Parallelism is the key to achieving high performance in computing. However, writing efficient and scalable parallel programs is notoriously difficult, and often requires significant expertise. To address this challenge, it is crucial to provide programmers with high-level tools to enable them to develop solutions easily, and at the same time emphasize the theoretical and practical aspects of algorithm design to allow the solutions developed to run efficiently under many different settings. This thesis addresses this challenge using a three-pronged approach consisting of the design of shared memory programming techniques, frameworks, and algorithms for important problems in computing.

The thesis provides evidence that with appropriate programming techniques, frameworks, and algorithms, shared-memory programs can be simple, fast, and scalable, both in theory and in practice. The results developed in this thesis serve to ease the transition into the multicore era.

The first part of this thesis introduces tools and techniques for deterministic parallel programming, including means for encapsulating nondeterminism via powerful commutative building blocks, as well as a novel framework for executing sequential iterative loops in parallel, which lead to deterministic parallel algorithms that are efficient both in theory and in practice. The second part of this thesis introduces Ligra, the first high-level shared memory framework for parallel graph traversal algorithms. The framework allows programmers to express graph traversal algorithms using very short and concise code, delivers performance competitive with that of highly-optimized code, and is up to orders of magnitude faster than existing systems designed for distributed memory. This part of the thesis also introduces Ligra+, which extends Ligra with graph compression techniques to reduce space usage and improves parallel performance at the same time, and is also the first graph processing system to support in-memory graph compression.

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

This is a revised version of the thesis that won the 2015 ACM Doctoral Dissertation Award.
Shared-Memory Parallelism Can Be Simple, Fast, and Scalable
ACM Books

Editor in Chief
M. Tamer Özsu, University of Waterloo

ACM Books is a new series of high-quality books for the computer science community, published by ACM in collaboration with Morgan & Claypool Publishers. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

Shared-Memory Parallelism Can Be Simple, Fast, and Scalable
Julian Shun, University of California, Berkeley
2017

Computational Prediction of Protein Complexes from Protein Interaction Networks
Sriganesh Srihari, The University of Queensland Institute for Molecular Bioscience
Chern Han Yong, Duke-National University of Singapore Medical School
Limsoon Wong, National University of Singapore
2017

The Handbook of Multimodal-Multisensor Interfaces, Volume 1: Foundations, User Modeling, and Common Modality Combinations
Editors: Sharon Oviatt, Inca Designs
Björn Schuller, University of Passau and Imperial College London
Philip R. Cohen, Voicebox Technologies
Daniel Sonntag, German Research Center for Artificial Intelligence (DFKI)
Gerasimos Potamianos, University of Thessaly
Antonio Krüger, German Research Center for Artificial Intelligence (DFKI)
2017

Communities of Computing: Computer Science and Society in the ACM
Thomas J. Misa, Editor, University of Minnesota
2017

Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining
ChengXiang Zhai, University of Illinois at Urbana–Champaign
Sean Massung, University of Illinois at Urbana–Champaign
2016

An Architecture for Fast and General Data Processing on Large Clusters
Matei Zaharia, Stanford University
2016
Reactive Internet Programming: State Chart XML in Action
Franck Barbier, *University of Pau, France*
2016

Verified Functional Programming in Agda
Aaron Stump, *The University of Iowa*
2016

The VR Book: Human-Centered Design for Virtual Reality
Jason Jerald, *NextGen Interactions*
2016

Ada’s Legacy: Cultures of Computing from the Victorian to the Digital Age
Robin Hammerman, *Stevens Institute of Technology*
Andrew L. Russell, *Stevens Institute of Technology*
2016

Edmund Berkeley and the Social Responsibility of Computer Professionals
Bernadette Longo, *New Jersey Institute of Technology*
2015

Candidate Multilinear Maps
Sanjam Garg, *University of California, Berkeley*
2015

Smarter Than Their Machines: Oral Histories of Pioneers in Interactive Computing
John Cullinane, *Northeastern University; Mossavar-Rahmani Center for Business and Government, John F. Kennedy School of Government, Harvard University*
2015

A Framework for Scientific Discovery through Video Games
Seth Cooper, *University of Washington*
2014

Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers
Bryan Jeffrey Parno, *Microsoft Research*
2014

Embracing Interference in Wireless Systems
Shyamnath Gollakota, *University of Washington*
2014
Shared-Memory Parallelism Can Be Simple, Fast, and Scalable

Julian Shun

University of California, Berkeley
To my family
# Contents

Preface  xiii  

**Chapter 1**  Introduction  1  
1.1  Shared-Memory Programming  4  
1.2  Shared-Memory Algorithm Design  10  
1.3  Shared-Memory Performance  14  
1.4  The Problem Based Benchmark Suite  16  
1.5  Contributions of this Book  17  

**Chapter 2**  Preliminaries and Notation  23  
2.1  Parallel Programming Model  23  
2.2  Algorithmic Complexity Model  24  
2.3  Parallel Primitives  26  
2.4  Graphs  26  
2.5  Strings  27  
2.6  Problem Definitions  27  
2.7  Experimental Environment  30  

**PART I**  PROGRAMMING TECHNIQUES FOR DETERMINISTIC PARALLELISM  33  

**Chapter 3**  Internally Deterministic Parallelism: Techniques and Algorithms  35  
3.1  Introduction  35  
3.2  Programming Model  38  
3.3  Commutative Building Blocks  43  
3.4  Internally Deterministic Parallel Algorithms  46  
3.5  Experimental Results  55  

Chapter 4  Deterministic Parallelism in Sequential Iterative Algorithms

4.1 Introduction
4.2 Analysis Tools
4.3 Algorithmic Design Techniques
4.4 Maximal Independent Set
4.5 Maximal Matching
4.6 Random Permutation
4.7 List Contraction
4.8 Tree Contraction
4.9 Limited Randomness
4.10 Experiments

Chapter 5  A Deterministic Phase-Concurrent Parallel Hash Table

5.1 Introduction
5.2 Related Work
5.3 Preliminaries
5.4 Deterministic Phase-Concurrent Hash Table
5.5 Applications
5.6 Experiments

Chapter 6  Priority Updates: A Contention-Reducing Primitive for Deterministic Programming

6.1 Introduction
6.2 Priority Updates
6.3 Contention in Shared Memory Operations
6.4 Applications of Priority Update
6.5 Experiment Study: Applications

PART II  LARGE-SCALE SHARED-MEMORY GRAPH ANALYTICS

Chapter 7  Ligra: A Lightweight Graph Processing Framework for Shared Memory

7.1 Introduction
7.2 Related Work
7.3 Framework
7.4 Applications
7.5 Experiments
Chapter 8  Ligra+: Adding Compression to Ligra  211
  8.1  Introduction  211
  8.2  Previous Work  212
  8.3  Ligra+ Implementation  213
  8.4  Experiments  219

PART III  PARALLEL GRAPH ALGORITHMS  231

Chapter 9  Linear-Work Parallel Graph Connectivity  233
  9.1  Introduction  233
  9.2  Linear-Work Low-Diameter Decomposition  237
  9.3  Linear-Work Connectivity  237
  9.4  Implementation Details  241
  9.5  Experiments  247

Chapter 10  Parallel and Cache-Oblivious Triangle Computations  259
  10.1  Introduction  259
  10.2  Preliminaries  261
  10.3  Triangle Counting  263
  10.4  Exact Triangle Counting  265
  10.5  Approximate Triangle Counting  268
  10.6  Extensions  269
  10.7  Evaluation  272
  10.8  Parallelization of the Pagh-Silvestri Algorithm  283
  10.9  Prior and Related Work  285

PART IV  PARALLEL STRING ALGORITHMS  289

Chapter 11  Parallel Cartesian Tree and Suffix Tree Construction  291
  11.1  Introduction  291
  11.2  Preliminaries  294
  11.3  Parallel Cartesian Trees  295
  11.4  Cartesian Trees and the ANSV Problem  300
  11.5  Experiments  302

Chapter 12  Parallel Computation of Longest Common Prefixes  315
  12.1  Introduction  315
  12.2  Preliminaries  318
12.3 Algorithms and Analysis  325
12.4 Experiments  331

Chapter 13  Parallel Lempel-Ziv Factorization  345
13.1 Introduction  345
13.2 Preliminaries  347
13.3 Parallel Lempel-Ziv Factorization Algorithm  349
13.4 Implementations  351
13.5 Experiments  352

Chapter 14  Parallel Wavelet Tree Construction  357
14.1 Introduction  357
14.2 Preliminaries  358
14.3 Related Work  359
14.4 Parallel Wavelet Tree Construction  360
14.5 Experiments  364
14.6 Parallel Construction of Rank/Select Structures  368
14.7 Extensions  369

Chapter 15  Conclusion and Future Work  373
15.1 Summary  373
15.2 Future Work  375

References  379
Index  413
Author’s Biography  427
Parallelism is the key to achieving high performance in computing. However, writing efficient and scalable parallel programs is notoriously difficult, and often requires significant expertise. To address this challenge, it is crucial to provide programmers with high-level tools to enable them to develop solutions easily, and at the same time emphasize the theoretical and practical aspects of algorithm design to allow the solutions developed to run efficiently under many different settings. This book addresses this challenge using a three-pronged approach consisting of the design of shared-memory programming techniques, frameworks, and algorithms for important problems in computing. The book provides evidence that with appropriate programming techniques, frameworks, and algorithms, shared-memory programs can be simple, fast, and scalable, both in theory and practice. The results developed in this book serve to ease the transition into the multicore era.

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

The second part of this book introduces Ligra, the first high-level shared-memory framework for parallel graph traversal algorithms. The framework allows programmers to express graph traversal algorithms using very short and concise code, delivers performance competitive with that of highly optimized code, and is up to orders of magnitude faster than existing systems designed for distributed memory. This part of the book also introduces Ligra+, which extends Ligra with graph compression techniques to reduce space usage and improve parallel performance at the same time, and is also the first graph processing system to support in-memory graph compression.
The third and fourth parts of this book bridge the gap between theory and practice in parallel algorithm design by introducing the first algorithms for a variety of important problems on graphs and strings that are both practical and theoretically efficient. For example, this book develops the first linear-work and polylogarithmic-depth algorithms for suffix tree construction and graph connectivity that are also practical, as well as a work-efficient, polylogarithmic-depth, and cache-efficient shared-memory algorithm for triangle computations that achieves a $2–5 \times$ speedup over the best existing algorithms on 40 cores.

This book is a revised version of the thesis that won the 2015 ACM Doctoral Dissertation Award.

Acknowledgments

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

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

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

I am thankful to Guy Blelloch, Phil Gibbons, Charles Leiserson, and Michael Mahoney for recommending me for both the CMU School of Computer Science Distinguished Dissertation Award as well as the ACM Doctoral Dissertation Award.
I am very grateful to both award committees for their hard work in reviewing my thesis, on which this book is based, and I am extremely honored that my work was selected for both of the awards in 2015. I am thankful to Google for their financial sponsorship of the ACM Doctoral Dissertation Award. The ACM Books program provided enormous support in revising this book.

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

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

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

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

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

Why have multicore machines become so widespread in just the past decade? Moore’s law states that the transistor density doubles approximately every 18

---

2. For example, the Intel Sandy Bridge-based Dell PowerEdge R930 can be configured with up to 96 cores and 6 TB of memory.
months [Moore 1965], and along with Dennard scaling, which states that transistor power density is constant [Dennard et al. 1974], this has historically corresponded to increases in clock speeds of single core machines of roughly 30% per year since the mid-1970’s [Leiserson and Mirman 2008]. However, since around the mid-2000’s, Dennard scaling no longer continued to hold due to physical limitations of hardware and, as a result, hardware vendors have turned to developing processors with multiple cores to deliver improved performance. These machines are referred to as shared-memory multicore machines, as the different cores have access to a shared global memory. This shift in processor technology has often been referred to as the “multicore revolution” [Leiserson and Mirman 2008]. Multicore technology has become ubiquitous today, with most personal computers, and even most cellular phones containing multiple cores. Therefore, writing parallel programs to take advantage of the multiple cores on a machine is crucial to obtaining scalable performance and enabling large-scale data to be processed.

In addition to multicore technology, parallel computing can come in the form of distributed systems, as mentioned above, graphics processing units (GPUs), and field programmable gate arrays (FPGAs). Unlike multicores, distributed systems can solve problems that do not fit in the memory of a single machine. However, compared to multicore shared-memory systems, communication and data replication in distributed systems often leads to high additional overheads. Therefore, for problems that can fit in memory, shared-memory multicores are generally significantly more efficient on a per-core, per-dollar, and per-joule basis than distributed-memory systems. For example, this book shows that the exact triangle count of the Yahoo! Web graph with over 6 billion edges can be computed in under 1.5 min and a suffix tree can be constructed on the 3 GB human genome in under 3 min on a modern 40-core machine, much faster than previous distributed-memory solutions (both in absolute performance and on a per-core basis) for the same problem. The data sets in these examples are among the largest considered in the literature for the corresponding problems, and easily fit on a multicore machine. While GPUs and FPGAs may be more efficient for certain problems, multicore machines are much more general-purpose, support larger memory sizes (useful for scaling to large data), and are considerably easier to program. This book argues that shared-

---

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

4. The techniques developed in this book are also applicable to Intel’s new Xeon Phi coprocessors, which support higher memory bandwidth than traditional multicore machines. However, currently their memory sizes are not sufficient for some of the larger data sets studied in this book.
memory multicores offer a *sweet spot* between programmability and efficiency. There has been a large body of work on developing efficient algorithms and frameworks for regular problems, where the parallelism is relatively well structured (e.g., problems in dense numerical linear algebra and scientific simulations), while less work has been done for irregular problems, where the parallelism is much less well-structured and highly dependent on input data (e.g., problems on graphs and strings). *This book studies shared-memory programming techniques, frameworks, and algorithms for a wide class of irregular problems and shows that shared-memory parallelism can be simple, fast, and scalable.*

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

- **Section 1.1** introduces nested fork-join parallelism, which is the type of parallelism studied here. This section then describes challenges in shared-memory programming, including obtaining determinism, controlling shared access, and developing high-level programming abstractions. The reader will obtain an overview of the contributions of this book to addressing these challenges.
- **Section 1.2** describes the Parallel Random Access Machine (PRAM) and work-depth models for analyzing parallel algorithms. This is followed by some highlights of the book’s contribution in bridging the gap between theory and practice in parallel algorithms via designing theoretically efficient algorithms that perform well on modern multicore machines.
- **Section 1.3** describes performance factors of multicore programs, including caching, memory contention, and parallel scalability. This section introduces techniques developed in this book that take into account these factors to improve performance.
- **Section 1.4** introduces a benchmark suite developed in this book to comprehensively evaluate solutions to given problems in terms of simplicity as well as theoretical and practical efficiency.
- The contributions of this book are summarized in Section 1.5.
1.1 Shared-Memory Programming

Languages

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

Another method for writing shared-memory multicore programs is to use simple constructs that indicate which parts of the program are safe to run in parallel, and allow a run-time scheduler to assign work to threads and perform load balancing on-the-fly. This approach is known as dynamic multithreading. Using dynamic multithreading languages such as Cilk [Frigo et al. 1998], OpenMP, Intel Threading Building Blocks, Habanero [Budimlic et al. 2011], and X10 [Charles et al. 2005], one can write clean parallel programs while letting the run-time scheduler perform the work allocation and load balancing. This approach frees the programmer from the low-level details of explicit thread management, leading to simpler code, while delivering comparable or improved performance. With advances in scheduling, it is now possible to write a wide class of parallel programs in this framework that are efficient, both in theory and in practice [Blumofe and Leiserson 1999], without having to tune the program to achieve balanced workloads.

Nested Fork-Join Parallelism

All of the algorithms and techniques studied in this book are designed for nested fork-join parallelism, in which procedures can be called recursively in parallel via a fork construct, and synchronized via a join construct [Blelloch 1992]. This includes parallel for-loops, which can be implemented using fork and join. Nested parallel computations can be defined inductively in terms of the composition of sequential and parallel components, and modeled as a directed acyclic graph (computation DAG). Dynamic multithreading languages such as Cilk support low-overhead primitives to implement fork-join parallelism [Leiserson 2010]. A broad class of parallel programs can be expressed with fork-join parallelism, and the programming techniques and frameworks developed in this book aim to enable programs written within this paradigm to be simpler and more efficient.
Determinism

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

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

One of the main approaches to developing efficient deterministic parallel solutions in this book is the deterministic reservations framework for parallelizing greedy sequential algorithms (Chapter 3). The approach consists of two phases—in the reserve phase, the iterates concurrently mark all of the data that they affect, and in the commit phase, iterates whose mark is still written on all of its affected data proceed with the computation on the data. Determining successful reservations is done in a deterministic manner, so that for a given round the same iterates succeed/fail on every execution. Parallel algorithms written in this framework return the same answer as their sequential counterparts, which gives determinism, and allows the parallel and sequential algorithms to be interchanged when necessary. The algorithms developed are also very simple, as the user only needs to
struct STStep {
  int u; int v;
  edge *E; res *R; disjointSet F;
STStep(edge* _E, disjointSet _F, res* _R)
  : E(_E), R(_R), F(_F) {};
bool reserve(int i) {
  u = F.find(E[i].u); //find component
  v = F.find(E[i].v); //find component
  if (u == v) return 0; //skip edge if endpoints belong to the
  // same component
  if (u > v) swap(u, v);
  R[v].reserve(i); //reserve larger component
  return 1; }
bool commit(int i) {
  if (R[v].check(i)) { F.link(v, u); return 1; } //link if reservation
  // was successful
  else return 0; }
};
void ST(res* R, edge* E, int m, int n, int psize) {
  disjointSet F(n); //deterministic union-find data structure
  speculative_for(STStep(E, F, R), 0, m, psize); //deterministic
  // reservations driver
}

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

specify the reserve and commit functions called by each iterate in the two corresponding phases, as well as corresponding data structures. For example, Figure 1.1 shows the C++ code for a spanning forest algorithm using deterministic reservations. disjointSet is a deterministic union-find data structure developed in this book, and speculative_for executes the deterministic reservations framework using the user-defined reserve and commit functions (more details will be discussed in Chapter 3).

Part I describes tools for writing internally deterministic parallel code [Blelloch et al. 2012, Shun et al. 2013, Shun and Blelloch 2014], drawing heavily on using commutative operations. This part also describes internally deterministic solutions to a broad set of benchmark problems using these tools, and shows that these solutions are efficient (competitive with existing nondeterministic solutions and
achieve good parallel speedup), scalable to large inputs, natural to reason about, not complicated to code [Blelloch et al. 2012], and also have good theoretical guarantees [Blelloch et al. 2012, Shun et al. 2015].

### Controlling Shared Access

Many parallel programs use *locks* to control access to shared resources. The granularity of locking (e.g., locking an entire data structure vs. locking a small part of the data structure) affects the performance, scalability, and programmability of a solution, with coarser-grained locking leading to simpler solutions and finer-grained locking leading to higher efficiency and scalability. Programming with locks, however, has disadvantages such as leading to deadlock or livelock, and writing efficient fine-grained lock-based programs is often very tedious. There has been significant work on writing parallel programs without locks by making use of atomic operations (e.g., compare-and-swap and fetch-and-increment) supported in hardware [Herlihy and Shavit 2012]. Proper use of atomics can lead to more efficient programs than fine-grained locking and has the advantage of having progress guarantees. All of the programming techniques, algorithms, and data structures developed in this book are lock-free, making use of atomic operations when necessary, while also being simple. An extremely useful atomic primitive called *priority update* for controlling shared access in deterministic programs [Shun et al. 2013] is introduced in Chapter 6, and is used throughout the algorithms in this book.

*Transactional memory* (TM) is a technique to simplify shared-memory programming by allowing users to specify regions of code that will execute atomically (see, e.g., [Harris et al. 2010] for an overview). This frees the programmer from having to lock critical sections in code, leading to simpler programs. There has been significant research in implementing transactional memory both in software and in hardware. However, the techniques developed in this book are unlikely to benefit from TM for two reasons: (1) the order in which transactions succeed in TM is not deterministic, and (2) the algorithms in this book have no lock-based critical sections—shared accesses are protected using only a single atomic instruction.

### Programming Frameworks

Another effort in simplifying shared-memory programming has been in developing higher-level frameworks and interfaces for writing parallel solutions. These range from general parallel programming libraries such as the Parallel Boost Graph Library [Gregor and Lumsdaine 2005], Multi-Core Standard Template Library (MCSTL) [Singler et al. 2007], SWARM [Bader et al. 2007], Galois [Pingali et al. 2011], and algorithms/containers provided as part of the Intel Threading Building Blocks, to domain-specific frameworks/languages such as GraphLab [Low et al.
Parents $= \{-1, \ldots, -1\}$ \hspace{1cm} \triangleright \text{initialized to all -1’s, indicating unexplored}

\begin{verbatim}
1 procedure UPDATE(s, d)
2 \hspace{1cm} return (CAS(&Parents[d], -1, s)) \hspace{1cm} \triangleright \text{atomically explore vertex}
\end{verbatim}

\begin{verbatim}
4 procedure COND(i)
5 \hspace{1cm} return (Parents[i] == -1) \hspace{1cm} \triangleright \text{check if unexplored}
\end{verbatim}

\begin{verbatim}
6 procedure BFS(G, r)
7 \hspace{1cm} Parents[r] = r
8 \hspace{1cm} Frontier = \{r\} \hspace{1cm} \triangleright \text{vertexSubset initialized to contain only } r
9 \hspace{1cm} while SIZE(Frontier) \neq 0 do
10 \hspace{1cm} Frontier = EDGEMAP(G, Frontier, UPDATE, COND) \hspace{1cm} \triangleright \text{explore next frontier}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.2.png}
\caption{Pseudocode for breadth-first search (BFS) in Ligra. The compare-and-swap function \texttt{CAS(loc, oldV, newV)} atomically checks if the value at location \texttt{loc} is equal to \texttt{oldV} and if so it updates \texttt{loc} with \texttt{newV} and returns \texttt{TRUE}. Otherwise, it leaves \texttt{loc} unmodified and returns \texttt{FALSE}.}
\end{figure}


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

This book shows that Ligra can process the largest publicly available real-world graphs in shared-memory and is much faster than existing high-level graph processing systems for the same applications. This work advocates performing large-scale graph analytics on a single shared-memory server instead of using distributed memory, and since the development of Ligra, there have been several other large-
scale graph processing frameworks [Nguyen et al. 2013, Roy et al. 2013, Kaler et al. 2014, Zhang et al. 2015] developed for shared-memory multicores, as well as a graph processing framework for GPUs sharing ideas with Ligra [Wang et al. 2015].

Concurrency
There has been a large body of research on concurrency in parallel programming, which studies how different threads interact with each other. Dealing with concurrency often requires considerable effort from the programmer because the behavior of concurrent programs is almost always nondeterministic due to the nondeterministic order in which the threads execute. The goal of this book is to hide the concurrency in parallel programs from the programmer by raising the level of abstraction and developing deterministic tools at this higher level of abstraction (e.g., deterministic reservations described in Chapter 3 and priority updates described in Chapter 6) and data structures (e.g., a deterministic phase-concurrent hash table described in Chapter 5) that the user can simply call in their programs. By raising the level of abstraction, the implementations of the tools can be nondeterministic (but hidden to the programmer), giving more flexibility and efficiency. This approach leads to deterministic parallel solutions that are simple to reason about, and that are also efficient at the same time.

Memory consistency issues often arise in concurrent programs as instructions can be reordered on multicore processors. However, in all of the solutions developed here, reads and writes to the same memory location are either separated by a synchronization point or use a compare-and-swap, which implicitly issues a memory barrier to prevent consistency issues. All of the solutions are sequentially consistent, which means that their results are consistent with some valid sequential execution of the program [Lamport 1979].

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

Parallel Random Access Machine

Algorithm designers have traditionally used the Parallel Random Access Model (PRAM) to analyze parallel algorithms for shared memory. In this model, every core has unit-time access to the shared global memory. An algorithm’s complexity is characterized by its asymptotic time \( T \) and number of cores \( P \), with the total number of operations being the product of the two terms. They can also be analyzed in the Work-Time Framework [JaJa 1992], in which the total number of operations \( W \) and number of parallel time steps \( T \) is specified. PRAM algorithms are written using flat parallelism, in which parallel operations over a single array is done synchronously at every time step. The algorithm must specify how work can be efficiently allocated among the cores on each step (known as the processor allocation problem). Using Brent’s scheduling principle [Brent 1974, JaJa 1992], an algorithm with \( W \) work and \( T \) time can be run in \( W/P + T \) time with \( P \) cores. Nested fork-join parallel algorithms cannot be directly expressed in the PRAM, and the parallelism in such algorithms must be flattened to work for the PRAM. Different classes of PRAM models differ in whether concurrent reads or writes are allowed, how to resolve write conflicts, and how to deal with contention (see, e.g., [JaJa 1992, Gibbons et al. 1999]). There have also been variants proposed that allow for asynchrony among the cores [Gibbons 1989, Cole and Zajicek 1989, Nishimura 1990, Gibbons et al. 1998], as well as a related model that provides parallel primitives on vectors [Blelloch 1990].

Work-Depth Model

The work-depth model is a model supporting nested fork-join parallelism.\(^5\) As discussed in Section 1.1, a nested parallel computation can be modeled as a computation DAG. An algorithm’s complexity is analyzed by computing its work \( W \), which is the sum of the costs of all the tasks in the computation DAG, and its depth \( D \), which is the maximum sum of costs of tasks on a directed path in the DAG (the longest sequential dependence). The maximum possible amount of parallelism (i.e., the maximum number of cores the computation can take advantage of) is \( W/D \). The complexity of PRAM algorithms translate to results in the work-depth model, however they can often be simplified, as the processor allocation step is not necessary and divide-and-conquer can be used. The work-depth model underlies

---

\(^5\) This contrasts with the Work-Time Framework, which is a framework for analyzing PRAM algorithms and does not allow for nested parallelism.
the design of programming languages such as NESL [Blelloch 1992] and Cilk [Frigo et al. 1998], and algorithms designed for the model can take advantage of dynamic multithreading languages. For example, a computation with work \( W \) and depth \( D \) using Cilk’s randomized work-stealing scheduler gives an expected running time of \( W/P + O(D) \) when running on \( P \) cores [Blumofe and Leiserson 1999]. The algorithms developed in this book are analyzed in the work-depth model, but they can easily be translated into PRAM algorithms.

**Traditional Design Goals**

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

**Bridging Theory and Practice**

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

\(^6\) Polylogarithmic-depth algorithms are also desirable for computational complexity reasons, as they fall in the class NC (Nick’s Class) containing problems that can be solved on circuits with polylogarithmic depth and polynomial size [Arora and Barak 2009].
Table 1.1  Work and depth bounds for the (randomized) algorithms developed in this book

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

Note: For the graph problems, $n = \text{number of vertices}$ and $m = \text{number of edges}$. For the other problems, $n$ is the input size. The depth of some of these algorithms can be improved with approximate compaction [Gil et al. 1991a], as described in their respective chapters.

\(^a\) Bounds are for constant-sized alphabets.

\(^b\) $\sigma = \text{alphabet size}$.

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

We will now briefly look at the performance of two of the algorithms developed in this book—triangle counting and suffix tree construction. For triangle counting, this book develops the first work-efficient, polylogarithmic-depth, and cache-friendly shared-memory algorithm (Chapter 10), which outperforms existing
1.2 Shared-Memory Algorithm Design

Figure 1.3 Experimental evaluation of triangle counting and suffix tree construction. (a) Speedup of our triangle counting algorithm relative to the fastest shared-memory algorithm (varies between the implementation in GraphLab [Gonzalez et al. 2012] and the one by Green et al. [2014]) on various synthetic graphs from [Shun et al. 2012] and real-world graphs from [Leskovec and Krevl 2014, Kwak et al. 2010] on 40 cores with two-way hyper-threading. (b) Parallel running times of suffix tree construction on the 3 GB human genome. * Reported times from the literature [Mansour et al. 2011, Comin and Farreras 2013]. ** Code from [Mansour et al. 2011] run on our 40-core machine with a memory budget of 160 GB.

shared-memory algorithms by a factor of 2–5× on 40 cores with two-way hyper-threading and achieves a parallel speedup ranging from 22× to 49× [Shun and Tangwongsan 2015]. The speedup of the algorithm with respect to the fastest existing shared-memory implementation on various graphs is shown in Figure 1.3(a). Additionally, this algorithm has stronger theoretical bounds than previous shared-memory algorithms. Compared to existing distributed-memory solutions, the algorithm is faster by at least an order of magnitude on a per-core basis on the largest graphs studied in the literature. For suffix tree construction, this book develops the first parallel algorithm with linear work and polylogarithmic depth that is also practical (Chapter 11) [Shun and Blelloch 2014]. On 40 cores with two-way hyper-threading, the algorithm achieves a 5.4–50.4× speedup over the best sequential algorithm [Kurtz 1999] on a variety of inputs. The algorithm can construct the suffix tree for the 3 GB human genome, one of the largest data sets reported in the literature for suffix tree construction, in under 3 min. Compared to the fastest num-
bers reported in the literature for suffix tree construction on the human genome, the algorithm is at least two times faster in practice, as shown in Figure 1.3(b), in addition to being theoretically more efficient.

1.3 Shared-Memory Performance

Cache Performance
Due to the high latency to access main memory, modern multicore machines have caches, which are smaller memories that support faster access times. Multicore machines can have multiple levels of caches, each with different sizes and access times, and furthermore caches may either be shared among cores or private to a single core. The caches thus form a hierarchy, and designing algorithms that make efficient use of the cache hierarchy is crucial for performance. The algorithms studied in this book involve many memory accesses, and thus their performance is largely determined by the number of cache misses. While this book does not explicitly analyze the cache performance of algorithms (with the exception of Chapter 10, which analyzes cache performance of triangle computations), they are all implemented to be cache-friendly, maximizing spatial and temporal locality when possible. Cache misses can also be factored into an algorithm’s theoretical complexity (see, e.g., [Frigo et al. 1999, Simhadri 2013]), although this is not the focus of this book.

Contention
On multicore machines, different private caches may reference the same objects in memory, and so there is the challenge of making sure that the cores’ views of the data are consistent. A cache coherence protocol dictates how this consistency is maintained among the caches (see, e.g., [Culler et al. 1998] for more details). Cache coherence protocols have a significant effect on the performance of shared memory accesses (see, e.g., the recent study by David et al. [2013]). In general, when updates are performed to a shared location concurrently by many different cores, the memory contention causes performance to worsen as the cache coherence protocol must perform significant work to ensure consistency among different caches. To reduce contention in shared-memory programs, Chapter 6 develops and advocates the usage of the priority update operation, which performs an actual update only when the value written has “higher priority” than the existing value, for a large class of applications. This book studies its performance both experimentally and theoretically under varying degrees of sharing, showing that it is much more efficient than many commonly used operations, and comparable in performance to other, less powerful operations. Figure 1.4(a) shows an experiment measuring the
1.3 Shared-Memory Performance

**Figure 1.4**
(a) Experiments measuring contention of various parallel operations. Times are for 5 runs of 100 million operations to varying number of memory locations on a 40-core Intel Nehalem machine (log-log scale). Since the number of operations is fixed, fewer locations implies more operations sharing those locations. (b) Average speedup of Ligra+ relative to Ligra on a variety of graph applications on 40 cores with two-way hyper-threading.

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

**Scalability**

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

Due to the irregular nature of the solutions studied in this book, they often spend a significant fraction of the time performing memory accesses, and their parallel scalability is often limited by the memory subsystem of the machine (e.g., memory bandwidth or cache contention). To alleviate this problem for graph applications, this book uses graph compression techniques in Chapter 8 to reduce memory usage, thus reducing the impact of the memory subsystem bottleneck, and as a result improving parallel performance and scalability. This book develops Ligra+ by integrating the graph compression techniques into Ligra, and shows that reduced space usage and improved parallel performance can be achieved at the same time [Shun et al. 2015]. The graph sizes are reduced to about half of the original size on average, and performance increases by about 14% on average on 40 cores. Figure 1.4(b) shows the average relative performance of Ligra+ compared to Ligra on various graph applications using 40 cores. Ligra+ is the first high-level graph processing system to support in-memory compression.

1.4 The Problem Based Benchmark Suite

To measure the programming simplicity, theoretical efficiency, and empirical performance among different solutions for given problems, my co-authors and I developed a benchmark suite, called the Problem Based Benchmark Suite (PBBS) [Shun et al. 2012], containing a set of well-known fundamental problems that is representative of a broad class of non-numeric applications arising in computing. Table 1.2 shows the problems currently in the benchmark suite (the definitions of these problems can be found in Section 2.6). Unlike most existing benchmarks, which are based on specific code, the PBBS benchmarks are defined in terms of the problem specifications—a concrete description of valid inputs and corresponding valid outputs, along with some specific inputs. Any algorithms, programming methodologies, specific programming languages, or machines can be used to solve the problems. The benchmark suite is designed to compare the benefits and shortcomings of different algorithmic and programming approaches, and to serve as a dynamically improving set of educational examples of how to parallelize applications. The PBBS has enabled comparisons in terms of simplicity, and theoreti-

7. The table has been modified from [Shun et al. 2012] to reflect the problems currently in the benchmark suite.
Table 1.2 Benchmarks in the Problem Based Benchmark Suite

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

cal/practical performance among various algorithms and programming techniques for the problems studied in this book. Many of the implementations developed in this book are part of the PBBS.

1.5 Contributions of this Book

This book seeks to address the three types of challenges arising in multicore programs, as outlined in Sections 1.1, 1.2, and 1.3, to make large-scale shared-memory parallelism more accessible. To address these challenges, I use a three-pronged approach studying programming techniques, algorithm design, and performance analysis for shared-memory multicores. These three areas are highly interrelated, and so each of the chapters of this book will inevitably cut across the different areas. An illustration placing each of the topics of this book into the closer two among the three categories is shown in Figure 1.5. This book provides evidence that with appropriate programming techniques, frameworks, and algorithms, shared-memory programs can be simple, fast, and scalable, both in theory and in practice.

I believe that the frameworks, tools, algorithms, and ideas developed in this book will enable more people to write efficient shared-memory parallel programs and take advantage of the power of multicore machines to perform large-scale computations. The code developed as part of this book is publicly available, and

---

8. While the book focuses on multicore solutions, this is not a constraint of the PBBS.
Figure 1.5 A pictorial organization of this book. The topics touch upon programming techniques, algorithm design, and performance analysis, and are placed in the closer two among the three areas in the figure.

has already been used by various researchers for benchmarking and developing their own shared-memory solutions.

I have developed the results of this book in collaboration with various co-authors: Guy Blelloch, Laxman Dhulipala, Jeremy Fineman, Phillip Gibbons, Yan Gu, Aapo Kyrola, Harsha Simhadri, Kanat Tangwongsan, and Fuyao Zhao. The following paragraphs describe the organization and contributions of this book.

Chapter 2 introduces the necessary definitions and notation used throughout. Then, Part 1 describes frameworks and techniques for simplifying deterministic parallel programming. The contributions of this part include:

- a new approach for writing efficient deterministic parallel programs using building blocks based on commutativity, and the design of several building blocks including priority updates, dictionaries, and disjoint sets (Chapters 3, 5, and 6);
- a novel technique called deterministic reservations for taking sequential loops with dependencies among iterations and parallelizing them deterministically (Chapters 3 and 4);
- a suite of deterministic parallel algorithms and data structures, including comparison sorting, a hash table-based dictionary, remove duplicates, ran-
dom permutation, list contraction, tree contraction, breadth-first search, spanning forest, minimum spanning forest, maximal independent set, maximal matching, suffix arrays, Delaunay triangulation, Delaunay refinement, quad/oct trees, k-nearest neighbors, N-body, and triangle ray intersect, along with experiments showing they are fast, scalable, and competitive with the best nondeterministic code for the same problem (Chapters 3–6);

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

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

• the first application of Nisan’s pseudorandom generator for space-bounded computations [Nisan 1992] to reducing the amount of randomness in low-depth parallel algorithms, in particular to reduce the amount of randomness in the random permutation and list contraction algorithms from $O(n \log n)$ to a polylogarithmic number of random bits (Chapter 4);

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

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

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

• the first comprehensive experimental study of the priority update operation versus other widely-used operations under varying degrees of sharing, demonstrating that it is up to orders of magnitude faster on modern Intel and AMD multicore machines (Chapter 6); and
many applications of the priority update operation in deterministic parallel programs, enabling good performance even under a high degree of write sharing (Chapter 6).

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

- the Ligra shared-memory graph processing framework containing just two simple functions—one for mapping computation over a subset of vertices and one for mapping computation over a subset of edges—sufficient to concisely express a broad class of graph traversal algorithms in shared-memory (Chapter 7);
- the generalization of the direction-optimizing idea used in breadth-first search [Beamer et al. 2012] to a large class of graph traversal algorithms to improve performance (Chapter 7);
- an experimental evaluation showing that the Ligra implementations are efficient and scalable to the largest publicly available real-world graphs in the literature, and outperform existing high-level graph processing frameworks by up to orders of magnitude (Chapter 7);
- the first high-level shared-memory graph processing system (Ligra) to process (in under a minute) the largest publicly available real-world graph, the Yahoo! Web graph with over 6 billion edges, showing the benefits of shared-memory for large-scale graph processing, and subsequently leading to several other shared-memory graph processing systems [Nguyen et al. 2013, Roy et al. 2013, Kaler et al. 2014, Zhang et al. 2015, Wang et al. 2015] (Chapter 7);
- Ligra+, the first high-level shared-memory graph processing system to use graph compression to reduce in-memory space usage, improving the scalability of shared-memory graph processing (Chapter 8); and
- an efficient implementation and experimental evaluation of Ligra+ showing that graph compression both reduces the space usage and also improves the parallel performance of graph algorithms (Chapter 8).

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

- the first practical linear-work and polylogarithmic-depth parallel algorithm for graph connectivity, a problem that has been open for over a decade (Chapter 9);
extensive empirical evaluation of the parallel connectivity algorithm, showing that it is competitive with existing parallel implementations, none of which are linear-work and polylogarithmic-depth (Chapter 9);

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

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

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

• a new and simple linear-work, polylogarithmic-depth parallel algorithm for building multiway Cartesian trees using divide-and-conquer, and various applications of Cartesian trees (Chapter 11);

• the first practical linear-work and polylogarithmic-depth parallel algorithm for suffix tree construction, developed using suffix arrays and multiway Cartesian trees (Chapter 11);

• the state-of-the-art parallel suffix tree implementation for shared-memory, achieving good parallel speedup (up to $24 \times$ on 40 cores) and outperforming existing parallel implementations by at least a factor of 2 (Chapter 11);

• new theoretically efficient and practical parallel algorithms for computing longest common prefixes, a useful primitive in suffix array (and suffix tree) construction (Chapter 12);

• the first comprehensive experimental evaluation of parallel longest common prefix algorithms, showing that the new algorithms achieve good parallel speedup, are up to $2.3 \times$ faster than the best existing algorithm on 40 cores, and lead to improved performance for suffix array construction (Chapter 12);

• the first practical linear-work and polylogarithmic-depth parallel algorithm for Lempel-Ziv factorization (based on suffix arrays), an essential operation in many data compression methods (Chapter 13);

• an extensive experimental study of the Lempel-Ziv factorization algorithm showing that it achieves good parallel speedups (up to $23 \times$ on 40 cores) and
outperforms the sequential algorithm with just 2 or more threads (Chapter 13);

- the first polylogarithmic-depth parallel algorithms for constructing wavelet trees, an essential component to many compressed data structures (Chapter 14); and

- a comprehensive empirical evaluation of the wavelet tree algorithms showing that they achieve good speedup over the sequential algorithm (up to $27 \times$ on 40 cores) and are up to 5.6 times faster than existing parallel implementations (Chapter 14).

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