0), with $0 \leq L(v) < n$. Associate with each edge (u, v) the value $\mathit{cache}(u, v)$, which stores the (old) value of $L(v)$ when the algorithm last followed edge (u, v). (Note that the cached value may be out-of-date, as $L(v)$ may have increased since then). Organize all outgoing edges into a data structure $\mathit{outgoing}(v)$, which will be described later in this section.

Also associate with v a collection of counters $N(v, j)$ and corresponding labels $\Lambda(v, j)$, for each $j \,:\, 0 \leq j < \lg(n)$, all initially 0. The jth counter $N(v, j)$ satisfies the following invariant: $0 \leq N(v, j) \leq 2^{j+2}$. The label $\Lambda(v, j)$ stores the value of v's label when $N(v, j)$ was last reset to 0. The $N(v, j)$ counter counts the number of incoming edges (u, v) that, when last followed, had $2^{j-1} < L(v) - L(u) \leq 2^j$.

I now describe the algorithm in more detail. When inserting the edge (u, v) into the dag, proceed as follows: (1) initialize $\mathit{cache}(u, v) \leftarrow 0$; (2) insert (u, v) into $\mathit{outgoing}(u)$;, and (3) follow the edge (u, v) as described by $\text{FOLLOW}(u, v)$ in Figure 12-2. This procedure updates v's label and other state, and then (selectively) calls $\text{FOLLOW}(v, *)$ recursively on the ongoing edges of v.

When executing $\text{FOLLOW}(u, v)$, first check whether $L(u) < L(v)$. If not, increase $L(v)$ by 1 in line 2, thus ensuring that the resulting labelling induces a valid topological ordering.

State maintained by each node $u \in V$:

$L(u)$, the label of u.
$\forall j \in \{0, 1, \ldots, \lg n\}$: $N(u, j)$, a counter bounded by 2^{j+2}.
$\forall j \in \{0, 1, \ldots, \lg n\}$: $\Lambda(u, j)$, an old label of u.
$\forall v : (u, v) \in E$: *cache*(u, v), an old label of v.
outgoing(u), an array of outgoing edges.

```
FOLLOW(u, v)
 1  if L(u) ≥ L(v)
 2     then L(v) ← L(u) + 1
 3     else ▷ Check if v has sufficiently many immediate
            ▷ predecessors to increase its label.
 4          j ← ⌈lg(L(v) − L(u))⌉
 5          N(v, j) ← N(v, j) + 1
 6          if N(v, j) = 2^{j+2}
 7            then L(v) ← max(L(v), Λ(v, j) + 2^j)
 8                 N(v, j) ← 0
 9                 Λ(v, j) ← L(v)
10  if L(v) has increased:
11     then for each (v, w) such that cache(v, w) ≤ L(v)
12                do FOLLOW(v, w)
    ▷ Done following the edge.
    ▷ Update (v, w) in u's outgoing edge set.
13  cache(u, v) ← L(v)
14  Update (u, v) in outgoing(u).
```

Figure 12-2: Pseudocode for updating labels, when following an edge in the dag. The triangles denote comments. Each node v maintains four data structures: a label $L(v)$, a counter $N(v, j)$ and old label $\Lambda(v, j)$ for $0 \leq j < \lg(n)$, and an outgoing-edge set *outgoing*(v) organized by *cache*(v, w) — a cached version of w's label.

If, on the other hand, we already have $L(u) < L(v)$, then examine whether v should have its label increased due to its counts of immediate predecessors. This step is where the label is increased more aggressively than in Section 12.1.

First, find the magnitude of difference between $L(u)$ and $L(v)$ in line 4 — specifically, find the smallest j such that $L(v) - L(u) \leq 2^j$. Then, increment the corresponding counter $N(v, j)$. If the counter $N(v, j)$ reaches its maximum value of 2^{j+2}, then perform the "more aggressive" update of v's label, increasing $L(v)$ by up to $2^j - 1$ in line 7. Next, reset $N(v, j)$ to 0 and record the corresponding $\Lambda(v, j)$ in lines 8 and 9.

This label update may seem strange at this point, but Lemma 12.2 will show that when $\Lambda(v, j) + 2^j > L(v)$, v has at least 2^{j+1} distinct immediate predecessors with label at least $\Lambda(v, j) - 2^j$. Thus, v has at least $\Lambda(v, j) - 2^j + 2^{j+1}$ predecessors, and the label is updated accordingly.

At this point, if v's label increases (whether in line 2 or line 7), then follow all outgoing edges whose targets have cached labels that are $\leq L(v)$, i.e., targets that may need to have their label increased. Finally, when the algorithm has finished following outgoing edges, associate v's new

label $L(v)$ with $cache(u, v)$ and update the edge (u, v) in u's outgoing edge list.

Some comments on implementation details appear next.

Implementing the outgoing-edge set

The outgoing-edge set data structure is straightforward to implement. The variable $outgoing(v)$ is an n-slot array, each slot containing a linked list of outgoing edges. If $cache(v, w) = x$, then a pointer to w is stored in a linked list at slot x in the array. Updating an edge (i.e., in line 14) is easy in that it simply involves removing it from one linked list and adding it to another.

To determine which edges to follow (in line 11), follow all outgoing edges in array slots between v's old label and v's new label. After following each edge, the target's label (and hence the edge's cached label) has increased beyond v's current label $L(v)$. Thus, when $\textsc{Follow}(u, v)$ returns, there are no outgoing edges stored in array slots $0, \ldots, L(v)$. As a result, each slot of the array need be examined only once during the entire execution.

Calculating in the RAM model

Some machine models allow for a constant-time lg calculation, but this chapter does not require such an assumption. Since the algorithm only compute this logarithm for at most n different values (i.e., the difference in the possible labels), these values can be precomputed and stored in a size-n logarithm table with all the necessary entries. Even a naive algorithm for computing base-2 logarithms of $\lg n$-bit numbers requires only $O(\log n)$ time using only additions (repeated doubling) and comparisons. The $O(n \log n)$ time for pre-computing the entire table is dwarfed by the $O(n^2 \log n)$ time of the topological-ordering algorithm.

Similarly, one could precompute a table for calculating 2^{j+2} using only additions, thereby not requiring a bitshift operation in the machine model.

Supporting predecessor/successor queries

As described so far, our data structure supports only queries of the form, "Does u precede v in the topological ordering?" It does not (efficiently) support queries of the form, "What is the next vertex in the topological ordering after u?" These queries are easy to support without increasing the asymptotic running time.

Throughout the execution of the algorithm, maintain a linked list matching the topological ordering. To search into the linked list, also maintain a balanced search tree (BST) (see [72, Chapter 12]), ordered/keyed by label, where ties are broken by unique vertex identifiers. Initially the vertices are simply sorted by identifier. Whenever v's label increases, remove v from the tree and reinsert it with key equal to its new label. Query the BST for v's predecessor u, and move v from its current location in the linked list, instead locating v after its BST-predecessor u. Since labels have maximum value of $n - 1$ (shown later in Lemma 12.2), a node v is reinserted at most $n - 1$ times, resulting in n BST insertions and predecessor queries, as well as n linked-list moves, per vertex. Each BST operations has cost $O(\log n)$ (using a reasonable BST implementation), and each linked-list operation has constant cost. The total additive cost of $O(n^2 \log n)$ for the $O(n^2)$ BST operations has no effect on the asymptotic running time of our algorithm.

Detecting a cycle

Thus far, we have assumed that no edge insertion introduces a cycle to the graph. It is easy to modify our algorithm to detect the introduction of a cycle, although the performance bounds only

hold until the first cycle is detected; that is, we provide no mechanism for deleting an edge aside from rebuilding the entire data structure.

To detect whether the addition of (u, v) introduces a cycle, simply check whether u is ever visited during the depth-first search performed while following edges. If so, there is a cycle. The nodes in the cycle can be found by a separate graph search.

12.3 Analysis

This section analyzes the algorithm from Section 12.2. There are three key statements and proofs here. First, the ordering induced by the labels is, in fact, a topological order. Second, the labels are bounded by n. Lastly, the total cost of inserting m edges is $O(n^2 \log n)$, and this analysis to be tight. The key observation for this last theorem is that no edge is followed too many times.

Correctness of the topological ordering

This first theorem shows that the data structure maintains a valid topological ordering.

Theorem 12.1 *After completely processing each edge insertion, if* $((u, v)) \in E$, *then* $L(u) < L(v)$.

Proof. Initially, the labels trivially induce a good ordering since there are no edges in the dag. Assume, for the sake of contradiction, that after some edge insertion the theorem is violated. Consider the first edge insertion causing a violation, and let edge (u, v) be an edge for which $L(u) > L(v)$. During the edge insertion, the label $L(u)$ must have increased, thereby causing the violation. If we subsequently followed (u, v), then $L(v)$ would have increased beyond $L(u)$ (in line 2).

Suppose, therefore, that we did not follow (u, v) after the last increase to $L(u)$. It follows that $cache(u, v) > L(u)$, as the edge was not followed in lines 11–12. We know that the cached label $cache(u, v) \leq L(v)$, since $L(v)$ is nondecreasing. Hence, $L(v) \geq cache(u, v) > L(u)$, which is a contradiction. □

Bounded labels

This next lemma shows that the value of each label is bounded by n. (This fact also ensures that each label can be stored in a single word, which has been implicit throughout.)

Lemma 12.2 *For* $v \in V$, *let* $Pred(v)$ *be the total number of predecessors of* v. *Then at any point during the algorithm execution,* $L(v) \leq Pred(v) < n$.

Proof. We proceed by induction over the number of edges that have been followed. Initially, $L(v) = 0$, and the claim holds trivially. Assume the lemma is true before following the edge (u, v). We show that it holds after following the edge. If label $L(v)$ does not increase, then the claim follows immediately, as $L(v) \leq Pred(v)$ by inductive hypothesis.

Suppose instead that $L(v)$ increases while following (u, v). There are two places that v's label increases: line 2 and line 7 of Figure 12-2. In the first case, $L(v) \leftarrow L(u) + 1$. By inductive hypothesis, $L(u) \leq Pred(u)$, and we know that $Pred(u) < Pred(v)$ since the predecessors of v include all the predecessors of u plus the node v itself. (Recall that v is considered to be a predecessor of itself.) Thus, $L(v) = L(u) + 1 \leq Pred(u) + 1 \leq Pred(v)$, and the claim follows.

In the second case, $L(v)$ increases in line 7 as a result of $N(v, j)$ increasing to 2^{j+2}. We fix j for the remainder of this proof. First, we observe that $L(v)$ increasing here implies that (prior to the

update) $L(v) < \Lambda(v,j) + 2^j$. (Otherwise the label $L(v)$ would remain unchanged.) Consider the 2^{j+2} increases to $N(v,j)$ since the last time it was reset to 0.

We claim that each distinct edge contributed at most 2 such increases to $N(v,j)$. Suppose for the sake of contradiction that some edge (u,v) contributed 3 or more increases to $N(v,j)$, and consider the last 3 such increases. For $x \in \{u,v\}$, let $L_1(x)$, $L_2(x)$, and $L_3(x)$ be the values of x's label immediately prior to these 3 increases, respectively. Since the counter increases, $L_i(v) - L_i(u) > 2^{j-1}$. (Recall that this inequality follows from the choice of j, see line 4.)

Moreover, since $cache(u,v)$ is updated after each edge is followed, and since the edge is only followed if the label $L(u)$ exceeds the cached value of $L(v)$, we can conclude that $L_2(u) \geq L_1(v)$ and $L_3(u) \geq L_2(v)$. Combining these two facts, we conclude that $L_2(v) > L_2(u) + 2^{j-1} \geq L_1(v) + 2^{j-1}$ and $L_3(v) > L_3(u) + 2^{j-1} \geq L_2(v) + 2^{j-1}$, which together imply that $L_3(v) > L_1(v) + 2^j$. Finally, since the counter does not reset between these increases, and $\Lambda(v,j)$ represents v's label at the time of the previous reset, $L_1(v) \geq \Lambda(v,j)$. It follows that $L(v) \geq L_3(v) > \Lambda(v,j) + 2^j$, which is a contradiction.

We therefore conclude that each edge contributes at most 2 increases to $N(v,j)$. Since there have been 2^{j+2} increases when the counter resets (causing the label of v to be increased), we conclude that there are at least $2^{j+2}/2$ distinct edges contributing to $N(v,j)$. Fix some u such that (u,v) contributes to $N(v,j)$. Each time edge (u,v) contributes to the count, we have $L(u) \geq L(v) - 2^j$ (again, by the way in which j was chosen on line 4). Thus, since $L(v) \geq \Lambda(v,j)$, we know that $L(u) \geq \Lambda(v,j) - 2^j$.

Thus, v has at least 2^{j+1} distinct immediate predecessors with label at least $\Lambda(v,j) - 2^j$. We call these vertices the "contributing predecessors."

Let x be a topologically earliest contributing predecessor. By inductive hypothesis, we have $Pred(x) \geq L(x)$. All of x's predecessors are predecessors of v. Moreover, none of the other contributing predecessors, nor v itself, are predecessors of x. Thus, $Preds(v) \geq Preds(x) + 2^{j+1} \geq L(x) + 2^{j+1} \geq (\Lambda(v,j) - 2^j) + 2^{j+1} = \Lambda(v,j) + 2^j$. Noting that the label $L(v)$ is increased to $\Lambda(v,j) + 2^j$ completes the proof. □

Complexity analysis of insertions

Let us now consider the total cost of m edge insertions, showing that the total cost is $O(n^2 \log n)$. The key observation is that no vertex is visited too many times, specifically, more than $O(n \log n)$ times. The main idea of the proof is the amortize the cost of following an edge directed towards v against the number of times that the label of v is increased.

Lemma 12.3 *In every execution, for every $v \in V$, vertex v is visited at most $O(n \log n)$ times.*

Proof. Whenever visiting v, either $L(v)$ increases (line 2), or $N(v,j)$ increases for some j (line 5). The former occurs at most $n-1$ times over the course of the algorithm.

For the latter case, consider the jth counter. Whenever the counter reaches 2^{j+2} it resets. Let $\Lambda_i(v,j)$ denote the value of v's label associated with the ith reset. Observe $\Lambda_{i+1}(v,j) \geq \Lambda_i(v,j) + 2^j$ due to the update in line 7, or more generally $\Lambda_i(v,j) \geq i2^j$. Since $n > L(v) \geq \Lambda(v,j)$, it follows that the maximum value of i here is $\lfloor (n-1)/2^j \rfloor$, and hence $N(v,j)$ can be increased at most

$$2^{j+2}\left(\lfloor (n-1)/2^j \rfloor + 1\right) \leq \frac{2^{j+2}n}{2^j} + 2^{j+2} \leq 4n + 2^{j+2}$$

times. Summing over all $O(\log n)$ values of j gives $O(n \log n)$ counter increments and hence visitations of v. □

Combining Lemma 12.3 with a cost analysis of each call to FOLLOW yields the total running time:

Theorem 12.4 *The total running time to perform up to m edge insertions is $O(n^2 \log n)$.*

Proof. To reach this bound, we calculate the cost of each call to FOLLOW, and multiply by the total number of times that any node is visited. Ignoring lines 11–12, each step of FOLLOW has constant cost. The only remaining cost is due to finding edges to follow in the outgoing-edge data structure. The outgoing-edge data structure is a size-n array of linked lists. We visit each array cell only once over the entire course of the algorithm, for an aggregate cost of $O(n)$. We charge the $O(1)$ cost of traversing the linked list against the outgoing edge followed. Multiplying the array-traversal cost for each outgoing-edge data structure by n vertices yields a total cost for our algorithm of $O(n^2 + F)$, where F is the total number of edge followings. Applying Lemma 12.3 completes the proof. □

Finally, the following theorem shows that the analysis is tight.

Theorem 12.5 *For any sufficiently large n, there exists a sequence of $\Theta(n^2)$ edge insertions on an n-vertex dag that causes our algorithm to follow $\Omega(n^2 \log n)$ edges.*

Proof. Without loss of generality, suppose $n = 3k - 4$, where $k \geq 2^3$ is a power of 2. The graph we construct consists of three categories of vertices: (1) vertices $u_0, u_1, \ldots, u_{k-1}$, (2) sets of vertices $S_0, S_1, \ldots, S_{\lg(k)-3}$ with $|S_j| = 2^{j+2}$ (so $\sum_j |S_j| = k - 4$), and (3) a set of vertices T with $|T| = k$. Initially there are no edges in the graph, and all labels are 0.

First, add edges (u_i, u_{i+1}) in order for $0 \leq i < k - 1$. After these edge additions, $L(u_i) = i$. These labels are invariant over the remainder of the edge insertions — we use these vertices as anchors to increase the labels of all the other vertices. In fact, the *only* time the labels of any other vertex $v \in (\bigcup_j S_j) \cup T$ will increase is when adding an edge (u_i, v).

The edge insertions proceed in phases ranging from 1 to k. In phase i, first insert edge (u_{i-1}, t) for all $t \in T$, thereby increasing $L(t)$ such that $L(t) = i$. Next, consider each j for which i is a multiple of 2^j. There are two cases.

Case 1: If $i = 2^j$, add edges (s_j, t) for all $s_j \in S_j$ and $t \in T$. Observe that before the edge addition, $N(t, j) = 0$, $\Lambda(t, j) = 0$, and $L(s_j) = 0 = L(t) - 2^j$. After the 2^{j+2}th edge insertion, $N(t, j)$ reaches 2^{j+2}. We have, however, that $L(t) \geq \Lambda(t, j) + 2^j$, and hence $L(t)$ does not increase. The counter $N(t, j)$ is subsequently reset to 0, and $\Lambda(t, j) \leftarrow L(t) = 2^j$. Finally, $\mathit{cache}(s_j, t) \leftarrow 2^j$ as well.

Case 2: Otherwise, $i \geq 2 \cdot 2^j$, and the edges (s_j, t) already exist. Instead, insert edges (u_{i-2^j-1}, s_j), for all $s_j \in S_j$. This edge insertion causes $L(s_j)$ to increase to the next multiple of 2^j. After the update, we have $L(s_j) = \mathit{cache}(s_j, t) = \Lambda(t, j) = i - 2^j$, and hence all edges (s_j, t) are followed. The counter $N(t, j)$ again resets to 0, $\Lambda(t, j) \leftarrow L(t) = i$, and finally $\mathit{cache}(s_j, t) \leftarrow i$.

In both cases, whenever the phase number i is a multiple of 2^j, we follow all edges (s_j, t) for all $s_j \in S_j$ and $t \in T$. Consider a fixed j. There are $|S_j| \cdot |T| = 2^{j+2}k$ such edges. Summing over all $k/2^j$ phases during which the phase number is a multiple of 2^j, there are $(2^{j+2}k)(k/2^j) = 4k^2 = \Omega(n^2)$ edge followings from vertices in S_j to vertices in T. Summing over all $\lg(k) - 2 = \Theta(\log n)$ values of j yields a total of $\Omega(n^2 \log n)$ edge followings. □

12.4 Concluding remarks

This chapter has shown how to solve the problem of incremental topological ordering where the total cost of inserting m edges into a graph containing n vertices is $O(n^2 \log n)$. It supports order queries of the form "Does u come before v in the topological ordering?" in $O(1)$ time, and, with

minor modifications, it can support successor/predecessor queries in $O(1)$ time. For dense graphs where $m \geq n^{4/3} \log^{2/3} n$, this algorithm is the most efficient to date.

As presented, the algorithm requires $O(n^2)$ space (in terms of machine words, not bits); using a priority queue to manage the outgoing edges should reduce the space to $O(m+n \log n)$ but increase the running time to $O(n^2 \log^2 n)$.

The major open question is whether the new techniques introduced in this paper can yield improvements in the sparse case. For our algorithm, there exists an instance of $n - 1$ edge insertions that requires $\Omega(n^2)$ work. For example, always adding edges to the front of a chain results in relabelling every node in the chain on every edge insertion. It would be interesting, however, if some variant of our approach leads to a good algorithm for sparse graphs.

Chapter 13

Conclusions

In this thesis, I have presented many algorithms and techniques for designing provably good parallel and cache-efficient algorithms. As problem-specific conclusions occurred at the end of each chapter, this chapter is reserved for more general discussions.

Many of the problems and solutions I presented are motivated by issues that arise in designing a good parallel-programming infrastructure. The goal in Chapters 2–3, for example, is to make dealing with concurrent accesses to data a bit easier for the programmer. Similarly, the scheduling algorithms in Chapters 4 and 5 should be incorporated into a runtime system and ignored by the average programmer. Even the cache-oblivious dictionaries in Chapters 8–10 and CSB multiplication algorithm in Chapter 6 can be rolled into libraries that may be incorporated into larger systems.

One of the main problems I addressed is concurrency control. Both Chapters 2 and 3 as well as Chapter 5 addressed this issue to some degree. Many questions remain. How can we analyze programs that use locks/transactions in general? Chapters 2, 3, and 8 all gave algorithms that use locks and attain provably good performance, but what about other algorithms? Is there a general methodology for locking that guarantees good performance? There are several different approaches in this thesis. For SP-maintenance and conflict detection, I limited the degree of locking. For CSB matrix-vector multiplication, I duplicated data to avoid contention. The concurrent CO B-trees, on the other hand, incorporated randomization. All of these focus on structuring an algorithm to reduce the cost of the concurrent accesses; what about analyzing an arbitrary class of programs (say fork-join programs) that use locks?

Another theme throughout this thesis has been amortization, which occurs in many of the algorithms. For both the SP-maintenance and conflict detection problems, I incorporated serial data structures into a shared, parallel data structure. These serial data structures included some amortized element, which presented a challenge as it seems that a slow amortized operation occurring at the wrong time could effectively increase the span of the computation. To compensate, the solutions I presented were able to bypass the amortization at certain points. I suspect, however, that this bypass is unnecessary, at least for these types of algorithms. Can we prove that these slow amortized steps do not, in fact, increase the runtime?

Amortization remains a powerful tool in sequential algorithms, but their effectiveness is still unclear in parallel programs. Yes, this thesis did include a couple of examples, but I did not solve the general challenge of parallelizing amortized codes. The first solution we were toying with for SP-maintenance, for example, was to take the SP-order code and simply parallelize the underlying order-maintenance data structure. This approach proved ineffective as it seemed possible to construct input computation dags that would cause high contention on a small region in the data structure. All I can say, however, is that what we tried did not work; we did not prove any lower

bounds on parallelizing an order-maintenance data structure.

Finally, I addressed several variants of cache-oblivious dictionaries in this thesis. The two main contributions were adding concurrency and exploring the cost tradeoff on operations. I am curious to see whether any of these techniques can be incorporated into other cache-oblivious data structures.

Bibliography

[1] Serge Abiteboul, Haim Kaplan, and Tova Milo. Compact labeling schemes for ancestor queries. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 547–556. Society for Industrial and Applied Mathematics, 2001.

[2] N. Abramson. The ALOHA system — another alternative for computer communications. In *Proceedings of AFIPS FJCC*, volume 37, pages 281–285, 1970.

[3] Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. In *Annual European Symposium on Algorithms (ESA)*, volume 1136 of *Lecture Notes in Computer Science*, pages 419–430, Barcelona, Spain, September 25-27 1996.

[4] Michael D. Adams and David S. Wise. Seven at one stroke: results from a cache-oblivious paradigm for scalable matrix algorithms. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 41–50, New York, NY, USA, 2006. ACM.

[5] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, Jun 2006.

[6] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[7] Kunal Agrawal, Michael A. Bender, and Jeremy T. Fineman. The worst page-replacement policy. *Theory of Computing Systems*, 44(2):175–185, 2009.

[8] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 163–174, Salt Lake City, Utah, USA, February 20–23 2008.

[9] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, October 2006. In conjunction with *International Conference on Architectutal Support for Programming Languages and Operating Systems.*

[10] Deepak Ajwani and Tobias Friedrich. Average-case analysis of online topological ordering. In Takeshi Tokuyama, editor, *ISAAC*, volume 4835 of *Lecture Notes in Computer Science*, pages 464–475. Springer, 2007.

[11] Deepak Ajwani, Tobias Friedrich, and Ulrich Meyer. An $O(n^{2.75})$ algorithm for online topological ordering. In *Proceedings of the 10th Scandinavian Workshop on Algorithm Theory*, volume 4059 of *Lecture Notes in Computer Science*, pages 53–64. Springer, 2006.

[12] Eric Allen, David Chase, Joe Hllett, Victor Luchango, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hoch stadt. The Fortress language specification, version 1.0 β. Technical report, Sun Microsystems, Inc., March 2007.

[13] Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, 1990.

[14] Stephen Alstrup, Philip Bille, and Theis Rauhe. Labeling schemes for small distances in trees. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 689–698, 2003.

[15] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 258–264. ACM Press, 2002.

[16] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Direct routing on trees. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 342–349, 1998.

[17] Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 947–953. Society for Industrial and Applied Mathematics, 2002.

[18] C. Scott Ananian, Krste Asanovic, Bradley q C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. *IEEE Micro*, 26(1):59–69, January 2006.

[19] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[20] Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 370–380, 1991.

[21] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 135–141, 1996.

[22] Arne Andersson and Ola Petersson. Approximate indexed lists. *Journal of Algorithms*, 29:256–276, 1998.

[23] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual Symposium on Theory of Computing (STOC)*, pages 335–342, Portland, Oregon, May 2000.

[24] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the*

34th Annual ACM Symposium on Theory of Computing (STOC), pages 268–276, Montréal, Canada, Québec. Canada, May 2002.

[25] Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 560–569, Burlington, VT, October 1996.

[26] Marta Arias, Lenore J. Cowen, and Kofi A. Laing. Compact roundtrip routing with topology-independent node names. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, pages 43–52. ACM Press, 2003.

[27] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, New York, NY, USA, June 1998. ACM Press.

[28] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computering research: A view from berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley, December 2006.

[29] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse. HPCS scalable synthetic compact applications #2. Version 1.1.

[30] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972.

[31] Laszlo A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[32] F. Belik. An efficient deadlock avoidance technique. *IEEE Transactions. on Computers*, 39(7), 1990.

[33] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[34] M. A. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proceedings of the 29th International Colloquium on Aotomata, Languages and Programming (ICALP)*, pages 195–207, 2002.

[35] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[36] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th Annual Symposium on Discrete Algorithms*, pages 29–38, 2002.

[37] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

[38] M. A. Bender, M. Farach-Colton, B. C. Kuszmaul, and Jim Sukha. Cache-oblivious B-trees for optimizing disk performance. Manuscript., 2005.

[39] 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. In *Proceedings of the 44th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 271–282, Cambridge, Massachusetts, October 2003.

[40] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 139–151, Rome, Italy, September 2002.

[41] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[42] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan Fogel, Bradley Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 81–92, San Diego, CA, USA, June 9–11 2007.

[43] Michael A. Bender, Martin Farach-Colton, Simai He, Bradley C. Kuszmaul, and Charles E. Leiserson. Adversarial contention resolution for simple channels. In *17th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 325–332, 2005.

[44] Michael A. Bender, Martin Farach-Colton, and Bradley Kuszmaul. Cache-oblivious string B-trees. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2006. To appear.

[45] Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert. Contention resolution with heterogeneous job sizes. In *Proceedings of the 14th Annual European Symposium on Algorith ms (ESA)*, pages 112–123, Zürich, Switzerland, September 2006.

[46] Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert. A new approach to incremental topological ordering. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1108–1115, 2009.

[47] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proceedings of the Seventeenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, Las Vegas, NV, USA, July 17–20 2005.

[48] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 133–144, Barcelona, Spain, June 27–30 2004.

[49] Jon Louis Bentley and James B. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[50] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[51] Guy E. Blelloch, Rezaul Alam Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 501–510, San Francisco, California, January 20–22 2008.

[52] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Brief announcement: Low depth cache-oblivious sorting. In *Proceedings of the Twenty-First Annual ACM Symposium on Parallel Algorthms and Architectures (SPAA)*, pages 121–123, Calgary, Canada, August 11–13 2009.

[53] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP*, pages 213–225, 1996.

[54] 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, August 25 1996.

[55] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[56] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, University of Texas at Austin, 1999.

[57] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov 2006.

[58] Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, April 1995.

[59] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, Baltimore, Maryland, May 2003.

[60] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th Annual Symposium on Discrete Algorithms (SODA)*, pages 39–48, San Francisco, California, January 2002.

[61] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 859–860, San Francisco, California, January 2000.

[62] Aydın Buluç and John R. Gilbert. On the representation and multiplication of hypersparse matrices. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, pages 1–11, April 2008.

[63] Umit Catalyurek and Cevdet Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS)*, page 118, Washington, DC, USA, 2001. IEEE Computer Society.

[64] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[65] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June 1998.

[66] Junghoo Cho, Hector Garcia-Molina, Taher Haveliwala, Wang Lam, Andreas Paepcke, Sriram Raghavan, and Gary Wesley. Stanford webbase components and applications. *ACM Transactions on Internet Technology*, 6(2):153–186, 2006.

[67] Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 71–80, San Diega, California, June 9–11 2007.

[68] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the Twentieth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 207–216, Munich, Germany, June 14–16 2008.

[69] Cilk Arts, Inc., Burlington, MA. *Cilk++ Programmer's Guide*, 2009. Available from `http://www.cilk.com/`.

[70] D. Comer. The ubiquitous B-Tree. *Computing Surveys*, 11:121–137, 1979.

[71] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[72] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[73] Christopher Y. Crutchfield, Zoran Dzunic, Jeremy T. Fineman, David R. Karger, and Jacob Scott. Improved approximations for multiprocessor scheduling under uncertainty. In *SPAA*, pages 246–255, 2008.

[74] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference*, pages 157–172, New York, NY, USA, 1969. ACM.

[75] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luch angco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[76] Timothy A. Davis. University of Florida sparse matrix collection. *NA Digest*, 92, 1994.

[77] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2006.

[78] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[79] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 122–127, May 1982.

[80] Paul F. Dietz, Joel I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Algorithm Theory—SWAT '94: 4th Scandinavian Workshop on Algorithm Theory*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142. Springer-Verlag, July 6–8 1994.

[81] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, New York City, May 1987.

[82] Paul F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*. Springer, July 11–14 1990.

[83] Edsger Wybe Dijkstra. Cooperating sequential processes. Technical Report EWD-123, 1965.

[84] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 1–10. ACM Press, 1990.

[85] Jack Dongarra. Sparse matrix storage formats. In Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors, *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM, 2000.

[86] Jack Dongarra, P. Koev, and X. Li. Matrix-vector and matrix-matrix multiplication. In Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors, *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM, 2000.

[87] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, New York, 1986.

[88] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. X10: an experimental language for high productivity programming of scalable systems. In *Proceedings of Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2005. In conjunction with *Symposium on High Performance Computer Architecture (HPCA)*.

[89] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package I: The symmetric codes. *International Journal for Numerical Methods in Engineering*, 18:1145–1151, 1982.

[90] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[91] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 22–25 1997.

[92] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, December 1991.

[93] Jeremy T. Fineman. Provably good race detection that runs in parallel. Master's thesis, Massachusetts Institute of Technology Department of Electrica l Engineering and Computer Science, August 2005.

[94] Klaus Finkenzeller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification.* John Wiley & Sons, second edition, 2003. E-book at `books24x7.com`.

[95] L.R. Ford and D.R. Fulkerson. *Flows in networks.* Princeton University Press Princeton, NJ, 1962.

[96] Simon French. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-shop.* Ellis Horwood, 1982.

[97] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2009, to appear)*, Calgary, Canada, August 2009.

[98] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, New York, New York, October 17–19 1999.

[99] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[100] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–280, Cambridge, Massachusetts, July 30 – August 2 2006.

[101] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, April 1985.

[102] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 210–219. Society for Industrial and Applied Mathematics, 2001.

[103] Alan George and Joseph W. Liu. *Computer Solution of Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[104] Mihály Geréb-Graus and Thanasis Tsantilas. Efficient optical communication in parallel computers. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 41–48, 1992.

[105] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[106] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl*, 13:333–356, 1991.

[107] Leslie Ann Goldberg, Mark Jerrum, Tom Leighton, and Satish Rao. Doubly logarithmic communication algorithms for optical-communication parallel computers. *SIAM Journal on Computing*, 26(4):1100–1119, August 1997.

[108] Leslie Ann Goldberg, Yossi Matias, and Satish Rao. An optical simulation of shared memory. *SIAM Journal on Computing*, 28(5):1829–1847, October 1999.

[109] Ronald Graham, Eugene L. Lawler, Jan Karel Lenstra, and Alexander R. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[110] Albert G. Greenberg, Philippe Flajolet, and Richard E. Ladner. Estimating the multiplicities of conflicts to speed their resolution in multiple access channels. *JACM*, 34(2):289–325, April 1987.

[111] Albert G. Greenberg and Shmuel Winograd. A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *JACM*, 32(3):589–596, July 1985.

[112] Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. *Advances in Computing Research*, 5:345–374, 1989.

[113] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E. Tarjan. Faster algorithms for incremental topological ordering. In *Proceedings of the 35th International Colloquium on Automata, Languages, and Programming*, July 2008.

[114] Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. Incremental topological ordering and strong component maintenance. *CoRR*, abs/0803.0792, 2008.

[115] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[116] Jim Handy. *The Cache Memory Book*. Academic Press, second edition, 1998.

[117] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, Third edition, 2003.

[118] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of ACM Symposium on Principles of Distributed Computin g (PODC)*, pages 92–101, 2003.

[119] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 289–300, San Diego, California, 1993.

[120] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[121] 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, 2004.

[122] Sandy Irani. Competitive analysis of paging. In *Developments from a June 1996 Seminar on Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 52–73, London, UK, 1998. Springer-Verlag.

[123] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In S. Even and O. Kariv, editors, *Proceedings of the 8th Colloquium on Automata, Languages, and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Acre (Akko), Israel, July 13–17 1981.

[124] P. Jayanti. A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, volume 1499 of *LNCS*, 1998.

[125] Ari Juels, Ronald L. Rivest, and Michael Szydlo. The blocker tag: Selective blocking of RFID tags for consumer privacy. In *Conference on Computer and Communications Security*, pages 103–111, 2003.

[126] Haim Kaplan, Tova Milo, and Ronen Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 954–963. Society for Industrial and Applied Mathematics, 2002.

[127] David R. Karger, Clifford Stein, and Joel Wein. Scheduling algorithms. *CRC Handbook of Computer Science*, 1997.

[128] Zardosht Kasheff. Cache-oblivious dynamic search trees. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2004.

[129] Irit Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Isreal Inst. of Tech., Haifa, May 2002.

[130] Irit Katriel. On algorithms for online topological ordering and sorting. Technical Report MPI-I-2004-1-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[131] Irit Katriel and Hans L. Bodlaender. Online topological ordering. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 443–450, Vancouver, British Columbia, Canada, January 2005.

[132] Michal Katz, Nir A. Katz, Amos Korman, and David Peleg. Labeling schemes for flow and connectivity. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 927–936, 2002.

[133] Telikepalli Kavitha and Rogers Mathew. Faster algorithms for online topological ordering. *CoRR*, abs/0711.0251, 2007.

[134] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 87–96, New York, NY, USA, 2008. ACM.

[135] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 2006.

[136] V.S Anil Kumar, Madhav V. Marathe, Srinivasan Parthasarathy, and Aravind Srinivasan. Scheduling on unrelated machines under tree-like precedence constraints. *Proceedings of the Eighth International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, 2005.

[137] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[138] E.L. Lawler and J. Labetoulle. On preemptive scheduling of unrelated parallel processors by linear programming. *Journal of the ACM (JACM)*, 25(4):612–619, 1978.

[139] F.T. Leighton, Bruce M. Maggs, and Satish B. Rao. Packet routing and job-shop scheduling in $o(\text{Congestion} + \text{Dilation})$ steps. *Combinatorica*, 14(2):167–186, 1994.

[140] Jure Leskovec, Deepayan Chakrabarti, Jon M. Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *PKDD*, pages 133–145, 2005.

[141] Guolong Lin and Rajmohan Rajaraman. Approximation algorithms for multiprocessor scheduling under uncertainty. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 25–34, San Diego, California, USA, 2007.

[142] Hsiao-Fei Liu and Kun-Mao Chao. A tight analysis of the katriel–bodlaender algorithm for online topological ordering. *Theoretical Computer Science*, 389(1-2):182–189, 2007.

[143] K. Patrick Lorton and David S. Wise. Analyzing block locality in Morton-order and Morton-hybrid matrices. *SIGARCH Computer Architecture News*, 35(4):6–12, 2007.

[144] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[145] P. D. MacKenzie, C. G. Plaxton, and R. Rajaraman. On contention resolution protocols and associated probabilistic phenomena. *JACM*, 45(2):324–378, March 1998.

[146] Grzegorz Malewicz. Parallel scheduling of complex dags under uncertainty. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 66–75, Las Vegas, Nevada, USA, 2005.

[147] Virendra J. Marathe, Michael F. Spear, Christ opher Heriot, Athul Acharya, David Eisenstat, and Michael L. Scherer III, William N. an d Scott. Lowering the overhead of nonblocking software transactional m emory. In *Proceedings of the Workshop of Languages, Compilers, and Hard ware Support for Transactional Computing (TRANSACT)*, June 2006.

[148] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. On-line graph algorithms for incremental compilation. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 70–86, June 1993.

[149] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53–58, 1996.

[150] Harry M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[151] Lyle A. McGeoch and Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.

[152] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, theorem 5, pages 198–199. Springer-Verlag, Berlin, 1984.

[153] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33. IEEE Computer Society Press, 1991.

[154] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *CACM*, 19(7):395–404, July 1976.

[155] M. M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *Proceedings of the 18th International Conference on Distributed Computing (DISC)*, volume 3274 of *LNCS*, 2004.

[156] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Feb 2006.

[157] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, March 1966.

[158] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[159] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the 3rd Annual Symposium on Discrete Algorithms (SODA)*, pages 367–375, Orlando, Florida, January 1992.

[160] Aydın 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 *Proceedings of the Twenty-First Annual ACM Symposium on Parallel Algorthms and Architectures (SPAA)*, pages 233–244, Calgary, Canada, August 11–13 2009.

[161] Jelani Nelson. External-memory search trees with fast insertions. Master's thesis, Massachusetts Institute of Technology Department of Electrica l Engineering and Computer Science, June 2006.

[162] 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, 2007.

[163] Itzhak Nudler and Larry Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

[164] S. M. Omohundro, C. Lim, and J. Bilmes. The sather language compiler/debugger implementation. Technical report, International Computer Science Institute, Berkeley, March 1992.

[165] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

[166] David J. Pearce and Paul H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proceedings of the 3rd International Workshop on Efficient Experimental Algorithms*, volume 3059 of *Lecture Notes in Computer Science*, pages 383–398. Springer, 2004.

[167] D.J. Pearce, P.H.J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the 3rd International Workshop on Source Code Analysis and Manipulation*, pages 3–12, Sept. 2003.

[168] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall Englewood Cliffs, NJ, 2002.

[169] Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[170] Naila Rahman, Richard Cole, and Rajeev Raman. Optimised predecessor data structures for internal memory. In *Proceedings of the 5th International Workshop on Algorithm Engineering*, volume 2141 of *LNCS*, pages 67–78, Aarhus, Denmark, August 2001.

[171] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 294–305, Austin, Texas, December 2001.

[172] G. Ramalingam and Thomas Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Information Processing Letters*, 51(3):155–161, 1994.

[173] Rajeev Raman and David Stephen Wise. Converting to and from dilated integers. *IEEE Transactions on Computers*, 57(4):567–573, 2008.

[174] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[175] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, second edition, 2003.

[176] Nobuo Sato and W. F. Tinney. Techniques for exploiting the sparsity of the network admittance matrix. *IEEE Trans. Power Apparatus and Systems*, 82(69):944–950, Dec. 1963.

[177] Sandeep Sen and Siddhartha Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 829–838, San Francisco, California, January 2000.

[178] Paolo Serafini. Scheduling jobs on several machines with the job splitting property. *Operations Research*, 44(4):617–628, 1996.

[179] Viral Shah and John R. Gilbert. Sparse matrices in Matlab*P: Design and implementation. In *International Conference on High Performance Computing*, pages 144–155, 2004.

[180] David B. Shmoys, Clifford Stein, and Joel Wein. Improved approximation algorithms for shop scheduling problems. *SIAM Journal on Computing*, 23(3):617–632, June 1994.

[181] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[182] Sleepycat Software. The Berkeley Database. `http://www.sleepycat.com`, 2005.

[183] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.

[184] Jim Sukha. Brief announcement: A lower bound for depth-restricted work stealing. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 124–126, Calgary, Canada, August 11–13 2009.

[185] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.2.3 Reference Manual*, April 2006.

[186] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22(2):215–225, April 1975.

[187] Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the Association for Computing Machinery*, 26(4):690–715, October 1979.

[188] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[189] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–10. ACM Press, 2001.

[190] W.F. Tinney and J.W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, Nov. 1967.

[191] Sivan Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–726, 1997.

[192] Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, May 1984.

[193] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, 2005.

[194] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[195] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

[196] Gideon Weiss and Michael Pinedo. Scheduling tasks with exponential service times on non-identical processors to minimize various cost functions. *Journal of Applied Probability*, 17(1):187–202, 1980.

[197] Dan E. Willard. Inserting and deleting records in blocked sequential files. Technical Report TM81-45193-5, Bell Laboratories, 1981.

[198] Dan E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 251–260, Washington, D.C., May 28–30 1986.

[199] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, April 1992.

[200] D.E. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 114–121, San Francisco, California, May 5–7 1982.

[201] Jeremiah Willcock and Andrew Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th Annual International Cconference on Supercomputing (ICS)*, pages 307–316, New York, NY, USA, 2006. ACM.

[202] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.