Models for Parallel Computation in Multi-Core, Heterogeneous, and Ultra Wide-Word Architectures

by

Alejandro Salinger

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Multi-core processors have become the dominant processor architecture with 2, 4, and 8 cores on a chip being widely available and an increasing number of cores predicted for the future. In addition, the decreasing costs and increasing programmability of Graphic Processing Units (GPUs) have made these an accessible source of parallel processing power in general purpose computing. Among the many research challenges that this scenario has raised are the fundamental problems related to theoretical modeling of computation in these architectures. In this thesis we study several aspects of computation in modern parallel architectures, from modeling of computation in multi-cores and heterogeneous platforms, to multi-core cache management strategies, through the proposal of an architecture that exploits bit-parallelism on thousands of bits.

Observing that in practice multi-cores have a small number of cores, we propose a model for low-degree parallelism for these architectures. We argue that assuming a small number of processors (logarithmic in a problem's input size) simplifies the design of parallel algorithms. We show that in this model a large class of divide-and-conquer and dynamic programming algorithms can be parallelized with simple modifications to sequential programs, while achieving optimal parallel speedups. We further explore low-degree-parallelism in computation, providing evidence of fundamental differences in practice and theory between systems with a sublinear and linear number of processors, and suggesting a sharp theoretical gap between the classes of problems that are efficiently parallelizable in each case.

Efficient strategies to manage shared caches play a crucial role in multi-core performance. We propose a model for paging in multi-core shared caches, which extends classical paging to a setting in which several threads share the cache. We show that in this setting traditional cache management policies perform poorly, and that any effective strategy must partition the cache among threads, with a partition that adapts dynamically to the demands of each thread. Inspired by the shared cache setting, we introduce the minimum cache usage problem, an extension to classical sequential paging in which algorithms must account for the amount of cache they use. This cache-aware model seeks algorithms with good performance in terms of faults and the amount of cache used, and has applications in energy efficient caching and in shared cache scenarios.

The wide availability of GPUs has added to the parallel power of multi-cores, however, most applications underutilize the available resources. We propose a model for hybrid computation in heterogeneous systems with multi-cores and GPU, and describe strategies for generic parallelization and efficient scheduling of a large class of divide-and-conquer algorithms.

Lastly, we introduce the Ultra-Wide Word architecture and model, an extension of the word-RAM model, that allows for constant time operations on thousands of bits in parallel. We show that a large class of existing algorithms can be implemented in the Ultra-Wide Word model, achieving speedups comparable to those of multi-threaded computations, while avoiding the more difficult aspects of parallel programming.

Acknowledgements

Many people have contributed, directly or indirectly, to a happy conclusion of this thesis. Lucky me, I had not only one, but two advisors to guide me through the tough path of a doctoral program. Most of the work in this thesis was carried out in direct collaboration with Professor Alejandro López-Ortiz. I am immensely grateful to him for guiding me in finding interesting problems, in identifying the right research questions, for providing me with crucial insights as to how to address them, and not less importantly, for helping me appreciate the strengths of my results. His invaluable advice has helped me make the right decisions both in my career as well as in other important aspects of my life. I am also deeply grateful to my co-supervisor Professor Ian Munro for providing me with his remarkable insights on research problems as well as with his wise career advice. Thank you Alex and Ian for enabling me to work in a friendly atmosphere. It has been a delight to work under your supervision.

I would also like to thank my thesis committee members —Professors Hiren Patel, Prabhakar Ragde, Roberto Solis-Oba, and Bernard Wong— for their comments and suggestions, which have contributed to the improvement of the contents and presentation of this thesis.

The results in this thesis would not have been possible without the important contribution of my co-authors. I thank Reza Dorrigiv, Alejandro López-Ortiz, Arash Farzan, Patrick Nicholson, and Robert Suderman for their ideas and hard work. I am also thankful to Jérémy Barbay, Francisco Claude, Robert Fraser, Shahin Kamali, Daniel Remenik, and Gelin Zhou and surely many others for the countless discussions that in one way or another shaped the research that led to the results in this thesis (and thanks Francisco for printing my thesis!).

I would like to thank Wendy Rush and Helen Jardine for offering their help whenever I needed it. Thank you as well to Professor Daniel Berry for the invitations to celebrate several holidays. It means a lot to those of us who are away from our homes.

I am also very grateful to the Natural Sciences and Engineering Research Council of Canada (NSERC) for providing me with funding through my supervisors and to the David R. Cheriton School of Computer Science for awarding me the Cheriton scholarship.

I have been very fortunate to have met many great friends during my stay in Waterloo. Thanks to everybody with whom I have shared the Algorithms Lab —in particular to the long-lived clique BAPF— you made the lab my $(n+1)$-th home. To everybody who has proudly worn the jersey of Hopeless Experts since I started playing in my first term. It has been my pleasure to share our successes and disappointments with you, and I thank you for letting me play team manager for so many years. To all the friends that have in one way or another made my PhD experience a very enjoyable one (and whose names I omit for fear that I might forget someone), I thank you and hope that our paths will cross again.

I wish to thank my family for their love and support. To my parents, René and Anamaría, who raised me to be who I am today and have supported me unconditionally in all my enterprises,

academic or otherwise. I love you and I thank you for being the best parents anyone can ask for. To the sunshine of my life, my son Nicolás, who has taken some stress out of my studies by making me realize what the really important things in life are. Finally, a big thank you to my lovely wife Natasha. Thank you for painfully reading through my thesis and showing me where I should and where I should not put a comma. Thank you as well for making sure that I could concentrate on my work by taking care of everything else during the busiest times. Thank you for being so understanding and for always being there to listen and provide thoughtful advice. Your encouragement and support are an invaluable contribution to this thesis and to everything else in my life. I love you!

Dedication

To my parents

Table of Contents

List of Tables xvii

List of Figures xix

List of Algorithms xxiii

1 Introduction 1

1.1 Summary of Results and Structure of this Thesis 3

2 Parallel Computation 7

2.1 Sequential Models of Computation 7

2.1.1 Turing Machine 8

2.1.2 Random Access Machine 9

2.2 Parallelism in Computation: Flynn's Taxonomy 10

2.3 Theoretical Modeling of Parallel Computation 11

2.3.1 The Shared-Memory Model and the PRAM 11

2.3.2 Performance Measures 13

2.3.3 Network Models 15

2.3.4 Communication 17

2.3.5 Directed Acyclic Graphs 18

2.3.6 Boolean Circuits and Parallel Complexity Classes 20

2.3.7 Alternating Turing Machines 24

2.3.8 Vector Machines . . . 24
2.3.9 P-Complete Problems . . . 25
2.3.10 Amdahl's Law . . . 26
2.4 Parallel Architectures . . . 27
2.5 Beyond the PRAM . . . 29
2.5.1 Variants of the PRAM Model . . . 29
2.5.2 Hierarchical Memory Models . . . 31
2.5.3 Bridging Models . . . 31
2.6 Aspects of Parallel Programming . . . 33
2.6.1 Scheduling Multi-Threaded Programs . . . 34
2.7 Basic Parallel Algorithm Design Techniques . . . 38
2.7.1 Balanced Trees . . . 38
2.7.2 Pointer Jumping . . . 39
2.7.3 Pipelining . . . 39
2.7.4 Divide and Conquer . . . 40
2.7.5 Partitioning . . . 40
2.7.6 Accelerated Cascading . . . 40
2.7.7 Symmetry Breaking . . . 41
2.8 The Multi-Core Era . . . 41
2.8.1 Multi-Core Architectures . . . 42
2.8.2 Models for Multi-Core Computation . . . 43
2.8.3 Graphic Processing Units . . . 59
2.9 Bit Parallelism and the Word-RAM . . . 62

3 LoPRAM: A Model for Low-Degree Multi-Core Parallel Computation 65

3.1 Model . . . 67
3.1.1 Thread Model . . . 67
3.1.2 Multiprocessing Model . . . 69

3.2 Optimal Algorithm Parallelization . . . 69
3.2.1 Divide and Conquer . . . 70
3.2.2 Dynamic Programming . . . 73
3.3 Experiments . . . 81
3.4 Conclusions . . . 83

4 On the Sublinear Processor Gap for Parallel Architectures 85
4.1 Overview of Arguments . . . 86
4.2 Exposition . . . 88
4.2.1 Limited Parallelism . . . 88
4.2.2 Natural Constraints . . . 88
4.2.3 Write Conflicts . . . 89
4.2.4 Processor Communication Network . . . 91
4.2.5 Buffer Overflow . . . 92
4.2.6 Divide-and-Conquer Algorithms . . . 92
4.2.7 Cache Imposed Bounds . . . 93
4.2.8 The Class $E(p(n))$ . . . 94
4.2.9 Parallelism in Turing Machine Simulations . . . 96
4.2.10 Amdahl's Law . . . 99
4.3 Conclusions . . . 100

5 Algorithms in the Ultra-Wide Word Model 101
5.1 The Ultra-Wide Word-RAM Model . . . 103
5.1.1 UW-RAM Subroutines . . . 105
5.2 Simulation of FS-RAM . . . 107
5.2.1 Implementing FS-RAM Operations in the UW-RAM . . . 108
5.2.2 Constant Time Priority Queue . . . 110
5.2.3 Constant Time Dynamic Prefix Sums . . . 111
5.3 Dynamic Programming . . . 112

5.3.1 Subset Sum . . . 112
5.3.2 Knapsack . . . 113
5.3.3 Generalizations of Subset Sum and Knapsack Problems . . . 114
5.3.4 Longest Common Subsequence . . . 115
5.4 String Searching . . . 121
5.4.1 Shift-And and Shift-Or . . . 121
5.4.2 Boyer-Moore-Horspool (BMH) . . . 124
5.5 Conclusions . . . 125

6 Paging and Online Algorithms **127**
6.1 Online Algorithms . . . 128
6.1.1 Competitive Analysis . . . 128
6.2 Paging . . . 129
6.2.1 Paging Algorithms . . . 130
6.2.2 Other Cost Models . . . 133
6.2.3 Alternative Performance Measures . . . 134
6.2.4 Paging with Multiple Request Sequences . . . 135

7 Paging for Multi-Core Shared Caches **139**
7.1 The Cache Model . . . 141
7.2 Bounds of Online Strategies for Minimizing Faults . . . 142
7.3 The Offline Problem . . . 150
7.3.1 Hardness of Multi-Core Paging . . . 151
7.3.2 Properties of Offline Algorithms for Final-Total-Faults . . . 156
7.3.3 Optimal Algorithms for Final-Total-Faults and Partial-Individual-Faults . . . 163
7.4 Conclusions . . . 166

8 Minimizing Cache Usage in Paging **169**
8.1 Paging with Cache Usage . . . 170
8.1.1 Applications . . . 172
8.1.2 Related Cost Models . . . 173
8.2 Offline Optimum . . . 173
8.3 Online Algorithms . . . 177
8.3.1 A Family of Cost-Sensitive Online Algorithms . . . 178
8.3.2 Bounds on the Competitive Ratio of A_α . . . 180
8.4 Real World Sequences . . . 185
8.5 Conclusions . . . 185

9 Toward a Generic Hybrid CPU-GPU Parallelization of Divide-and-Conquer Algorithms **191**
9.1 Related Work . . . 193
9.2 A Hybrid CPU-GPU Model . . . 194
9.3 Generic Divide-and-Conquer Parallelization . . . 195
9.3.1 Breadth-First Structure . . . 196
9.3.2 Conversion to GPU Code . . . 197
9.3.3 Example: Divide-and-Conquer Sum . . . 197
9.4 Work Division and Scheduling Strategies . . . 198
9.4.1 Basic Hybrid Work Division . . . 198
9.4.2 Advanced Hybrid Work Division . . . 200
9.5 Case Study: Mergesort . . . 203
9.5.1 Basic Hybrid Implementation . . . 205
9.5.2 Advanced Hybrid Implementation . . . 206
9.5.3 GPU Optimizations . . . 206
9.5.4 Experimental Results . . . 207
9.6 Conclusions . . . 211

10 Conclusions **213**

References **217**

List of Tables

2.1 Flynn's taxonomy 11

2.2 I/O complexity of various problems in the Parallel External Memory model 45

2.3 Low-depth cache oblivious algorithms 49

2.4 Multi-Core oblivious algorithms in the Hierarchical Model 58

3.1 Sequential and parallel time complexities for various divide-and-conquer algorithms in the LoPRAM model 74

4.1 Low degree parallelism: summary of performance according to processor count . . 86

5.1 Wide word memory access operations supported by the UW-RAM 105

8.1 Description of simulation sequences for paging with cache usage 186

9.1 Specification of hybrid platforms used in experiments 207

9.2 Estimated parameters of platforms used in experiments 207

List of Figures

1.1 Growth in processor performance 2

2.1 The shared-memory model with p processors 12
2.2 A 3×3 mesh network of diameter 4 17
2.3 A 3-dimensional hypercube network 17
2.4 A 3-dimensional butterfly network 18
2.5 Example of a schedule of a Directed Acyclic Graph 19
2.6 Example of Amdahl's Law 27
2.7 A schematic representation of a multi-core processor 42
2.8 The Parallel External Memory model (PEM) 44
2.9 A multi-core cache model 50
2.10 Controlled-PDF schedule 52
2.11 A component in the Multi-BSP model 55
2.12 A hierarchical multi-level caching model 57
2.13 Conceptual GPU architecture 62

3.1 Example of an execution tree for mergesort in the LoPRAM 69
3.2 Execution tree of a divide-and-conquer algorithm in the LoPRAM 71
3.3 Dependency graph of the dynamic programming recurrence for Matrix Chain Multiplication 77
3.4 Times and speedups for parallel mergesort in the LoPRAM 81
3.5 Times and speedups for parallel Strassen's matrix multiplication in the LoPRAM . 82

3.6 Times and speedups for parallel Matrix Chain Multiplication in the LoPRAM . . . 83

4.1 Expected number of write access collisions . . . 91

5.1 A wide word in the Ultra-Wide Word architecture . . . 104
5.2 Illustration of the transpose operation in the Ultra-Wide Word architecture . . . 106
5.3 Yggdrasil FS-RAM memory layout . . . 108
5.4 Example of dynamic programming tables for Longest Common Subsequence . . . 118

7.1 Multi-Core paging example . . . 144
7.2 Schematic illustration of the sequence for the proof of the lower bound on the competitive ratio of a shared LRU strategy . . . 149
7.3 Schematic illustration of the NP-completeness proof of Partial-Individual-Faults . . . 151
7.4 Example of forcing a fault . . . 157
7.5 State diagram for proof of optimality of lazy algorithms . . . 159
7.6 Example of execution in proof of optimality of lazy algorithms . . . 160

8.1 Example of interval representation of a request sequence . . . 175
8.2 Cost ratio, fault rate, and average cache used of various paging algorithms on the sequence "espresso" . . . 187
8.3 Cost ratio, fault rate, and average cache used of various paging algorithms on the sequence "gs" . . . 188
8.4 Cost ratio, fault rate, and average cache used of various paging algorithms on the sequence "acroread" . . . 189
8.5 Cost ratio, fault rate, and average cache used of various paging algorithms on the sequence "grobner" . . . 190

9.1 Basic hybrid work division . . . 199
9.2 Advanced hybrid work division . . . 200
9.3 Level of the recursion tree reached and fraction of total work by the GPU as a function of the CPU-GPU work ratio for hybrid mergesort . . . 204
9.4 Advanced hybrid work division for mergesort example . . . 204

9.5 Empirical estimation of maximum number of GPU parallel threads 208
9.6 Empirical estimation of scalar performance ratio between CPU and GPU 209
9.7 Hybrid mergesort speedups as a function of CPU-GPU work ratio 209
9.8 Hybrid mergesort speedups as a function of the input size 210
9.9 Times and speedups of GPU mergesort with parallel merge 211
9.10 Best estimated and empirical work ratios and transfer levels for hybrid mergesort . 212

List of Algorithms

3.1 Parallel dynamic programming 78

5.1 Wide-word transpose 106
5.2 Wide-word reverse transpose 107
5.3 FS-RAM read 109
5.4 FS-RAM write 110
5.5 UW-RAM LCS-length 119
5.6 Shift-And 122
5.7 UW-RAM Shift-And 123
5.8 UW-RAM BMH 125

7.1 Minimum final total faults 165
7.2 Partial individual faults 167

8.1 Minimum cache usage cost 177

9.1 Generic divide-and-conquer 196
9.2 Breadth-first divide-and-conquer 196
9.3 GPU generic function 197
9.4 Divide-and-conquer sum 197
9.5 GPU sum 198
9.6 Recursive mergesort 203

9.7 Breadth-first mergesort 205
9.8 Advanced hybrid mergesort 206

Chapter 1

Introduction

In 1965, Gordon Moore, co-founder of Intel Corporation, observed that the number of transistors in integrated circuits doubled approximately every two years and predicted that this trend would continue for the foreseeable future [Moore, 1965]. Since the number of transistors in a microprocessor is closely related to its speed, this observation, commonly known as Moore's law, has also been regarded to predict that the speed of processors would double every 18 months. This form of Moore's law remained remarkably accurate for about four decades, with a notable period of steady increases of about 50% per year in performance since the early 1980's with the appearance of RISC (Reduced Instruction Set Compiler) architectures [Hennessy and Patterson, 2007]. However, due to a combination of problems arising from power dissipation, little progress in instruction-level-parallelism, and the non-matching improvement of memory latency, the yearly improvement in performance decreased to about 20% in the early 2000's (see Figure 1.1). The continuous reduction of the size of integrated circuits is nearing fundamental physical limits. Consequently, major processor manufacturers have turned to seeking improvements in performance through placing multiple *cores* on the same chip, with each core running more slowly than previous sequential processors. Using the same power as their preceding faster processors, multi-core chips would enjoy an aggregate improvement in performance. This marked an inflexion point in the history of personal computing, as the dominance of scalar processors has given way to parallel computers, which in the past were reserved mostly for High Performance Computing (HPC).

Multi-cores have become the dominant processor architecture with 2, 4, and 8 cores widely available, and with plans of an increasing number of on-chip cores in the near future. Researchers and practitioners must adapt to the new scenario. While in the past a hardware upgrade to a newer, faster processor implied a significant performance improvement for unmodified sequential applications, currently the only way to attain significant performance gains is through parallelism. As a consequence, research in multiple areas of Computer Science has turned to parallelism.

Research in parallel computation is by no means new —parallel computers and distributed

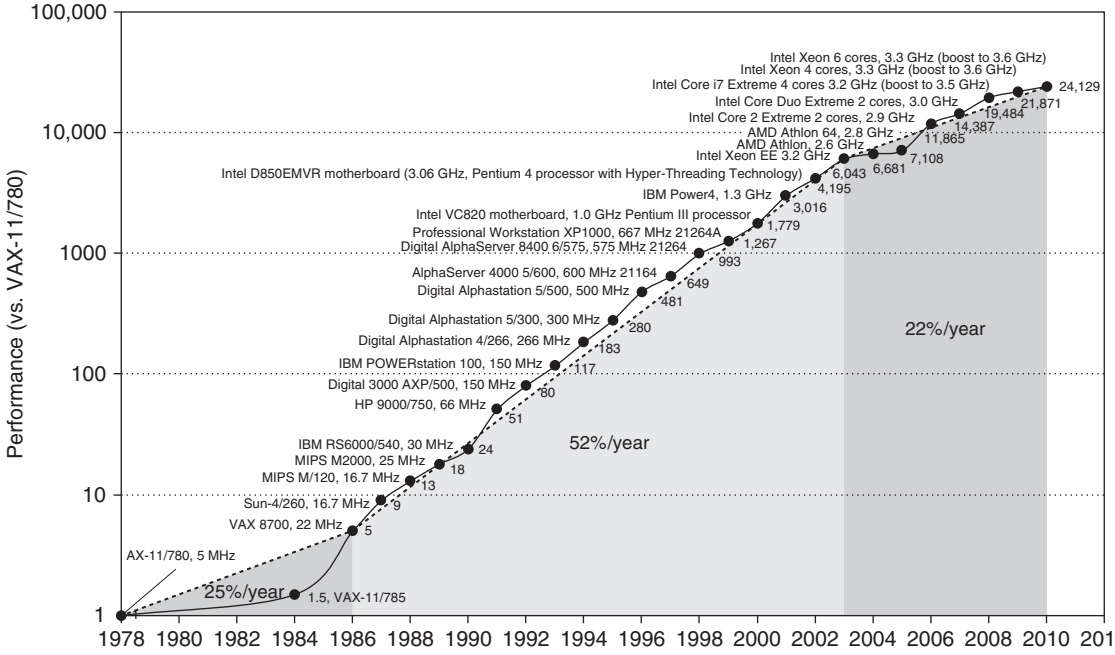


Figure 1.1: Processor performance compared to the VAX 11/780 in the period 1978-2010. The reduction in processor performance in the early 2000's led to processor manufacturers shifting to the development of multi-core processors. Figure obtained from [Hennessy and Patterson, 2011, Figure 1.1] and used with permission of Elsevier Inc.

systems have been developed for many years, and there exists a vast body of research on the subject, including theoretical models and algorithms. Nevertheless, the focus of parallel computation in the past was mainly on high performance computing, and only a small fraction of computer scientists and practitioners were familiar with the technology and theory behind it. The appearance of multi-cores marks a significant change in the focus of parallel computation. Parallelism is here, whether we like it or not, and it affects all aspects of computation, from the most performance driven scientific computations to the most popular every day applications running on a low end laptop. This change of focus implies that developing parallel programs is not anymore a task restricted to eager experts only, but could soon become part of the job of every software developer.

There are many challenges that arise in this new scenario, ranging from the development of accessible programming environments for parallel programming to determining the proper changes in the curriculum of Computer Science students to reflect the new requirements in parallel programming skills. The Theoretical Computer Science community is not exempt from the scenario change, and researchers have been faced with the fundamental problem of backing technological

advancements with models of computation and algorithms for the new multi-core architecture. For many years the Random Access Machine (RAM) model of computation faithfully represented the architecture of computers. With the advent of multi-core architectures this ceased to be the case. Currently there is no established model of computation that reflects the characteristics of commodity computers. How should we adapt existing models of parallel computation to the multi-core scenario? What algorithms are still efficient in the new architecture, and what new algorithms and algorithmic techniques could result in more efficient implementations? How can we properly manage the resources that multiple cores share in a multi-core chip? What are the trade-offs in multi-core chip design? Is it preferable to add more cores or more cache? These are only a few examples of the myriad of questions that continue to raise interesting and relevant theoretical problems, leading to a revival of research in theoretical parallel computation.

Adding to the relevance of parallel computation in today's computing landscape, the emergence of programming languages for general purpose programming on Graphic Processing Units (GPUs) has led to an increasingly large number of applications and algorithms that use these massively parallel devices to boost performance. Originally intended for exclusively executing data-parallel graphic tasks, GPUs are now regarded as commodity accelerators, whose affordable prices have made them ubiquitous in desktop and laptop computers. GPUs bring similar challenges to the ones raised by multi-cores. While there are many implementations of algorithms in GPUs, there are currently no established theoretical models to design and analyze algorithms in these devices, a task which is hindered by the diversity of designs and programming models across vendors and models.

In this thesis we address several theoretical aspects of parallel computation in these new architectures, from modeling of computation in multi-cores and heterogeneous multi-core and GPU platforms, to multi-core cache management strategies, through the proposal of a new parallel architectural feature in processor development, together with accompanying model of computation and algorithms.

1.1 Summary of Results and Structure of this Thesis

We begin by reviewing the aspects of parallel computation that are most relevant to our work in Chapter 2. We describe existing models of parallel computation, paying special attention to the shared memory model and the Parallel Random Access Machine (PRAM). We cover parallel architectures, parallel programming, and algorithmic techniques. We then describe parallelism in multi-cores, GPUs, and at the processor's word level, together with related computational models.

In Chapter 3 we introduce the LoPRAM model of computation for multi-core architectures. This model is based on the observation that the number of processors in current multi-core chips

is small compared to the large scale parallelism targeted in past models and parallel architectures. More specifically, we assume that the number of cores p is bounded by a logarithmic function on the input size, i.e., $p = O(\log n)$. We describe the characteristics of the model, including its multi-threading model and scheduler, and show how it naturally leads to work-optimal parallel algorithms for a large class of divide-and-conquer and dynamic programming algorithms via simple modifications to sequential implementations. The assumption of a small number of processors is the key factor in obtaining the optimal speedup. In contrast, in the PRAM model the design, analysis and implementation of work-optimal algorithms for the assumed $\Theta(n)$ processors proved to be one of the biggest challenges in practice for its adoption.

In Chapter 4 we further explore how a low number of processors affects the design and performance of a parallel system. We argue that design and implementation issues of algorithms and architectures are significantly different—both in theory and in practice—between computational models with high and low degrees of parallelism. We report an observed gap in the behavior of a parallel architecture depending on the number of processors that separates the performance, design, and analysis of systems with a sublinear number of processors and systems with linearly many processors. More specifically, we observe that systems with either logarithmically many cores or with $O(n^\alpha)$ cores (with $\alpha < 1$) exhibit a qualitatively different behavior than a system with $\Theta(n)$ cores in terms of the design of work-optimal algorithms, communication between processors, hardware implementations, generic simulations, and achievable speedups, among others. The evidence we present suggests the existence of a sharp theoretical gap between the classes of problems that can be efficiently parallelized with $o(n)$ processors and with $\Theta(n)$ processors, unless $\mathsf{P} = \mathsf{NC}$.

In Chapter 5 we explore an alternative view of parallelism in the form of an ultra-wide word processor. We introduce the Ultra-Wide Word architecture and model, an extension of the word-RAM model, that allows for constant time operations on thousands of bits in parallel. We argue that a large class of word-RAM algorithms can be implemented in the Ultra-Wide Word model, obtaining speedups comparable to multi-threaded computations while keeping the simplicity of programming of the sequential RAM model. We show that this is the case by describing implementations of Ultra-Wide Word algorithms for dynamic programming and string searching. In addition, we show that the Ultra-Wide Word model can be used to implement a non-standard memory architecture, which enables the sidestepping of lower bounds of important data structure problems such as priority queues and dynamic prefix sums.

Next, we turn our attention to more practical aspects of computation in multi-cores and GPUs, including paging strategies for shared caches in multi-core processors. The paging problem models a two-level memory hierarchy consisting of a fast but small cache of size k, and a slow memory of infinite size. The input to the problem is a sequence of page requests. A request is a *fault* if the corresponding page is not in the cache, and a *hit* otherwise. A paging algorithm must decide which pages to keep in cache at each request in order to minimize the number of faults. In Chapter 6 we review the paging problem in more detail, including different cost models, measures

of performance, and related work in the case of various sequences sharing a cache.

In Chapter 7 we study the paging problem for multi-core processors, which extends the classical paging problem to a setting in which several processes simultaneously share the cache. Building on a previously proposed model by Hassidim [2010] that allows paging strategies to schedule requests, we propose a more conventional model in which requests must be served as they arrive. We study the problem of minimizing the number of faults, deriving bounds on the competitive ratios of natural strategies to manage the cache. We show that traditional online paging algorithms are not competitive in our model. We then study the offline paging problem and show that the problem of deciding if a request can be served such that at a given time each sequence has faulted at most a given number of times is NP-complete and that its optimization version is APX-hard (for an unbounded number of sequences). We show as well that although offline algorithms can benefit from properly aligning future requests by means of faults, an algorithm that does so by forcing faults on pages that it has in its cache has no advantage over a lazy algorithm which evicts pages only when faults occur. Lastly, we describe offline algorithms for the decision problem and for minimizing the total number of faults that run in polynomial time in the length of the sequences.

Motivated by the paging problem in shared caches, in Chapter 8 we introduce the minimum cache usage problem, an extension to classical sequential paging in which algorithms must account for the amount of cache they use. Thus, the cost of an algorithm is a combination of the number of faults and the amount of cache it uses, with the weight of each measure depending on the intended application. We show that traditional online paging algorithms achieve the same competitive ratio k as in the classic model, but that, as expected, they do not adapt to the differences between cache and fault costs. We then describe a simple family of online algorithms for the problem that achieve a competitive ratio of 2 if $\alpha < k$, where $\alpha = f/c$ is the ratio between fault and cache cost, and $\max\left\{k, \frac{\alpha(k+1)}{\alpha+k-1}\right\}$ if $\alpha \geq k$. We further parametrize the analysis by considering the locality of reference of the sequence, and show that for sequences with high locality of reference the competitive ratio of our algorithms is at most 2. An experimental evaluation on real-world memory traces shows that our algorithms are close to optimal. We show as well that the offline problem admits a polynomial time algorithm. In doing so, we define a reduction of paging with cache usage to weighted interval scheduling on identical machines.

In Chapter 9 we address hybrid computation in a heterogeneous architecture with a multi-core processor and a GPU. Given the streaming-processing characteristics of GPUs, most practical applications so far are on highly data-parallel algorithms. Many problems, however, allow for task-parallel solutions or a combination of task and data-parallel algorithms. For these, a hybrid CPU-GPU parallel algorithm that combines the highly parallel stream-processing power of GPUs with the higher scalar power of multi-cores is likely to be superior. In this chapter we describe a generic translation of any sequential implementation of a divide-and-conquer algorithm into an implementation that benefits from running in parallel in both multi-cores and GPUs. This

translation is generic in the sense that it requires little knowledge of the particular algorithm. We then present a schedule and work division scheme that adapts to the characteristics of each algorithm and the underlying architecture, conveniently balancing the workload between GPU and CPU. Our experiments show a 4.5x gain over the single core recursive implementation, while demonstrating the accuracy and practicality of the approach.

Finally, in Chapter 10 we present the conclusions of this thesis and directions for future research.

Chapter 2

Parallel Computation

Parallel computation is a broad field that encompasses several areas of Computer Science, from theoretical parallel complexity classes to practical implementations of computer clusters with hundreds of processing nodes. In this chapter we provide an overview of various aspects of parallel computation. Before a brief review of classic sequential models of computation in Section 2.1, we turn to a description of Flynn's taxonomy and the different forms of parallel computation in Section 2.2. We then describe in Section 2.3 several models of theoretical parallel computation, with a special focus on the Parallel Random Access Machine (PRAM) model. In Section 2.4 we describe the most important types of parallel architecture designs. We describe in Section 2.5 the various models that attempt to improve the PRAM model by including relevant practical considerations. We review in Section 2.6 important aspects of writing parallel programs, including scheduling of multi-threaded computations. We describe next the most used techniques in parallel algorithm design in Section 2.7.

We then turn to parallelism in modern multi-core and many-core processors in Section 2.8, where we describe the multi-core and Graphic Processing Unit (GPU) architectures, and the recent models that have been proposed for multi-core computation. We end this chapter in Section 2.9 with a description of bit-parallelism —a form of parallel computation which takes advantage of the intrinsic parallelism of instructions in sequential processors—, together with the word-RAM model of computation.

2.1 Sequential Models of Computation

Before we turn to models of parallel computation, let us review two classic models of sequential computation: the Turing Machine and the Random Access Machine (RAM).

2.1.1 Turing Machine

The Turing machine is a simple but powerful mathematical model of a general purpose computer. A Turing machine consists of an infinite length tape and a finite set of rules to operate on symbols on the tape. The input to a computation is a set of symbols initially on the tape. The machine carries out a computation by reading symbols from the tape and writing symbols to the tape according to its set of rules, possibly stopping at some point. Formally, a Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ [Sipser, 1996], where

- Q is a finite set of states.
- Σ is a finite input alphabet, not containing a blank symbol $\sqcup$.
- Γ is a finite set of tape symbols, with $\sqcup \in \Gamma$.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\triangleleft, \triangleright\}$ is the transition function.
- $q_0 \in Q$ is the start state.
- $q_{accept} \in Q$ is the accept state.
- $q_{reject} \in Q$ is the reject state, with $q_{accept} \neq q_{reject}$.

A tape of a Turing machine is an infinite one-dimensional sequence of cells, each of which can hold one symbol, bounded to the left by a leftmost cell and unbounded to the right. The tape has a head which at any given time is placed on one cell and can be moved during the computation. The computation of a Turing machine M starts with the tape containing a string $w \in \Sigma^*$ on the leftmost cells, followed by blanks ('$\sqcup$'), with the tape's head on the leftmost cell. The state of M is initially q_0. At each step, M reads the symbol under the head and acts according to the transition function δ, possibly changing its state and either writing a symbol on the tape under its head or moving the head one cell to the left ($\triangleleft$) or right ($\triangleright$). The computation continues until M enters the accept or reject states, in which case we say M halts. Otherwise, M continues forever. A *configuration* $C = (q, T, h)$ of a Turing machine is given by its state $q \in Q$, the contents of its tape $T \in \Gamma^m$, and the position of the head in the tape h, where m is the largest cell number with a non-blank symbol. An accepting configuration is any configuration containing q_{accept}, and a rejecting configuration is a configuration containing q_{reject}. We say that M accepts (rejects) an input w if it enters an accepting (rejecting) configuration and halts. M *recognizes* a language L if it enters an accepting configuration on every input $w \in L$, and it *decides* language L if it accepts on every input $w \in L$ and it rejects on every input $w \notin L$.

The time of a Turing machine computation is the number of steps that it takes until it halts, and its space equals the maximum number of cells that its scans. For a Turing machine M

that always halts, its time (space) complexity is the function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the maximum time (space) of M on a computation on an input of size n.

There exist several variants of Turing machines that generalize and extend the definition above. Some notable examples are multi-tape Turing machines, machines with bidirectional tapes, multi-head Turing machines , restricting input alphabets to be binary, or restricting some tapes to be read- or write-only (see [Sipser, 1996; Arora and Barak, 2009] for definitions).

Turing machines are the base model for the theories of decidability and complexity of languages, this is, the study of the sets of languages that can be decided by Turing machines, and the resources (in terms of time and space) that the decision of languages require. Changes in the definitions of the original Turing machine model and its variants do not significantly affect the results that can be derived from them since each of the models can simulate the rest [Arora and Barak, 2009]. In terms of decidability theory this means that the classes of languages that can be decided remain the same across variants of Turing machines, while in terms of complexity it means that simulations can be done efficiently (i.e., with at most polynomial slowdown). This invariance with respect to changes is known as robustness [Sipser, 1996].

2.1.2 Random Access Machine

The Random Access Machine (RAM) model of computation [van Emde Boas, 1990] models the Von Neumman architecture: a processing unit and an infinite set of registers (memory) which store both the instructions of a program and the data. This model faithfully represents the design of modern computer architectures, and thus the RAM has become the standard model used in the design and analysis of algorithms for general purpose computers. The basic model consists of a control unit where a program is stored, a program counter, a set of accumulator registers, and an infinite set of memory registers. The set of instructions of the model allows moving data between accumulators and memory, performing arithmetic operations, and influencing the flow of control of the program by changing the value of the program counter either unconditionally or depending on the value of an accumulator.

The set of arithmetic instructions that are available vary across definitions of the model. The basic RAM model allows for only addition and subtraction, while more powerful models extend the instructions with multiplication, division, and bitwise Boolean instructions.

There are, in general, two ways of measuring the time complexity of a computation in the RAM model. In the *uniform* cost measure, every instruction takes one unit of time, regardless of the size of the values on which the instruction operates. In the *logarithmic* cost measure, the cost of an instruction is the sum of the of the lengths of the data involved in the instruction. Clearly, the time complexity of a uniform cost RAM is at most the one of a logarithmic cost RAM, and the gap between the two measures might be excessive depending on the instructions available. While the overhead in simulating a uniform cost RAM by a logarithmic cost RAM with

only arithmetic and boolean instructions is polynomially bounded, if multiplication is allowed then this overhead is exponential. Consequently, the uniform cost RAM with multiplication is not considered a *reasonable* sequential model in terms of the *invariance thesis*, which says that "reasonable machine models can simulate each other within a polynomially bounded overhead in time and a constant-factor overhead in space" [van Emde Boas, 1990]. In fact, $\mathsf{P} = \mathsf{NP}$ under this model [Pratt and Stockmeyer, 1976; Hartmanis and Simon, 1974].

In the analysis of algorithms in the RAM model, space is normally measured in terms of the number of registers used, which is justified by the fact that values in registers are bounded in terms of input values. However, in the theory of machine models, space is measured in terms of the size of the values stored in each register, which conforms to the logarithmic measure of time. A measure that allows the space complexity of RAMs to be fully equivalent of that of a multi-tape Turing machine is to charge every register with the size of the maximum value ever stored during a computation and to add up the cost over all registers up to the maximum address reached. The cost contributed by a register is zero if the register is unused and the sum of its address and maximum size value otherwise [van Emde Boas, 1990].

2.2 Parallelism in Computation: Flynn's Taxonomy

The most natural way of conceiving a parallel computer is as a collection of similar processors physically close to each other and with facilities to communicate and synchronize. However, parallelism in computation can take many forms, not necessarily implemented by this conception of a parallel computer. What defines a computation as parallel or sequential depends on what we consider to be the unit of information in the computation and the operations involved in it. For example, a computation on a single processor subtracting two 64-bit numbers and comparing the result to zero can be regarded as sequential if the goal is to determine which number is larger, and as parallel if the goal is to verify if there is any value of i for which the i-th bit in both words differs in value. While a subtraction of two 64-bit numbers is implemented by a parallel circuit in hardware, it is considered as a basic sequential operation in the RAM model.

A first approach to classify different kinds of parallelism in computation is to separate them along the dimension of data and instructions. The following classification is known as Flynn's taxonomy [Flynn, 1972]; it separates computation depending on whether different processors execute the same or different instructions, and on whether they operate on the same or different data. Flynn's taxonomy defines the following categories, which are shown in Figure 2.1.

Single-Instruction-Single-Data (SISD): a sequential processor executing one instruction stream on one data stream.

		Data: Single	Data: Multiple
Instructions	Single	SISD	SIMD
	Multiple	MISD	MIMD

Table 2.1: Flynn's classification of types of parallel computations.

Single-Instruction-Multiple-Data (SIMD): each processor executing the same instruction stream on different data. A simple example of such a computation is an elementwise addition of two vectors in which the addition of each element is performed by a different processor. Graphic Processing Units (GPUs) implement this mode of parallelism (see Section 2.8.3).

Multiple-Instruction-Single-Data (MISD): processors execute different instruction streams on the same data. MISD can refer to simultaneous computation on replicated data, which can be used for fault tolerance purposes, as well as to a pipelined execution [Flynn and Rudd, 1996].

Multiple-Instruction-Multiple-Data (MIMD): processors execute different instruction streams on different data. This is the least restrictive mode of parallelism, and it is the one implemented by modern multi-core processors.

2.3 Theoretical Modeling of Parallel Computation

In this section we review several models of parallel computation and the most relevant performance measures. We also define the most important parallel complexity classes.

2.3.1 The Shared-Memory Model and the PRAM

The shared-memory model is the natural extension of the RAM model. It consists of a number of processors, each with its own private memory and executing its own local program, communicating with one another by exchanging data through a shared global memory. Processors can either be *synchronous*, executing in lockstep under the control of a common clock, or they can be *asynchronous*, each of them running under its own clock [JáJá, 1992]. Figure 2.1 depicts the concept of this model.

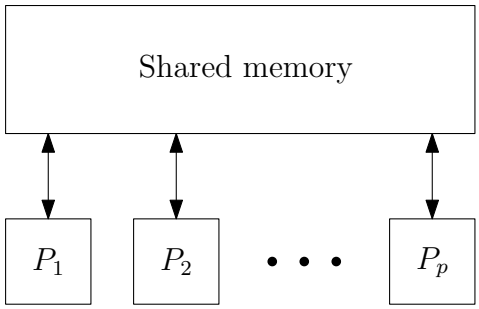


Figure 2.1: The shared-memory model with p processors.

The dominant model for theoretical research on parallel computation is the Parallel Random Access Memory (PRAM) model [Fortune and Wyllie, 1978], which is the synchronous version of the shared-memory model.

The standard PRAM model is a Multiple-Instruction-Multiple-Data (MIMD) type, since at any time each processor may execute different instructions on different data (see Section 2.2). A PRAM computation consists of a sequence of unit-cost operations of the following type: read, compute, and write [Gibbons, 1989]. In a read operation, a processor can read a global memory location and copy it into its local memory. In a compute operation, it can execute a single RAM operation on local data, storing the result in its local memory. In a write operation, a processor can write from its local memory to the shared memory.

There are different variants of PRAM models according to how they deal with memory contention. The most powerful one is the Concurrent-Read-Concurrent-Write (CRCW) PRAM, which allows unit-cost concurrent reads and concurrent writes on the same memory address, with different varieties depending on how a concurrent write is resolved: arbitrarily, by priority, randomly, or by allowing concurrent writing only when all processors attempt to write the same value (known as common CRCW) [JáJá, 1992]. On the other end, the Exclusive-Read-Exclusive-Write (EREW) PRAM model does not allow any simultaneous access to a single memory location, while the CREW model allows simultaneous read access only.

It is easy to see that any algorithm that works on an EREW PRAM works on a CREW PRAM and any algorithm that works on a CREW PRAM works on a CRCW PRAM of any type. On the other hand, any algorithm that works on a priority CRCW PRAM (the most powerful of the CRCW PRAMs [Karp and Ramachandran, 1990]) can be simulated by an EREW PRAM with the same number of processors and with a factor of $O(\log p)$ of overhead in the parallel time, where p is the number of processors [Eckstein, 1979; Vishkin, 1983; Fich et al., 1988]. In addition, a priority CRCW PRAM can be simulated by a common CRCW PRAM with no loss in time, provided that sufficiently many processors are available [Kucera, 1982].

2.3.2 Performance Measures

The performance of sequential algorithms is measured primarily in terms of worst-case time and to a lesser extent in terms of the space used. The performance of parallel algorithms has more aspects to consider. Naturally, parallel time is a key performance measure, although it is not enough to use time as the only measure of performance; the amount of resources that an algorithm uses is also relevant. For example, an algorithm A that runs in $O(n)$ parallel time with $\Theta(n^2)$ processors might not always be preferable to an algorithm B that runs in time $O(n \log n)$ but with $O(n)$ processors. Perhaps $\Theta(n^2)$ processors are not available in practice, and simulating algorithm A with fewer processor might result in an implementation which is slower than B. Algorithm B, although slower than A, performs less work per processor used and has a smaller cost in terms of the total processor-time product.

As this example illustrates, there are several considerations of relevance in the performance of a parallel algorithm. We already mentioned parallel time and the number of processors. We consider both these measures to be functions of the input size n. The number of processors is not always explicit in the description of an algorithm, and this is actually a desirable property. An algorithm should work for a non-fixed number of processors. It is thus common to have the running time as a function of both the size of the input and the number of processors used. Hence, we can talk about the performance of a parallel algorithm by specifying a time function with the number of processors p as a parameter or in terms of the minimum of this function and the value of p that realizes it. We now formalize these notions and define other important measures of performance of parallel algorithms.

Definition 2.1 (Work) *We define the* work $W(n)$ *of an algorithm as the total number of operations it performs on an input of size* n*.*

A common way of referring to the performance of a parallel algorithm is by its parallel time and work, known as the work-time framework [JáJá, 1992]. Like the number of processors in the example above, the work of an algorithm is also a second indicator of its performance and, depending on the scenario, a slow algorithm with small work might be preferable to a faster algorithm whose work is larger. Let $T(n)$ be the time of the fastest sequential algorithm for a given problem. We say that a parallel algorithm for that problem is *work-optimal* if $W(n) = \Theta(T(n))$. A work-optimal algorithm performs the same number of operations as the best sequential algorithm for the problem. A measure that is related to the work is the cost of an algorithm.

Definition 2.2 (Cost [JáJá, 1992]) *The cost of an algorithm A that solves a problem of input size n in parallel time $T_p(n)$ with $p(n)$ processors is defined as $c(n) = T_p(n) \times p(n)$.*

Note that although related, cost and work are not equal in general. The work of an algorithm is independent of the number of processors, while the cost depends on the number of processors used.

For example, consider a simple parallel algorithm to add n numbers $a_0, \ldots, a_{n-1}$, with $n = 2^k$ for some k. The algorithm works by first adding pairs of numbers in parallel: $a'_i = a_{2i} + a_{2i+1}$ for $0 \leq i \leq n/2$, and then recursively adding the numbers in a' until there is only one number left. The total work of this algorithm is $W(n) = O(n)$, which is independent of any number of processors. Since the fastest sequential algorithm to solve the problem takes time $\Theta(n)$, this algorithm is work-optimal. Now, if we use $p(n) = n/2$ processors, and at each level of the recursion a processor is assigned a pair of elements to add, the total time of the algorithm is $T_p(n) = O(\log n)$, since there are $\log n$ recursion levels and the parallel time is constant at each level. The cost of the algorithm would then be $O(n \log n)$. The two measures are, however, closely related and, as we shall see, they might coincide depending on the number of processors used by the algorithm.

A measure that describes the performance gains of a parallel algorithm over the best sequential algorithm for a given problem is the *speedup*:

Definition 2.3 (Speedup [JáJá, 1992]) *Let* Π *be a problem whose fastest sequential algorithm runs in serial time* $T(n)$*. Let* A *be a parallel algorithm for* Π *that runs in parallel time* $T_p(n)$ *with* p *processors. Then, the* speedup *achieved by* A *is*

$$S_p(n) = \frac{T(n)}{T_p(n)}.$$

Naturally, $S_p(n) \leq p$. We say that a parallel algorithm achieves optimal speedup if $S_p(n) = \Theta(p)$.

The work-time framework enables the presentation of an algorithm independently of the number of processors and a particular assignment of tasks to processors. For example, the simple algorithm to add n numbers described above can be presented as a $O(\log n)$ time and $O(n)$ work algorithm. This high level description of algorithms is even independent of an underlying parallel architecture. However, when implementing the algorithm in an actual parallel computer, the assignment of tasks to processors must be specified. The number of processors limits the number of parallel operations that can be carried out in one step of the algorithm, and thus the specification of this algorithm must take this into account. Conveniently, one can keep the original algorithm's description and apply Brent's Lemma to obtain the time bounds for an execution on a parallel computer with a fixed number of processors.

Brent's Lemma [Brent, 1974] Consider a parallel algorithm that runs in parallel time $t(n)$ and requires work $W(n)$. Suppose that at each timestep i the algorithm performs $w_i(n)$ operations. A PRAM implementation of this algorithm requires $m = \max_i\{w_i(n)\}$ processors in order to run in time $t(n)$. If the number of processors available is $p < m$, we can simulate each set

of $w_i(n)$ operations in time $\lceil w_i(n)/p \rceil \leq w_i(n)/p + 1$. Hence, the complete algorithm can be simulated in at most $\sum_{i=1}^{t(n)} w_i(n)/p + 1 = W(n)/p + t(n)$ parallel steps.

Thus, using Brent's Lemma one can specify the time of an algorithm on any number of processors. For the addition algorithm in the example with time $O(\log n)$ and work $O(n)$, the parallel time with $p \leq n/2$ processors becomes $T_p(n) = O(n/p + \log n)$. Since for this problem $T(n) = O(n)$, this algorithm achieves optimal speedup for any $p \leq n \log n$. In general, given a work-optimal algorithm of parallel time $t(n)$ and work $W(n) = T(n)$, by Brent's Lemma the time of the algorithm with p processors is $T_p(n) = O(T(n)/p + t(n))$ and thus its speedup is $S_p(n) = \Omega\left(\frac{T(n)}{T(n)/p+t(n)}\right)$. Hence, the algorithm achieves optimal speedup so long as $p = O(T(n)/t(n))$ [JáJá, 1992].

Similarly, for a (not necessarily work-optimal) algorithm of parallel time $t(n)$ and work $W(n)$, since the running time on a p-processor PRAM is $T_p(n) = O(W(n)/p + t(n))$, the corresponding cost is $c(n) = O(W(n) + t(n)p)$, and hence cost and work are asymptotically equal if $p = O(W(n)/t(n))$.

A work-optimal algorithm in general might not necessarily run in the fastest possible parallel time for a problem. An algorithm which does achieve the best possible parallel running time is called a *work-time optimal algorithm* [JáJá, 1992].

Finally, we define the notion of efficiency of a parallel algorithm.

Definition 2.4 (Efficiency [JáJá, 1992]) *Let A be parallel algorithm for a problem Π that runs in parallel time $T_p(n)$ with p processors. The* efficiency *of A is given by*

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}.$$

Note that $T_1(n)$, the time of the parallel algorithm A with one processor, is not necessarily equal to the time $T(n)$ of the best sequential algorithm for the problem, and thus the efficiency of an algorithm might be different from its speedup. While the speedup is a measure of the advantage of a parallel algorithm with p processors relative to a problem, the efficiency measures the effective utilization of p processors relative to the particular parallel algorithm A.

2.3.3 Network Models

In a shared-memory model like the PRAM model, processors have access to a common memory which is used for communication and synchronization purposes. In addition, each processor may have a local private memory. In a network model of parallel computation, we have a network

of processors, each with a local private memory, and communication takes place through the network. The most common application of this model is to distributed computation, in which a group of separate computers is connected by an interconnection network, although it can also be applied to systems in which processors are in the same computer and may have access to a shared memory in addition to being connected by a network.

In the network model, processors communicate with one another by exchanging messages, a scheme known as the *message passing model* [JáJá, 1992]. When a processor sends a message to another processor, it executes a *send message* instruction and resumes its execution immediately. When receiving a message, a processor suspends the execution of its program until the data from the sending processor is received.

Different network models are defined according to the topology of the network that connects the processors. Several parameters can be used to evaluate the topology of a network, for example, the *diameter*, which measures the maximum distance between any two nodes in the network; the *maximum degree* of a node; and the *edge connectivity* of the graph, defined as the minimum number of link deletions required to cause the network to become disconnected.

We now describe some of the common network topologies: linear array, ring, mesh, torus, hypercube, and butterfly.

The Linear Processor Array and the Ring In this model, a set of p processors $P_1, \ldots, P_p$ are connected in a linear list. The diameter of the network is $p-1$, the maximum degree is 2 and the edge connectivity is 1. In a ring model, processors are connected in a circular list. The diameter of the network in this case is $\lfloor p/2 \rfloor$, and the maximum degree and the edge connectivity are 2. The ring model is also known as a one-dimensional torus.

The Mesh In this model, $p = m^2$ processors are arranged in an $m \times m$ grid. The diameter of the network is $2\sqrt{p}-2$, the maximum degree of a node is 4, and the edge connectivity is 2. Given its $\Theta(\sqrt{p})$ diameter, many computations in this model require $\Omega(\sqrt{p})$ parallel steps [JáJá, 1992]. Figure 2.2 shows a 3×3 mesh. The 2D torus is the analogous of the ring in one dimension: each row and column forms a ring, with extreme nodes connected to each other directly. Compared to the mesh, the maximum degree is still 4, but the diameter becomes $2\lfloor \sqrt{p}/2 \rfloor$ and the edge connectivity 4. Meshes and tori can be generalized to higher dimensions.

Hypercubes A hypercube of dimension d consists of $p = 2^d$ processors indexed with numbers $0, \ldots, p-1$. Two processors are connected to each other if the binary representation of their indices differs in exactly one bit. The diameter of a d-dimensional hypercube is $d = \log p$. Every node has degree d and thus the edge connectivity is d as well. The virtues of this model are its small diameter, its regularity, its graph theoretic properties, and that many computations are simple and fast in it [JáJá, 1992]. Figure 2.3 shows a 3-dimensional hypercube network.

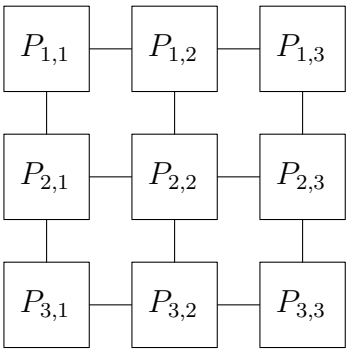


Figure 2.2: A 3×3 mesh network of diameter 4.

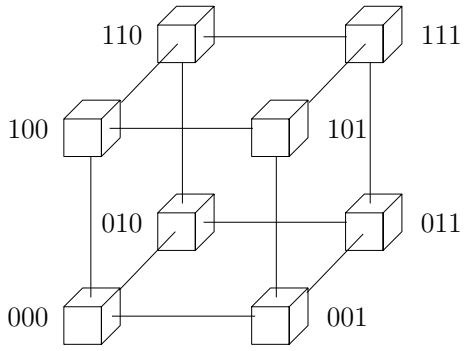


Figure 2.3: A 3-dimensional hypercube network. The index of each processor is given in binary.

Butterfly One disadvantage of the hypercube is that the degree of each node increases with the size of the network [Leighton, 1992]. The butterfly is a variant of the hypercube with similar computational properties but with bounded degree. A butterfly network of d dimensions has $p = (d+1)2^d$ nodes at various levels and rows. Each node is specified by a pair (w, i), in which i is the level of the node, with $0 \leq i \leq d$, and w is an d-bit binary number representing the row of the node [Leighton, 1992]. Two nodes (w_1, i_1) and (w_2, i_2) are connected if $i_2 = i_1 + 1$ and $w_1 = w_2$ (a straight edge), or if $i_2 = i_1 + 1$ and w_1 and w_2 differ in the i_2-th bit only, with bit indices starting from 1 (a cross edge). Figure 2.4 shows a three-dimensional butterfly network. The butterfly is similar in structure to the hypercube. In fact, a hypercube is equivalent to a butterfly with the nodes in each row merged together. Hence, the butterfly shares some of the nice properties of the hypercube, such as its recursive structure and its small diameter ($O(\log p)$) [Leighton, 1992]. Moreover, its maximum degree is 4, as each node is connected to at most two other nodes in the same row and to at most two other nodes in other rows, and its edge connectivity is 2.

2.3.4 Communication

An important feature of parallel computers is the communication between processors. In parallel architectures with distributed memory, processors communicate with each other by sending messages through a network. If the number of processors is large, not all processors are adjacent and a strategy for passing messages from one processor to another is required, a problem known as *routing*.

In shared-memory architectures, communication takes place through this memory, and each processor can communicate directly with any other. Together with time complexity of parallel algorithms, communication complexity plays an important role in measuring the performance of parallel algorithms. In the PRAM model, communication complexity is defined as the worst-case

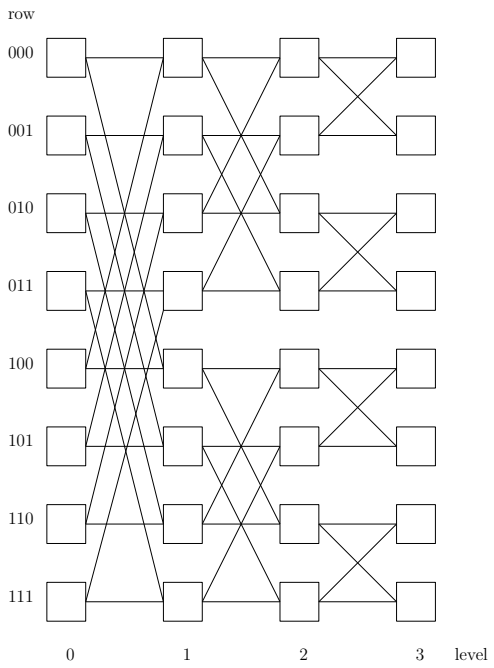
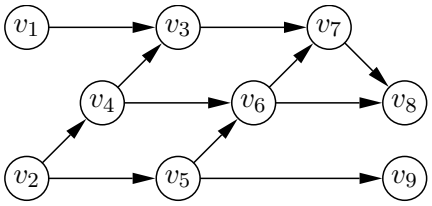


Figure 2.4: A 3-dimensional butterfly network. Figure adapted from [Leighton, 1992].

traffic between shared memory and any local memory of a processor [JáJá, 1992]. Algorithms that are optimal in terms of time complexity may not be optimal when considering communication as an additional performance parameter, and an alternative algorithm may be required in order to achieve optimality.

2.3.5 Directed Acyclic Graphs

A directed graph is a graph $G = (V, E)$ in which each edge $(u, v) \in E$ is outgoing from node u and incoming to node v. A path from a node u to a node v is a sequence of nodes $w_0, w_1, \ldots, w_k$ with $w_0 = u$, $w_k = v$, and such that $(w_i, w_{i+1}) \in E$, for $0 \leq i < k$. A cycle is a path containing at least one edge in which the first and the last nodes are the same [Cormen et al., 2001]. A directed acyclic graph (DAG) is a directed graph with no cycles. Any computation, parallel or otherwise, can be represented by a DAG that specifies the tasks to be executed and their precedence constraints. Nodes in a DAG without incoming edges are called *root* nodes, while nodes without outgoing edges are called *leaves*. In a DAG representation of a computation, each root node represents an input, each leaf represents an output, and internal nodes represent tasks or actions. The edges of the graph describe the precedence order between actions. There is an



$V_1 = \{v_1, v_2\}$
$V_2 = \{v_4, v_5\}$
$V_3 = \{v_3, v_6, v_9\}$
$V_4 = \{v_7\}$
$V_5 = \{v_8\}$

Figure 2.5: Example of a 3-schedule of a DAG of depth 5. V_i contains the nodes scheduled at step i. The schedule requires $\tau = 5$ steps.

edge between nodes u and v if the action for u must complete before the action for v can start. In this case, we say that u is a *parent* of v and v is a *child* of u. We say that u is an *ancestor* of v and v is a *descendant* of u if there is a path between u and v. The *depth* or *level* of a node v is the number of nodes in the longest path between a root node and v. The depth of a DAG is the maximum depth of any node in the graph [Blelloch et al., 1999]. Figure 2.5 shows an example of a DAG of depth 5. A task can be assumed to have unit time duration, as longer tasks can be modeled as consecutive unit time tasks. This representation is completely independent of a parallel architecture [JáJá, 1992].

In the DAG model it is assumed that any processor can access the data computed by any other processor without incurring in additional cost. A particular implementation of an algorithm can be specified by a *schedule*.

Definition 2.5 (Schedule of a DAG [Blelloch et al., 1999]) *A schedule of a DAG is a sequence of sets $V_1, \ldots, V_\tau$ where set V_i contains the nodes that are visited or scheduled at time step i, with the following properties:*

- *the sets $V_1, \ldots, V_\tau$ form a partition of V, i.e., no node belongs to more than one set, and*
- *if a node $u \in V_i$ is an ancestor of a node $v \in V_j$, then $i < j$, that is, a node is scheduled only after all its ancestors are visited.*

Thus, nodes that are scheduled in the same step are unordered. A p-schedule, for $p \geq 1$, is a schedule in which at most p nodes are scheduled at the same time, and it is *parallel* if $p > 1$ and *sequential* otherwise. Two common sequential schedules are *breadth-first* (or *level-order*) and *depth-first*. In a breadth-first 1-schedule, a node is scheduled only after all nodes of lower levels have been scheduled. A node is *ready* if it is a root node, or if all of its ancestors were scheduled. A depth-first 1-schedule, at each step, schedules a root if there are no scheduled nodes with a ready child. Otherwise, it schedules a ready child of the most recently scheduled node with a ready child [Blelloch et al., 1999].

Assuming unit time duration tasks, the time complexity of a schedule is the number of steps τ. The following well-known lemma places a lower bound on the number of steps for any p-schedule:

Lemma 2.1 *For all $p \geq 1$, any p-schedule of a DAG with n nodes and depth d requires at least $\max\{n/p, d\}$ steps.*

An example of a p-schedule with $p = 3$ is shown in Figure 2.5.

The lower bound in Lemma 2.1 is matched up to a factor of 2 by any greedy schedule. A greedy schedule is one in which no processor is idle if there is a task ready to execute. More specifically, using the terminology in Definition 2.5, a p-schedule is greedy if at each step i with r ready nodes, $|V_i| = \min\{p, r\}$ [Blelloch et al., 1999]. The following lemma bounds the number of steps of any greedy schedule:

Lemma 2.2 ([Blumofe and Leiserson, 1993]) *The number of steps of a greedy p-schedule of a DAG of n nodes and depth d is at most $n/p + d$.*

It follows from the lower bound in Lemma 2.1 that a greedy schedule gives a 2-approximation to the optimal schedule of a DAG, since $n/p + d \leq 2 \cdot \max\{n/p, d\}$.

Lemma 2.2 extends the result of Brent's Lemma (see Section 2.3.2) to greedy schedules on DAGs [Blumofe and Leiserson, 1999]. Assuming unit cost tasks, the number of nodes in a DAG corresponds to the work of the computation, while the depth corresponds to the minimum parallel time achievable. A level-order schedule of a DAG with a number of processor p' large enough so that at each step all ready nodes are scheduled can thus be seen as a parallel algorithm of work n and parallel time d. By Brent's Lemma, this computation can be scheduled on $p \leq p'$ processors in time $T_p(n) = n/p + d$.

In Section 2.6.1 we describe work stealing and parallel depth first, two of the most studied schedules in the DAG model of parallel computation. We further describe their properties with respect to cache performance in the context of multi-cores in Section 2.8.2.

2.3.6 Boolean Circuits and Parallel Complexity Classes

A theoretical computational model that has been widely used in Computational Complexity is the *circuit model* [Karp and Ramachandran, 1990]. This model is of special interest in parallel computation, since it is the basis for the definition of parallel complexity classes.

Definition 2.6 (Boolean Circuit [Karp and Ramachandran, 1990]) *A boolean circuit is a directed acyclic graph (DAG), in which nodes are labeled as* input*,* constant*, AND, OR, NOT, or* output *nodes. The in-degree of input and constant nodes is 0, the in-degree of AND and OR*

nodes is 2, and NOT and output nodes have in-degree 1. Output nodes have out-degree 0. A boolean circuit with n input nodes and m output nodes computes a function $f : \{0,1\}^n \to \{0,1\}^m$. *We define the* size *of a circuit as the number of edges in the DAG, although it can also be defined as the number of gates or the sum of both gates and edges. The* depth *of a circuit is the length of the longest path between an input and an output node.*

The size of a circuit is a measure of the amount of computational resources, while the depth of a circuit measures the time required for the computation of its boolean function, assuming unit time gates.

A boolean circuit has fixed input and output sizes and thus a different circuit is required for inputs of different sizes. Our usual notion of computation, however, is that one algorithm handles inputs of any size [Greenlaw et al., 1995]. The notion of circuit can be generalized as a model for computing functions of any input and output sizes by considering *families of circuits*, with one circuit for each input size. Without loss of generality, we can assume that the size n of an input string determines the size $\ell(n)$ of the output string [Karp and Ramachandran, 1990].

Definition 2.7 (Family of Circuits) *A family of circuits is an infinite sequence* $\mathcal{C} = \{C_i\}$, *where circuit* C_i *has size* i *and output size* $\ell(i)$.

We are interested in solving problems using families of circuits. A problem can be considered as a transducer of strings over $\{0,1\}$, i.e., a function from $\{0,1\}^n$ to $\{0,1\}^m$. We say that a family of circuits $\mathcal{C}$ solves a problem P if the function computed by C_i defines precisely the string transduction required by P for inputs of length i [Karp and Ramachandran, 1990].

We restrict families of circuits to be *uniform* and, in particular, *logspace uniform*. A family of circuits is logspace uniform if there is a logarithmic space-bounded Turing machine that on input 1^n generates a description of C_n, for each n. Without a uniformity restriction, families of circuits can be excessively powerful. For example, a circuit family could be defined such that C_n outputs 1 if the n-th Turing machine halts on its own description and 0 otherwise. This family of functions would then solve an undecidable problem [Greenlaw et al., 1995]. Intuitively, the notion of uniformity implies that the circuit construction should not exceed the computational power of the constructed circuit [Greenlaw et al., 1995]. In this sense, logspace uniformity has the property that circuits can be constructed in a reasonable time, both sequentially and in parallel [Greenlaw et al., 1995].

The Class NC

We now define complexity classes for problems that can be solved with families of circuits as follows.

Definition 2.8 (CKT$(C(n), D(n))$ [Karp and Ramachandran, 1990]) *For $C(n) \geq n$ and $D(n) \geq \log n$, a problem P is in the class $CKT(C(n), D(n))$ if there is a logspace uniform family of circuits $\mathcal{C} = \{C_n\}$ solving P such that C_n is of size $O(C(n))$ and depth $O(D(n))$.*

Since the size of a circuit is related to its work and its depth is related to its parallel time, the class defined above relates problems to the work and parallel time required to solve them. The following classes group problems that are solvable in polylogarithmic time with polynomial amount of work.

Definition 2.9 (NC^k [Karp and Ramachandran, 1990]) *The class NC^k, $k > 1$, is the class of problems that belong to $CKT(poly(n), O(\log^k n))$, where $poly(n) = \cup_{d \geq 1} O(n^d)$* [1].

Definition 2.10 (NC [Karp and Ramachandran, 1990])

$$\mathsf{NC} = \bigcup_{k \geq 1} \mathsf{NC}^k$$

NC is generally accepted as the class of problems that are solvable with a high degree of parallelism with a feasible amount of resources [Cook, 1981]. This class is robust with respect to variations of the model. For example, NC is precisely the class of problems that are solvable in a PRAM in polylogarithmic time with a polynomial number of processors [Papadimitriou, 1994].

By removing the restriction on the bounded in-degree of AND and OR gates in the circuit, we obtain the *unbounded fan-in circuit* model. The class UCKT for this model is defined analogously to the CKT class defined in Definition 2.8, and we define the class AC^k as $\mathsf{AC}^k = \text{UCKT}(poly(n), O(\log^k n))$ [2] and $\mathsf{AC} = \cup_{k \geq 0} \mathsf{AC}^k$. It is interesting to note that $\mathsf{AC} = \mathsf{NC}$ [Karp and Ramachandran, 1990].

Under the assumption of a bounded amount of local computation per processor per unit time in a PRAM, a strong relation between the computational power of unbounded fan-in circuits and CRCW PRAMs can be established [Stockmeyer and Vishkin, 1984]. Let CRCW$(P(n), T(n))$ be the class of problems solvable on a CRCW PRAM in time $O(T(n))$ with $O(P(n))$ processors. Any unbounded fan-in circuit in UCKT$(S(n), D(n))$ can be simulated by a CRCW PRAM in time $O(D(n))$ with $S(n)$ processors, and hence UCKT$(S(n), D(n)) \subseteq$ CRCW$(S(n), D(n))$.

A CRCW PRAM can also be simulated by an unbounded fan-in circuit, although special care has to be taken when simulating memory accesses. It can be shown that a single step of the

[1] The classes NC^0 and NC^1 require a different notion of uniformity. Appropriate uniformity notions for NC^0 and NC^1 are given, respectively, in terms of deterministic logarithmic time random access Turing machines (DLOGTIME uniformity), and in terms of alternating Turing machines (which we describe in Section 2.3.7) [Greenlaw et al., 1995].

[2] The notions of uniformity usually applied in the definition of AC^k are logspace uniformity for $k \geq 1$, and DLOGTIME uniformity in the case $k = 0$ [Greenlaw et al., 1995].

CRCW PRAM can be simulated by an unbounded fan-in circuit of polynomial size and constant depth and thus $\text{CRCW}(P(n), T(n)) \subseteq \text{UCKT}(poly(P(n)), T(n))$. If we define CRCW^k as the class of problems solvable by CRCW PRAM algorithms in time $O(\log^k)$ with a polynomial number of processors, we have $\text{CRCW}^k = \mathsf{AC}^k$, for $k \geq 1$, and thus $\cup_{k\geq 1}\text{CRCW}^k(poly(n), \log^k) = \mathsf{NC}$ [Stockmeyer and Vishkin, 1984]. Since we noted that simulations between the different types of PRAMs (EREW, CREW and CRCW) can be carried out with a $O(\log P)$ overhead in the parallel time (see Section 2.3.1), we have

$$\text{PRAM}(poly(n), polylog(n)) = \mathsf{NC},$$

where $polylog(n) = \cup_{d\geq 1} O((\log n)^d)$ and the PRAM processor and time bounds can refer to any of the PRAM models [Karp and Ramachandran, 1990].

Given the small number of processors in multi-core chips, our focus is primarily on moderate parallelism. The classes described above capture parallel algorithms that achieve polylogarithmic times and assume a large (polynomial) number of processors. The classes of problems defined above in terms of PRAM algorithms could be defined as well for lower degrees of parallelism, although with a different notion of parallel time achieved. With a sublinear number of processors, the times achievable by parallel algorithms are closer to the sequential complexities of problems, and thus a class definition that specifies the parallel time relative to the original sequential complexity and the number of processors explicitly can be more informative than the standard definition.

We can adapt the definitions above to the case of a sublinear number of processors as follows. We define as $\text{PRAM}_s(P(n), S(n))$ the class of problems that are solvable in a PRAM (of a particular type) with $P(n)$ processors and $S(n)$ speedup. For example, $\text{PRAM}_s(\log^k n, \log^\ell n)$ with $k \geq \ell$ is the class of problems that can be sped up by a $\log^\ell n$ factor with $\log^k n$ processors. This definition captures the extent to which a problem can be efficiently parallelized with a specific number of processors. Classes like $\text{PRAM}_s(\log^k n, \log^\ell n)$ and $\text{PRAM}_s(n^\alpha, n^\beta)$, with $\beta \leq \alpha \leq 1$, can give a more refined partition of problems in terms of their susceptibility to be parallelized with a small number of processors, a feature which is not captured by the class NC. Similar definitions of parallel complexity classes that capture efficiency are the classes ENC and EP [Kruskal et al., 1990], which are defined as the class of problems that can be parallelized optimally (up to constant factors) with a polynomial number of processors, achieving polylogarithmic and polynomial parallel time, respectively.

In Chapter 4 we elaborate on complexity classes for parallel problems that can be parallelized with a sublinear number of processors and show a strict separation between the problems that can be sped up optimally with a polynomial number of processors and those that can achieve optimal speedups with a logarithmic number of processors.

2.3.7 Alternating Turing Machines

Another formal model of parallel computation is the *alternating Turing machine (ATM)*. An ATM is a nondeterministic Turing machine whose states can be either existential or universal. As in regular Turing machines, a configuration of an ATM at one point of a computation is given by its current state, the content of its tapes, and the position of the heads on the tapes. A configuration is accepting if its state is an accepting state, it is existential if its state its existential, or it is universal if its state is universal. A regular nondeterministic Turing machine accepts a computation if any of its configurations is accepting. In ATMs, a computation from a given configuration α is accepting if:

- α is accepting or,
- α is existential and $\exists\beta$ such that α leads to β and β is accepting, or
- α is universal and $\forall\beta$ such that α leads to β, β is accepting.

An ATM accepts an input x if its computation from the initial configuration with input x is accepting. We define the class $\mathrm{ATM}(S(n), T(n))$ as the class of languages that can be accepted by an ATM in time $O(T(n))$ and space $O(S(n))$. It can be shown that for $k \geq 1$ $\mathrm{ATM}(\log n, \log^k n) = \mathsf{NC}^k$ [Ruzzo, 1981], and hence

$$\mathrm{ATM}(\log n, polylog(n)) = \mathsf{NC}.$$

Here, in order to allow sublinear computation times, a random access model can be used to read the input tape, allowing the machine to write a number indicating the address on the input tape of the symbol it reads, which takes $O(\log n)$ time.

2.3.8 Vector Machines

A vector machine [Pratt and Stockmeyer, 1976; Simon, 1977] consists of a collection of bit processors together with a set of registers that can store bit vectors. In a vector machine, the set of possible operations for a processor are boolean operations on registers, complementation of the contents of a register, conditional jump on zero, right and left shift of contents or registers, and a mask instruction that inhibits some processors from executing the next instruction.

The input is provided in a subset of the registers called the *input* registers. Each step of the computation consists of the i-th processor reading the i-th bit of the operands specified in a common instruction, executing the instruction and writing the result in the i-th bit of an output vector. In case of the shift operations, the i-th processor writes the result in the appropriate

shifted position. This way, processors are able to communicate with each other. At the end of a computation, the result is written in the *output* registers.

We define $\text{VM}(S(n), T(n))$ as the class of problems that can be solved on a vector machine with $O(S(n))$ processors in $O(T(n))$ time. When the number of processors of a vector machine is polynomial and we allow polylogarithmic time, the problems that we can solve with a vector machine are exactly those we can solve with a PRAM or a boolean circuit with the same amount of resources, i.e., $\text{VM}(poly(n), polylog(n)) = \mathsf{NC}$ [Karp and Ramachandran, 1990].

2.3.9 P-Complete Problems

A problem is in the class P if it can be decided in polynomial time by a (sequential) Turing machine. P can also be defined as the class of problems that can be solved in polynomial time on a RAM, and it is generally regarded as the class of problems that can be efficiently solved on a sequential processor [JáJá, 1992] [3]. Since an algorithm solvable in polylogarithmic time with a polynomial number of processors can be run with one processor in time bounded by a polynomial, every problem in NC is in P. However, it is not known if every problem in P is also in NC. This open question is a fundamental problem in Computational Complexity. If this was the case, it would mean that every algorithm that is solvable efficiently in a sequential model of computation can also be solved efficiently in parallel [Karp and Ramachandran, 1990].

We say that a decision problem A is *logspace reducible* to a decision problem B if there exists a function $f : \{0,1\}^* \to \{0,1\}^*$ such that f is computable by a logspace Turing machine and, for all $x \in \{0,1\}^*$, $x \in A$ if and only if $f(x) \in B$. A problem $A \in \mathsf{P}$ is P-complete if every problem $A' \in \mathsf{P}$ is log-space reducible to A. P-complete problems are of special interest because if any P-complete problem is in NC, this would imply that $\mathsf{P} = \mathsf{NC}$. No efficient parallel algorithms have been found for P-complete problems, leading to the belief that these problems are inherently sequential. The following are two examples of P-complete problems.

Example 2.1 (Monotone Circuit Value) *The Monotone Circuit Value problem (MCV) is the standard problem used to show that other problems are* P*-complete (by reduction from MCV). The input to this problem is a boolean circuit without NOT gates, such that each gate has fan-in 2 and there is a single output value, together with an assignment of a constant value 0 or 1 to each input line. The problem is to determine the output of the circuit. It can be shown that any computation of a Turing machine taking time bounded by a polynomial can be transformed in logspace to the specification of a MCV problem, and thus any problem in* P *is reducible to MCV [Karp and Ramachandran, 1990; Goldschlager, 1977].*

[3] Strictly speaking, an algorithm with a running time of $\Theta(n^{100})$ is infeasible for even moderate values of n. However, problems that are shown to be in P are usually shown to admit $O(n^k)$ algorithms for some small constant k and small constants hidden in the asymptotic notation (in some cases after several iterations of reducing the running times from algorithms with larger constants) [Arora and Barak, 2009].

Example 2.2 (Maximum-Flow) *The input to this problem is a directed graph $G = (V, E)$ with a vertex s called the source and a vertex t (different from s) called the sink. Each of the edges $e \in E$ is assigned a nonnegative capacity $c(e)$. Given such graph G and a value f, the problem is to determine whether the value of the maximum flow from s to t is at least f [Greenlaw et al., 1995; Lengauer and Wagner, 1990].*

Showing that a problem is P-complete is good evidence that it cannot be efficiently parallelized. However, there may be problems that are not efficiently parallelizable and that are not P-complete. In fact, there are various problems that are solvable in sequential polynomial time for which their parallel complexity is unknown. It is also not known if these problems are P-complete. Examples of such problems are the existence of a perfect matching in an arbitrary graph, integer greatest common divisor, and modular integer exponentiation [Karp and Ramachandran, 1990]. Although there exist parallel algorithms to solve these problems, none of these achieve polylogarithmic time with a polynomial number of processors. Thus, it is not known whether these problems are in NC. A problem that is neither in NC nor P-complete is called a P-intermediate problem. A proof of the existence of these problems would answer the P = NC question negatively, showing that not all problems are highly parallelizable.

2.3.10 Amdahl's Law

Consider a system whose performance is improved by enhancing a part of it. For example, the system can be a computer and the enhancement to replace a component by a faster one or the system could be a computer program with the enhancement being an optimization of some function in the program. Amdahl's Law states that the performance of the enhanced system is limited by the fraction of the performance to which the enhancement applies [Hennessy and Patterson, 2007]. In the context of parallel computation, we can apply the law to the speedup that can be obtained by parallelizing a fraction of a program. Let $T_1(n)$ denote the sequential time of a program. Suppose that a fraction f of the program's original running time is parallelized optimally with p processors. Amdahl's Law states that the overall execution time due to this improvement is

$$T_p(n) = \frac{T_1(n)f}{p} + T_1(n)(1 - f).$$

Hence, the speedup obtained with the improvement is

$$S_p(n) = \frac{T_1(n)}{T_p(n)} = \frac{1}{\frac{f}{p} + (1 - f)}.$$

Figure 2.6 shows the overall speedup achievable as a function of the number of processors for various fractions. This figure illustrates the limitations imposed by Amdahl's Law; the maximum

speedup achievable by a parallelization that speeds up a fraction f of the original performance is $1/(1-f)$. For example, for an enhancement that perfectly parallelizes 80% of a program, no matter how many processors are used, gains cannot go beyond a 5-fold speedup.

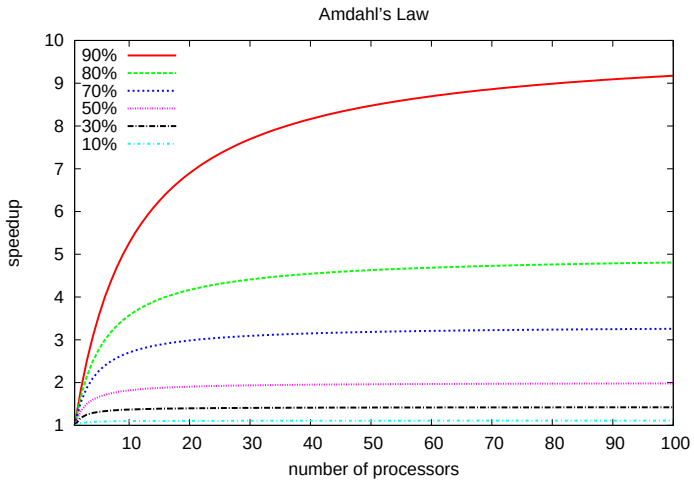


Figure 2.6: Example of Amdahl's Law: overall speedup as a function of the number of processors for a program in which a given fraction can be parallelized optimally.

2.4 Parallel Architectures

There are different ways to implement parallelism in computers. Here we describe the most common types of architectures according to how parallelism is exploited.

Pipelined Processors Consider a task S that can be broken in a number of subtasks $s_1, \ldots, s_k$. Assuming each subtask takes the same time t, the total time of a sequential execution of S would be $t \cdot k$. If we need to execute n tasks sequentially, the total time would be $n \cdot t \cdot k$. Pipelining takes advantage of the observation that after executing subtask s_1 for the first task, the subtasks of the second task can start being processed, concurrently with the ones of the first task. With pipelining, the n tasks could be executed in time $k \cdot t + (n-1) \cdot t$, where the first term corresponds to the time taken to execute the first task, and the second term corresponds to a result being produced every t units of time for the rest of the tasks [Parrot, 1987]. In a pipelined processor architecture, subtasks are handled by different processing units, increasing the number of executed operations per unit of time.

Array Processors Also known as vector processors, this architecture consists of a set of processing units, each with its own memory, running synchronously in SIMD mode directed by a single control unit. Vector processors exploit data-parallelism by executing the same instruction simultaneously on large data stored in the form of linear arrays or vectors. An example of this architecture is the family of Cray computers, which also uses pipelining in combination with vector processors. This architecture is also implemented by Graphic Processing Units (GPUs), which we describe in Section 2.8.3.

Symmetric Multiprocessing (SMP) In this architecture, a number of identical processors are attached to a bus that connects them to a shared memory. Processors commonly execute in MIMD mode and communicate with each other through the shared memory. In practice, there is contention between processors for memory access, which limits the number of processors allowed by this setup. This is the architecture implemented by multi-core processors (see Section 2.8.1).

Distributed Architectures In this type of architecture, multiple processors have each its own local private memory and are connected to each other in a network. Processors commonly run in MIMD mode and communication between processors takes place by message passing through the communication network. Communication latency is usually high compared to the cost of local computation and therefore algorithms for distributed architectures try to minimize the effects of latency by interleaving communication and computation, a technique known as *latency hiding*. Network latency depends on the physical distance between processors as well as on the network's topology (see Section 2.3.3).

Data Flow Architectures Data flow computers represent computations as graphs of dependencies. In this architecture, the order of execution of instructions does not depend on the order in which the programmer states their order but on the availability of data [Parrot, 1987]. If data is available for several instructions at the same time, then these instructions can be executed in parallel. Hence, if the architecture provides multiple processing elements and instructions and data are matched, then a complete program can be executed in parallel.

The data flow architecture can be seen as a pipelined ring structure connecting a set of processing units, a matching unit, and an instruction store. The data flows around the ring entering the matching unit, which arranges the data into sets of matched operands. These operands obtain the corresponding instruction from the instruction store, after which the processing elements are activated. Any resulting data is then put again on the ring.

2.5 Beyond the PRAM

The PRAM model has been fruitful from a theoretical perspective, mainly because it meets the high standards of descriptive simplicity set by the RAM. However, it has failed in accurately expressing the characteristics of real parallel architectures. Characteristics like the cost of non-local memory references and bandwidth limits, relevant in practice, are not included in this model, and hence the PRAM has proved to be unrealistic. Including these and other characteristics in parallel models has been the goal of various attempts to refine and improve the PRAM model. However, no agreement has been reached in terms of what characteristics should be modeled and no single model has been broadly accepted as the most suitable. Part of the success of the RAM model is due to the assignment of a unit cost to both memory and arithmetic operations, which allows for the use of asymptotic analysis. This balance between memory access and arithmetic operations is not present in parallel computation, and hence asymptotic analysis is less effective [Maggs et al., 1995]. Given the broad use of asymptotic analysis in the RAM model, the acceptance of a parallel model that considers constant factors to account for the difference between memory and arithmetic operations has been rather difficult. A second drawback of the PRAM lies in the difficulty of deriving algorithms that take full advantage of the presumed $\Theta(n)$ processors.

Following the presentation in [Maggs et al., 1995], in this section we present some of the most important models of parallel computation that were proposed to improve upon the PRAM and more accurately model the reality of (pre-multi-core) parallel architectures.

2.5.1 Variants of the PRAM Model

In what follows, we describe models that enhance the PRAM model by including some of the characteristics that are most relevant in practice.

Memory Access In many architectures it is not possible to have simultaneous access to the same memory location by more than one processor, be it for reading or writing. In Section 2.3.1, we described variants of the PRAM that restrict concurrent access to memory, restricting either writing (CREW PRAM) or both reading and writing (EREW). Although these models preserve the unit cost per memory access, they introduce the notion of serial access. Other models that introduce contention criteria for simultaneous memory access are the Module Parallel Computer (MPC) [Mehlhorn and Vishkin, 1984], which allows only one access to the same module of memory at each step; the Local-memory PRAM (LPRAM) [Aggarwal et al., 1990], which is basically a CREW PRAM where each processor is provided with unlimited amount of local memory; and the QRQW PRAM [Gibbons et al., 1994], which introduces a cost of memory access proportional to the number of processors attempting to access the same memory location at a given step.

Synchronization In the PRAM model, processors are synchronized by a global clock. In reality, different instructions do not have the same time duration. For example, an addition often is faster than a floating point multiplication or a global memory access [Gibbons, 1989]. Hence, different processors execute instructions at different speeds, and the computation is *asynchronous*. Asynchronous execution is considered in models like the APRAM [Cole and Zajicek, 1989], which introduces the *round complexity* as a measure of the total number of ticks of a virtual clock that can tick only after every processor has executed at least one instruction, thus reflecting the real time of execution from start to termination of an algorithm when running on an asynchronous machine.

The Asynchronous PRAM [Gibbons, 1989] is a PRAM in which processors run asynchronously, executing their instructions independently of other processors. Besides the read, write, and compute operations, it introduces synchronization steps. In a synchronization step between a set S of processors, all processors in S wait for all the rest of the processors in S to arrive at a logical point in the computation before proceeding. In this model a processor cannot read a memory location written by another one before a synchronization step involving the two processors takes place between these memory accesses. Thus, there are no race conditions. The cost model considers the time of processors' instructions plus the time for synchronization steps, with the total running time being the maximum between the running times of the local programs of each processor.

Another model that considers synchronization between intervals of asynchronous execution is the XPRAM [Valiant, 1990b]. In this model, operations are executed in *supersteps*. In each superstep, a processor executes a number of local operations and sends or receives messages to implement global memory accesses. The cost of a superstep for a processor is the sum of the number of its local operations and memory accesses, the latter multiplied by a constant overhead factor. The runtime of a superstep is the maximum cost of that superstep over all p processors. Synchronization takes place every $L = \log p$ global operations, with no synchronizations within this period.

Latency In practice, the time delay of a non-local memory access is higher than that of a local access or an arithmetic operation. In the context of memory accesses, the term *latency* denotes the time between a memory access request and the retrieval of the content. The LPRAM model [Aggarwal et al., 1990] suggests a fixed constant cost for a global memory access. The BPRAM [Aggarwal et al., 1989], an elaboration of the LPRAM, considers a fix cost for the first global memory access, and a variable cost for each additional access to the same memory block, thus providing incentives for locality of reference and block transfer.

Bandwidth The *bandwidth* of a channel is the maximum rate at which data can be communicated through it. Bandwidth is considered in the *Distributed Random Access Machine (DRAM)*

model [Leiserson and Maggs, 1988], in which memory accesses are implemented by routing messages through a network. The goal of the model is to reflect limited bandwidth in the communication network, thus providing incentives to limit access to remote data. Another model that considers limited communication between processors is the PRAM(m) [Mansour et al., 1994], which modifies the PRAM model by restricting the size of the global shared memory to m memory locations, with m being much smaller than the number of processors p (e.g., $m = \sqrt{n}$). The model is a priority CRCW PRAM in which the number of messages that can be transmitted between processors at any time is restricted by the size of the global memory.

2.5.2 Hierarchical Memory Models

Computers store data in various types of physical memory units, each of different size and access time. A large amount of work has been devoted to include these characteristics in theoretical models for the design and analysis of algorithms. These models seek a more refined reflection of memory access costs by defining a memory hierarchy with increasing sizes and access costs at each level. These models take into account data movements and provide incentives for exploiting data movement in parallel in the form of block transfers and parallel transfers.

Early examples of sequential hierarchical models are the HMM [Aggarwal et al., 1987a] and the BT [Aggarwal et al., 1987b] models. The HMM model considers k levels of memory, the i-th level having size 2^i and access time $i + 1$, although other cost functions are considered. The BT model extends the HMM, incorporating the possibility of moving large data in blocks. The parallel versions of these models, P-HMM and P-BT, replicate the sequential models P times and connect processors and memory through a network allowing parallel data movement.

Hierarchical memories play an important role in multi-core computation, as the way in which parallel algorithms make use of on-chip private and shared caches can significantly influence their overall performance. In fact, most models proposed for multi-cores focus on cache efficiency. We will describe the most relevant models in detail when we review models for multi-core computation in Section 2.8.2.

2.5.3 Bridging Models

As argued by Valiant [1990a], the success of the von Neumann model of sequential computation relies on its ability to serve as an efficient bridge between software and hardware, as it enables programs to run efficiently on real computers. He goes on to claim that in order for a general-purpose parallel-computing to be widely used, an analogous unifying *bridging model* is required.

The *bulk-synchronous parallel* BSP model [Valiant, 1990a] is proposed as a viable candidate for this role. It is defined as the combination of three attributes: a number of processors with

local memory, a router that delivers messages point-to-point between pairs of processors, and facilities for periodic synchronization of all or a subset of the processors. Computation can be synchronized at most every ℓ steps, which reflects the cost of a synchronization operation [Maggs et al., 1995]. It also implies a communication latency because remote accesses must wait after a synchronization operation in order for it to take effect. In addition, a parameter g is introduced as the ratio of the number of local operations to the steps required for message transmission, which enforces bandwidth limitations. This model does not charge a cost for injecting a message in the network, thus the only communication overhead is the travel time of a message of the network. However, the travel time can be hidden by performing local operations that do not require remote memory access. In this sense, the BSP model provides incentives for latency hiding [Maggs et al., 1995].

Another model proposed as a bridging model is the Candidate Type Architecture [Snyder, 1986]. This model considers two parameters: a fixed number of sequential von Neumann computers and a constant communication cost. The computers execute asynchronously with a global synchronization mechanism, and they are connected in a network of bounded degree. The two parameters and a two-level memory hierarchy provide incentives for locality of reference. However, there is no incentive for latency hiding, bandwidth management, or synchronization avoidance, since in this model synchronization is free and there are no bandwidth constrains.

The LogP model [Culler et al., 1993] is another example of a bridging model that uses the BSP model as a starting point to develop a model that would more accurately reflect the existing parallel machines at its time. The LogP is a model of distributed memory multiprocessors in which processors communicate by point-to-point messages. Unlike the BSP model, the LogP models asynchronous execution. It specifies four parameters: an upper bound L on communication latency, a communication overhead o defined as the time that a processor is engaged in transmission or reception of a message, a gap g that specifies the minimum time between two consecutive communications of a processor, and a number P of processor modules. In addition, the network has a finite capacity. The structure of the network is not specified, although these parameters model its performance characteristics. This model encourages coordination of work assignment and data placement, so as to reduce communication bandwidth requirements and the frequency of remote references. Also, the network capacity restriction encourages a careful scheduling of computation, overlapping of computation with communication, and balanced communication patterns between processors. The development of the LogP model is notable in that it comprises the efforts by researches of backgrounds in theoretical, software, and hardware disciplines, suggesting that benefits of the model can be obtained in various areas, both in the design and analysis of algorithms as well as in programming and performance [Maggs et al., 1995].

The LogGP [Alexandrov et al., 1995] model extends the LogP model to capture both short and long messages. The model thus includes parallel machines that have special support for long messages, which can provide a much higher bandwidth for long messages than for short messages.

2.6 Aspects of Parallel Programming

The goal of a programming language is to bridge the gap between the abstract description of algorithms and computer architectures. Programming languages should provide a good balance between simplicity and expressibility, at the same time retaining performance.

There exist several methods for programming parallel computers. Programming languages and tools differ from each other mainly in the programming paradigm they follow. The most common programming paradigms are message-passing, shared-memory, and data-parallel.

Message-Passing In the message-passing paradigm, processes have access to local memory and communicate with one another by explicitly sending and receiving messages. Message passing can be implemented either synchronously or asynchronously.

Shared-Memory In the shared-memory model, processes share a global common memory with a global address space. The programmer specifies which parts of a program can be executed in parallel by explicitly initiating and terminating threads. Communication between processes takes places by writing and reading data in the shared memory. This requires some method of synchronization in order to avoid inconsistencies or race conditions. There exists various *synchronization primitives* that allow threads to be coordinated. Some of these are:

- **Barrier**: when simultaneous threads are running, a thread that reaches a barrier in the code stops its execution and can only proceed after the rest of the threads have reached this point.
- **Semaphore**: A semaphore [Dijkstra, 1974] is a data structure that restricts access to a shared resource. A semaphore has a value that indicates how many units of the shared resource are free. Semaphores can be accessed through two operations P and V. When a thread wants to access the shared resource restricted by a semaphore s, it calls $P(s)$ to check if the counter associated with s is greater than zero, meaning that the resource is available. If this is the case, the counter is decreased by one. Otherwise, the thread waits until the counter gets incremented by another thread. The operation V is called when a resource is to be made available again, and it increases the value of the counter. Both operations P and V must be atomic and are usually implemented using *compare-and-swap* or *test-and-set* hardware instructions, which allow conditional modification of variable in an atomic fashion.
- **Mutex**: A mutex or *lock* is basically a semaphore with value 1. This is, there is only one unit of the resource available at a given time. The word mutex is short for *mutually*

exclusive, which indicates the use of this primitive. A mutex is used when a thread reaches a *critical section* of a program or when it needs to access a common shared variable. The use of locks and semaphores can lead to a situation known as *deadlock*. A deadlock between two processes occurs when each of them is waiting for the other one to release a resource, a scenario that can be extended to many processes. Possible solutions to avoid this condition include determining a partial order between resources so that requests are done in certain order and having *preemptive* threads, i.e., threads can be interrupted by the scheduler.

Data-Parallel In the Data-Parallel paradigm, each process operates on different parts of the same data. Hence, the approach is commonly a SIMD one. Logically, there is one thread of control, which performs operations sequentially and in parallel.

It is important to note that the programming paradigm of a language does not necessarily have to match the underlying architecture. For example, a shared-memory machine could be programmed with multiple threads sharing a common virtual address space or as different processes running on distinct processors, each with its own local address space, using messages to exchange data, which would actually be implemented as memory copies.

2.6.1 Scheduling Multi-Threaded Programs

Parallel programming in the shared-memory paradigm is most commonly implemented through multi-threading. In such implementation parallelism is expressed through the creation of asynchronous threads that can be executed in parallel, usually with synchronization points along the execution. A crucial component of the execution of a multi-threaded program is the schedule, which specifies the assignment of threads to processors and their relative order of execution. The execution schedule of a multi-threaded program can be embedded in the program itself, it can be implemented by an external scheduler, or it can be a combination of both. In Section 2.3.5 we defined the schedule of a computation represented by a directed acyclic graph (DAG). In this section we describe two of the most relevant schedules for multi-threaded computation, work stealing and parallel depth first, both of which assume the DAG model of computation.

The problem of computing the schedule of a DAG is offline if the scheduler has prior knowledge of the entire graph and online otherwise. In the latter case, the scheduler learns the structure of the DAG as the computation proceeds. For algorithms in which the structure of the computation depends only on the size of the input, the DAG can be known in advance, and a schedule can be computed offline. Matrix multiplication is an example of such algorithm. Quicksort, on the other hand, is an example of an algorithm that requires dynamic scheduling, as the split sizes are not known in advance [Blelloch et al., 1999]. In general, the DAG of the computation is not known in advance, and schedulers must make decisions based on partial knowledge. Both work stealing

and parallel depth first admit online implementations for computations with certain structural restrictions.

Work Stealing

Consider a multi-threaded computation in which each thread is divided in unit-time tasks. The high level idea of work stealing [Burton and Sleep, 1981] is that a processor that runs out of assigned tasks steals a task that is waiting to be executed by another processor. More specifically, each processor has a double-ended queue (deque) in which it stores its tasks that are ready to execute. Whenever the active task spawns another task, the new task is added to the bottom of the deque. The active task enters a waiting state, and the processor starts executing the task at the bottom of the queue. When a task is finished, if the deque is not empty, the processor takes the next task from the bottom of the deque. Note that on one processor this leads to the standard depth-first order of execution. In a parallel setting, if the deque of a processor is empty, this processor (the thief) chooses another processor (the victim) and attempts to take the task at the top of the victim's deque. If the thief succeeds, it starts executing the stolen task. Otherwise (the victim's deque might be empty or a task might be stolen by another thief), it continues trying to steal from other processors [Blumofe et al., 1996b]. A random work-stealing scheduler is one in which the thieves choose victims randomly. In general, the choice of stealing strategy may be of critical importance to the efficient performance of an algorithm. Thus, the term work stealing subsumes a large family of schedules of varying performance.

Blumofe and Leiserson [1999] present a randomized work-stealing algorithm for multithreaded computations. A multithreaded computation is modeled as a DAG of tasks connected by edges of three types: *continue* edges specify the sequential order of tasks within a thread; *spawn* edges connect a task that creates or spawns a new thread to the first task of the created thread, thus forming an activation tree; and *join* edges connect a task to one that depends on it to execute. Since generic multithreaded computations with arbitrary dependencies can be impossible to schedule efficiently [Blumofe and Leiserson, 1993], the focus of this randomized work-stealing algorithm is on *fully strict* computations. A *strict* computation is one in which all join edges go from a thread to an ancestor in the activation tree; a *fully strict* computation in one in which all join edges go from a thread to its parent thread. Fully strict computations include backtrack search computations, divide-and-conquer, as well as dataflow computations [Blumofe and Leiserson, 1999]. Blumofe and Leiserson show that randomized work stealing has good performance guarantees in terms of time, space, and expected communication between processors. More specifically, they show that for a fully strict computation represented by a DAG of depth d and total work W, the expected running time of a multi-threaded execution with p processors using work stealing, including scheduling overhead is $T_p = O(W/p + d)$, which matches Lemma 2.2. Let S_1 denote the minimum space required for a 1-processor execution of the computation. A parallel execution with work stealing requires at most $S_1 p$ space, thus achieving a linear expansion with

respect to the sequential computation. Finally, the expected number of total bytes communicated between processors during execution is $O(p\,d\,S_{max})$, where S_{max} is the size of largest activation frame of any thread[4]. This bound matches the minimum required communication for some fully strict computations for any schedule [Blumofe and Leiserson, 1999]. This model of computation and the randomized work-stealing algorithm are implemented in Cilk [Blumofe et al., 1996c], a C-based runtime system for multithreaded programming.

In a parallel computation scheduled with work stealing, tasks that are close in the computational DAG tend to be executed in the same processor [Acar et al., 2002], thus contributing to the data locality of the multithreaded computation. This property makes work stealing a schedule with generally good private cache performance. We now describe works that prove bounds on the private cache complexity of work stealing.

Blumofe et al. [1996b] analyze the performance of work stealing on fully strict computations on a distributed shared-memory system with DAG consistency, as implemented by the BACKER coherence algorithm [Blumofe et al., 1996a]. In a system with p processors, each with a private cache of size C, the time complexity of a parallel computation scheduled with work stealing is $T_p(C) = O(T_1(C)/p + m\,C\,d)$, where $T_1(C)$ is the sequential execution time including page faults, m is the cost of a page fault, and d is the depth of the corresponding DAG. The cache complexity Q_p, measured in number of faults, is bounded by $Q_1(C) + O(p\,C\,d)$, where Q_1 is the cache complexity of a sequential execution. Thus, the parallel cache complexity of the computation is only an additive term larger than the sequential one.

A similar result was shown by Acar et al. [2002] for *nested-parallel* computations on a shared memory system. A nested parallel computation is one in which the corresponding DAG is a series-parallel graph. This class of computation includes multi-threaded programs with parallel loops and fork and joins, and most computations that can be expressed in Cilk and Nesl [Blelloch, 1996]. For such computations it is shown that work stealing achieves an expected cache complexity $Q_p(C) \leq Q_1(C) + O(p\,C\,d\,\lceil m/s \rceil)$, where m is time cost of a cache miss, and s the steal time. It is shown as well that for general computations the parallel cache complexity can be much worse than the sequential one. More specifically, there is a family of computations with sequential time $T_1(n) = \Theta(n)$ and cache complexity $Q_1(C) = 3C$ that when scheduled even on two processors exhibit a cache complexity that is linear on the size of the input, i.e., $Q_2(C) = \Theta(n)$. Moreover, it is shown that the expected running time of a nested-parallel computation is $O(T_1(C)/p + m\lceil m/s \rceil Cd + (m+s)d)$, where $T_1(C)$ is the sequential execution time including cache misses.

[4]In the model of multithreaded computation in [Blumofe and Leiserson, 1999], an activation frame is a block of memory allocated for each thread for the storage of values on which it computes. The activation frame for a thread remains allocated for the duration of the thread's execution.

Parallel Depth First

Another schedule that exhibits provable time bounds and space usage is the PDF-schedule proposed by Blelloch et al. [1999]. Using the DAG model of a computation in which a node is a unit-time task, they define a parallel schedule that respects the priorities of tasks given by a sequential schedule. The idea is that a parallel schedule that deviates little from a sequential schedule can result in resource bounds that are close to those of the sequential schedule [Blelloch et al., 1999].

Blelloch et al. define a p-schedule $\mathcal{S}$ to be based on a 1-schedule $\mathcal{S}_1$ if at each step i of $\mathcal{S}$, the earliest k_i nodes that are ready in $\mathcal{S}_1$ are scheduled in $\mathcal{S}$, for some $k_i \leq p$ [Blelloch et al., 1999]. Intuitively, the p-schedule respects the priorities of the sequential schedule in the sense that at any step in the p-schedule, ready nodes are scheduled respecting the order in which these nodes are scheduled in $\mathcal{S}_1$. It could be the case, however, that nodes are scheduled in a different order in $\mathcal{S}$ and $\mathcal{S}_1$. In general, a p-schedule that schedules more than one node in several steps will schedule out-of-order with respect to most 1-schedules. For a given p-schedule, these nodes are called *premature* [Blelloch et al., 1999]. Blelloch et al. show that for any p-schedule based on a 1-schedule, the number of premature nodes is at most $(p-1)(d-1)$, where d is the depth of the DAG.

The upper bound on the number of premature nodes can be used to show that the space used by a p-schedule based on a 1-schedule is at most $S_1 + O(pd)$, where S_1 is the maximum space used by the sequential schedule and d is the depth of the DAG. Note that this improves the space bound of $S_1 p$ of work stealing whenever $d \leq S_1$, which is true for most parallel computations since S_1 is at least the size of the input and the depth of parallel algorithms is typically smaller than the size of the input [Blelloch et al., 1999]. In addition, a greedy p-schedule based on a 1-schedule is uniquely defined and takes at most $W/p + d$ steps on a DAG of work W and depth d (see Lemma 2.2).

Blelloch et al. define a PDF schedule as a p-schedule based on a depth-first schedule, and describe an algorithm to implement this schedule for the class of nested-parallel languages while keeping the space and time bounds. A nested-parallel computation of work W and depth d that allocates at most $O(W)$ space and uses sequential space S_1 can be implemented in parallel with a PDF schedule in $O(W/p + d)$ time and $S_1 + O(pd)$ space on a PRAM with prefix-sums [Blelloch, 1989] or in $O(W/p + d \log p)$ time and $S_1 + O(pd \log p)$ space on a PRAM without prefix-sums [Blelloch et al., 1999].

Subsequent work showed bounds for the PDF scheduler for computations with synchronization variables [Blelloch et al., 1997]: a computation of work W, depth d, sequential space S_1 and σ synchronizations can be executed in parallel in $S_1 + O(pd \log pd)$ space and $O(W/d + \sigma \log(pd)/p + d \log(pd))$ time on a p-processor CRCW PRAM with fetch-and-add primitive [Gottlieb et al., 1983]. If the DAG is planar, then the bounds become $S_1 + O(pd \log p)$ for space and $O(W/d + d \log p)$ for time, independent of the number of synchronizations.

Parallel schedulers based on sequential schedulers also exhibit good performance on systems with shared caches. Blelloch and Gibbons [2004] show that a computation on a cache of size C_1 that incurs M_1 cache misses under a sequential schedule $\mathcal{S}_1$ will incur no more than M_1 misses under a parallel schedule based on $\mathcal{S}_1$, provided that the shared cache has size $C_p \geq C_1 + (p-1)d$. In other words, only an additive factor over the sequential cache size is required in order for the parallel computation to incur no extra misses compared to the sequential computation. Moreover, for greedy p-schedules the parallel computation takes $W/p + d$ steps, where W is the total work of the sequential computation, and d is the depth of the computation under the p-schedule. In this model, memory accesses are an action in the DAG, and they have different weights whether their access is a cache hit or a cache miss. Thus, the weights of these actions depend on the order in which they are executed. Hence, the depth of the DAG (defined as the path of maximum weight) is schedule dependent. Note that, in general, p and d are small compared to cache sizes, as the number of processors in shared cache architectures is small, and there are many parallel algorithms with polylogarithmic depth, including the class NC [Blelloch and Gibbons, 2004].

The extra cache required for a parallel computation in order not to incur more faults than the sequential one is asymptotically tight in the sense that there exist computations for which a smaller cache is not enough. More specifically, Blelloch and Gibbons show that for any shared cache size $C_p \leq C_1 + \frac{(p-1)d_p}{3}$, there exists a computational DAG G such that a depth-first schedule $\mathcal{S}_1$ of G incurs M_1 misses, and any greedy p-schedule based on $\mathcal{S}_1$ incurs more than M_1 misses with a cache size C_p, where d_p is the depth of the DAG under the parallel schedule [Blelloch and Gibbons, 2004].

In Section 2.8.2 we describe various models that have been proposed for multi-core computation. In many of these models algorithms and scheduler are separate entities. The work-stealing and PDF schedulers that we describe in this section are two of the state-of-the-art schedulers for private and shared cache systems, respectively, and thus many of the models and results for multi-core computing are based on these schedulers.

2.7 Basic Parallel Algorithm Design Techniques

Various paradigms have been found to be useful for solving a wide range of problems in parallel. In this section we describe some of these paradigms and the problems to which they can be applied, following the presentation in [JáJá, 1992].

2.7.1 Balanced Trees

This strategy consists on building a balanced binary tree on the input elements of the problem. An internal node usually holds information about the data stored at the leaves of the subtree

rooted at this node. The computation is carried out by traversing the tree forward and backward to and from the root.

Consider the prefix-sums problem. Given a sequence of elements $\{x_i\}_{i=1}^n$ and a binary associative operation $*$, we wish to compute the n partial sums $s_i = x_1 * \ldots * x_i$, for $1 \leq i \leq n$. Even though at first glance this problem seems inherently sequential, a fast parallel algorithm can be obtained using a balanced binary tree. During the forward traversal of the tree, each node represents the application of $*$ to its two children. Thus, after this traversal, each node v stores the sum of the elements stored at the leaves of the subtree rooted at v. Then, in the backward traversal we compute the prefix sums corresponding to the nodes at each level of the tree, so that in the last step the prefix sums corresponding to the input elements are computed. An algorithm using this idea takes time $O(\log n)$ with a total of $W(n) = O(n)$ operations, which is work-time optimal in the EREW and CREW PRAM [JáJá, 1992].

2.7.2 Pointer Jumping

This technique is useful in processing data stored in linked lists or rooted-directed trees. A root-directed tree is a tree where there is a directed edge from every node (but the root) to its parent. Consider a forest F of rooted-directed trees and the problem of finding for each node, the root of its tree. F is specified in an array P such that $P(i) = j$ if j is the parent of i in a tree in F ($P(i)$ is set to i if i is a root). Initially, the successor $S(i)$ of each node is set to its parent $P(i)$. The technique of pointer jumping consists of repeatedly updating the successor of a node by the successor's successor until the root of the tree is reached. An algorithm that uses this technique can find the roots of all nodes in time $O(\log h)$ and work $W(n) = O(n \log h)$, where h is the maximum height of any tree in the forest, and n is the number of total nodes. Since a linear-time sequential algorithm exists for this problem, this algorithm is not work-optimal (unless h is a constant).

2.7.3 Pipelining

Pipelining consists of dividing tasks in several subtasks $t_1, \ldots, t_k$ such that once t_1 is completed, we can begin to process the sequence of subtasks of a second task concurrently and at the same rate as the previous task. This technique can be used, for example, for inserting a sequence of k elements into a 2-3 tree by inserting each element before all the operations corresponding to the insertion of the previous element are finished, resulting in a $O(\log n)$ time algorithm with $O(k \log n)$ operations [JáJá, 1992]. This technique has also been used to obtain optimal speedups on priority queue algorithms [Munro and Robertson, 1979].

2.7.4 Divide and Conquer

This technique consists of partitioning the input in parts of approximately equal sizes, recursively solving the subproblems for each part, and combining the results to obtain the solution of the original problem. The idea is to solve the subproblems concurrently. The effectiveness of the strategy depends on how efficiently we can partition the input and combine the results to subproblems. In some cases, a straightforward application of this technique does not lead to optimal time algorithms, mainly because of the difficulty of speeding up the merging phase. One alternative to speeding up the merging phase is to use an n^α $(0 < \alpha < 1)$ divide-and-conquer strategy, which partitions the input in n^α parts rather than into a small number of them. A second alternative is to improve the merging phase by using pipelining, in a combined strategy known as cascading divide-and-conquer [JáJá, 1992].

In Chapter 3 we analyze the divide-and-conquer strategy in the LoPRAM model, showing that for a large class of problems we can obtain optimal speedup even in the cases in which the division and merging phases are not parallelized.

2.7.5 Partitioning

This strategy is similar to divide-and-conquer in that the input is decomposed in various parts in order to solve subproblems concurrently. However, while in the divide-and-conquer strategy the main work lies in the merging phase, the partitioning technique does most of the work in dividing the input in subproblems so that the results can be easily combined later. For example, a work-optimal algorithm for merging two sequences of length n can be obtained by first choosing approximately $n/\log n$ elements of each A and B in order to partition them into blocks of almost equal length and then concurrently merging each of pair of blocks of length $O(\log n)$ with an optimal sequential algorithm. This algorithm runs in $O(\log n)$ time with a total of $O(n)$ operations on the CREW PRAM model [JáJá, 1992].

2.7.6 Accelerated Cascading

This technique consists of combining an optimal but slow algorithm and a non-optimal but fast algorithm for the same problem. For example, such technique can be used to obtain an optimal parallel algorithm for the problem of computing the maximum of a set of distinct elements. A balanced tree can be used to compute the maximum in $O(\log n)$ time using $O(n)$ operations, which is optimal in terms of work, but it can be done faster. Using a doubly logarithmic-depth tree computation, the maximum of the set can be computed in $O(\log \log n)$ time on the CRCW PRAM but with a total of $O(n \log \log n)$ operations, which makes it non-optimal in terms of work. Accelerated cascading can be used to obtain a fast work-optimal algorithm by first starting with

the optimal balanced tree algorithm until the size of the problem reaches a certain threshold and then shifting to the fast doubly-logarithmic tree algorithm. The result is an $O(\log \log n)$ time and $O(n)$ work algorithm on the CRCW PRAM, which is work-time-optimal for this model when $O(n)$ processors are available [JáJá, 1992].

2.7.7 Symmetry Breaking

This technique can be used in problems where elements in the problem are similar and a mechanism should be used to partition the elements into classes that can be processed in parallel. For example, consider the problem of k-coloring a directed cycle. The main difficulty in solving this problem in parallel lies in the apparent symmetry of the input. In order to assign different colors to different vertices, these have to be somehow distinguished from one another. However, all vertices seem equal. In order to break the symmetry, we can consider the binary representation of colors. Initially, a different color can be assigned to each vertex, and the number of colors can be reduced by a simple procedure that ensures that the colors of neighboring vertices are different by taking advantage of their binary representation. Symmetry breaking can also be implemented using randomization [JáJá, 1992].

2.8 The Multi-Core Era

The steady improvement in performance in the form of higher clock rates enjoyed by the microprocessor industry for almost four decades came to an abrupt halt in the early 2000's. In light of the little progress in Instruction Level Parallelism (ILP), the significant disparity between improvements in processor clock rate and off-chip memory latency, and more importantly the difficulties related to heat and power dissipation, leading processor manufacturers shifted away from seeking performance increases through the improvement of scalar processing and turned to parallelism, realized through the design of chips containing various relatively simpler computing units or *cores*. Compared to single core processors, multi-core processors can run at lower frequencies utilizing the power normally given to a single core processor, slowing down serial processing [Gustafson, 2011] but leading to an increase in aggregate performance [Ramanathan, 2006] [5].

Is the shift to multi-cores the end of Moore's law? Not quite. The number of transistors that we are able to put on a chip still doubles every two years. While previously the more available transistors were devoted to increase the clock rate of a single core, now they are used to include

[5] Some multi-core designs, however, allow to dynamically increase the operating frequency of cores to improve performance, subject to the number of active cores and to limits on the processor power consumption and temperature [IntelTurboBoost]. This feature could potentially be used to improve the performance of serial computation.

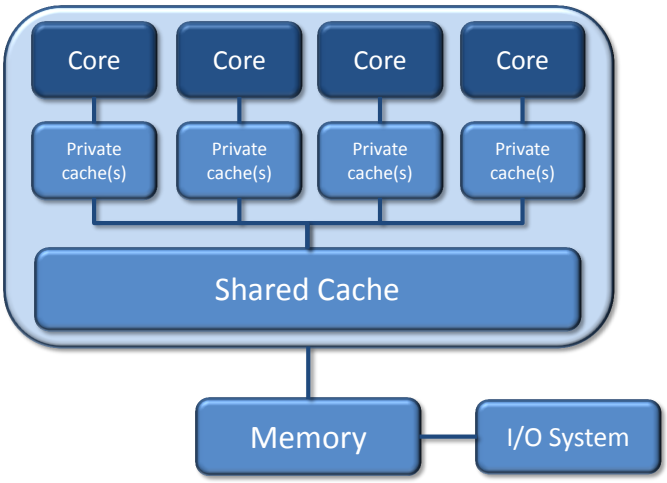


Figure 2.7: A schematic representation of a multi-core processor. Shared caches might be shared by all or some cores. Figure adapted from [Hennessy and Patterson, 2011, Chapter 5].

more cores in one chip. In this sense, a newly revised version of Moore's law stated that the number of cores on a chip would double every two years.

2.8.1 Multi-Core Architectures

In a multi-core chip, each core consists of registers, arithmetic and floating point units, and control logic [Riesen and Maccabe, 2011]. Cores in a multi-core processor share some resources such as second- or third-level caches, memory, and I/O buses [Hennessy and Patterson, 2007]. While designs vary across manufacturers and models and continue to evolve, typically each core has a private level one cache, and higher levels of cache are shared by either all or subsets of all cores. The centralized shared memory design of multi-cores with uniform access time from any core places them under the category of Symmetric Multi-Processors (SMP). Architectures with this feature are also known as uniform memory access (UMA) [Hennessy and Patterson, 2007]. This design can be scaled to a few dozen processors [Hennessy and Patterson, 2007], as with a larger number of processors it becomes hard to keep the uniform latency from all cores. Figure 2.7 depicts a schematic representation of a multi-core processor with a centralized shared memory.

Multi-core architectures belong to the category of MIMD architectures, as each core can execute different programs on different data. They allow the simultaneous execution of multiple processes with separate address spaces as well as of multiple threads sharing the same address

space. In a shared memory multi-core processor, communication between threads occurs through the shared address space via load and store operations [Hennessy and Patterson, 2007]. Data that is accessible by multiple cores is denoted as shared data. When this type of data is cached, its value might be replicated across the private caches of various cores, which reduces access latency as well as memory bandwidth and contention. However, replicating data introduces the problem of maintaining a consistent view of data by different cores, a problem known as *cache coherence* [Hennessy and Patterson, 2007]. Solutions to this problem, known as cache coherence protocols, are based on keeping track of the state of shared cache blocks. Depending on the technique used, protocols belong to one of two classes: *directory based*, in which the sharing status of a block is kept in one centralized directory, and *snooping*, in which cache controllers of each core monitor some broadcast medium connecting caches (a bus or switch) to determine if they currently have a copy of a block that is being requested by another core [Hennessy and Patterson, 2007].

As the number of cores and their speed grow, the single bus and single physical memory design of multi-cores cannot keep up with the increased memory and coherence traffic. Designers have thus used multiple buses and interconnection networks as well as multiple physical memory banks, hence increasing memory bandwidth while at the same time retaining the uniform access time to memory [Hennessy and Patterson, 2007]. In these designs, cores might be grouped in tiles, with tiles communicating through the interconnection network, sometimes referred to as Network on Chip (NoC). Moreover, cache coherency might be supported through software protocols rather than by hardware (see, e.g. [Howard et al., 2010]). Hence, some newer multi-core processors can also be considered as distributed memory systems or a combination of the latter and an SMP [Riesen and Maccabe, 2011].

2.8.2 Models for Multi-Core Computation

Multi-cores have become the standard processor architecture with two, four, and eight core processors generally available for personal computing, with the latest coprocessor releases for servers featuring 60 x86-compatible cores [IntelXeonPhi]. As parallel computing pervades general purpose computing, the software development community must adapt to the new scenario, and parallel applications should become commonplace. Implementing parallel programs requires developing parallel algorithms, whose design and analysis require, in turn, a suitable model of computation. Given the variety of designs and the constant evolution of multi-core architectures, this is a challenging task. As in any model, some characteristics of the architecture are simplified and others are emphasized. We now review some of the models that have been proposed recently for multi-core computing. Most of these focus not only on computational time but also on the cache complexity of algorithms, either of private caches, shared caches, or both.

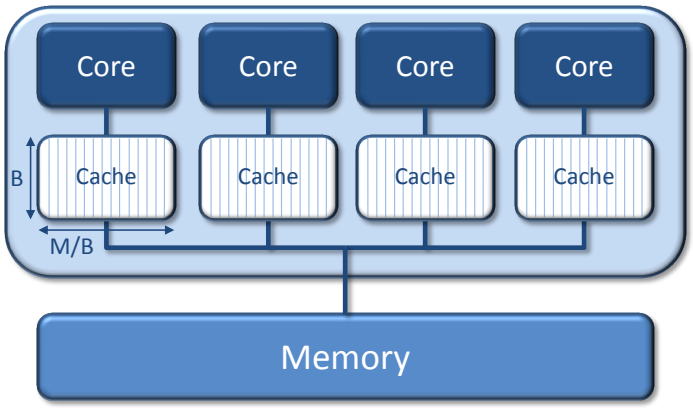


Figure 2.8: The Parallel External Memory model. Figure adapted from [Arge et al., 2008].

Private Cache Models and Applications

Various algorithms have been designed for multi-cores focusing on private cache performance under different models. These models have in common that they seek algorithms that minimize the amount of data transferred between higher levels of memory and the private caches of each core. They differ in the assumptions with respect to algorithms having knowledge of the parameters of the underlying architecture (cache and line sizes), and with respect to the mechanism used to manage data transfer to and from caches (explicitly by the algorithm or by the system's scheduler). We now describe some of these models together with the time and cache complexity bounds of the most important examples of algorithms designed in each model.

Parallel External Memory Extending the *External Memory* model of Aggarwal and Vitter [1988] to a multiprocessor setting, the *Parallel External Memory* model (PEM) by Arge et al. [2008] models a system with p processors and a two-level memory hierarchy consisting of a large shared memory and private caches for each core. Each cache is used exclusively by the corresponding processor, with all communication between processors taking place through the shared memory. All computation by a processor must be on data residing in its private cache. Each cache has size M, partitioned in blocks of size B. Data is transferred from and to main memory in blocks of size B. Figure 2.8 depicts the PEM model. As in the PRAM model, the rules with respect to simultaneous access to shared memory define the CRCW, CREW, and EREW versions of the model (see Section 2.3.1).

The PEM model combines the classic PRAM model with the external memory model. As in the PRAM, processors are assumed to be synchronous, and PRAM algorithms can be directly

	Problem	I/O complexity	Assumptions
1	-Sorting	$\Theta\left(\frac{n}{pB}\log_{\frac{n}{B}}\frac{n}{B}\right)$	$p \leq \frac{n}{B^2}$
2	-Weighted list ranking -Tree contraction -Expression tree evaluation -Euler tour	$\Theta(\text{sort}_p(n))$	$p \leq \frac{n}{B^2 \log B \log^{(t)} n}$ $p \leq \frac{n}{B^2}$
3	-Lowest common ancestor	$\Theta\left(1 + \frac{q}{n}\text{sort}_p(n)\right)$	$p \leq \frac{n}{B^2 \log B \log^{(t)} n}$
4	-Minimum spanning tree -Connected and biconnected components -Ear decomposition	$\Theta\left(\text{sort}_p(\lvert V\rvert) + \text{sort}_p(\lvert E\rvert)\log\left(\frac{\lvert V\rvert}{pB}\right)\right)$	$p \leq \frac{\lvert V\rvert+\lvert E\rvert}{B^2 \log B \log^{(t)} n}$
5	-Line segment intersection reporting	$\Theta\left(\text{sort}_p(n) + \frac{k}{pB}\right)$	$p \leq \min\{\frac{n}{B^2}, \frac{n}{B\log n}\}$

Table 2.2: I/O complexity of various problems in the Parallel External Memory model [Arge et al., 2008, 2010; Ajwani et al., 2011]. In the above, n is the input size, p is the number of processors, B is the cache block size, and $sort_p(n) = \Theta((n/pB)\log_{n/B}(n/B))$ is the I/O complexity of sorting n elements in the PEM model with p processors; $\log^{(t)} n$ denotes the composition of t log functions, where t is a constant that arises in a subroutine of the algorithm for weighted list ranking. For the lowest common ancestor problem q denotes the number of queries. V and E are the set of vertices and edges, respectively, of the input graph for the minimum spanning tree, connected and biconnected components, and ear decomposition problems. k denotes the output size in the line segment intersection reporting problem. All results are in the CREW PEM model and assume that $M = B^{O(1)}$.

simulated in the PEM model. However, the performance measure of the PEM model, known as the I/O complexity, is not the number of parallel operations but the number of parallel block transfers between memory and private caches. Thus, direct simulations of PRAM algorithms might result in inefficient implementations in the PEM model [Arge et al., 2008].

Algorithms for several fundamental problems have been designed in this model. Table 2.2 shows a summary of existing PEM algorithms together with their I/O complexity and required assumptions on the number of processors. The PEM algorithms for sorting and line segment intersection reporting are optimal [Arge et al., 2008; Ajwani et al., 2011]. The algorithms on lists, trees, and graphs (columns 2-4 in Table 2.2) achieve an optimal p-fold speedup over their external memory model counterparts [Arge et al., 2010].

Ideal Distributed Cache The Parallel External Model [Arge et al., 2008] described above features algorithms that are responsible for their schedule on multiple processors. Another approach to the design of parallel algorithms for multi-cores is to rely on an external scheduler that

is not controlled by the algorithm itself. We have described such approaches for a private cache model with work stealing and for a shared cache model with the PDF-scheduler (see Section 2.6.1) Moreover, algorithms in the PEM model know and explicitly use parameters such as the sizes of caches and blocks. Other models consider algorithms that are separated from the scheduler and that are oblivious to some or all parameters of the underlying architecture. One of these models is the ideal distributed cache model for parallel machines by Frigo and Strumpen [2006]. This model extends the ideal (sequential) cache model by Frigo et al. [1999], which introduced the concept of *cache oblivious* algorithms.

The ideal distributed cache model consists of p asynchronous processors, each with a private *ideal* cache, connected to a large shared memory. An ideal cache is fully associative and implements the optimal offline cache replacement policy (see Section 6.2). Each cache has size Z words, and it is partitioned in lines of L words, which are the units of transfer between memory and caches. A processor can only access data in its private cache, and it incurs a cache miss when it requires access to a word not in cache. The cache complexity of a computation is defined as the number of cache misses starting and ending with an empty cache [Frigo and Strumpen, 2006]. This model assumes that caches are *non-interfering*, and thus the number of cache misses can be analyzed independently for each processor. How realistic this assumption is depends on how cache consistency is maintained. For example, caches are non-interfering in the dag-consistent model of Blumofe et al. [1996a], in which consistency is implemented by the BACKER algorithm (see Section 2.6.1). Caches are also non-interfering in the private cache model of Acar et al. [2002] if the computation is race-free [Frigo and Strumpen, 2006].

Frigo and Strumpen derive a bound on the cache complexity of a parallel computation in this model in terms of its sequential cache complexity. A trace of a multithreaded computation is a sequence of the instructions of the computation in an order that is consistent with the partial order defined by the parallel computation. A segment of a trace is defined to be a subsequence of consecutive instructions in the trace. A scheduler is assumed to partition a trace into segments and assign segments to processors respecting the data dependencies of the parallel computation. For a scheduler that partitions a trace M of a computation into S segments, Frigo and Strumpen show that the total number of cache misses of a computation in the ideal distributed cache model satisfies $Q_p(M) \leq S \cdot f(|M|/S)$, where f is a concave function such that $Q(A) \leq f(|A|)$ for all segments A of M.

This result can be combined with the cache complexity bounds of Acar et al. [2002] (see Section 2.6.1) to derive bounds on the cache complexity of Cilk programs with randomized work stealing. Frigo and Strumpen show that for a program of work W and depth d with memory consistency maintained by the BACKER protocol [Blumofe et al., 1996a] $Q_p = O(S \cdot f(W/S))$ and $S = O(pd)$ with high probability, where, again, f is a concave function such that $Q(A) \leq f(|A|)$ holds for all segments A.

Frigo and Strumpen apply these results to Cilk programs for matrix multiplication and one-

dimensional stencil computations, obtaining probabilistic upper bounds on the cache complexity of multithreaded algorithms for these problems. For example, the recursive Cilk program for multiplying two $n \times n$ matrices in [Blumofe et al., 1996a] has work $W = O(n^3)$ and depth $d = O(n)$. Deriving a concave function f from the cache complexity of sequential matrix multiplication algorithms [Frigo et al., 1999], the cache complexity Q_p of the parallel program is $Q_p(n, Z, L) = O(n^3/(L\sqrt{Z}) + S^{1/3}n^2/L + S)$, where $S = O(pn)$ with high probability [Frigo and Strumpen, 2006]. Note that this bound strictly subsumes the bounds of $Q_p(n, Z) = O(n^3/(\sqrt{Z}) + Zpn)$ of Blumofe et al. [1996a], though it is not optimal [Frigo and Strumpen, 2006].

Addressing False Sharing False sharing is a consequence of maintaining cache coherency among multiple caches belonging to different processors. It occurs when two or more processors access different portions of a common block (thus each has a copy of the block in its private cache), and at least one of them writes into some location of the block. The cache coherency protocol invalidates the block for the rest of the processors, which have to fetch an updated copy the next time they access it [Cole and Ramachandran, 2012]. False sharing can occur due to a partition of data that does not match block boundaries or to cores working on small tasks and accessing the same cache block [Cole and Ramachandran, 2012]. Note that false sharing refers to the case when two or more cores access the same block in cache, but not the same data. Hence, misses due to false sharing would not occur if block updates did not propagate to other caches.

Cole and Ramachandran [2012] study the effect of false sharing on multi-core computations on private caches and develop algorithms (some of which are obtained by adapting existing algorithms) with low false sharing costs for several problems, such as scans, matrix transposition, matrix multiplication, and fast Fourier transform (FFT), among others. These algorithms are Hierarchical Balanced Parallel (HBP) computations (introduced in [Cole and Ramachandran, 2010]) that are *block-resilient*, which establishes that any shared block is accessed $O(B)$ times and hence incurs $O(B)$ misses due to false sharing (where B is the number of words in a cache block). Algorithms are oblivious to the parameters of the architecture, and achieve a relatively low cost due to false sharing misses under most schedulers [Cole and Ramachandran, 2012]. The cost due to false sharing is measured in terms of block delays, which is measured in units of cache misses. The block delay incurred by a block β is defined as the number of times β is moved from one cache to another due to false sharing misses during a time interval, and it is an upper bound on the delay incurred by a task due to false sharing misses when accessing the block β [Cole and Ramachandran, 2012]. Specific bounds are shown under a centralized scheduler S_C [Chowdhury and Ramachandran, 2010] and randomized work stealing [Cole and Ramachandran, 2011].

For example, for matrix multiplication of two $n \times n$ matrices with the standard recursive method (recursively multiplying eight $n/2 \times n/2$ matrices and performing four matrix additions to combine the results) and with Strassen's algorithm [Strassen, 1969] under the S_C scheduler on p processors, the block delay cost is $O(Bp \log p)$, which is $O(Q(n, M, B))$ (the respective

sequential cache complexity) if $n^2 = \Omega(B^{4/\lambda} M^{1/\lambda} (p \log p)^2/\lambda)$, where $\lambda = 3$ for standard matrix multiplication and $\lambda = \log_2 7$ for Strassen's algorithm [Cole and Ramachandran, 2012].

Shared Cache Models and Applications

In Section 2.6.1 we described the PDF-schedule, a parallel schedule of tasks in a DAG based on a depth-first sequential schedule. We mentioned the results by Blelloch and Gibbons which prove bounds on the shared cache complexity of parallel computations when executed with a PDF-schedule [Blelloch and Gibbons, 2004]. These results apply to the case of machines supporting simultaneous multi-threading (SMT), to shared-memory multi-processors (regarding main memory as a shared cache), as well as to multi-cores [Blelloch and Gibbons, 2004].

Recall from Section 2.6.1 that for a nested-parallel computation with sequential cache complexity $Q(n, M, B)$, where n is the size of the input, M the size of the cache, and B the size of a line in the cache, the shared cache complexity of a parallel execution with the PDF-scheduler on p processors is at most the sequential one, provided that the shared cache is an additive factor pBd larger than M, where d is the depth of the computation. In other words, $Q_p(n, M + pBd, B) \leq Q(n, M, B)$.

Low Depth Cache Oblivious Algorithms The properties of the PDF-scheduler for shared caches has inspired the design of algorithms for multi-cores that exhibit good shared cache performance. The overhead above in the cache size depends both on the depth of the computation and on the number of processors. In the case of multi-cores, the number of processors is relatively small. Based on these observations, Blelloch et al. [2010] propose an approach to obtain cache-oblivious parallel algorithms with good parallel cache complexity by designing nested-parallel algorithms with good sequential cache complexity and low depth.

The same considerations apply to systems with private caches, observing that for a shared memory machine with private caches using a work-stealing scheduler $Q_p(n, M, B) < Q(n, M, B) + O(pMd/B)$ with high probability [Acar et al., 2002]. In this case the overhead in cache complexity is also proportional to the depth of the computation.

While there exist many sequential cache-oblivious algorithms with good cache complexity, several of these do not show natural parallelizations with low depth [Blelloch et al., 2010]. Blelloch et al. describe cache-oblivious algorithms with optimal work, polylogarithmic depth, and cache complexities matching those of the best sequential algorithms for a set of important problems. These results are based on a new sorting algorithm with $O(\log^2 n)$ depth and optimal cache complexity $O((n/B) \log_M n)$. Table 2.3 summarizes the bounds obtained for various problems.

Blelloch et al. extend the cache complexity bounds based on sequential cache complexities to hierarchies of caches. They consider a Parallel Multi-level Distributed Hierarchy (PMDH), in

	Problem	Depth	Cache complexity
1	-Sorting	$O(\log^2 n)$ $O(\log^{1.5} n)$ (randomized, w.h.p.)	$O\left(\frac{n}{B}\log_M n\right)$
2	-List ranking -Euler tour on trees	$O(D_{\text{sort}}(n)\log n)$	$O(Q_{\text{sort}}(n))$
3	-Tree contraction	$O(D_{\text{sort}}(n)\log^2 n)$	$O(Q_{\text{sort}}(n))$
4	-Lowest common ancestor	$O(D_{\text{sort}}(n)\log n)$	$O(\lceil k/n\rceil Q_{\text{sort}}(n))$
5	-Minimum spanning forest -Connected components	$O(D_{\text{sort}}(n)\log^2 n)$	$O\left(Q_{\text{sort}}(\lvert E\rvert)\log\left(\frac{\lvert V\rvert}{\sqrt{M}}\right)\right)$
6	-Sparse matrix vector multiply	$O(\log^2 n)$	$O\left(\frac{m}{B}+\frac{n}{M^{1-\epsilon}}\right)$

Table 2.3: Depth and sequential cache complexity of algorithms for various problems. In the above, n is the input size, M is the cache size, B is the cache line size, $D_{\text{sort}}(n)$ and $Q_{\text{sort}}(n)$ are the depth and cache complexity of the sorting algorithm in row 1. For the lowest common ancestor problem k denotes the number of queries. V and E are the set of vertices and edges, respectively, of the input graph for the minimum spanning forest and connected components problems. For the sparse matrix vector multiply problem, m is the number of nonzero entries in the matrix, which has n^ϵ separators. All bounds assume $M = \Omega(B^2)$.

which each processor has a private multi-level cache hierarchy, as well a Parallel Multi-level Shared Hierarchy (PMSH), where all processors share a multi-level cache hierarchy [Blelloch et al., 2010]. In both models, all computation occurs on data in the first level cache. Caches are inclusive, fully associative, and implement the optimal replacement policy. In addition, for PMDH, as in the two-level cache hierarchy model of Frigo and Strumpen [2006], caches are assumed to be non-interfering, with a consistency model that is a variant of the DAG consistency cache model of Blumofe et al. [1996a].

In the PMDH model, Blelloch et al. show that for a nested-parallel computation of depth d scheduled with a work-stealing scheduler the cache complexity at each level of the hierarchy satisfies $Q_p(n, M_i, B_i) \leq Q(n, M_i, B_i) + O(pM_i d/B_i)$ with probability $1-\delta$, where M_i and B_i are the cache and line sizes at level i, respectively, $Q(n, M_i, B_i)$ is the sequential cache complexity of the computation at each level, and δ is an arbitrarily small positive constant [Blelloch et al., 2010]. Similarly, when scheduled on a PMSH using a PDF scheduler, the cache at each level i incurs fewer than $Q(p(M_i - B_i d'), B_i)$, and the computation takes at most $W'/p + d'$ steps, where d' and W' are, respectively, the depth and work of the computation including the latencies of data misses [Blelloch et al., 2010].

Private and Shared Caches

Blelloch et al. [2008] propose a model for multi-cores with both private and shared caches, reflecting the reality that multi-core chips feature small private caches per core and a large shared

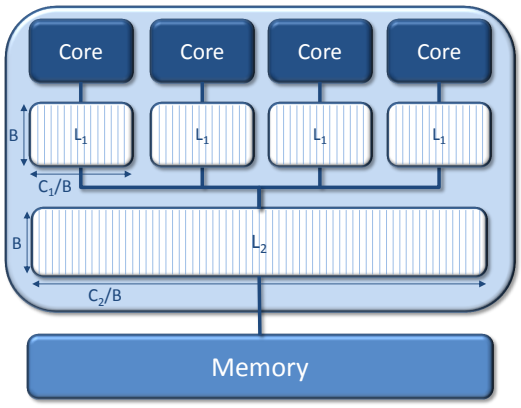


Figure 2.9: The multi-core cache model of Blelloch et al. Figure adapted from [Blelloch et al., 2008].

cache. We now describe this model as well as the results that have been obtained in it for divide-and-conquer and dynamic programming algorithms.

Scheduling Divide-and-Conquer Algorithms in Multi-Cores The multi-core model described by Blelloch et al. [2008] consists of p cores, each with a private L_1 cache of size C_1, and with all cores sharing an L_2 cache of size $C_2 \geq p \cdot C_1$. All cores share an unbounded memory, whose data is organized in blocks of size B (see Figure 2.9). Whenever a core wants to access a block from memory, it must first bring it to its L_1 cache if it is not already there. If the access is a write access and the block is in the L_1 cache of other cores, these copies are invalidated, which is enforced automatically by hardware [Blelloch et al., 2008]. Results in this model assume a least-recently-used (LRU) cache eviction policy (see Section 6.2).

An important challenge in this model is to design an online scheduler with good performance both in terms of private and shared caches, which sometimes have competing demands [Blelloch et al., 2008]. For good private cache performance, it is desirable to have processors working on disjoint data, whereas for good shared cache performance, processors should be working on the same data simultaneously. The online scheduler of this multi-core cache model is a *Controlled-PDF scheduler*, which presents good cache performance for the class of hierarchical divide-and-conquer algorithms: algorithms in which the divide and combine steps can in turn be solved by divide-and-conquer algorithms. Examples of algorithms in this class are mergesort with divide-and-conquer merge [Akl and Santoro, 1987], recursive matrix addition, cache-oblivious matrix multiplication [Frigo et al., 1999], and Strassen's matrix multiplication [Strassen, 1969], among others.

A Controlled-PDF schedule divides a computational DAG G in L_1- and L_2-supernodes. L_2-supernodes group nodes in G whose space fits in L_2 cache, and L_1-supernodes within each L_2-

supernode correspond to computation that fits in L_1 cache. The scheduler then considers each L_2-supernode at a time in a depth-first order and schedules the L_1-supernodes within each L_2-supernodes in parallel according to a PDF-schedule (see Section 2.6.1). Each L_1-supernode is scheduled to execute entirely in one core, and only once all L_1-supernodes within an L_2-supernode have finished, the scheduler continues with the next L_2-supernode (see Figure 2.10).

In order for the scheduler to determine the sizes of supernodes for a particular algorithm, the user must specify a space function $S(n)$ equal to the space required to solve the problem sequentially. The space requirements for a hierarchical divide-and-conquer algorithm can be determined by solving a recurrence that combines the spaces used by the algorithms involved, and assuming a depth-first execution. For example, for the recursive addition of two $n \times n$ matrices, $S(n) = S(n/2) + O(n^2)$, and thus $S(n) = O(n^2)$. For Strassen's matrix multiplication, which uses the recursive matrix addition, the space $S(n)$ satisfies the same recurrence and therefore $S(n) = O(n^2)$. In addition to the space usage, the user must also specify the ratio r between the parallel and sequential space usage of the algorithm. In order to achieve the desired cache bounds of a parallel execution, the model requires this ratio to be constant. This requirement might not be satisfied by a direct breadth-first parallelization of some algorithms. For example, the direct parallelization of Strassen's matrix multiplication requires space $p^{1-\frac{2}{\log_2 7}} n^2$, and hence the space grows with the number of processors. Therefore, a modified version of the hierarchical algorithm must be devised [Blelloch et al., 2008].

Given a space function $S(n)$ and ratio r, the scheduler chooses $n_1 = S^{-1}(C_1)$ and $n_2 = S^{-1}\left(\frac{\alpha}{r} S(n_1)\right)$, where α is such that $C_2 \geq \alpha \cdot C_1$. Then, n_1 and n_2 are the input sizes of recursive calls that form L_1- and L_2-supernodes, respectively. The scheduler is aware of the particular cache sizes of the underlying system, but the algorithm need not know about them.

When scheduled with a Controlled-PDF schedule, a hierarchical divide-and-conquer algorithm will incur a number of cache misses in both L_1 and L_2 caches that is within a constant factor of the number of misses of a sequential execution [Blelloch et al., 2008].

Regarding the parallel time, note that the Controlled-PDF scheduler is not a greedy one, as processors may be idle while waiting for the completion of L_2-supernodes, which is due to the fact that each L_1-supernode is executed sequentially in one processor. This is a consequence of the scheduler favouring cache performance over parallelism. In order to achieve a p-fold speedup in the parallel time, some assumptions have to be made with respect to the relation between cache sizes. In turn, these assumptions depend on the inherent parallelism of the algorithm, and in particular on the inherent parallelism of L_2-supernodes, as we explain now.

The Controlled-PDF executes an L_2-supernode in time $T_p(n_2) = T(n_2)/p + T_\infty(n_2, n_1)$, where $T(n)$ is the sequential time of the algorithm and $T_\infty(n_2, n_1)$ is the inherent parallelism of the L_2-supernode under the Controlled-PDF schedule (i.e., the execution time with an infinite number of processors). Then, the schedule will achieve optimal speedup if $T(n_2)/p = \Omega(T_\infty(n_2, n_1))$. Parallelism is achieved in the portion of the algorithm that reduces the input from n_2 to n_1 through

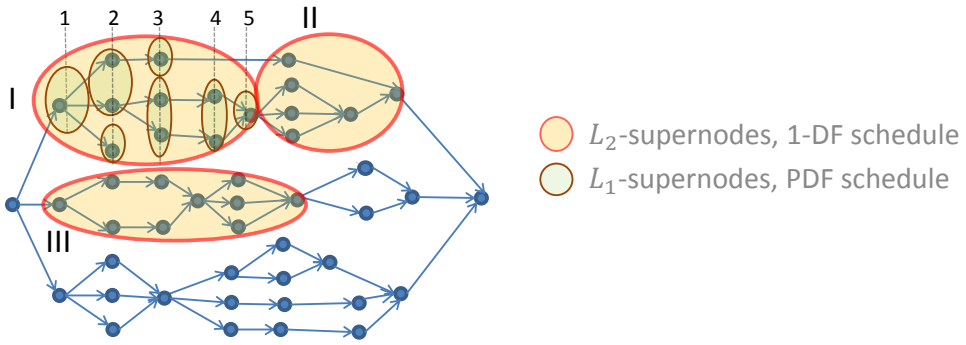


Figure 2.10: Example of a controlled-PDF schedule [Blelloch et al., 2008]. L_2-supernodes are scheduled sequentially in depth-first order and L_1-supernodes are scheduled with a PDF-scheduler. Roman and Arabic numerals indicate, respectively, a possible order of execution of L_2-supernodes and L_1-supernodes within an L_2-supernode.

recursive calls. Since n_2 depends on the parameter α, the ability to obtain optimal speedups will depend on whether α is large enough to satisfy the above condition. The model requires $\alpha \geq p$, but a larger α might be required to satisfy the condition. How large the value of α should be depends on the parallelism of the algorithm [Blelloch et al., 2008]. For example, for recursive matrix addition, $T(n) = n^2$, $n_1 = \sqrt{C_1}$, $T_\infty(n_2, n_1) = C_1 + \log \alpha$, and $\alpha \geq p$. For recursive merge, $T(n) = n$, $n_1 = C_1$, $T_\infty(n_2, n_1) = C_1 + \log \alpha \cdot \log(\alpha C_1)$, and $\alpha \geq p\left(1 + \frac{\log \alpha \cdot \log(\alpha C_1)}{C_1}\right)$ [Blelloch et al., 2008].

Dynamic Programming and Cache Performance Chowdhury and Ramachandran [2008] present cache-efficient dynamic programming algorithms for the multi-core model described in [Blelloch et al., 2008] as well as in models with private caches only [Frigo and Strumpen, 2006] and shared caches only [Blelloch and Gibbons, 2004]. They develop algorithms for three general classes of dynamic programming (DP) problems: Local Dependency DP (LDDP) (e.g., longest common subsequence, pairwise sequence alignment with affine gap cost), Gaussian Elimination Paradigm (GEP) (e.g., all-pairs shortest-paths, LU decomposition), and Parenthesis problems (e.g., matrix chain multiplication). A combination of LDDP and GEP yields results for a problem in a fourth category they call RNA secondary structure prediction with simple pseudo-knots.

The general strategy to solve dynamic programming problems used by Chowdhury and Ramachandran is to divide DP tables in tiles and recursively solve the problem in each tile. For each class of DP problems, and for each cache model, they describe a tiling sequence for each level of the recursion and give a parallel schedule that yields good performance in terms of both parallel time and cache efficiency [Chowdhury and Ramachandran, 2008].

For example, consider the longest common subsequence (LCS) problem:

Definition 2.11 (Longest Common Subsequence (LCS)) *Let $S = s_1 s_2 \dots s_n$ be a string. A subsequence of S is a string $S' = s_{i_1} s_{i_2} \dots s_{i_k}$ with $i_1 < i_2 < \dots < i_k$, i.e., a string containing a subset of not necessarily contiguous characters of S, in the same order they appear in S. Given two strings S and T, the LCS problem asks for the longest string that is a subsequences of both S and T.*

For a string S, let $S[i..j]$ denote the substring of S containing characters $s_i s_{i+1} \dots s_j$. Let $D[i, j]$ denote the length of the longest common subsequence between $S[1..i]$ and $T[1..j]$. Then the length of LCS between S and T can be solved using the following recurrence [Cormen et al., 2001]:

$$D[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ D[i-1, j-1] & \text{if } i, j > 0 \text{ and } s_i = t_j \\ \max\{D[i, j-1], D[i-1, j]\} & \text{if } i, j > 0 \text{ and } s_i \neq t_j \end{cases} \tag{2.8.1}$$

For an LCS instance with two strings of length n, the two-dimensional table D is divided in tiles, which are then computed recursively. In each level of the recursion subtables are divided in τ^2 sub-tables of equal size, where $\tau = t[d]$ is the parameter of the tiling sequence at recursion depth d. Sub-tables are then computed recursively in $2\tau - 1$ parallel steps along the diagonals of the table. Parameters of the tiling sequence and the schedule are defined as follows for each of the cache models [Chowdhury and Ramachandran, 2008]:

- If $t[d] = 2$ for all d, an execution with one processor yields that the cache-oblivious sequential algorithm in [Chowdhury and Ramachandran, 2006], whose I/O complexity is $O(n^2/(MB))$, where M is the cache size and B the length of a cache line.
- For a model of private caches, the tiling sequence has parameters $t[0] = p$ and $t[d] = 2$ for $d \geq 1$. That is, the table is initially divided in p^2 sub-tables of size $(n/p)^2$ each. The i-th processor is assigned the i-th column of sub-tables in the first level of the recursion, after which each sub-table is computed sequentially in each processor. This results in a diagonal-by-diagonal execution of the table based on the sub-tables. The parallel running time obtained is $T_p(n) = O(n^2/p + n)$, and the number of cache misses is $O(n^2/(MB))$ (under the assumptions that $n \geq pM$ and $B \leq M$) [Chowdhury and Ramachandran, 2008].
- For the case of a single shared cache, tiling parameters are $t[d] = 2$ for $d < \log(n/p)$ and $t[\log(n/p)] = p$. Hence, at level $d = \log(n/p)$ each tile corresponds to a $(n/p) \times (n/p)$ sub-table. The schedule is such that all processors work together in single tiles. Once again the parallel time is $T_p(n) = O(n^2/p + n)$, while the cache complexity is $O(n^2/(MB))$.

- Finally, for the multi-core cache model with private L_1 caches of size C_1 and a shared L_2 cache of size C_2 the strategy combines the approaches for the private-only and shared-only models. The tiling parameters are $t[r] = p$ for $r = \log(n/C_2)$ and $t[d] = 2$ otherwise. Thus, this strategy splits the table until the size of subproblems corresponding to a tile equals the size of L_2 caches, after which it applies the strategy for private caches. In this case the running time is again $T_p(n) = O(n^2/p + n)$, and the L_1 and L_2 cache complexities are $O(n^2/(C_1B))$ and $O(n^2/(C_2B))$, respectively.

Similar results are obtained for the rest of the DP paradigms, resulting in cache-efficient parallel executions up to the critical path length of the dynamic programming algorithms [Chowdhury and Ramachandran, 2008].

Hierarchical Models

Most of the results that we have described so far focus on settings with private caches to each core, and possibly a shared cache among all cores. In many multi-core designs, however, caches may be shared by a subset of all cores, with higher level caches being shared by a larger number of cores. In this section we describe results that consider more general hierarchical models, in which the layout of caches and cores follows a tree-like structure: leaves correspond to cores, and internal nodes to caches, with each cache being shared among all the leaves in their corresponding subtree.

Multi-BSP: A Bridging Model for Multi-Core Computing The Multi-BSP model proposed by Valiant [2011] is an adaptation of the BSP model [Valiant, 1990a] to multi-cores (see Section 2.5.3). The model has explicit parameters for various important aspects of parallel computation in multi-cores: number of processors, sizes of memories and caches, and communication and synchronization costs. The model seeks algorithms that are efficient for any combination of values of these parameters, and hence that are portable across platforms of varying characteristics.

The Multi-BSP model consists of multiple levels of components and is recursive: each level j consists of 4 parameters (p_j, g_j, L_j, m_j), where p_j is the number of level $(j-1)$ components, g_j is the data rate at which the j-th component communicates with the $(j+1)$-th component, L_j is the synchronization cost, and m_j is the size of the memory. The components at level 0 are the cores or processors, and the components at higher levels are either caches or memories. An instance of the model is a tree structure of some depth d, which is the number of levels. Figure 2.11 shows a generic component of the model at some level j. It is assumed that the last level memory is sufficient to support the computation. In particular $m_d \geq n$. In addition, it is assumed that $m_i \geq m_{i-1}$ for all i, and that $g_d = \infty$.

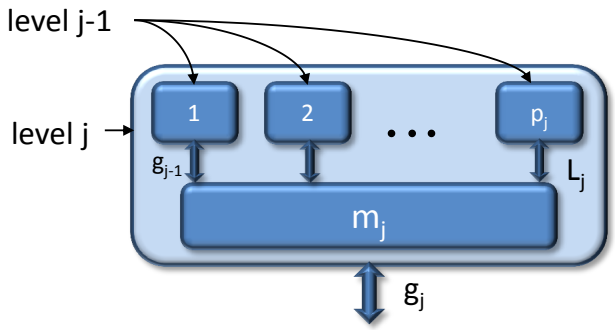


Figure 2.11: A component at level j in the Multi-BSP model. It contains p_j components at level $j-1$, which synchronize at a cost of L_j, and communicate at a rate g_{j-1} with the memory of size m_j of the component at level j. Figure adapted from [Valiant, 2011].

For example, an instance with $d = 1$ and parameters $(p_1 = 1, g_1 = \infty, L_1 = 0, m_1)$ is the von Neumann model, while an instance with parameters $(p_1 \geq 1, g_1 = \infty, L_1 = 0, m_1)$ is the PRAM, where m_1 is the size of the memory [Valiant, 2011]. An attempt to model a more sophisticated architecture would be that of modeling an instance of p Sun Niagara UltraSparc T1 multi-cores, connected to an external storage. In this case the instance has 3 levels with the following parameters: level 1: $(p_1 = 4, g_1 = 1, L_1 = 3, m_1 = 8$ Kb) (each core runs 4 threads and has an L1 cache of size 8 Kb); level 2: $(p_2 = 8, g_2 = 3, L_2 = 23, m_2 = 3$ Mb) (each chip has 8 cores with a shared L2 cache of size 3 Mb); level 3: $(p_3 = p, g_3 = \infty, L_1 = 108, m_3 = 128$ Gb) (p chips with external memory of size m_3 accessible via a network at a rate g_2). In this example, L_j parameters are not exactly synchronization costs but latency parameters given in the chip specification. In addition, caches are managed implicitly by hardware and not by the algorithm. Thus, while the relative values of the g and L parameters are meaningful, it is hard to obtain their exact values [Valiant, 2011].

The notion of optimality of an algorithm A in this model is defined relative to a given baseline algorithm B in terms of the relation between their computation steps, parallel communication costs, and synchronization costs. An algorithm A is said to be optimal with respect to algorithm B if the parallel computation steps, parallel communication costs, and synchronization costs of A are within multiplicative constant factors of those of B, and the total computation steps are within additive constant factors of the number of steps of B. The constant factors may depend on the level d but not on any other parameter of the model.

Valiant gives matching upper and lower bounds for Multi-BSP algorithms for matrix multiplication, fast Fourier transform, comparison-based sorting, as well as associative composition. In the latter problem, given an array A of n elements of a set X, an associative binary operation $*$ on X, and a set of disjoint contiguous subarrays of A, the goal is to compute the composition of each subarray under $*$, with this being the only operation allowed on elements of X [Valiant,

2011].

The main differences between this model and most of the models that we have described is obviously the large number of parameters, the fact that algorithms are aware of these parameters, and that the scheduling and management of data across components is done explicitly by the algorithm. While the simplicity in the design and analysis of algorithms suffers from this proliferation of parameters, the goal of the model is that the design of some fundamental algorithms, although difficult, will result in efficient algorithms that are portable for all machines.

Next, we describe other models that consider hierarchical caches but in which algorithms and scheduler are separate entities and moreover. Moreover, algorithms in these models have no notion of the parameters of the architecture.

Oblivious Algorithms in the HM Model Chowdhury et al. [2010] present the HM model, a multi-level hierarchical multi-core model in which caches at increasing levels have increasing sizes and are shared by larger groups of cores. The HM model, which had been briefly considered previously in [Chowdhury and Ramachandran, 2008], consists of p processors, organized in a hierarchical cache structure of h levels, where at each level $1 \leq i \leq h-1$ there are q_i L_i caches of size C_i and block size B_i, and with a shared memory of arbitrarily large size at level h. The number of level-$(i-1)$ caches that share a given L_i cache is denoted by p_i. The model assumes $p_1 = 1$ (i.e., each core has a private L_1 cache) and that $p_h = 1$ (i.e., the last level cache is shared by all cores). Also, the increasing sizes of caches at higher levels satisfy $C_i \geq c_i \cdot p_i C_{i-1}$ for all i and for suitable constants c_i [Chowdhury et al., 2010]. Figure 2.12 shows a representation of the HM model. Note that the multi-core model of Blelloch et al. [2008] corresponds to a 3-level HM model.

The algorithms described by Chowdhury et al. for the HM model are parameter oblivious (i.e., they have no notion of the parameters of the underlying model); however, they are allowed to provide hints to the runtime scheduler. The proposed algorithms use two types of scheduler hints: coarse-grained contiguous (CGC) and space-bound (SB), plus a combination of both (CCG $\Rightarrow$ SB). Roughly speaking, CGC is useful for computations with parallel for loops, while SB and CCG $\Rightarrow$ SB are useful for algorithms that spawn tasks recursively [Chowdhury et al., 2010]. With CGC scheduling, a series of parallel tasks acting on contiguous data are divided into subtasks that act on contiguous segments, assigning subtasks to contiguous cores for parallel execution. In SB scheduling, the algorithm specifies an upper bound on the space used by each task that is spawned during the computation. Intuitively, this allows the scheduler to execute an entire task of size s in the cores under the smallest level i cache L_i such that $s \leq C_i$ in a way that the only cache misses at level i will be those ones related to reading the input and writing the output [Chowdhury et al., 2010]. With CCG $\Rightarrow$ SB, a collection of subtasks forked from a task that was assigned to an L_i cache for some level i are distributed evenly across caches at a lower level, such that the subtasks fits in that level's cache, and the parallelism is fully exploited. This

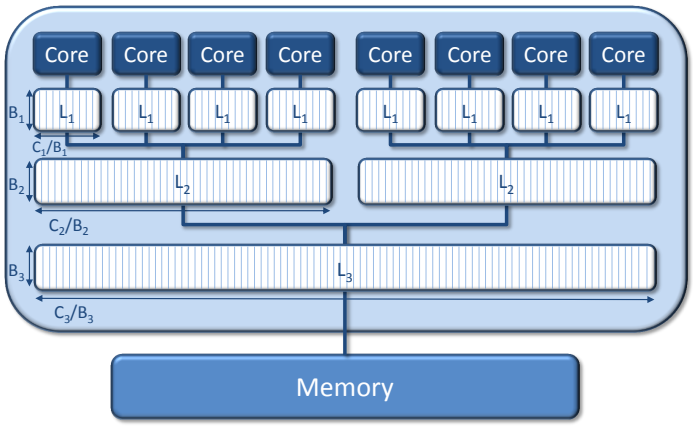


Figure 2.12: An example of the Hierarchical Model (HM) with $h = 4$ and $p_1 = 1$, $p_2 = 4$, $p_3 = 2$, and $p_4 = 1$. Figure adapted from [Chowdhury et al., 2010].

type of scheduling is useful, for example, for algorithms that fork a large number of parallel tasks [Chowdhury et al., 2010].

Using the above scheduling strategies, Chowdhury et al. show bounds on the parallel time and cache complexity at each level i (defined as the maximum number of block transfers between any L_i cache an its corresponding L_{i+1} cache in any direction) for various multi-core oblivious algorithms. The problems and bounds obtained are shown in Table 2.4. These bounds are optimal in this model for prefix sums, matrix transposition, fast Fourier transform, sorting, and Gaussian Elimination Paradigm (GEP), for the assumed number of processors and assuming tall caches (i.e., $C_i = \Omega(B_i^2)$ at each level i, $1 \leq i \leq h - 1$).

The Parallel Cache Oblivious Model Blelloch et al. [2011] propose the Parallel Cache Oblivious (PCO) model, a variation of the sequential Cache Oblivious model of Frigo et al. [1999], to analyze the cache complexity of parallel computations in a hierarchy of caches. This is another example of a model in which the algorithm and scheduler are separate. The algorithm specifies its parallelism independently of the architecture, and the scheduler is responsible for executing the computation on a given architecture satisfying performance bounds. Moreover, the bounds given by the scheduler are a function of performance bounds of the algorithm that can be characterized and analyzed also independently of a particular architecture [Blelloch et al., 2011]. The underlying architecture assumed in this work is modeled as a variation of the Parallel Memory Hierarchy (PMH) model [Alpern et al., 1993], which is similar to the HM model of Chowdhury et al. [2010] described above.

The PCO model is a cache cost model for parallel computations in which the cache complexity $Q^*(n, M, B)$ is analyzed in terms of a single cache of size M and block size B, as in the

	Problem	Time	Cache	Assumptions
1	Prefix sum	$\Theta(n/p)$	$\Theta(n/(q_i B_i))$	$p \leq n/(B_1 \log n)$
2	Matrix transposition	$\Theta(n^2/p)$	$\Theta(n^2/(q_i B_i))$	$p \leq n^2/B_1$
3	Matrix multiplication	$\Theta(n^3/p)$	$\Theta(n^3/(q_i B_i \sqrt{C_i}))$	$p \leq \min\{n^2, C_{h-1}/(2^{h-2} C_1)\}$
4	GEP	$\Theta(n^3/p)$	$\Theta(n^3/(q_i B_i \sqrt{C_i}))$	$p \leq \min\left\{\frac{n^2}{\log^2}, \frac{C_{h-1}}{C_1 \prod_{i=2}^{h-1}\left(2\log^2\left(\frac{C_i}{C_{i-1}}\right)\right)}\right\}$
5	FFT	$\Theta(n \log n/p)$	$\Theta\left(\frac{n \log_{C_i} n}{q_i B_i}\right)$	$p \leq n/B_1$
6	Sorting	$\Theta(n \log n/p)$	$\Theta\left(\frac{n \log_{C_i} n}{q_i B_i}\right)$	$p \leq n/(B_1 \log\log n)$
7	List ranking	$\Theta(n \log n/p)$	$\Theta(n/(q_i B_i)) \log_{C_i} n +$ $(\log\log n)^2 \log(n/B_i))$	$p \leq n/((B_1 + \log\log n) \log n \log\log n)$

Table 2.4: Time and cache complexity bounds for multi-core oblivious algorithms in the Hierarchical Model (HM) [Chowdhury et al., 2010]. GEP stands for Gaussian Elimination Paradigm [Chowdhury and Ramachandran, 2006] and FFT for fast Fourier transform.

sequential cache oblivious model. In addition to using the same analysis of the sequential model for sequential sequences of instructions, the PCO extends the sequential model by considering the cache complexity during the execution of parallel blocks of a computation.

Blelloch et al. show that space-bounded schedulers can be used to bound the cache complexity of algorithms in a hierarchical cache machine in terms of their PCO cache complexity. Space-bounded schedulers were introduced by Chowdhury et al. [2010] in their HM model, which we described above. A space-bounded scheduler takes into account the space used by tasks to schedule them in an appropriate cache level, namely the lower level cache in which the task fits. Blelloch et al. show that a nested-parallel computation executed in a PMH with any space-bounded scheduler achieves a cache complexity that matches the bounds given by the PCO model, i.e., the number of cache misses at each level i in the hierarchy is at most $Q^*(n, M_i, B_i)$, where M_i and B_i are the cache and block size at level i.

However, the running time of the computation achieved might vary with greedy space-bounded schedulers, performing well only when computations are well balanced [Blelloch et al., 2011]. In general, since in a hierarchical model caches are associated with a fix set of processors, if the space and processor requirements of a task differ significantly, this will lead to either idle processors or excessive cache misses. In fact, Blelloch et al. show that there exist computations with plenty of parallelism overall and a small cache miss count according to the PCO model, for which any schedule on a PMH with one shared cache will either execute tasks almost sequentially or incur a large number of cache misses [Blelloch et al., 2011].

Hence, Blelloch et al. extend the PCO model to account for space-parallelism imbalance.

This modification introduces a parameter α to the model which is an estimate of the degree of parallelism of a computation as a function of its size. Intuitively, a computation of size S should be able to use $O(S^{\alpha})$ processors effectively [Blelloch et al., 2011]. They propose a modified space-bounded scheduler that enables the mapping of bounds obtained in the modified PCO model to a PMH machine even for irregular computations. This scheduler allows the execution of parallel subtasks at different levels of the hierarchy, thus adapting to the imbalance of tasks' sizes, and achieves an efficient running time (with cache misses evenly balanced across processors) so long as the parallelism of the machine is enough compared to that of the algorithm [Blelloch et al., 2011].

2.8.3 Graphic Processing Units

A Graphic Processing Unit (GPU) is a massively parallel microprocessor designed to accelerate the graphic tasks required by modern computer applications [Garland, 2011]. Graphic tasks are inherently parallel: graphic scenes typically contain thousands or millions of polygon primitives. Rendering a scene involves the processing of each vertex of every primitive, and every pixel touched by a primitive. Primitives, vertices, and pixels can be processed independently of each other, and thus the workload assigned to a GPU consists of millions of independent tasks that can be processed in parallel [Garland, 2011]. At the same time, computer graphics are throughput-oriented tasks, since scenes are typically rendered at a fixed frame rate, and thus the amount of work done per unit of time influences the visual experience of an application. This throughput-oriented design of GPUs often sacrifices single task performance for throughput of a set of tasks [Garland, 2011].

The massive power and low cost of GPUs have motivated the development of languages and tools to enable the programming and use of these devices for general applications beyond graphic tasks. While initially programmers had to implement programs adhering to the graphic terminology and express data and operations in terms of graphic primitives, the development of more general C-like languages and programming environments such as CUDA (Compute Unified Device Architecture) [Nickolls et al., 2008] by NVIDIA, later followed by the vendor-independent OpenCL (Open Computing Language) [Stone et al., 2010], has facilitated the development of general GPU applications, what has become to be known as General Purpose Computing on GPUs (GPGPU).

The terminology related to a GPU's architecture and programming model differ across vendors. However, the architectures of both NVIDIA and ATI GPUs are in general similar, and so are the programming models of their respective most used environment, CUDA and OpenCL. We describe the programming model and architecture in generic terms, and point out the differences between environment and vendors when necessary.

GPU Architecture

The architecture of a GPU consists of a large array of parallel processors and a set of fixed-function units for tasks such as work distribution, rasterization, and image processing [Garland, 2011]. The array of parallel processors is composed of several multithreaded Streaming Multiprocessors (SMs), each of which supports on the order of a thousand coresidents threads [Garland, 2011]. For example, in an NVIDIA GPU "Fermi" architecture, each SM has 32 scalar cores, and it can support 1,536 coresident threads.

Each SM contains a collection of processing elements (PEs) or cores, each of which provides fully pipelined integer and floating point arithmetic units [Garland, 2011]. In contrast to CPUs cores, SM cores do not use techniques to speed-up single thread performance in the form of out-of-order execution or branch predictions. Given the throughput-oriented design of the architecture, this allows less area of the chip to be devoted to control logic, leaving more area for functional units [Garland, 2011].

GPU Programming Model and Thread Scheduling

Before we describe the execution and scheduling of threads, let us describe the GPU programming model. A *platform* consists of a *host* connected to one or more *devices* [Khronos OpenCL, 2012]. Usually, the host corresponds to a CPU and a device to a GPU. A CUDA or OpenCL program consists of host programs which manage the execution of *kernels* on the device. For example, consider a simple function to multiply two vectors [Hennessy and Patterson, 2011]:

```
add(float * a, float * b, float * c, int n){

    for(int i = 0; i < = n;  i++){
       c[i] = a[i] * b[i];
    }
}
```

The function above is suitable for a data-parallel SIMD parallelization, as the same operation is done on different data items, independently of each other. In order to be executed in parallel on a GPU, the above function must be written as a kernel. A kernel will be executed by multiple threads, each of them responsible for the multiplication of two entries of the arrays. Each such thread or instance of a kernel is called a *work-item* in OpenCL [Khronos OpenCL, 2012], and a *CUDA Thread* in CUDA. In general, all work-items execute the same code, but the execution path they follow and the data on which they operate might differ across work-items. Each work-item is uniquely identified by a threadID, which can be obtained by the item to enable a specific

execution path. In the case of the example, the threadID determines the position in the array on which the thread will operate on.

The entire set of work-items or threads is known in general as a vectorized loop, a *grid* (CUDA), or an *index range* (OpenCL) [Hennessy and Patterson, 2011]. Threads are grouped in *thread blocks* (CUDA) or *work-groups* (OpenCL). All threads in the same block execute concurrently in a single streaming multiprocessor (CUDA) or *computing unit* (OpenCL). Threads in the same block are grouped into 32-thread units called *warps* (CUDA) or *wavefronts* (OpenCL), which are the fundamental unit of scheduling in a SM. A warp of threads executes in lockstep, one instruction at a time. At each timestep, the SM scheduler can execute an instruction of any ready warp, possibly interleaving the execution of instructions of various wraps. Scalar cores in a SM are divided in two groups of 16, and each group is associated with a separate warp scheduler, thus allowing the execution of two warps simultaneously [Garland, 2011].

For example, suppose that in the vector multiplication example $n = 8196$. The grid is the GPU code that executes over all these elements. With a block size of 512, there are 16 blocks in the grid. Each block then contains 16 SIMD threads of 32 threads each (a warp) [Hennessy and Patterson, 2011]. Thus, all threads corresponding to the first 512 elements of the arrays will execute in the same SM, with each group of 32 threads operating on consecutive elements executing concurrently in lock-step.

A SM uses a Single-Instruction-Multiple-Thread (SIMT) architecture, an instance of the SIMD broader class of architectures. A thread warp executing the same path executes together, effectively realizing a SIMD execution. However, when threads within a warp follow different paths, for example, because of a divergent conditional, the scheduler executes all paths one after the other, with threads being activated only when executing the path they followed. While the execution of such divergent paths is correct, there is a loss of efficiency proportional to the number of divergent paths [Garland, 2011].

GPU Memory Model

GPUs contain different memory regions. At the lowest level, individual threads or work-items can have their own dedicated set of registers [Garland, 2011; Hennessy and Patterson, 2011]. In addition, each thread can access a local memory (CUDA) or private memory (OpenCL). Streaming Multiprocessors are also equipped with a high speed and low latency shared memory, which is shared by all threads on the same block. This kind of memory (shared by work-items in the same work-group) is called local memory in OpenCL. A *global memory* allows access by work-items in all work-groups. A region of this global memory is known as the *constant memory*, and it is used by the host to allocate memory objects in the device [Khronos OpenCL, 2012]. Some architectures also feature an L1/L2 cache hierarchy. For example, in the Fermi architecture each SM has an L1 cache and there is an L2 cache that is shared across SMs [Garland, 2011].

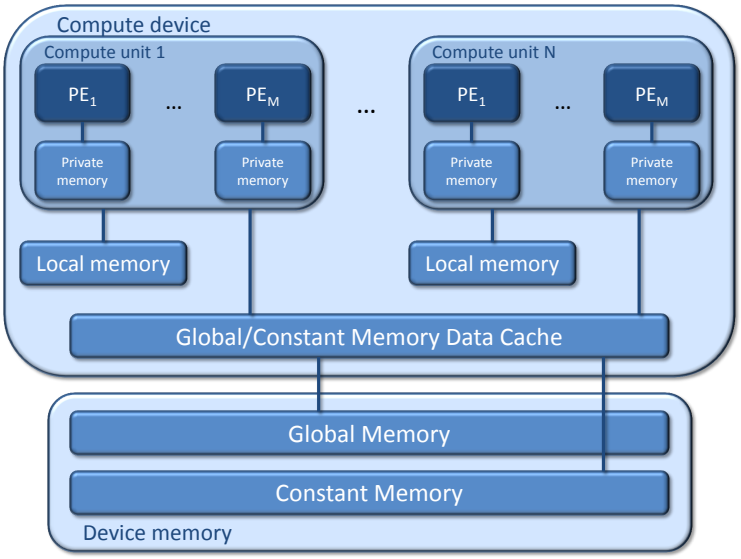


Figure 2.13: A conceptual depiction of the OpenCL device architecture. Figure adapted from [Khronos OpenCL, 2012].

A conceptual depiction of an OpenCL device together with its memory regions is shown in Figure 2.13.

The performance of a GPU application is highly dependent on its memory access patterns. In general, accesses to local memory are much faster than accesses to global memory. In particular, programs should seek memory accesses to contiguous data by work-items in the same group, which are known as *coalesced* memory operations.

2.9 Bit Parallelism and the Word-RAM

The word-RAM is a variant of the RAM model in which a word has length w bits, and the contents of the memory are assumed to be integers in the range $\{0, \ldots, 2^{w-1}\}$ [Hagerup, 1998]. This implies that $w \geq \log n$, where n is the size of an input problem, and that the size of the memory is at most $2^{O(w)}$; otherwise a memory cell cannot be addressed using a constant number of words.

The word-RAM includes the usual load, store and jump instructions of the RAM model, allowing for immediate operands and for direct and indirect addressing. In this model, arithmetic operations on two words are modulo 2^w, and the instruction set includes left and right shift operations (equal to multiplication and division by powers of two) and boolean operations. All

instructions take constant time to execute. There are different versions of the word-RAM model depending on the instruction set assumed to be available. The *restricted model* is limited to addition, subtraction, left and right shifts, and boolean operations AND, OR, and NOT. These instructions augmented with multiplication constitute the *multiplication model*. Finally, the AC^0 model assumes that all functions computable by an unbounded fan-in circuit of polynomial size (in w) and constant depth are available in the instruction set and execute in constant time. This definition includes all instructions from the restricted model and excludes multiplication. We refer to the reader to the survey by Hagerup [1998] for a more extended description of the model and a discussion of its practicality.

Algorithms in the word-RAM model take advantage of the intrinsic parallelism in instructions that operate on words. The simplest examples are boolean operations: in one instruction we can compute the AND or OR of w sets of 1 bit each. In general, word-RAM algorithms exploit this parallelism by operating on various elements in parallel using operations on w-bits words. One example of such operation that is useful in many word-RAM algorithms is the comparison of several pairs of elements in parallel. This can be achieved by packing these elements in *fields* within one word. We now describe how to implement a fieldwise comparison in constant time.

Comparators Suppose that a w-bit word is divided in f-bit fields, with each field representing an $(f-1)$-bit number. Let G and F be two such words and let F_i and G_i denote the contents of the i-th field in F and G, respectively. Let us assume that we want to identify all F_i such that $F_i \geq G_i$. Fieldwise comparisons can be done by setting the most significant bit of each field in F as a test bit and computing $H = F - G$. The most significant bit of the i-th field in H will be 1 if and only if $F_i \geq G_i$ [Hagerup, 1998]. Now, if we want to operate only on the values of F that are greater than or equal to their corresponding values in G, we can mask away the rest of the values as follows. We first mask away all but the test bits in H. Then, a mask M with ones in all bits of the relevant fields and zeros everywhere else (including test bits) can be obtained by computing $M = H - (H >> (f-1))$. The result of $(M \ \& \ F)$ contains then only the values of fields that pass the test [Hagerup, 1998]. Clearly this operation takes constant time, and it can be easily adapted to other standard comparisons. Example 2.3 shows the parallel comparison of 8 pairs of values packed in fields of $f = 4$ bits in words of $w = 32$ bits.

Example 2.3 *Example of a parallel elementwise comparison of tuples* $F = (1, 5, 0, 4, 7, 4, 5, 2)$ *and* $G = (6, 2, 1, 0, 4, 4, 6, 1)$*, using 32-bit words and fields of* $f = 4$ *bits (with the most significant bit at the far left). The resulting word represents a tuple* $R = (R_1, R_2, \ldots, R_8)$*, where* $R_i = F_i$ *if* $F_i \geq G_i$*, and* $R_i = 0$ *otherwise. I.e.,* $R = (0, 5, 0, 4, 7, 4, 0, 2)$.

F	=	1001	1101	1000	1100	1111	1100	1101	1010
G	=	0110	0010	0001	0000	0100	0100	0110	0001
$H = F - G$	=	0011	1011	0111	1100	1011	1000	0111	1001
Mask	=	1000	1000	1000	1000	1000	1000	1000	1000
$H = H$ & Mask	=	0000	1000	0000	1000	1000	1000	0000	1000
$H' = H >> (f-1)$	=	0000	0001	0000	0001	0001	0001	0000	0001
$M = H - H'$	=	0000	0111	0000	0111	0111	0111	0000	0111
$R = M$ & F	=	0000	0101	0000	0100	0111	0100	0000	0010

In the word-RAM model, word-level parallelism is the only source of increased efficiency with respect to traditional RAM algorithms, and hence w is the maximum speedup that can be obtained [Hagerup, 1998].

There are various algorithms for fundamental problems that take advantage of word-level parallelism or a bounded universe, some of which fit into the word-RAM model, although are not explicitly designed for it [Arlazarov et al., 1970]. Much attention has been given to sorting and searching, for which known lower bounds in the comparison model do not carry to the word-RAM model [Fredman and Willard, 1993]. For example, in a word-RAM model with multiplication, sorting n words can be done in $O(n \log\log n)$ time and $O(n)$ space deterministically [Han, 2004], and in expected $O(n\sqrt{\log\log n})$ time and $O(n)$ space using randomization [Han and Thorup, 2002]. Word-RAM techniques have also been applied in many different areas, such as succinct data structures [Jacobson, 1989; Munro, 1996], computational geometry [Chan, 2006; Chan and Patrascu, 2009], and text indexing [Grossi et al., 2003].

In Chapter 5 we introduce a model that extends the word-RAM model by adding a w^2-bit word Arithmetic Logic Unit (ALU) that implements all operations of the word-RAM model, but keeping memory accesses at w-bits. We show that many word-RAM algorithms can be implemented in the new model, achieving speedups comparable to those of multi-threaded computation, while keeping the simplicity of sequential programming inherent to the RAM model.

Chapter 3

LoPRAM: A Model for Low-Degree Multi-Core Parallel Computation

Modern microprocessor architectures have gradually incorporated a certain degree of parallelism in the form of advances such as pipelined architectures and SIMD vector extensions. The appearance of multi-cores in 2004 went a step forward, replicating full processing units that could read and execute instructions in parallel. Microprocessors with 2 and 4 cores became widely available. While the degree of parallelism provided by any of these solutions was rather small and as such it was best studied as a constant speedup over the traditional and/or transdichotomous RAM model, plans of an increasing number of cores by leading manufacturers meant that a constant speedup would no longer accurately reflect the amount of resources available.

In this Chapter we present the LoPRAM model, a new model of low degree parallelism which better reflects recent multi-core architectural advances. We argue that in current architectures the number of processors available can effectively be assumed to be $O(\log n)$. This parallels the development of the transdichotomous-RAM in which the presence of bit-level parallelism went from being subsumed as a small constant speed-up to the $w = O(\log n)$-bit word model in which the speedup is a function of w (see Section 2.9).

Theoretical parallel computation was already a well developed field, with the PRAM [Fortune and Wyllie, 1978] being the dominant model. This model generally assumed $\Theta(n)$ or an even larger number of processors working synchronously with zero communication delay and often with infinite bandwidth among them (see Section 2.3.1). If the number of processors available in practice was smaller, the $\Theta(n)$ processor solution could be emulated using Brent's Lemma [Brent, 1974] (see Section 2.3.2). The PRAM model, while fruitful from a theoretical perspective, proved unrealistic, and various attempts were made to refine it in a way that would better align to what could effectively be achieved in practice. We describe models that improved upon the PRAM in

Section 2.5. In practice, there were various important drawbacks of the PRAM model, such as the cost of synchronization, the cost of interprocessor communication, the cost-effectiveness of a massively parallel machine and, more importantly, the enormous difficulty in developing and implementing work-optimal algorithms (i.e., linear speedup) for a computer with $\Theta(n)$ processors. Even relatively simple tasks such as sorting required considerable thought before a work-optimal PRAM algorithm could be developed [Cole, 1988]. The state of parallel algorithm research consequently entered into a dormant state in the second half of the 1990s. Recent developments in multi-core architectures have brought back the possibility of parallel architectures in practice, which has revived the study of parallel algorithms. However, to the best of our knowledge, the assumption of a logarithmic level of parallelism as well as its theoretical implications had yet to be noted in the literature.

As with the classical RAM model, the LoPRAM supports different degrees of abstraction. Depending on the intended application and the performance parameters required, the design and analysis of an algorithm can consider issues such as the memory hierarchy, interprocess communication, low level parallelism, or high-level thread-based parallelism. Our main focus is on the higher level, thread-based parallelism. Naturally, the more abstract the model the easier it is to reason on it at the expense of fidelity in the analysis. As we shall see, the design and analysis of algorithms at this higher level is often sufficient to achieve optimal speedup. This, of course, does not preclude the use of low level optimizations when necessary. This parallels the classical RAM model in which issues such as caching, secondary storage, and other such hardware characteristics can be incorporated or ignored as it may be deemed most appropriate.

Our results[1] We apply the LoPRAM model to the design and analysis of algorithms for multi-core architectures for a sizable subset of problems and show that we can readily obtain optimal speedups. This is in contrast to the PRAM model, in which work-optimal algorithms were often rather challenging research questions. Notable examples of these are sorting [Cole, 1988] and some fundamental problems on directed graphs, such as determining whether a graph is acyclic or constructing a topological ordering of an acyclic graph, which are examples of problems affected by the so called *transitive closure bottleneck* [Karp and Ramachandran, 1990; Kao and Klein, 1990]. More explicitly, we show that a large class of divide-and-conquer algorithms can be parallelized using the high level LoPRAM thread model while achieving optimal speedup. This leads to a parallel version of the master theorem for divide-and-conquer algorithms [Cormen et al., 2001], which can be readily applied to any divide-and-conquer algorithm whose complexity can be obtained with the original theorem. We show that dynamic programming algorithms can also be easily parallelized in the LoPRAM model by describing a generic strategy that takes a dynamic programming solution to a problem and generates a suitable schedule to solve it in parallel.

[1] Results in this chapter appeared in [Dorrigiv et al., 2008] and are joint work with Reza Dorrigiv and Alejandro López-Ortiz.

Interestingly, the assumption that there is a logarithmic bound on the degree of parallelism is key in the analysis of the techniques given. As well, communication cost remains modest under the assumption of low-degree parallelism. Indeed, with this bound in place a full processor network based on the complete graph is realizable. We show experimental results on both divide-and-conquer and dynamic programming parallelizations in the model, with close to optimal parallel executions, showing the simplicity and practicality of parallel algorithms in this model.

This chapter is organized as follows. In Section 3.1 we introduce the LoPRAM, a formal model for multi-core computing. In Section 3.2 we show that under this model a large class of divide-and-conquer and dynamic programming algorithms can be parallelized to achieve p-fold speedups in this model in a rather straightforward fashion. We show experimental results in Section 3.3. Lastly, in Section 3.4 we present concluding remarks and future directions of research.

3.1 Model

The core of a LoPRAM is a PRAM with $p = O(\log n)$ processors running asynchronously in multiple-instruction multiple-data (MIMD) mode. The read and write model, while architecture dependent, can generally be assumed to be Concurrent-Read Exclusive-Write (CREW) [Gibbons and Rytter, 1988; JáJá, 1992]. To support this model, semaphores and automatic serialization on shared variables are available—either hardware or software based—in a transparent form to the programmer. If an unserialized variable is concurrently written, this has undefined arbitrary behaviour—including suspension of execution.

The model naturally supports a high level abstraction that simplifies the design and analysis of parallel programs. The application benefits from parallelism through the use of threads. We show that in many instances this leads to parallel algorithms with p-fold speedup derived from simple modifications to sequential algorithms.

3.1.1 Thread Model

Two main types of threads are provided: standard threads and PAL-threads (Parallel Algorithmic Light threads). Standard threads are executed simultaneously and independently of the number of cores available; they are executed in parallel if enough cores are available or by using multitasking if the thread count exceeds the degree of parallelism, just as in a regular RAM. PAL-threads, on the other hand, are executed at a rate determined by the scheduler. If there are any PAL-threads pending, at least one of them must be actively executing, while all others remain at the discretion of the scheduler. They could be assigned resources, if they are available, or they could be made to wait inactive until resources free over. Once a thread has been activated, it remains active until it relinquishes control when it either completes, it reaches a synchronization point, or it blocks

to wait for an event. Threads are not preempted, which is important to avoid potential deadlock introduced by the scheduler[2].

Pending PAL-threads are activated in a manner consistent with order of creation as resources become available. While primitives can be defined for ad-hoc ordering of PAL-threads activation, by default threads are inserted into an ordered tree. The root of the tree is the main thread with new threads attached to the node corresponding to the activating parent-thread. The scheduler activates the nodes in the tree in parent-child order, i.e., first the parent thread is activated, which issues PAL-threads calls for its children. The parent thread is now in a wait state, and child threads are activated in order of creation. If there are cores available, PAL-threads are assigned to the available cores, otherwise the requests remain pending and the core of the parent thread is assigned sequentially to the children. If no further children remain pending, then control is returned to the parent thread. If cores become available, pending requests are activated in the order given by the preorder traversal of the tree.

Execution concludes when there are no further threads to activate, and the main thread exits. Consider for example the code for a parallel implementation of the classical sequential mergesort written using suitable C extensions for the LoPRAM:

```
void mergeSort(int numbers[], int temp[], int array_size)
{ m_sort(numbers, temp, 0, array_size - 1); }

void m_sort(int numbers[], int temp[], int left, int right) {
  int mid = (right + left) / 2;
  if (right > left) {
    palthreads { // execute in parallel if possible
      m_sort(numbers, temp, left, mid);
      m_sort(numbers, temp, mid+1, right);
    }  // implicit join
    merge(numbers, temp, left, mid+1, right);
  }
}
```

The semantics of the primitive `palthreads` are to create a PAL-threads call for each of the function calls within its scope. These threads are created as children of the current thread in the specific order given. Observe that there is an implicit wait at the end of the `palthreads` statement which can be deactivated using a "`palthreads { ... } nowait;`" construct with an explicit thread join later on if so needed. Note that we introduce this syntax only for the purposes of the example and that it is not inherent to the LoPRAM model. An example of the execution of `mergeSort` with an input of size 16 and 4 processors is shown in Figure 3.1.

[2]Still, a deadlock might occur as a result of threads' actions in an incorrectly designed implementation.

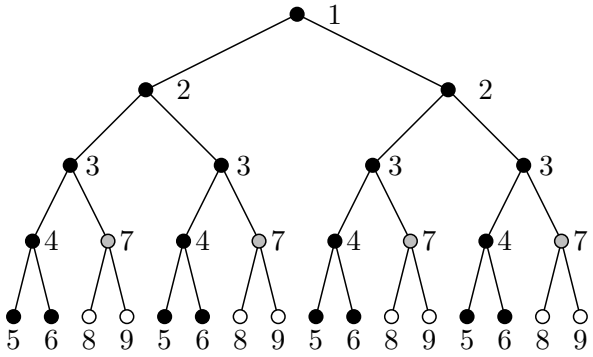


Figure 3.1: Example of an execution tree for mergesort with an input of length $n = 16$ and $p = 4$ processors. Black nodes represent active pal-requests, gray nodes represent calls that have been pal-requested but that are not active yet, while white nodes are calls that have not been pal-requested. The number by each node indicates the time step in which the call corresponding to that node is pal-requested. The picture shows the execution at $t = 6$.

We note that the LoPRAM scheduler resembles the work-stealing scheduler [Burton and Sleep, 1981] (see Section 2.6.1), and in fact both schedulers result in the same thread execution order on the mergesort example given above and in regular divide-and-conquer algorithms in general. However, these schedulers have different implementations: while an implementation of work stealing is distributed as threads belong to each processor, the LoPRAM scheduler is centralized, as newly created threads are added to a single pool from which they are dispatched by the thread scheduler.

3.1.2 Multiprocessing Model

In actuality, the number of cores made available by the operating system may vary as the level of multiprogramming in the system changes. Hence, in the analysis of the algorithm the number of processors available is denoted as p, with the assumption that this number is bounded from above by $O(\log n)$, i.e., p is $O(\log n)$ but p is not necessarily $\Theta(\log n)$. The algorithm must execute properly for any value of p. The running time is then a function of n and p.

3.2 Optimal Algorithm Parallelization

In this section we present two classes of problems which allow for ready parallelization under the LoPRAM model. Note that these same classes were not, in general, readily parallelizable under

the classic PRAM model.

3.2.1 Divide and Conquer

Consider the class of divide-and-conquer algorithms whose time complexity is described by a recurrence which can be resolved using the master theorem. We show that when these algorithms are executed in a straightforward parallelization on a LoPRAM, their execution time is given by a parallel version of the master theorem which reports optimal speedup.

Consider a recursive divide-and-conquer sequential algorithm whose time complexity $T(n)$ is a recurrence of the form:

$$T(n) = aT(n/b) + f(n), \tag{3.2.1}$$

where $a > 1$ and $b > 1$ are constants, and $f(n)$ is a nonnegative function. By the master theorem, $T(n)$ is such that [Cormen et al., 2001]:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{if } f(n) = O(n^{\log_b(a)-\epsilon}) \quad \text{(Case 1)} \\ \Theta(n^{\log_b a} \log n), & \text{if } f(n) = \Theta(n^{\log_b a}) \quad \text{(Case 2)} \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ and} \\ & af(n/b) \leq cf(n), \text{ for some } c < 1 \quad \text{(Case 3)} \end{cases}$$

In the above, $\epsilon > 0$ is some constant. We are interested in the time complexity of such algorithms when p processors are available. We assume that recursive calls can be assigned to different processors, which can execute their instances independently of those of others. All of the processors finish their computations before the results are merged. We denote by $T_p(n)$ the running time of a parallel algorithm that uses p processors and by $T(n) = T_1(n)$ its sequential version.

Sequential Merging We first consider problems for which the merging phase of the algorithm can only be done sequentially in each instance. Multiple processors can still be used to merge subproblems of different instances, but only one processor deals with a particular instance. The following theorem states the bounds for $T_p(n)$.

Theorem 3.1 *Let $T_p(n)$ be the time taken by a recursive algorithm that uses $p = O(\log n)$ processors whose sequential version has time complexity given by $T(n) = aT(n/b) + f(n)$, where $a > 1$ and $b > 1$ are constants, and $f(n)$ is a nonnegative function. Then, the time $T_p(n)$ is a recurrence of the form:*

$$T_p(n) = T\left(\frac{n}{b^{\log_a p}}\right) + \sum_{i=0}^{\log_a(p)-1} f\left(\frac{n}{b^i}\right). \tag{3.2.2}$$

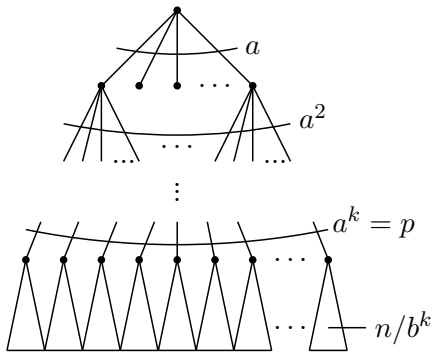


Figure 3.2: Execution tree of a divide-and-conquer algorithm with p processors: a thread is created for each recursive call until $a^k = p$ calls have been made. At this point, each thread executes the sequential version of the algorithm on an input of size n/b^k. Figure 3.1 shows this tree for an execution of mergesort.

The bounds for $T_p(n)$ are given by:

$$T_p(n) = \begin{cases} O(T(n)/p), & \text{if } f(n) = O(n^{\log_b(a)-\epsilon}) & \text{(Case 1)} \\ O(T(n)/p), & \text{if } f(n) = \Theta(n^{\log_b a}) & \text{(Case 2)} \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ and} & \\ & af(n/b) \leq cf(n), \text{ for some } c < 1 & \text{(Case 3)} \end{cases}$$

Proof: Since there are p processors available, some recursive steps of the algorithm can be performed in parallel. However, when the number of simultaneous calls exceeds p, we need to solve the rest sequentially. Since the algorithm divides the problem into a subproblems, at the k-th level of the recursion tree we will have a^k subproblems. Thus, when $k = \log_a p$ we will have p subproblems and no more processors are available for the subsequent recursive calls[3]. Then, at this point, as there are no more free cores available, the sequential version of the algorithm is used, with an input of size $n/b^{\log_a p}$ (see Figure 3.2). Observe that this condition is **never explicitly tested** by the scheduling algorithm, rather it is a natural consequence of the proposed order of execution of the parent child threads. Note that it cannot be the case that all the recursive calls are performed in parallel and that there is no sequential component to be executed, as it is not hard to see that this would only happen if $b^{\log_a p} \geq n$, which would mean that $p \geq n^{\log_b a} = \omega(\log n)$, but we assume $p = O(\log n)$.

[3]We have assumed that p is a power of a for the sake of simplicity. Note, however, that if this is not the case the same bounds hold for $p' = a^{\lfloor \log_a p \rfloor} \leq p$ processors, since in this case $p/a \leq p' \leq p$, and thus $T(n)/p' = \Theta(T(n)/p)$.

The cost of merging the solutions of the subproblems using p processors is given by the sum of the cost at each level of the recursion tree, namely: $\sum_{i=0}^{\log_a(p)-1} f\left(n/b^i\right)$. Hence, we can write the time of the parallel algorithm as in Equation (3.2.2). Now, as with the standard version of the master theorem, we prove each case separately.

Case 1 Since $f(n) = O(n^{\log_b(a)-\epsilon})$, by the master theorem, we have $T(n) = \Theta(n^{\log_b a})$. Substituting in Equation (3.2.2) we have

$$T_p(n) = O\left(\left(\frac{n}{b^{\log_a p}}\right)^{\log_b a} + \sum_{i=0}^{\log_a(p)-1}\left(\frac{n}{b^i}\right)^{\log_b(a)-\epsilon}\right) = O\left(\frac{n^{\log_b a}}{p} + \frac{n^{\log_b(a)-\epsilon}a}{a-b^\epsilon}\right)$$

Since $T(n) = \Theta(n^{\log_b a})$, the first term of the sum above is $O(T(n)/p)$, while the second term is strictly smaller for any $p = O(n^\epsilon)$. Since we assume $p = O(\log n)$, then $T_p(n) = O(T(n)/p)$.

Case 2 Since $f(n) = \Theta(n^{\log_b a})$, we have $T(n) = \Theta(n^{\log_b a}\log n)$. Then,

$$\begin{aligned} T_p(n) &= O\left(\left(\frac{n}{b^{\log_a p}}\right)^{\log_b a}\log\left(\frac{n}{b^{\log_a p}}\right) + \sum_{i=0}^{\log_a(p)-1}\left(\frac{n}{b^i}\right)^{\log_b a}\right) \\ &= O\left(\frac{n^{\log_b a}}{p}(\log n - \log b^{\log_a p}) + n^{\log_b a}\cdot\frac{a}{a-1}\right) \end{aligned}$$

The first term dominates as $(\log n)/p = \Omega(1)$. Therefore $T_p(n) = O(T(n)/p)$.

Case 3 In this case we prove that the total time is dominated by the time that it takes to merge the solutions of the subproblems to produce the final solution in the second level of the recursion tree. Recall that we assume that this process is done sequentially. Thus, no benefit is gained from using p processors in this case. Starting with recurrence (3.2.1), we know that $af(n/b) \le cf(n)$. By a simple induction argument, it can be shown that $f(n/b^i) \le (c/a)^i f(n),\ 0 \le i \le \log_b n$. In addition, we have $T(n) = \Theta(f(n))$. Hence,

$$T_p(n) = O\left(f\left(\frac{n}{b^{\log_a p}}\right)\right) + \sum_{i=0}^{\log_a p-1}\left(\frac{c}{a}\right)^i f(n) = O\left(f(n) + f(n)\frac{a}{a-c}\right) = O(f(n)).$$

On the other hand, $T(n) \ge f(n)$ and thus we have $T_p(n) = \Omega(f(n))$. ∎

Parallel Merging We now consider the special case when we can merge the results of subproblems in parallel optimally. The time complexity of the sequential algorithm is given by Equation (3.2.1). We claim that the parallel master theorem for this setting is as before with the exception of case 3 for which we have

$$T_p(n) = \Theta(f(n)/p), \quad \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ and } af(n/b) \leq cf(n), \text{ for some } c < 1 \tag{3.2.3}$$

For the merging phase, at the i-th level of the recursion tree we have to merge a total of a^i solutions of subproblems of size n/b^i. Each of them takes time $f(n/b^i)$, and since we assume they can be done in parallel, the total time of the merging phase at that level is given by $(a^i/p)f\left(n/b^i\right)$. Thus, we obtain an analogous expression to Equation (3.2.2):

$$T_p(n) = T\left(\frac{n}{b^{\log_a p}}\right) + \sum_{i=0}^{\log_a(p)-1} \frac{a^i}{p} f\left(\frac{n}{b^i}\right). \tag{3.2.4}$$

For cases 1 and 2 the dominant term is the first summand of the right hand side of this equation, and hence we obtain the same expression as before. For Case 3, using the arguments analogous to the previous sequential case be obtain

$$T_p(n) \quad \leq \quad O\left(f(n)\left(\frac{c}{a}\right)^{\log_a p}\right) + \frac{f(n)}{p}\frac{1}{1-c} \leq O\left(\frac{f(n)}{p}\right) + O\left(\frac{f(n)}{p}\right),$$

and since we have $T_p(n) \geq f(n)/p$, we conclude that $T_p = \Theta(f(n)/p)$.

Table 3.1 shows the recurrences and sequential time complexities of important divide-and-conquer algorithms, together with their parallel complexities given by Theorem 3.1. A similar result is presented independently in [Blelloch et al., 2008] for a class of hierarchical divide-and-conquer algorithms, in which the divide and combine phases are in turn divide-and-conquer algorithms as well. Algorithms presented in [Blelloch et al., 2008] include mergesort with divide-and-conquer merge [Akl and Santoro, 1987], and matrix multiplication algorithms, among others (see Section 2.8.2 for a more detailed description of these results). Our results hold for a more general class of divide-and-conquer algorithms for which the divide and combine phases may not necessarily be expressed directly as divide-and-conquer procedures with enough parallelism to achieve optimal speedup.

3.2.2 Dynamic Programming

Dynamic programming is suitable for solving problems that have optimal substructure as well as overlapping subproblems. The solutions to such subproblems are then combined into the solution of a larger problem. In many cases these subproblems can be solved in parallel, up to a degree that depends on the problem itself, and hence a certain degree of parallelism is achievable.

Algorithm	Recurrence	$T(n)$	$T_p(n)$
Mergesort	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n \log n)$	$O\left(\frac{n}{p} \log n\right)$
Strassen's matrix mult. [Strassen, 1969]	$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$	$O(n^{\log_2 7})$	$O\left(\frac{n^{\log_2 7}}{p}\right)$
Delaunay triangulation [Dwyer, 1987]	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n \log n)$	$O\left(\frac{n}{p} \log n\right)$
Polygon triangulation [Chazelle, 1991]	$T(n) = 2T\left(\frac{n}{2}\right) + O(\sqrt{n})$	$O(n)$	$O\left(\frac{n}{p}\right)$
Convex Hull	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n \log n)$	$O\left(\frac{n}{p} \log n\right)$
Closest pair	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n \log n)$	$O\left(\frac{n}{p} \log n\right)$

Table 3.1: Sequential and parallel time complexities for various divide-and-conquer algorithms in the LoPRAM model.

In the past, parallel versions of certain dynamic programming algorithms have been proposed. In a seminal paper, Apostolico et al. [1990] studied parallel algorithms for the string editing problem and other related problems by considering the Directed Acyclic Graph (DAG) corresponding to the problem and computing this graph in parallel. Galil and Park [1991] studied various dynamic programming problems, presenting a unified framework for the parallel computation of these problems using the closure methods and the matrix product methods as general tools for developing parallel algorithms. Later, Bradford [1994] developed a characterization that models dynamic programming tables by graphs, leading to polylogarithmic algorithms for the optimal matrix chain ordering, the optimal construction of binary trees and the optimal convex polygon triangulation. Bradford showed how to transform these problems to a minimum cost parenthesizing on a weighted semigroup, which is then transformed to a shortest path problem on a weighted directed graph. Most of these studies consider a few dynamic programming problems and provide parallel algorithms that are specific to those problems. In general, they assume a classical PRAM model with $\Theta(n)$ processors, meaning that the algorithm is designed so that it can take advantage of that many processors, shall they be available. In our case we restrict ourselves to $p = O(\log n)$ processors.

We show that dynamic programming algorithms can be parallelized by providing a general procedure that, given the specification of the dynamic programing solution to a problem, generates a scheduling strategy to solve it in parallel. The idea is similar to the previous work cited above in that we also reduce the original problem to computing the DAG corresponding to the dynamic programming specification of the solution.

Our goal is to compute the solution to the dynamic programming problem with as much parallelism as possible, with a general strategy that works for any problem whose specification is given in an explicit dynamic programming expression of its solution. Let M be an object

which stores partial solutions. Let $\mathcal{I}$ and $\mathcal{S}$ be the spaces of partial inputs and partial solutions, respectively. Let $h : \mathcal{I} \to \mathcal{S}$, $g : \mathcal{I} \to \{0, 1\}$, and $f : \{\mathcal{S}^k \times \mathcal{I}\} \to \mathcal{S}$ be functions, where $k \in \mathbb{N}$. We assume that this solution is of the form:

$$M[x] = \begin{cases} h(x) & \text{if } g(x) = 0 \quad \text{(base case)} \\ f(\{M[y_i]\}_{y_i \prec x}, x) & \text{otherwise} \end{cases} \tag{3.2.5}$$

For the dynamic programming solution to be effective, we require that the object M which stores partial solutions remains of reasonable size, that it can be efficiently indexed using a partial input x as key and that the recursive order $y_i \prec x$ be efficiently constructible in a bottom-up fashion. Alternatively, if the solution cannot be computed efficiently in a bottom-up fashion, we can use *memoization*, which stores the partial solutions as they are required in the top-down expansion of the recursion. In most cases these two techniques are equivalent, though there are known cases in which the use of one over the other (for either of them) is provably superior [Cormen et al., 2001].

Example 3.1 *Consider the* matrix chain multiplication problem*: given a sequence of matrices $(A_1, A_2, \ldots, A_n)$ to be multiplied together, we wish to compute their product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ (using the natural matrix multiplication algorithm) in a way so that the total number of scalar multiplications is minimized. The number of scalar multiplications is different for different parenthesizations of the product. Thus, the goal is to find the parenthesization that minimizes the number of scalar multiplications overall.*

The dynamic programming solution can be defined as follows [Cormen et al., 2001]: Let $M[i, j]$ be the minimum number of scalar multiplications required to compute the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$. The answer to the problem will then be $M[1, n]$. The best parenthesization of the product of matrices A_i through A_j involves deciding where to split this products into 2 products, this is, there exists an index k, $i \leq k < j$ such that the best choice for the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ will be to compute it as $(A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j)$. The cost of multiplying two matrices of dimensions $p_1 \times p_2$ and $p_2 \times p_3$ is $p_1 p_2 p_3$. If matrix A_i has dimension $p_{i-1} \times p_i$, then the last product of the two matrices in parenthesis has cost $p_{i-1} p_k p_j$. Hence, we can formulate the following recursion to compute $M[i, j]$:

$$M[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{M[i, k] + M[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases} \tag{3.2.6}$$

In general, we cannot expect a generic strategy to find highly parallel (i.e., polylogarithmic time) solutions for any dynamic program, as it has been shown that simple recurrences can be used to solve P-complete problems [Ibarra and Trân, 1994], and hence it is unlikely that such recurrences are in NC. Observe that a recurrence is one form of Equation (3.2.5), hence in principle solving the recurrence is equivalent to executing a dynamic programming algorithm. Note,

however, that we seek a moderate level of parallelism (i.e., a speedup proportional to $\log n$), and hence the limitation described above (if $\mathsf{P} \neq \mathsf{NC}$) does not apply in our case. Furthermore, it has been shown that there exist polylogarithmic time parallel algorithms for several formulations of recurrences that encode many of the most common dynamic programs, including linear recurrences of any order [Kogge, 1974], and many two dimensional recurrences [Ibarra and Trân, 1994]. For instance, recurrences that can be recast as a shortest path problem can be solved in $O(\log^2 n)$ parallel time. Examples of these are recurrences to solve the edit distance problem and the longest common subsequence problem (see Example 3.2) [Ibarra and Trân, 1994].

Dependency Graph

The recursion described in Equation (3.2.5) can be modeled by a graph G called the *dependency graph*. G has a vertex v_x corresponding to each subproblem x (or equivalently partial result $M[x]$). There is an edge from the v_x to v_y if and only if subproblem y depends directly on subproblem x. In this DAG, the source vertices correspond to the base cases. We consider computing this DAG in parallel. The speedup will be proportional to the amount of parallelism embedded in the graph. To be more precise, we consider the dependency graph as a partially ordered set (poset) on the subproblems. Then the subproblems in an antichain of this poset are independent and can be processed at the same time in parallel. We can partition G into antichains and then process the subproblems in every antichain in parallel. At each time we find an antichain that does not have any dependencies on the subproblems that have not yet been processed. We process subproblems in that antichain in parallel and then move on to the next antichain. A dual of Dilworth's theorem states that the size of the largest chain in a poset equals the smallest number of antichains into which the poset may be partitioned [Dilworth, 1950; Mirsky, 1971]. Suppose that $c_1 c_2 \ldots c_l$ is a largest chain in the poset. At step i we process c_i together with other elements in its antichain, i.e., elements that are incomparable with c_i.

These antichains capture the degree of parallelism that is readily apparent in the recursive description of the problem. For example, in the case of most common examples of two dimensional tables for dynamic programming, antichains take the form of rows, columns, or diagonals, thus readily exhibiting the inherent parallelism of the graph. In other cases, such as one dimensional dynamic programs, the DAG is a path and hence the parallelism available is less evident, although such programs might still admit efficient parallel algorithms. For example, consider a simple linear recurrence of the form $x_i = a_i \cdot x_{i-1} + b_i$, where a_i, b_i are constants and x_0 equals some initial value. Although the antichains of the corresponding DAG have all length one, the recurrence can be efficiently solved in parallel with an algorithm based on prefix-sums [Kogge, 1974].

Figure 3.3 shows the dependency graph for an instance of the matrix chain multiplication problem of size 4. In this case of this problem, each diagonal is an antichain, and hence most antichains have $\Theta(n)$ vertices.

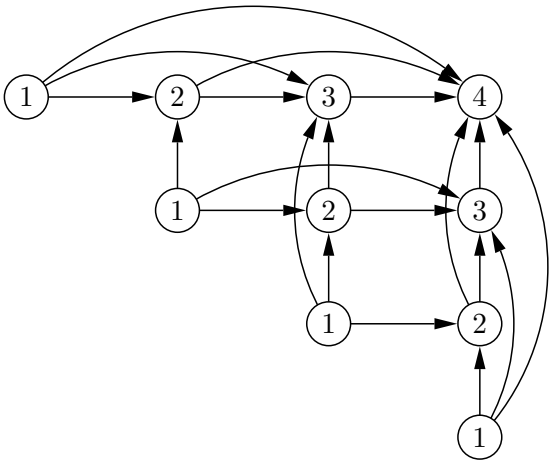


Figure 3.3: Dependency graph of the dynamic programming recurrence for matrix chain multiplication for $n = 4$ (see Equation (3.2.6)). The numbers in the vertices indicate the rank of the antichain to which the vertex belongs, based on the order given by the level of vertices in each antichain.

Computing the Dependency Graph in Parallel

Given the specification $\mathcal{D}$ of a dynamic programming solution of the form of Equation (3.2.5) and an input I, we give a parallel algorithm for solving the problem. Each vertex v has a counter c_v that indicates, at any time, the number of vertices that v depends on directly and that have not been computed yet. Initially, the value of the counter of each of vertex equals its indegree in the dependency graph, which is computed based on $\mathcal{D}$. The computation of partial solutions in the graph begins with the creation of a PAL-thread for each base case vertex. After a thread computes the value corresponding to a vertex v, it determines its outgoing neighbors according to $\mathcal{D}$ and decreases the values of their counters. If this leads to a 0 value for some counters $\{c_{u_1}, c_{u_2}, \cdots, c_{u_k}\}$, i.e., $u_1, u_2, \cdots, u_k$ are ready to be computed, the same thread creates other PAL-threads to compute these vertices and these get executed depending on the availability of processors. Algorithm 3.1 describes this strategy in pseudocode for a specification $\mathcal{D}$ and input I.

Algorithm 3.1 uses two subroutines: computeCounter and computeOutgoingNeighbours. For a large class of dynamic programming algorithms, these subroutines can be easily computed using $\mathcal{D}$ and I. Consider the matrix chain multiplication problem described in Example 3.1. Recall the specification $\mathcal{D}$ of the dynamic programming solution:

Algorithm 3.1 parallel_dp$(\mathcal{D}, I)$

```
1: V ← ∅
2: for each subproblem u in parallel do
3:     c_u ← computeCounter(u, D, I)
4:     add u to V
5: for each u ∈ V such that c_u = 0 do
6:     PAL-threads { computeVertex(u) } nowait
```

computeVertex(u)

```
1: compute u
2: N ← computeOutgoingNeighbors(u, D, I)
3: for each v ∈ N do
4:     c_v ← c_v − 1
5:     if c_v = 0 then
6:         PAL-threads { computeVertex(v) } nowait
```

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{M[i,k] + M[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Based on this specification, the subproblem $M[i,j]$ depends on $M[i,k]$ and $M[k+1,j]$ for $i \le k < j$. Thus computeCounter would initialize the counter of $M[i,j]$ to $2(j-i)$. The following subproblems depend on $M[i,j]$: $M[i,k]$ for $j < k \le n$ and $M[k,j]$ for $1 < k \le j$. These are the subproblems returned by computeOutgoingNeighbours$(M[i,j], \mathcal{D}, I)$.

Example 3.2 *Consider the longest common subsequence (LCS) problem which, given two strings S and T, asks for the longest string that is a subsequences of both S and T (see Definition 2.11 in Section 2.8.2). For a string S, let $S[i..j]$ denote the substring of S containing characters $s_i s_{i+1} \dots s_j$, and let $M[i,j]$ denote the longest common subsequence between $S[1..i]$ and $T[1..j]$. Recall from Equation 2.8.1 in Section 2.8.2 that $M[i,j]$ can be computed with the following recurrence:*

$$M[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ M[i-1,j-1] & \text{if } i,j > 0 \text{ and } s_i = t_j \\ \max\{M[i,j-1], M[i-1,j]\} & \text{if } i,j > 0 \text{ and } s_i \ne t_j \end{cases} \tag{3.2.7}$$

According to the recurrence above, computeCounter would initialize the counter of $M[i,j]$ to 0 if $i = 0$ or $j = 0$, and to 3 otherwise. Similarly, computeOutgoingNeighbours$(M[i,j], \mathcal{D}, I)$ returns $\{M[i+1,j], M[i,j+1], M[i+1,j+1]\}$ if $i < |S|$ and $j < |T|$, $\{M[i+1,j]\}$ if $i < |S|$ and $j = |T|$, $\{M[i,j+1]\}$ if $i = |S|$ and $j < |T|$, and the empty set if $i = |S|$ and $j = |T|$ (an implementation

that also takes into account the contents of the input strings could instead initialize counters and compute outgoing neighbours differently depending on whether the corresponding characters are equal or not).

If $T(n)$ is the time that it takes to compute the solution of the problem, our goal is to compute the solution optimally in time $T_p(n) = O(T(n)/p)$. There are two main aspects that affect the speedup factor:

1. **The amount of parallelism implicit in the DAG.** As stated earlier, the dependency graph of the dynamic programming algorithm can affect the amount of parallelism. If most antichains have size smaller than p, then our method cannot obtain much parallelism (e.g., consider the extreme case of having a path as the dependency graph). For most dynamic programming algorithms of dimension more than one, antichains are usually large enough to support parallelism. For the two examples above, the antichains are diagonals and as we have $p = O(\log n)$, most antichains have size larger than p.

2. **The slowdown from simultaneous attempts to update a vertex counter.** Observe that updating the counters of the neighbours of a vertex cannot always be done in parallel in a CREW model. Hence, we use a standard simulation technique to obtain CRCW behaviour on a CREW PRAM with a $\log p$ slowdown factor [Fich et al., 1988].

Ignoring overheads for simultaneous lookups as described above, our strategy achieves a p-fold speedup over the sequential dynamic programming algorithms for the matrix chain multiplication and longest common subsequence problems. In the case of matrix chain multiplication, the total number of vertices in the DAG is $\Theta(n^2)$ and its depth is $\Theta(n)$. Since the value associated with each vertex can be computed in $O(n)$ time, which is also the time it takes to update counters of outgoing neighbours, the total work is $O(n^3)$ and the critical path of the computation takes $O(n^2)$ time. Since the LoPRAM scheduler is greedy, by Lemma 2.2 the parallel time of Algorithm 3.1 is $T_p(n) = O(n^3/p + n^2)$, which is for $p = O(\log n)$ is $O(n^3/p)$, and thus we achieve a p-fold speedup over the $\Theta(n^3)$ time sequential dynamic programming algorithm. Note that this parallel algorithm is not work-optimal, as there exists an $O(n \log n)$ time sequential algorithm for this problem [Hu and Shing, 1982, 1984].

Similarly, the DAG corresponding to Recurrence 3.2.7 for the LCS problem has $\Theta(n^2)$ vertices and $\Theta(n)$ depth (for two strings of length $\Theta(n)$ each). As each vertex can be computed in constant time, the parallel time of Algorithm 3.1 in this case is $T_p(n) = O(n^2/p + n) = O(n^2/p)$, which is a again a p-fold speedup. This is not work-optimal, however, as there exists an $O(n^2/\log n)$ algorithm for LCS [Masek and Paterson, 1980] (in the case in which sequences are drawn from a set of bounded size [Cormen et al., 2001]). In Section 5.3.4 in Chapter 5 we use ideas of this faster LCS algorithm to obtain a parallel algorithm for this problem in the Ultra-Wide Word model.

Parallel Memoization

Memoization is a strategy to solve problems that have a similar recursive decomposition as in Equation (3.2.5), but in which the execution of the algorithm is carried out recursively in a top-down fashion. It differs from usual divide-and-conquer recursive algorithms in that the first time each subproblem is solved, its result is stored in order to avoid further computations of the same result. Initially, all the subproblems contain a value that indicates that the solution has not yet been computed. Before a subproblem is solved, its entry is looked up in the structure. If the entry has a previously computed value, this value is used and no further computation for this subproblem is carried out. Otherwise, the subproblem is solved, and the solution is stored in the structure.

The parallelization of an algorithm that uses memoization is similar to the one introduced earlier on for divide-and-conquer algorithms: each recursive call is assigned to a different PAL-thread, with the difference that threads might return immediately a previously computed value, or wait until a value being computed by another thread becomes available. Suppose that a thread t is created to compute a value $M[x]$. We have three cases:

- If $M[x]$ has already been computed by another thread, then t returns this value.
- If $M[x]$ has not been computed and its computation has not been initiated by another thread, then t records $M[x]$ as "in progress" and proceeds to compute the value. If $M[x]$ corresponds to a base case, then the value is computed and returned. Otherwise, let $M[y_1], \ldots, M[y_r]$ be the values that $M[x]$ depends on. Thread t launches a PAL-thread for each of these values and enters a wait state, becoming ready again when all its children are done.
- If $M[x]$ is not present but recorded as already in progress by another thread, t registers a notify condition on solution and blocks in non-busy wait, so the processor is now free to serve another PAL-thread. Once the solution is available, the thread is ready and resumes execution as determined by the scheduler.

Note that with this strategy there is no possibility of deadlock, as dependencies between threads correspond to dependencies in the computational DAG of the corresponding dynamic program, which is cycle-free.

Probes to test if a solution of a subproblem is available or in progress can be implemented with atomic update operations to a shared status variable. Note that, unlike the bottom-up approach that we described previously, there is no overhead factor due to simultaneous memory accesses, as once the status of a subproblem is set to available or in progress, the rest of the accesses by simultaneous threads are read-only.

As before, the speedup factor depends on the inherent parallelism of the dependency graph of the problem.

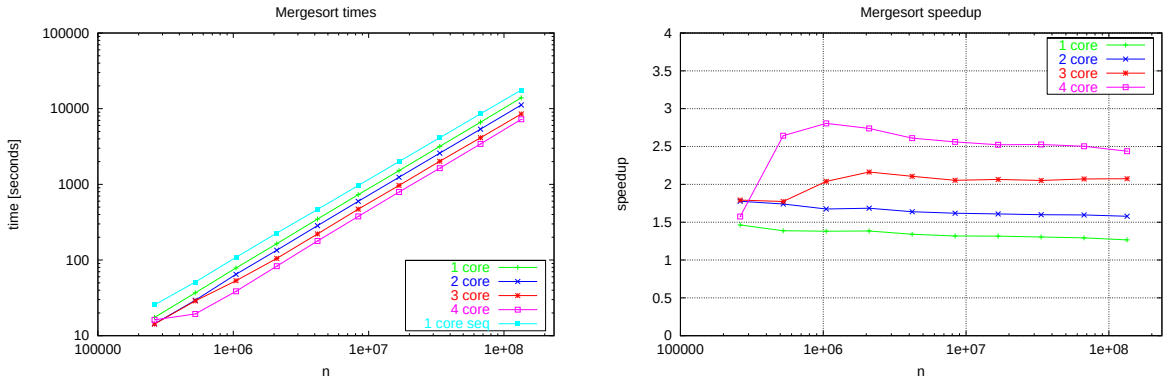


Figure 3.4: Left: mergesort times for randomly generated inputs with sizes from 2^{18} to 2^{27}. For each input size the time shown is the average of 25 runs on different inputs. Right: Speedup with respect to the sequential execution.

3.3 Experiments

We implemented a parallel scheduler for the LoPRAM model as described in Section 3.1.1 and tested it with two implementations of divide-and-conquer algorithms: mergesort and Strassen's matrix multiplication. The parallel implementations of these algorithms are essentially the sequential ones with PAL-threads for recursive calls.

The algorithms were implemented in C++ with MFC (Microsoft Foundation Class), and the experiments were run on an Intel® Core™ 2 Extreme CPU Q6850 (4 cores) at 3 GHz, with 8 Mb Cache and 4 Gb RAM running Windows Vista™ Business 64-bit. Figures 3.4 and 3.5 show the times of the parallel execution of mergesort and matrix multiplication, respectively, with 1, 2, 3, and 4 cores, as well as the times of the sequential recursive implementations. The speedup with respect to the sequential times are shown on the right. The parallel execution with one core is an execution with the LoPRAM scheduler with 1 as the parameter for the number of available processors.

We observe that for matrix multiplication we obtain nearly optimal p-fold speedups over the sequential implementation, while for mergesort speedups are slightly lower than p. The difference in performance between the two algorithms can be explained in that as the merging phase of matrix multiplication is more expensive than the one of mergesort ($\Theta(n^2)$ vs $\Theta(n)$), the overhead due to the parallel scheduling of recursive calls is less significant for the first algorithm, leading to almost optimal speedup in all instances. In addition, the fact that for mergesort the parallel execution with one core is faster than the sequential execution is explained in that the operating system schedules the sequential execution *across all four cores* of the system, incurring in an extra cost for context switching when the computation is continued in another core (which includes

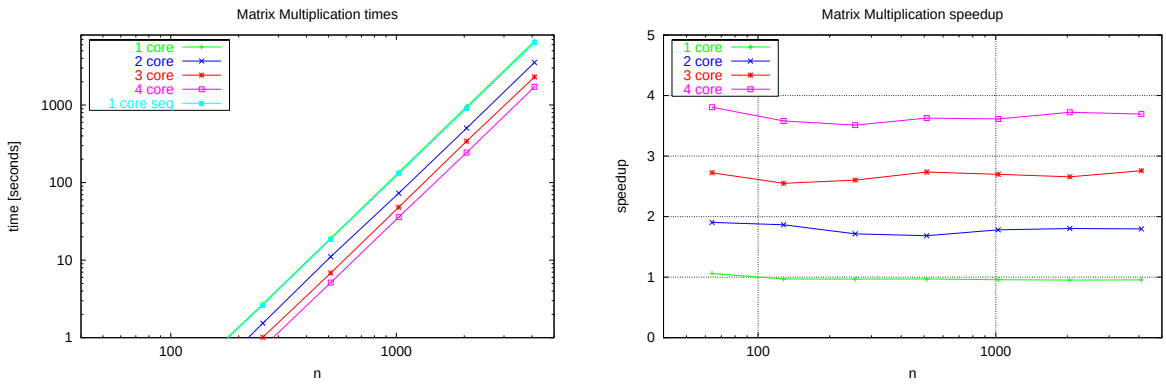


Figure 3.5: Left: Matrix multiplication times for randomly generated inputs with sizes from 64 to 4096. For each input size the time shown is the average of 5 runs on different inputs. Right: Speedup with respect to the sequential execution.

the cost associated with the reduced reuse of data in private caches). In contrast, the LoPRAM implementation with $p = 1$ pins the execution on one physical core.

These experiments show that straightforward modifications to these sequential divide-and-conquer algorithms are sufficient, at least in principle, to obtain significant and nearly optimal speedups in the LoPRAM model.

We also implemented the generic dynamic programming strategy as described in Algorithm 3.1 and used it to solve instances of the matrix chain multiplication problem. Recall that according to the dynamic programming recurrence for this problem (Equation 3.2.6), the outgoing neighbours of the subproblem $M[i, j]$ in the corresponding dependency graph are the subproblems $M[i, k]$ for $j < k \leq n$ and $M[k, j]$ for $1 < k \leq j$. Thus, a vertex can have $\Theta(n)$ neighbours, which means that updating the counters of a vertex's neighbours in the computeVertex method in Algorithm 3.1 can take $\Omega(n)$ time. Moreover, each counter update should be done atomically in order to avoid concurrency issues when two or more threads try to update the same counter. Since such operation requires a usually expensive system call, the time spent updating counters of neighbours becomes significant.

In the case of this particular problem, we can simplify this operation by taking advantage of the structure of the dependency graph. Consider a vertex v representing the value of the subproblem $M[i, j]$. All incoming edges of v originate in vertices corresponding to entries $R = \{M[i, k], k < i\}$, or $C = \{M[k, j], k < j\}$. Since this holds as well for any entry in R and C, the vertices corresponding to each set R and C in the dependency graph form a chain. This implies that when processing entries $M[i, j-1]$ and $M[i-1, j]$, all entries in R and C have already been processed, and thus after the values of $M[i, j-1]$ and $M[i-1, j]$ have been computed, all entries on which $M[i, j]$ depends have been computed. Therefore, when processing a vertex v in the

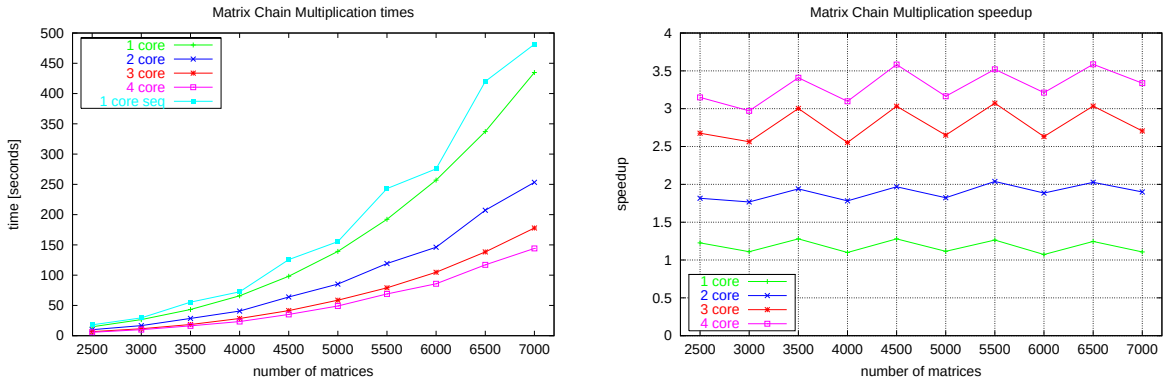


Figure 3.6: Left: Matrix chain multiplication times as a function of the number of matrices in the input. For each input size the time shown is the average of 10 runs on different inputs. Right: Speedup with respect to the sequential execution.

graph, we do not need to consider all dependencies in the same row or column but only the ones originating from the vertices in the immediately preceding diagonal antichain. This allows us to reduce the size of the outgoing neighbours of each vertex to at most 2 (vertices above and to the right in Figure 3.3), which reduces the time required to update counters. We still require all values in R and C in order to compute the value of $M[i, j]$, and thus computing each value still takes $\Theta(n)$ time for most cells. However, the reduction in both the number of system calls and the slowdown due to simultaneous attempts to update counters leads to significant improvements in running time compared to the original generic solution. The times and speedups obtained for the simplified dynamic programming solution for this problem are shown in Figure 3.6. Once again, we obtained nearly optimal speedups with respect to the original dynamic programming algorithm with a simple strategy applicable to large class of dynamic programming algorithms.

3.4 Conclusions

We introduced the LoPRAM, a new model for parallel computation that is faithful to current multi-core architectures, avoids many of the pitfalls of the previous PRAM model, namely difficulty of programming and expensive processor communication infrastructure, and allows for significant classes of problems to be parallelized with little effort. Our model supports a high level abstraction that simplifies the design and analysis of parallel programs. We provided LoPRAM parallel algorithms with optimal speedup for divide-and-conquer and dynamic programming problems by applying simple modifications of the corresponding sequential algorithms. These serve as starting examples of algorithm design and analysis in the LoPRAM model and give an insight of the techniques for parallelization that we seek to apply to a wider set of problems.

Future directions of research include refining the model by considering, for example, cache complexity and processor communication, as well as finding other families of algorithms for which we can apply general parallelization techniques to obtain simple parallel algorithms with p-fold speedups. We believe that the assumption $p = O(\log n)$ can significantly aid this process which, given the low degree of parallelism in multi-cores, would contribute to the development of applications that take full advantage of the parallelism offered by the architecture.

Chapter 4

On the Sublinear Processor Gap for Parallel Architectures

In Chapter 3 we introduced a model of low-degree parallelism in multi-cores, showing the benefits of assuming a small number of processors. In this chapter we explore the influence of the number of processors further in parallel computation in general, reporting an observed gap in the behavior of parallel architectures depending on the number of processors. This gap appears repeatedly in both empirical cases, when studying practical aspects of architecture design and program implementation as well as in theoretical instances when studying the behaviour of various parallel algorithms.

There is a vast experience in the study and development of algorithms for the PRAM architecture. In this case, the standard assumption (though often unstated) was that the number of processors p was linear on the size of the input, i.e., $p = \Theta(n)$ (see, e.g., [Greenlaw et al., 1995] for a thorough discussion). Indeed, the definition of the class NC, which is often equated with the class of problems that can be efficiently parallelized on a PRAM, allows for up to polynomially many processors. Hence, algorithms were designed to handle the case when $p = \Theta(n)$ or $p = \Theta(n^k)$ for $k \geq 1$ and if the actual number of processors available was lower, this could readily be handled by Brent's Lemma using a suitable scheduler [Brent, 1974; Bender and Phillips, 2007] (see Section 2.3.2). A fruitful theory was developed under these assumptions, and papers in which $p = o(n)$ were relatively rare.

Our results[1] We analyze and report on the influence of the assumed number of processors on several aspects of the performance of various types of parallel architectures. Because of its current prevalence, we focus especially on multi-core architectures, which actually feature a

[1] Results in this chapter appeared in [López-Ortiz and Salinger, 2013].

Processor count	$\Theta(n)$	$\Theta(n^\alpha)$	$\Theta(\log n)$
Merge sort	✗	✗	✓
Master theorem			
-Case 1	✗	✓	✓
-Case 2	✗	✓	✓
-Case 3	✗	✗	✗
Amdahl's law	✗	✓ (if $\alpha \leq 1/2$)	✓
Collision	✗	✓ (if $\alpha \leq 1/2$)	✓
Buffering	✗	✓	✓
Network size	✗	✓ (if $\alpha \ll 1/2$)	✓
TM simulation	✗	✗	✓

Table 4.1: Optimal performance for each case according to processor count, with $0 < \alpha < 1$.

relatively small number of processors, and hence advantages that can be identified for parallel systems with a small number of processor count can lead to benefits in parallel computation in these architectures. However, we also report on aspects of parallel computation that are relevant in general in other architectures, such as memory collisions, communication in distributed architectures, and network sizes, as well as in more theoretical aspects like complexity classes and simulations of other models. Our observations suggest the existence of fundamental differences in the qualities of parallel systems with sublinear and linear number of processors, and that exploiting the advantages of the former can lead to more practical and conceptually simpler designs of both parallel architectures and algorithms, ultimately increasing their adoption and reducing development costs. We present an overview of arguments and observations in Section 4.1, followed by the detailed exposition of arguments in Section 4.2. Section 4.3 presents concluding remarks for this chapter.

4.1 Overview of Arguments

In this section we briefly list the arguments in favour of considering a limited degree of parallelism. We emphasize that we did not start from the outset with this goal, but rather we sought to develop algorithms and tools (both practical and theoretical) for current multi-core architectures. The observations within are derived from both theoretical investigations and practical experiences in which time and time again we found that there seems to be a qualitative difference between a model with $O(\log(n))$ processors and one with $\Theta(n)$ processors, with, surprisingly, the advantage being for the weaker, i.e., $O(\log(n))$ model. Table 4.1 shows a summary of our observations for the considered processor counts. There is strong evidence of a sublinear cliff, beyond which development and implementation of efficient PRAM algorithms for many problems is substantially

harder if not completely impossible, unless $\mathsf{P} = \mathsf{NC}$. In several instances among the evidence observed, the phenomenon had been observed earlier by others [Greenlaw et al., 1995; Kruskal et al., 1990]. We now list our arguments briefly, before we expand on each of them individually in the next section.

1. The number of cores in current multi-core processors is nearly a constant, but first, if it is truly a constant, there is not much we can say about parallel speedups, and second, it seems to be steadily though slowly growing.
2. In analogous fashion to the word-RAM, the number of bits in a word could be an arbitrary w but really it is most likely $\Theta(\log n)$, since it is also an index into memory, and memory is usually polynomial on n.
3. The probability of collision on a memory access is only acceptably low for up to $O(\sqrt{n})$ processors.
4. The number of interconnects on a CPU network is prohibitively large for a large number of processors.
5. Serialization at the network end is too costly, i.e., if more than two processors want to talk to a single processor at the same time, this processor has to listen to them serially.
6. There are natural $\log n$ and n^ϵ barriers in the complexity of designing algorithms.
7. Efficient cache performance requires bounded number of processors in terms of cache sizes, which are always assumed to be below n, and often as well in terms of the ratio of shared and private cache sizes, which is well below 100.
8. We define the class of problems which can be sped up using a logarithmic number of processors and show that it contains ENC and EP [Kruskal et al., 1990] and, furthermore, this containment is strict.
9. For Turing machines we can automatically increase performance when simulating with a parallel computer using random access memories, with natural constraints limiting the speedup to a $\Theta(\log n)$ factor.
10. Amdahl's law suggests that programs can only noticeably benefit from parallelism if the number of processors is proportional to the relative difference between the execution time of the serial and parallel portions of a program.

4.2 Exposition

In this section we briefly expand on each of the points above. We aim to keep each argument as short as possible, since the entirety of the case is more important than any individual point.

4.2.1 Limited Parallelism

In principle, it is possible to build a computer with an arbitrary degree of parallelism. In practice, PRAMs algorithms and architectures focused on $\Theta(n)$-processor architectures, while relying on Brent's Lemma for cases when the number of processors was below that. In contrast, multi-core processors have aimed for a much smaller number of cores. In principle, this number could be modeled as a constant. However, this is unrealistic as the number of cores continues to grow—albeit slowly—with desktop computers having transitioned over the last decade from single core to dual core to quad core and presently eight cores and sixteen cores already shipping at the higher end of the spectrum. Additionally, it has been observed that generally speaking larger inputs justify larger investments in RAM and CPU capacity, so a function of n is much more reflective of real life constraints. This suggests that the number of cores is a function which grows slowly on the input size n, since there is a high processor cost. Let $\mathcal{P}(n)$ denote this function. Natural candidates for $\mathcal{P}(n)$ are $\Theta(\log n)$ and $\Theta(n^\alpha)$ for $\alpha < 1$, though there are other possibilities. Over the next subsections we shall consider various candidates for $\mathcal{P}(n)$.

4.2.2 Natural Constraints

The ability to index memory using a computer word as an address in a program's virtual memory suggests that the size of the word is $w = \Omega(\log M)$, where M is the memory size, though this does not necessarily need to be the case[2]. Memory itself is usually a polynomial function of the input size, i.e., $M = \Theta(n^k)$ for some $k \geq 1$, with $k = 1$ being a common value. Substituting $M = \Theta(n^k)$ in $w = \Omega(\log M)$ gives $w = \Omega(\log n)$. This is assumed in the word-RAM model, in order for algorithms to be able to refer to any input element. A common assumption in word-RAM papers is actually $w \approx \log n$, which enables constant-time lookup-table implementations of some functions on words while keeping table sizes sublinear (see, e.g., [Munro, 1996]), and restricts the size of pointers in succinct data structures that could otherwise increase their space usage (see, e.g., [Bose et al., 2009]).

Hence, the word size, which in the early days of computing was treated as a constant, namely 4 or 8 bits, became better understood as in fact proportional to the logarithm of the input size, that

[2]In practice, there have been architectures in which the memory size was strictly greater than 2^w. Currently, in the Intel architecture the size w places a limit on the largest addressable space, but this has not always been the case (e.g., the 8088 processor).

is $\Theta(\log n)$. Similarly, in modern multi-core computers, the number of processors has remained relatively bounded (in contrast to commercial PRAMs or GPUs which support anywhere from hundreds to thousands of processors). This relatively slow growth (at least as compared to most other usually exponential growing performance hardware indices) on the number of processors can thus be best modeled as $\log n$ in similar fashion to the word size.

4.2.3 Write Conflicts

We now analyze memory contention between threads as a function of the number of processors, when write memory accesses are assumed to be distributed uniformly at random among memory cells.

Consider a multi-threaded server application receiving requests from several clients simultaneously. Assume that these requests are served by parallel threads running on p processors that share the system's memory. Such an application is likely to have several portions of the computation accessing shared data such as database tables, buffers, and other shared data structures. Write access to shared data involves synchronization to avoid race conditions, usually implemented by synchronization primitives such as barriers and locks. In general, regardless of how synchronization is implemented, a simultaneous memory access to the same memory cell involves an overhead, either due to serialization or data invalidation. Let us call a simultaneous access by a pair of threads a *collision*. We define a collision in terms of pairs of threads. Thus, a simultaneous access to the same memory cell by t threads is counted as $\binom{t}{2}$ collisions.

We are interested in analyzing the influence of the number of processors on the number of collisions during a period of computation. The uncertainty added by the timing of client requests suggests that write access to shared memory can be modeled as a random process with a certain probability of collision. A crude but reasonable approximation is to model the memory accesses of each process as uniformly distributed over memory cells at each step.

We investigate the expected number of collisions for p threads accessing m memory cells, uniformly at random at each timestep of a period of service time. Clearly, the smaller the number of processors the lower the probability of collision. The question is for what value of p as a function of m does this probability become negligible. Note that in general the size of the memory is usually modeled as a growing function of a program's input size, with $m = O(n^k)$ being a common assumption. Thus, it is reasonable to analyze the number of collisions as m grows.

This reduces to a balls-and-bins scenario (see, e.g., [Feller, 1968]). Let us first consider the total number of overall collisions in one step. Let C be a random variable denoting this number. The probability that two memory accesses are to the same cell is $1/m$. Since there are $\binom{p}{2}$ pairs of memory accesses, the expected number of collisions in one step is $E[C] = \frac{p(p-1)}{2m}$. As m grows,

this expression tends to 0 if $p = o(\sqrt{m})$, tends to infinity if $p = \omega(\sqrt{m})$, and it converges to a positive constant for $p = \Theta(\sqrt{m})$. Figure 4.1 (left) shows the expected number of collisions as m grows for various processor counts.

Now we consider an alternative expression for memory access conflicts, namely the number of cells involved in collisions at each step. Thus, if three or more accesses are to the same cell, the event counts as one conflict. Let X be a random variable denoting the number of memory cells which suffer a collision when there are p simultaneous memory accesses. The probability of a memory cell not being accessed is $(1 - 1/m)^p$, and thus the expected number of accessed cells is $m - m(1 - 1/m)^p$. Then, for p accesses the expected number of cells for which there is more than one access is $E[X] = p - m + m\,(1 - 1/m)^p$.

Assume that $p = m^\alpha$ with $\alpha \leq 1$. The expression above is then

$$E[X] \approx m^\alpha - m + me^{-m^{\alpha-1}}.$$

Using the Taylor expansion of $e^{-m^{\alpha-1}}$ we obtain

$$\begin{aligned} E[X] &\approx m^a - m + m\left(1 - m^{\alpha-1} + \frac{m^{2(\alpha-1)}}{2} + \text{ l.o.}\right) \\ &\approx \frac{m^{2\alpha-1}}{2}. \end{aligned}$$

Again, when m tends to infinity, the above tends to 0, 1/2, or diverges if α is less, equal, or greater than 1/2, and thus the threshold again is for $p = \Theta(\sqrt{m})$ (see Figure 4.1 (right)). Clearly the smaller p is, the fewer the expected the collisions.

Case 1. If $p = m$, then $E[X] = m/e$, and $E[C] = (m-1)/2$. Thus, in each step about 37% of memory cells have more than one processor trying to access them and about half of the accesses result in collisions.

Case 2. If $p = \sqrt{m}$, then on average over the execution of the program there is a collision every two steps.

Case 3. If $p = o(\sqrt{m})$, the expected number of collisions goes to zero as m grows.

Suppose that every instruction takes unit time if there is no collision and $s \geq 1$ units of time otherwise. The expected number of collisions per processor per step is $\frac{(p-1)}{2m}$, and thus the expected slowdown in performance due to collisions is $\frac{s(p-1)}{2m}$, which is negligible for $p = o(m/s)$.

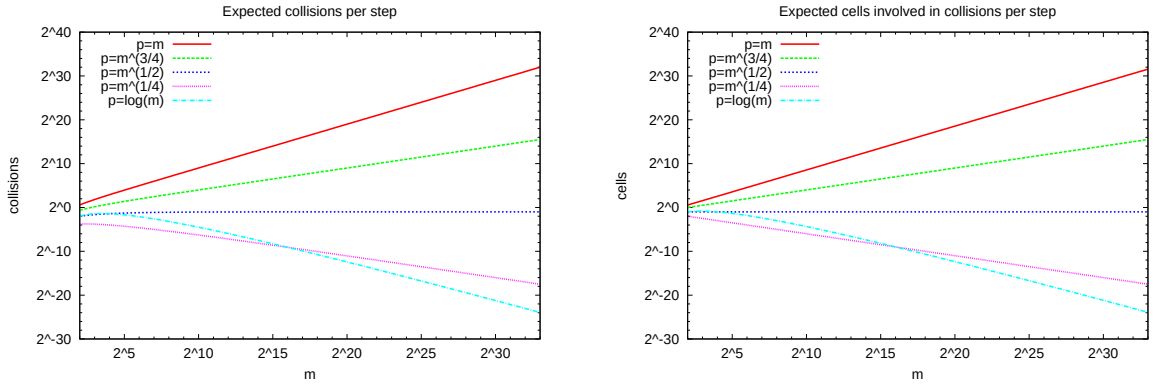


Figure 4.1: Expected number of collisions per step (left) and number of memory cells involved in collisions (right) for various processor counts as a function of the memory size m (in logarithmic scale).

4.2.4 Processor Communication Network

Traditionally, parallel computers use either shared memory or a processor communication network (or both) to exchange information between the various processing units. The advantage of shared memory is that no additional hardware is required for it; the disadvantages are issues of synchronization and memory contention. Hence, a widely explored alternative is the use of an ad-hoc processor communication network connecting the processors. In general, from the perspective of performance, a full communication network is the preferable network architecture. However, when the number of processors is assumed to be very large this is unfeasible. For example, for the case of $\Theta(n)$-processors of many commercial PRAM implementations, the number of interconnects required would have been $\Theta(n^2)$ which is prohibitive. Thus there was extensive study of alternative network topologies which reduced the complexity of the network while attempting to minimize the penalty in performance derived from the smaller network. Among the most successful such architectures we have the hypercube, the butterfly, and the torus (see Section 2.3.3).

We observe now that full processor communication network becomes a realistic possibility if the number of processors is $O(\log n)$ or even possibly $O(n^\alpha)$ for some $\alpha \ll 1/2$. For example, for a modest (by present standards) input size of $n = 2^{27} = 134,217,728$, even $n^{1/2}$ processors would require an impossible number of interconnects on the full graph ($\binom{\sqrt{n}}{2} \approx 6.7 \times 10^7$). A complete network of $\log n = 27$ processors, on the other hand, would require 351 interconnects, which are well within the realm of current architectures.

4.2.5 Buffer Overflow

Aside from issues of network topology, in practice it is natural to assume that each processor in a communication network can handle at most a small constant number of messages at once. If more than a constant number of processors send messages to a single processor, said messages would queue at the receiving end for further processing. In this section we consider a natural communication model in which in each instruction cycle a processor may send a message to at most one other processor. In practice, depending on the specific application the probability of collision may range anywhere from zero for the execution of independent threads to one for, say, a master processor serializing requests to some shared lock. As a compromise, we model again this process as if the processors chose their destination uniformly at random. Let p be the number of processors; then the maximum number of collisions observed at the most loaded buffer is $(\ln p / \ln \ln p)(1 + o(1))$ with high probability [Raab and Steger, 1998]. For input sizes $n > 2^{22}$, buffer handling with $p = n$ can introduce delays of about twice as many instruction cycles than with $p = \log n$, with the difference growing unboundedly (albeit slowly) for larger input sizes.

4.2.6 Divide-and-Conquer Algorithms

Divide-and-Conquer algorithms are naturally suited for parallelization. Instances at the same level of the recursion tree are independent and can be scheduled to be executed in parallel. This is especially well suited for multi-threaded systems, as each recursive call can simply be handled by a separate thread. This strategy requires no parallelization of the divide and combine phases of the recursion, which can be executed by each thread just as in the sequential algorithm. We showed in Chapter 3 that this easy parallelization yields optimal speedups for a large class of divide-and-conquer algorithms, but only for a bounded number of processors. Thus, in a system with a logarithmic or sublinear number of processors, obtaining the maximum possible speedup for this class of algorithms is simple and can be realized with a general strategy that is independent of the algorithm itself.

Consider a divide-and-conquer algorithm whose time complexity can be written as $T(n) = aT(n/b) + f(n)$. The master theorem [Cormen et al., 2001] yields the time bounds for a sequential execution of such an algorithm. A parallel version of this theorem can be obtained by analyzing the parallel time $T_p(n)$ of an execution in which recursive calls are executed in parallel and scheduled with the LoPRAM scheduler or work stealing [Burton and Sleep, 1981] with a bounded number of processors (see Section 3.2.1):

$$T_p(n) = \begin{cases} O(T(n)/p), & \text{if } f(n) = O(n^{\log_b(a)-\epsilon}) \text{ and } p = O(n^\epsilon) & \text{(Case 1)} \\ O(T(n)/p), & \text{if } f(n) = \Theta(n^{\log_b a}) \text{ and } p = O(\log n) & \text{(Case 2)} \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ and } af(n/b) \leq cf(n), \text{ for some } c < 1 & \text{(Case 3)} \end{cases} \tag{4.2.1}$$

Optimal speedups are achieved in Cases 1 and 2 only for $p = O(n^\epsilon)$ for $\epsilon > 0$, and $p = O(\log n)$, respectively. In Case 3, the time is dominated by the sequential divide and conquer time $f(n)$ at the top of the recursion.

We note that it is possible to obtain optimal speedups with larger numbers of processors for many divide-and-conquer algorithms. However, this invariably requires parallelizing the divide and combine phases of the algorithm, as otherwise the sequential time $f(n)$ of the divide and combine phases dominates the parallel time. In fact, as shown in Section 3.2.1, if an optimal parallel algorithm for the divide and combination phases is known, then all cases above yield optimal speedup, and the bounds of the processors can be relaxed. Then Case 3 becomes:

$$T_p(n) = \Theta(f(n)/p), \quad \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ and } af(n/b) \leq cf(n), \text{ for some } c < 1$$

Now Case 1 requires $p = O\left(\frac{n^{\log_b a}}{\log n}\right)$, Case 2 requires $p = O\left(n^{\log_b a}\right)$, while Case 3 requires $p = O(f(n)/\log n)$.

The result of Equation (4.2.1) shows that for a system with a small number of processors the implementation of parallel divide-and-conquer algorithms that achieve the full speedup offered by the architecture is simple and can be implemented without the unnecessary complexity of implementing specific parallel algorithms for the divide and combine phases of the algorithms.

When considering cache performance of divide-and-conquer algorithms, a bounded number of processors can also be advantageous. Blelloch et al. [2008] show that the class of *hierarchical* divide-and-conquer algorithms —algorithms in which the divide and combine phases can also be implemented as divide-and-conquer algorithms— can be parallelized to obtain optimal speedups and good cache performance when scheduled with a Controlled-PDF scheduler (see Section 2.8.2). While a Brent's Lemma type of implementation of some of the algorithms in [Blelloch et al., 2008] can achieve optimal speedups for a large number of processors (e.g., matrix addition and cache oblivious matrix multiplication algorithms can both be sped up optimally up to n^2 processors) [Blelloch et al., 2008], the optimal speedup and cache performance bounds under the Controlled-PDF scheduler is only achieved for a much smaller number of processors, bounded by the ratio between shared and private cache sizes, and even smaller in some cases, as we shall see in the next section.

4.2.7 Cache Imposed Bounds

Cache contention is a key factor in the efficiency of multi-core systems. Various multi-core cache models have been proposed which focus on algorithms and schedulers with provable cache performance. We describe some of these models in more detail in Section 2.8.2. Many of the results involving shared and private caches performance require bounds on the number of processors related to the size of the input and/or to the relative sizes of private and shared caches.

The Parallel External Memory (PEM) model [Arge et al., 2008] models p processors, each with a private cache of size M, partitioned in blocks of size B. A sorting algorithm given in this model is asymptotically optimal for the I/O bounds for at most $p \leq n/B^2$ processors, and it is actually proven that $p \leq n/(B \log B)$ is an upper bound for optimal processor utilization for any sorting algorithm in the PEM model [Arge et al., 2008]. This algorithm is used in further results in the model for graph and geometry problems [Arge et al., 2010; Ajwani et al., 2010, 2011]. Thus the assumption that $p \leq n/B^2$ is carried on to these results as well, some of which actually require $p \leq n/(B \log n)$ and even $p \leq \frac{n}{B^2 \log B \log^{(t)} n}$, where $\log^{(t)} n$ denotes the composition of t log functions, and t is a constant.

Shared cache performance is studied in [Blelloch and Gibbons, 2004], which compares the number of cache misses of a multi-threaded computation running on a system with p processors and shared cache of size C_p to those of a sequential computation with a private cache of size C_1. It is shown that under the PDF-scheduler [Blelloch et al., 1999], the parallel number of misses is at most the sequential one if $C_p \geq C_1 + pd$, where d is the critical path of the computation. This implies that $p \leq (C_p - C_1)/d$, which is less than n (as otherwise all the input would fit in the cache) and is usually sublinear, as d is rarely constant and is $\Omega(\log n)$ for many algorithms. Thus, for many algorithms the bound on the parallel misses holds for $p = O(n/\log n)$.

As mentioned in Section 4.2.6, Blelloch et al. [2008] study hierarchical divide-and-conquer algorithms in a multi-core cache model of p processors with private L_1 caches of size C_1 and a shared L_2 cache of size C_2. An assumption of the model is that $p \leq \frac{C_2}{C_1} \ll n$, since the input size is assumed not to fit in L_2. It is shown that under a Controlled-PDF scheduler, parallel implementations achieve optimal speedup and cache complexity within constant factors of the sequential cache complexity for a class of hierarchical divide-and-conquer algorithms. Optimality for some algorithms, such as Strassen's matrix multiplication and associative matrix inversion even require $p \leq (C_2/C_1)^{\frac{1}{1+\epsilon}}$ [Blelloch et al., 2008].

Cache efficient dynamic programming algorithms have been designed in this multi-core model with the same $p \leq \frac{C_2}{C_1}$ assumption [Chowdhury and Ramachandran, 2008], as well as in a shared cache model with $p \leq C_2/B$, where B is the block size. Thus, although the time complexity of parallel dynamic programming allows a large number of processors for optimal speedups (e.g., $T_p = O(n^3/p + n)$ for Gaussian elimination paradigm problems, which is optimal for $p \leq n^2$), the efficiency in cache performance restricts the level of parallelism.

Observe that presently the ratio between L_2 shared cache and private L_1 cache is in the order of 4 to 100 depending on the specific processor architecture.

4.2.8 The Class $E(p(n))$

The class NC can be defined as the class of problems which can be solved in polylogarithmic time using polynomially many processors (see Section 2.3.6). It is believed that NC $\neq$ P and hence

that there are known problems which do not admit a solution in time $O(\log^k n)$, for any $k \geq 1$. In our case we are interested in the study of problems which can be sped up using $O(\log n)$ or $O(n^\alpha)$ processors for $\alpha < 1$. Kruskal et al. [1990] introduced the classes ENC and EP which encode the classes of problems that allow optimal speed up (up to constant factors) using polynomially many processors on a CRCW PRAM. The class ENC has polylogarithmic running time, while the class EP has polynomial running time. They also define the related classes SNC, ANC, SP, and AP, which are analogous to ENC and EP in terms of the required running times but allow for some inefficiency. In general, one could introduce the class $\mathcal{C}(p(n), S(n))$ as the class of problems that allow a speedup of $S(n)$ with $p(n)$ processors. Thus, following the notation in [Kruskal et al., 1990], we define the class $E(p(n)) = \mathcal{C}(p(n), p(n))$, which is the class of problems that can be solved using $O(p(n))$ processors in time $O(T(n)/p(n))$, where $T(n)$ is the running time of the best sequential solution to the problem. We are particularly interested in the classes $E(\log n)$ and $E(n^\alpha)$ for $\alpha < 1$ [3].

The class ENC is a sharpening of the well known class NC. Recall that the class NC requires maximal speedup down to polylogarithmic time even at the cost of a polynomial amount of inefficiency (i.e., the ratio between parallel and sequential work). In contrast, ENC requires the same speedup but bounds the inefficiency to a constant factor.

The class $E(\log n)$ bounds the inefficiency to a constant which implies a speed up of $\Theta(\log n)$ on the sequential solution to the problem. By Brent's Lemma we can show that $E(\log n)$ includes the problems in classes ENC and EP. Since we investigate problems that are most worth parallelizing, we restrict this inclusion to problems with at least sequential linear time.

Theorem 4.1 *Let Π be a problem with sequential time $t(n) = \Omega(n)$. Then,*

1. $\Pi \in ENC \Rightarrow \Pi \in E(\log n)$
2. $\Pi \in EP \Rightarrow \Pi \in E(\log n)$

Proof: Since $ENC \subseteq EP$ [Kruskal et al., 1990], statement (2) implies (1). Let us show statement (2). Let $\Pi \in EP$ be a problem with sequential running time $t(n)$. Let A be an algorithm that solves Π in time $O(t(n)^\epsilon)$ with p processors, where $\epsilon < 1$ [Kruskal et al., 1990]. Since A is work-optimal, the total work done by A is $O(t(n))$. Then, by Brent's Lemma [Brent, 1974] (see Section 2.3.2), we can simulate A in $T'_p(n) = O(t(n)/p' + t(n)^\epsilon)$ time with $p' \leq p$ processors. The simulation achieves optimal speedup for any $p' = O(t(n)^{1-\epsilon})$. Since for $E(\log n)$ we have $p' = O(\log n)$, which is $O(t(n)^{1-\epsilon})$ for any $t(n) = \Omega((\log n)^{1/(1-\epsilon)})$ (and in particular for $t(n) = \Omega(n)$), then $\Pi \in E(\log n)$. ∎

[3] For consistency in the class comparisons, we assume a CRCW PRAM as in [Kruskal et al., 1990], though these classes can be defined for other PRAM types (EREW, CREW) as well as for asynchronous models (such as multi-cores).

The reverse is not the case, i.e., not all problems that are in $E(\log n)$ are in ENC, unless $\mathsf{P} = \mathsf{NC}$: there are known P-complete problems which allow optimal speedup using a polynomial number of processors [Fujiwara et al., 2000; Vitter and Simons, 1986], and thus they are in EP [Kruskal et al., 1990] (and hence in $E(\log n)$). If any such problem is in ENC, this would imply $\mathsf{P} = \mathsf{NC}$. We conjecture that the same is the case for $E(\log n)$ and EP.

This gives a theoretical separation between the problems that can be sped up optimally using polynomially many processors and those that can be sped up using a logarithmic number of processors.

Similarly, $E(n^\alpha)$ bounds the inefficiency to a constant which implies a speed up of $\Theta(n^\alpha)$ on the sequential solution to the problem. We show that $E(n^\alpha)$ includes most problems (with at least linear time sequential complexity) in ENC. For the same reasons described above, not all problems in $E(n^\alpha)$ are in ENC, for any $\alpha < 1$.

Theorem 4.2 *Let Π be a problem with sequential time $t(n) = \Omega(n)$. Then, $\Pi \in ENC \Rightarrow \Pi \in E(n^\alpha)$.*

Proof: The proof is analogous to the proof of Theorem 4.1. Let $\Pi \in ENC$ and let A be an algorithm that solves Π in time $O(\log^k(t(n)))$ with p processors, for some $k \geq 1$. Since A is work-optimal the total work done by A is $O(t(n))$. Then, by Brent's Lemma, we can simulate A in $T'_p(n) = O(t(n)/p' + \log^k(t(n)))$ time with $p' \leq p$ processors. The simulation achieves optimal speedup for any $p' = O(t(n)/\log^k(t(n)))$. Since this holds for $p' = O(n^\alpha)$ with $\alpha < 1$ and any $t(n) = \Omega(n)$, it follows that $\Pi \in E(n^\alpha)$. ∎

4.2.9 Parallelism in Turing Machine Simulations

In Section 4.2.2 we argued that there are natural constraints in the amount of inherent parallelism of computing models. In this section we extend these arguments to show the limitations of the speedup that can be obtained from the Four Russians technique [Arlazarov et al., 1970] when used for Turing machine simulations[4].

Hopcroft et al. [1975] showed that a deterministic Turing machine M running in time $T(n)$ can be simulated in a RAM in time $O(T(n)/\log T(n))$ by precomputing and storing the result of the computations of M for $\Theta(\log T(n))$ steps starting from every possible configuration. Then, M can be simulated $\Theta(\log T(n))$ steps at a time by successive lookups to the precomputed table. A similar technique was used by Dymond and Tompa [1983] to show that a Turing machine running in $T(n)$ time can be simulated by a PRAM in time $O(\sqrt{T(n)})$ using an exponential number of processors and memory addressing on words of size $O(\sqrt{T(n)})$. We describe a simulation by a

[4]See Section 5.3.4 for a brief introduction to the Four Russians technique in the context of dynamic programming.

multi-core computer that combines both techniques (table lookups and parallelism), and argue about its limitations based on realistic assumptions about the number of processors as well as word and memory sizes.

Outline Let M be a single-tape deterministic Turing machine[5] that performs $T(n)$ computation steps on an input of length n (and hence it always halts). Assume that M's alphabet is binary. The general idea of the simulation is to treat a block of contiguous bits of M's tape as a word in RAM. By precomputing M's resulting configuration after b steps when starting with each possible block, we can then simulate b steps of M at a time by successively looking up the next configuration of M from the precomputed table. Let $g(n)$ denote the precomputation time. If each access to the precomputed table takes constant time, then the total time of the simulation is $T_p(n) = \frac{T(n)}{b(n)} + g(n)$.

Precomputation phase Since in b steps M can only alter the contents of b cells, for a given position within the tape we need only to consider the content of the b cells to the left and b cells to the right of the current position in order to compute the resulting configuration after b steps. A block configuration of M is a tuple (s, B), where s is a state, and B is a $(2b+1)$-bit string representing the contents of a segment of M's tape around some position of the head. For each possible block configuration c, we store in $A[c]$ the resulting configuration when running M starting from c (i.e., the new state and block contents), plus information about how many positions the head moved and in which direction. The latter is necessary to determine where the new block should be centered in M's tape. A block configuration c uses $|c| = 2b + 1 + d \leq kb$ bits, where d is the constant number of bits required to indicate a state of M and k is a constant. Then there are at most 2^{kb} starting block configurations. The resulting configuration after b steps can be computed by direct simulation of M. Since these blocks can be computed independently in parallel for all possible starting configurations, preprocessing takes $g(n) = 2^{kb}b/p$ steps using p processors. Note that the precomputation requires M's specification but is independent of a particular input.

Simulation phase Suppose that the configuration table A has already been computed and it is stored in RAM on the multi-core machine. If the length of each configuration is smaller than the machine's word length, then A can be indexed by each configuration and each entry can be accessed in constant time. In this case A can be stored as an array of configurations, indexed by initial configuration. Given M and an input x, and starting with the initial configuration c_0, the multi-core simulates M (using one processor) by applying $c_{i+1} = A[c_i]$, and updating the contents of M's tape at each step, until an accepting or rejecting configuration is reached. Thus,

[5] Although it is straightforward to extend the simulation to a k-tape Turing machine, for simplicity we consider the single-tape case, which serves our purpose of discussing the limits in the parallelism offered by the approach.

the simulation completes M's computation in $T(n)/b$ table lookups. The total number of steps of the simulation, including preprocessing, is

$$T_p(n) = \frac{T(n)}{b} + \frac{2^{kb}b}{p}.$$

Restrictions on parameters There are natural restrictions that limit the speedup that can be achieved with the above technique: the word size, the size of table A, and the efficiency in terms of processor use. It is usual to regard memory as a polynomial in the input size. The number of configurations is 2^{kb} and hence A requires that many words of memory. This implies that, for a memory of size n^r, for some r, $b \leq (r/k)\log n = O(\log n)$. Moreover, in order to be able to access entries of A in constant time using block configurations as addresses, we require $bk \leq w$, where w is the word size. This is consistent with the common assumption $w = \Theta(\log n)$ (see Section 4.2.2). Furthermore, assume that in order to enable larger speedups we allow $b = \omega(\log n)$ and allow a table of superpolynomial size. Then, in order for the simulation time to dominate over preprocessing we would require $2^{kb}b/p = O(T(n)/b)$, and thus $p \geq n^{\omega(1)}/T(n)$, which would be prohibitive for any polynomial time $T(n)$.

Note that the parallelism exploited by this approach is both in terms of the parallel computation of the table A with multiple processors and in terms of the ability to manipulate various bits simultaneously to compute a result through a constant time table lookup. As we argue above, the second form of parallelism can only be exploited up to the manipulation of $\Theta(\log n)$ bits. The use of various processors is only for the precomputation phase, which is embarrassingly parallel. Thus, in principle, we could benefit from the use of a polynomial number of processors for this approach. In fact, the larger the number of processors, the larger $T(n)$ can be while still having the simulation phase dominating the total time. However, the maximum speedup factor to which the approach can lead is the size of the block b. Hence, for optimal processor utilization, the maximum number of processors that we can use is $p = b$. In this case we have that total time is $T_p(n) = O(T(n)/p + 2^{kp}p/p)$. For the simulation time to dominate, we then require $2^{kp} = O(T(n)/p)$, and thus $p = O(\log(T(n)))$. For any polynomial time Turing machine, this implies $p = O(\log n)$.

We note that the arguments in this section do not preclude the existence of other approaches that could result in optimal simulation times without the restrictions described above.

Faster recursive precomputation Finally, as a side note, we present an improvement to the precomputation phase that leads to a smaller precomputation time and hence to a wider range of values of $T(n)$ for which the simulation time dominates. We achieve this by recursively applying the same simulation on the computation of each entry of A. Let g_i and b_i denote the precomputation time and block length of the i-th level simulation, respectively. Thus $g_m = g(n)$ is

the total precomputation time and $b_m = b$ is the block length of the final simulation as described above. Since the computation of each entry of A can now be sped up by b_{m-1}, we have

$$g_m = \frac{2^{kb_m}}{p}\frac{b_m}{b_{m-1}} + g_{m-1},$$

where k is a constant such that for all i, a configuration in level i has size at most kb_i. We then set $b_{m-i} = \frac{b}{2^i}$ for all $0 \leq i \leq m = \log b$ Then, $b_{m-i}/b_{m-i-1} = 2$, which is the length of the critical path at each recursive level. Then,

$$g_{m-i} = \max\left\{\frac{2^{kb_{m-i}+1}}{p}, 2\right\} + g_{m-i-1} = 2 \cdot \max\left\{\frac{2^{(2^{-i}kb)}}{p}, 1\right\} + g_{m-i-1}$$

Note that $2^{2^{-i}bk}/p \leq 1$ for $i \geq \log b + \log k - \log\log p$. Let $i^\star = \log b + \log k - \log\log p$. Since $g_0 = 0$,

$$\begin{aligned}
g_m &= \sum_{i=0}^{m-1} g_{m-i} - g_{m-i-1} \\
&= \frac{2}{p}\sum_{i=0}^{i^\star} 2^{(2^{-i}kb)} + \sum_{i=i^\star+1}^{m-1} 2 \\
&\leq \frac{2}{p}(2^{kb} + i^\star 2^{kb/2}) + 2(\log\log p - \log k - 1) \\
&\leq \frac{2^{kb+2}}{p} + 2\log\log p
\end{aligned}$$

Therefore, the total simulation time is now $T_p(n) = O\left(\frac{T(n)}{b} + \frac{2^{kb}}{p} + \log\log p\right)$. For example, if $b = (\log n)/k$, and $p = \log n$, the time is $T_p(n) = O\left(\frac{T(n)}{\log n} + \frac{n}{\log n}\right)$. Without the improvement, we would require $T(n) \geq n\log n$ in order for the simulation to dominate, while now $T(n) \geq n$ is enough.

4.2.10 Amdahl's Law

Consider a program whose execution has a serial part that cannot be parallelized (unless $\mathsf{P} = \mathsf{NC}$) represented by $S(n)$ and a fully parallelizable part denoted by $P(n)$. Then the parallel time with

p processors is: $T_p(n) = S(n) + P(n)/p$ and the speedup is represented by

$$\frac{T_1(n)}{T_p(n)} = \frac{S(n) + P(n)}{S(n) + P(n)/p}.$$

Observe now that for $p = \Theta(n)$ we get that the parallel program is noticeably faster only if $S(n) = O(P(n)/n)$. For $p = \Theta(n^\alpha)$ we get that the parallel program is noticeably faster only if $S(n) = O(P(n)/n^\alpha)$. Lastly, for $p = \Theta(\log n)$ we get that the parallel program is noticeable faster if $S(n) = O(P(n)/\log n)$. Observe that most practical algorithms on large data sets run in time $O(n \log n)$ or less, with the sequential part often corresponding to I/O operations, i.e., reading the input. This means that the likeliest value for which one can obtain optimal speedup corresponds to $p = P(n)/S(n)$ which is often (though not always) $\log n$.

4.3 Conclusions

We presented a list of theoretical arguments and practical evidence as to the existence of a qualitative difference between the classes of problems that can be sped up with a sublinear number of processors and those that can be sped up with polynomially many processors.

We also showed that in various specific instances, even though there are optimal algorithms for either case, it is conceptually and practically much simpler to design an algorithm for a sublinear number of processors. The benefits of a low processor count extend to issues of processor communication, buffering, memory access, and cache bounds.

We introduced classes that describe the problems that allow for optimal speed up, up to constant factors, for logarithmic and sublinear number of processors and show that they contain a strictly larger class of problems that the PRAM equivalents introduced by Kruskal, Rudolph, and Snir in 1990 [Kruskal et al., 1990], unless $\mathsf{NC} = \mathsf{P}$.

The discontinuities identified in behaviour and performance of parallel systems for logarithmic and sublinear number of processors make these particular processor count functions theoretically interesting, practically relevant, and worth of further exploration.

Chapter 5

Algorithms in the Ultra-Wide Word Model

In this chapter we introduce the Ultra-Wide Word architecture and model of computation, an alternate view of parallelism for a modern architecture in the form of an ultra-wide word processor. This can be implemented by replacing one or more cores of a multi-core chip with a very wide word Arithmetic Logic Unit (ALU) that can perform operations on a very large number of bits in parallel.

The idea of executing operations on a large number of bits simultaneously has been successfully exploited in different forms. In Very Long Instruction Word (VLIW) architectures [Fisher, 1983], several instructions can be encoded in one wide word and executed in one single parallel instruction. Vector processors allow execution of one instruction on multiple elements simultaneously, implementing Single-Instruction-Multiple-Data (SIMD) parallelism. This form of parallelism led to the design of supercomputers such as the Cray architecture family [Russell, 1978], and is now present in Graphics Processing Units (GPUs) as well as in Streaming SIMD Extensions (SSE) extensions to scalar processors.

As CPU hardware advances, so does the model used in theory to analyze it. The increase in word size was reflected in the word-RAM model in which algorithm performance is given as a function of the input size n and the word size w, with the common assumption that $w = \Theta(\log n)$. In its simplest version, the word-RAM model allows the same operations of the traditional RAM model. Algorithms in this model take advantage of bit-level parallelism through packing various elements in one word and operating on them simultaneously (see Section 2.9). Although similar to vector processing, the word-RAM provides more flexibility in that the layout of data in a word depends on the algorithm, and data elements can be packed in an arbitrary way. Unlike VLIW architectures, the Ultra-Wide Word model we propose is not concerned with the compiler

identifying operations which can be done in parallel but rather with achieving large speedups in implementations of word-RAM algorithms through operations on thousands of bits in parallel.

As multi-core chip designs evolve, chip vendors try to determine the best way to use the available area on the chip, and the options traditionally are an increased number of cores or larger caches. We believe that the current stage in processor design allows for the inclusion of an architecture such as the one we propose. In addition, ease of programming is a major hurdle to the eventual success of parallel and multi-core architectures. In contrast, bit parallelism as exploited by the word-RAM model does not suffer from this drawback: there is a large selection of word-RAM algorithms (see, e.g., [Andersson and Thorup, 2007; Han, 2004; Hagerup, 1998; Chan, 2006]) that readily benefit from bit parallelism without having to deal with the more difficult aspects of concurrency such as mutual exclusion, synchronization and resource contention. In this sense, the advantage of an on-chip ultra-wide word architecture is that it would enable word-RAM algorithms to achieve speedups comparable to those of multi-threaded computations, while at the same time keeping the simplicity of sequential programming that is inherent to the RAM model. We argue that this is the case by showing several examples of implementations of word-RAM algorithms using the wide word, usually with simple modifications to existing algorithms, and extending the ideas and techniques from the word-RAM model. We also show how the Ultra-Wide architecture can be used to simulate a non-standard memory layout, which has been used to sidestep known lower bounds in important data structure problems [Brodnik et al., 2005, 2006].

In terms of the actual architecture, we envision the Ultra-Wide ALU together with multi-cores on the same chip. Thus, the Ultra-Wide architecture adds to the computing power of current architectures. The results we present in this chapter, however, do not use multi-core parallelism.

Our results[1] We introduce the Ultra-Wide Word architecture and model, which extends the w-bit word-RAM model by adding an ALU that operates on w^2-bit words. We show that several broad classes of algorithms can be implemented in this model. In particular:

- We describe Ultra-Wide Word implementations of dynamic programming algorithms for the subset sum problem, the knapsack problem, the longest common subsequence problem, as well as many generalizations of these problems. Each of these algorithms illustrates a different technique (or combination of techniques) for translating an implementation of an algorithm in the word-RAM model to the Ultra-Wide Word model. In all these cases we obtain a w-fold speedup over word-RAM algorithms.
- We also describe Ultra-Wide Word implementations of popular string searching algorithms: the Shift-And/Shift-Or algorithms [Baeza-Yates and Gonnet, 1992; Wu and Manber, 1992]

[1] Results in this chapter are joint work with Arash Farzan, Alejandro López-Ortiz, and Patrick K. Nicholson.

and the Boyer-Moore-Horspool algorithm [Horspool, 1980]. Again, we obtain a w-fold speedup over the original algorithms.

- Finally, we show that the Ultra-Wide Word model is powerful enough to simulate a non-standard memory architecture in which bytes can overlap, which we shall call FS-RAM [Fredman and Saks, 1989]. This allows us to implement data structures and algorithms that circumvent known lower bounds for the word-RAM model.

The rest of this chapter is organized as follows. In Section 5.1 we describe the Ultra-Wide architecture and model of computation. We show in Sections 5.2 how to simulate the FS-RAM memory architecture. In Sections 5.3 and 5.4 we show examples of UW-RAM implementations of algorithms for dynamic programming and string searching. We present concluding remarks in Section 5.5.

5.1 The Ultra-Wide Word-RAM Model

The Ultra-Wide word-RAM model (UW-RAM) we propose is an extension of the word-RAM model. The word-RAM model is a variant of the RAM model in which a word has length w bits, and the contents of memory are integers in the range $\{0, \ldots, 2^{w-1}\}$ [Hagerup, 1998]. This implies that $w \geq \log n$, where n is the size of the input, and a common assumption is $w = \Theta(\log n)$ (see Section 4.2.2). Algorithms in this model take advantage of the intrinsic parallelism of operations on w-bit long words. We provide a more detailed introduction to the word-RAM model in Section 2.9.

The UW-RAM extends the word-RAM model by introducing an Ultra-Wide ALU with w^2-bit *wide words*, where w is number of bits in a word-RAM model. The Ultra-Wide Arithmetic Logic Unit (ALU) supports the basic operations available in a word-RAM model with multiplication on the entire word at once. Thus, the supported operations are: addition, subtraction, left and right shift, bitwise boolean operations, and multiplication. In principle, we allow multiplication, although the results of this chapter require only two multiplications by constants, which can be replaced by straightforward AC^0 operations. The model maintains the standard w-bit ALU as well as w-bit memory addressing. In general, we will use the parameter w for the word size in the description and analysis of algorithms, although in some cases we explicitly assume $w = \Theta(\log n)$. In terms of real world parameters, the wide word in the Ultra-Wide ALU would presently have between 1,000 and 10,000 bits and could increase even further in the future.

In reality, the addition of an ALU that supports operations on thousands on bits would require appropriate adjustments to the data and instruction caches of a processor as well as to the instruction pipeline implementation. Similarly to the abstractions made by the RAM and word-RAM models, the UW-RAM model ignores the effects of these and other architectural

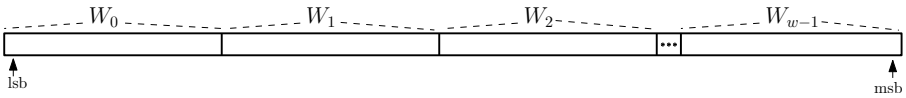


Figure 5.1: A wide word in the Ultra-Wide Word architecture. The wide word is divided in w blocks of w bits each, shown here in increasing number of block from left to right.

features and assumes that the execution of instructions on ultra-wide words is as efficient as the execution of operations on regular w-bit words, up to constant factors.

Provided that the UW-RAM supports the same operations as the word-RAM, the techniques to achieve bit-level parallelism in the word-RAM extend directly to the UW-RAM. However, since the word-RAM assumes that a word can be read from memory in constant time, many operations in word-RAM algorithms can be implemented through table lookups. For example, counting the number of one bits in a word of $w = \log n$ bits can be implemented through two table lookups to a precomputed table that stores the number of set bits for each number of $\log n/2$ bits. The space used by the table is $\sqrt{n}$ words. We cannot expect to achieve the same constant time lookup operation with words of w^2 bits since the size of the lookup tables would be prohibitive. However, the memory access operations of our model allow for the implementation of simultaneous table lookups of several w-bit words within a wide word, as we shall explain below.

Before describing the memory access operations supported by the UW-RAM, we introduce some notation. Let W denote a w^2-bit word. Let $W[i]$ denote the i-th bit of W, and let $W[i..j]$ denote the contiguous block of W from bit i to bit $j > i$, inclusive. The least significant bit of W is at $W[0]$, and thus $W = \sum_{i=0}^{w^2-1} W[i] \times 2^i$. For the sake of memory access operations, we divide W into w-bit blocks that can access different w-bit words in memory. Let W_j denote the j-th contiguous block of w bits in W, for $0 \leq j \leq w-1$, and let $W_j[i]$ denote the i-th bit within W_j. Thus, $W_j = W[jw..(j+1)w-1]$ and $W = \sum_{j=0}^{w-1} 2^j \times (\sum_{i=0}^{w-1} W_j[i] \times 2^i)$. The division of a wide word in blocks is solely intended for certain memory access operations, but basic operations of the model have no notion of block boundaries. Figure 5.1 shows a representation of a wide word, which depicts bits with increasing significance from left to right. Thus, shifts to the left (right) by i are equivalent to division (multiplication) by 2^i. In the description of operations with wide words we generally refer to variables with uppercase letters and to regular variables that use one w-bit word with lower case. In addition, we use $\vec{0}$ to denote a wide word with value 0. We use standard C-like notation for operations AND ('&'), OR ('|'), NOT ('$\sim$') and shifts ('<<','>>').

Memory access operations In this architecture, w (not necessarily contiguous) words from memory can be transferred into the w blocks of a wide word W in constant parallel time. These blocks can be written to memory in parallel as well. The memory access operations provided by the model that involve wide words are of three types: block, word, and content. Let us describe

Name	Input	Semantics
read_block	W, j, base	$W_j \leftarrow$MEM[base+j]
read_word	W, base	for all j in parallel: $W_j \leftarrow$MEM[base+j]
read_content	W, base	for all j in parallel: $W_j \leftarrow$MEM[base+W_j]
write_block	W, j, base	MEM[base+j]$\leftarrow W_j$
write_word	W, base	for all j in parallel: MEM[base+j]$\leftarrow W_j$
write_content	W, V, base	for all j in parallel: MEM[base+V_j]$\leftarrow W_j$

Table 5.1: Wide word memory access operations supported by the UW-RAM. In the above, MEM denotes regular RAM memory, which is indexed by addresses to words[2], and base is some base address.

read accesses (write accesses are analogous). A *block access* reads a single w-bit word from memory to a given block of a wide word. A *word access* reads w contiguous w-bit words from memory to an entire wide word in constant time. Finally, a *content access* uses the contents of a wide word W as addresses to read (possibly non-contiguous) words of memory simultaneously: for each block W_j within W, this operation reads from memory the w-bit word located at W_j (plus possibly a base address). The specifics of these operations are shown in Table 5.1.

Note that reading several (possibly non-contiguous) words from memory simultaneously is an assumption that is already made by any shared memory multiprocessing model. While, in reality, simultaneous access to all addresses in actual physical memory (e.g., DRAM) might not be possible, in shared memory systems, such as multi-core processors, the slowdown is mitigated by truly parallel access to private and shared caches, and thus the assumption is reasonable. We therefore follow this assumption in the same spirit. Furthermore, under the assumption that $w = \Theta(\log n)$, the UW-RAM supports the parallel memory access of $\Theta(\log n)$ words, which matches the amount of parallelism in memory accesses assumed in the multi-core models of this thesis (Chapters 3 and 7).

5.1.1 UW-RAM Subroutines

We now describe some operations that will be used throughout the UW-RAM implementations that we describe in later sections. A procedure we call *transpose* serves to bring together bits from all blocks into one block in constant time, while a procedure called *reverse transpose* is the inverse function. We will also use parallel comparators, a standard technique used in word-RAM algorithms, which we describe in Section 2.9.

[2]A more sophisticated version of the model could consider accessing half-words and individual bytes as well, which would contribute to space savings for some algorithms.

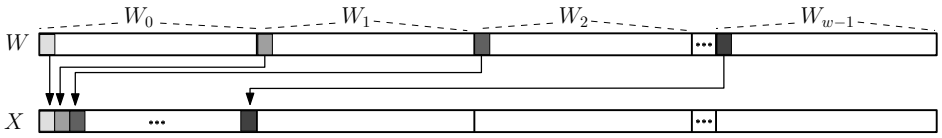


Figure 5.2: The *transpose* operation takes a wide word W whose set bits are restricted to the first bit of each block and compresses them to the first block of a wide word.

Algorithm 5.1 transpose(W)

$X \leftarrow W \times \frac{2^{w^2}-2^w}{2^{w-1}-1}$ {brings bits to contiguous positions in $X[(w-1)w+1..w^2]$}
$X \leftarrow (X << (w-1)w+1)$ & (2^w-1) {shifts bits to X_0 and cleans the rest of X}
return X

Transpose Let W be a wide word in which all bits are zero except possibly for the first bit of each block. The transpose operation copies the first bit of each block of a word W to the first block of a word X. I.e., if $X = \text{transpose}(W)$, then $X[j] \leftarrow W_j[0]$ for $0 \leq j < w$, and the rest of the bits of X are zero (see Figure 5.2).

The transpose operation can be implemented by using the *compression* operation described by Brodnik [1995, Algorithm 4.4]. This operation takes a (regular size) word x whose bits are all zeros but possibly for the bits in t blocks of k bits each that can have set bits (not necessarily equal across blocks). These blocks start at regular intervals of s bits[3]. The operation returns a word y with the contents of all blocks of x concatenated at the beginning of the word, and zeros in the rest of the word. More specifically, $y[i+j\cdot k] \leftarrow x[i+j\cdot s]$ for $0 \leq i < k$ and $0 \leq j < t$ [Brodnik, 1995]. This operation can be implemented in constant time by first multiplying x by the constant $c = \frac{2^{ts}-2^{tk}}{2^{s-k}-1}$ (which moves bits to contiguous positions in $x[(t-1)s+k..(t-1)s+tk+k-1]$), then shifting the result to the left by $(t-1)s+k$, and finally doing a bitwise AND with $2^{tk}-1$ [Brodnik, 1995][4].

In the case of the transpose operation we have $t = s = w$ and $k = 1$. The pseudocode for the transpose procedure is shown in Algorithm 5.1. Note that the three constants used in the procedure are fixed for any call to transpose, and thus they can be *hardwired* in the procedure. Note as well that the multiplication in the compression operation moves bits to $X[(w-1)w+1..w^2]$, with the last bit at position w^2. In order to accommodate this operation, we assume that the wide word has a constant number of extra bits after the (w^2-1)-th bit.

[3]This is called an (s,k)-sparse register in [Brodnik, 1995].
[4]The constant c is an integer whose derivation is described in [Brodnik, 1995].

Algorithm 5.2 reverse_transpose(W)

$C \leftarrow \frac{2^{w^2}-1}{2^w-1}$ $\{C_j = 1$ for all $j\}$
$X \leftarrow W \times C$ $\{X_j = W_0$ for all $j\}$
$D \leftarrow \frac{2^{(w+1)w}-1}{2^{w+1}-1}$ $\{D_j = 2^j$ for all $j\}$
$X \leftarrow X$ & D $\{X_j[j] = W[j]\}$
$M \leftarrow C >> (w-1)$ $\{M_j = 2^{w-1}$ for all $j\}$
$X' \leftarrow M - X$ {for all j, $X'_j[w-1] = 0 \Leftrightarrow X_j[j] = 1\}$
$X \leftarrow (\sim X'$ & $M) << (w-1)$ $\{X_j[0] = W[j]$ for all $j\}$
return X

Reverse Transpose This operation is the inverse of the transpose operation. It takes a word W whose set bits are all in the first block and spreads them across blocks of a word X so that $X_j[0] \leftarrow W[j]$ for $0 \leq j < w$. We implement this operation by modifying the spreading operation in [Brodnik, 1995, Algorithm 4.3] to avoid reversing the order of the bits, and to end with all bits at the beginning of each block. We first replicate the contents of the first block in all blocks and then extract the j-th bit of each block j. We then move this bit to the first bit of the block via four constant time operations. Algorithm 5.2 describes the details of this procedure. Again, the constants involved are fixed and can be hardwired in the procedure.

The *transpose* and *reverse transpose* operations require one multiplication by a constant each, and these are all the wide word multiplications that we use in our results. Our model allows multiplications on wide words in order to enable the implementation of a more general class of operations and algorithms. However, if a simpler underlying hardware design is desired, *transpose* and *reverse transpose* can also be implemented with straightforward circuits of constant depth. In this case, our model can be regarded as a restricted model with two non-standard AC^0 operations (see Section 2.9).

Comparators Many word-RAM algorithms perform operations on pairs of elements in parallel by packing these elements in *fields* within one word. In Section 2.9 we described how to implement fieldwise comparisons in constant time, a useful operation in word-RAM algorithms. This operation can be directly implemented on wide words in constant time. We shall assume that direct comparisons as well as operations that build on these (such as taking the fieldwise maximum between two words) are available and take constant time [Hagerup, 1998].

5.2 Simulation of FS-RAM

The FS-RAM is a model of computation first introduced by Fredman and Saks [1989] and further described by Brodnik [1995]. In the standard RAM model of computation, memory is organized

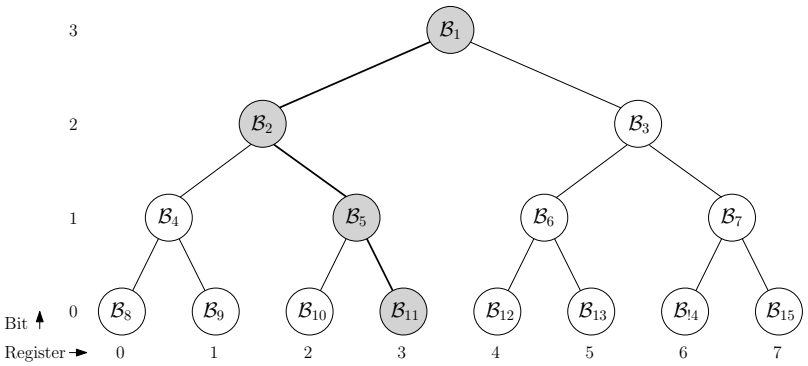


Figure 5.3: Yggdrasil memory layout: each node in a complete binary tree is an FS-RAM bit, and registers are defined as paths from a leaf to the root. For example, register 3 contains bits $\mathcal{B}_{11}, \mathcal{B}_5, \mathcal{B}_2$, and $\mathcal{B}_1$ (shaded nodes). Figure adapted from [Brodnik et al., 2005].

in registers or words, each word containing a set of bits. Any bit in a word belongs to that word only. In contrast, in the FS-RAM model words can overlap, that is, a single bit of memory can belong to several words[5]. The topology of the memory, i.e., a specification of which bits are contained in which words, defines a particular variant of the FS-RAM model. Variants of this model have been used to sidestep lower bounds for important data structure problems. For example, Brodnik et al. [2005] use a variant of FS-RAM called *Yggdrasil* in a data structure that achieves constant time for the operations insert, delete, membership, min, max, deletemin, deletemax, predecessor, and successor over a bounded universe of integers (known as the *discrete extended priority queue problem* [van Emde Boas et al., 1977]). In the Yggdrasil variant of FS-RAM, words in memory are organized as paths from leaves to the root in a complete binary tree. Thus, bits might belong to several paths. Figure 5.3 shows an example of this FS-RAM layout.

We show how the UW-RAM can be used to implement memory access operations for any given FS-RAM of word size at most w bits in constant time. Thus, the time bounds of any algorithm in the FS-RAM model carry directly to the UW-RAM.

5.2.1 Implementing FS-RAM Operations in the UW-RAM

Let $\mathcal{B}_1, \ldots, \mathcal{B}_B$ denote the bits of FS-RAM memory. A particular FS-RAM memory layout can be defined by its *appearance sets*, this is, the locations of each bit $\mathcal{B}_i$ in the FS-RAM memory [Brodnik, 1995]. For example, in the Yggdrasil model depicted in Figure 5.3, the appearance set of $\mathcal{B}_1$ is $\{\text{reg}[i].\text{bit}[3] | i = 0, \ldots, 7\}$, the one of $\mathcal{B}_4$ is {reg[0].bit[1],reg[1].bit[1]}, and the one

[5]The original name chosen by Fredman and Saks is Random Access Machine with Byte Overlap (RAMBO), reflecting the nature of the memory model.

Algorithm 5.3 fs-ram_read(t)

1: read_word($W, \mathcal{R}[t]$) $\{W_j \leftarrow \mathcal{R}[t, j]\}$
2: read_content(W, A) $\{W_j \leftarrow A[\mathcal{R}[t, j]]\}$
3: W $\leftarrow$ transpose(W)
4: write_block($W, 0, \&ret$) $\{ret \leftarrow W_0\}$
5: **return** ret

of $\mathcal{B}_{12}$ is {reg[4].bit[0]}. Equivalently, the layout can be specified by the registers and the bits contained in them. In the example above, reg[0]=$\mathcal{B}_8\mathcal{B}_4\mathcal{B}_2\mathcal{B}_1$, and in general reg[$i$].bit[$j$]= $\mathcal{B}_k$, where $k = \lfloor i/2^j \rfloor + 2^{m-j-1}$ ($m = 4$ in the example) [Brodnik et al., 2005].

In order to implement memory access operations on a given FS-RAM using the UW-RAM, we need to represent the memory layout of FS-RAM in standard RAM. We assume that the FS-RAM memory layout is given as a table $\mathcal{R}$ that stores, for each register and bit within the register, the number of the corresponding FS-RAM bit. Thus, if reg[i].bit[j]= $\mathcal{B}_k$, for some k, then $\mathcal{R}[i, j] = k$. This is without loss of generality, as this representation can be easily precomputed from the appearance sets. We assume that $\mathcal{R}$ is stored in row major order.

Given an FS-RAM memory of r registers of $b \leq w$ bits each and $B \leq br$ distinct appearance sets, we want to store its contents in RAM. For this, we simply store each bit $\mathcal{B}_i$ in a different word. Thus, $A[i]$ stores the value of $\mathcal{B}_i$, where A is an array of integers in RAM. The total space used by this representation is then Bw bits, where w is the number of bits in a RAM word. Naturally, we could store more than one bit in each word of A; however, this representation allows us to avoid concurrent writes to a same word.

Given an index t of a register of an FS-RAM represented by $\mathcal{R}$, we can read the values of each bit of reg[t] from RAM and return the b bits in a word. Doing this sequentially for each bit might take $O(b)$ time. Using the wide word we can take advantage of parallel reading and the transpose operation to retrieve the contents of reg[t] in constant time. Let reg[t]= $\mathcal{B}_{i_0} \ldots \mathcal{B}_{i_{b-1}}$. The read operation first reads the value of a bit $\mathcal{B}_{i_j}$ into block W_j of W by assigning $W_j = A[\mathcal{R}[t, j]]$. The second step consists of one transpose operation, after which the b bits are stored in W_0. Algorithm 5.3 shows the read operation, which takes constant time.

In order to implement the write operation reg[t]= $\mathcal{B}_{i_0} \ldots \mathcal{B}_{i_{b-1}}$ of FS-RAM, we first set $W_0 = \mathcal{B}_{i_0} \ldots \mathcal{B}_{i_{b-1}}$ and perform a reverse transpose operation to place each bit $\mathcal{B}_j$ in block W_j. We then write the contents of each W_j in $A[\mathcal{R}[t, j]]$. Algorithm 5.4 shows this operation, which takes constant time as well.

Since the read and write operations described above are sufficient to implement any operation that uses FS-RAM memory (any other operation is implemented in RAM), we have the following result:

Theorem 5.1 *Let $\mathcal{R}$ be any FS-RAM memory layout of r registers of at most b bits each and B*

Algorithm 5.4 fs-ram_write($t, \mathcal{B} = \mathcal{B}_{i_0} \ldots \mathcal{B}_{i_{b-1}}$)

1: read_block($W, 0, \mathcal{B}$) $\{W_0 \leftarrow \mathcal{B}\}$
2: W $\leftarrow$ reverse_transpose(W)
3: read_word($V, \mathcal{R}[t]$) $\{V_j \leftarrow \mathcal{R}[t, j]\}$
4: write_content(W, V, A) $\{A[\mathcal{R}[t, j]] \leftarrow W_j\}$

distinct appearance sets, with $b \leq w$ and $\log B \leq w$. Let A be any FS-RAM algorithm that uses $\mathcal{R}$ and runs in time T. Algorithm A can be implemented in the UW-RAM to run in time $O(T)$, using $rb + B$ additional words of RAM.

Proof: Table $\mathcal{R}$ indicating the FS-RAM bit identifier for each register and bit within register can be stored in rb words of RAM, while the values of each bit can be stored in B words of RAM. Since both fs-ram_read and fs-ram_write are constant time operations, any t-time operation that uses FS-RAM memory can be implemented in UW-RAM in the same time t. In the case that $\mathcal{R}$ is not given, it can be computed from the appearance sets in $O(rb)$ time. This translation from appearance sets to $\mathcal{R}$ is fixed for the same FS-RAM layout and needs only to be precomputed once. ∎

By Theorem 5.1 we can implement any arbitrary FS-RAM memory layout and word-RAM algorithm with a moderate space overhead. Note that since any FS-RAM implementation requires at least B bits of FS-RAM memory, the relative overhead in space is reduced. The space overheads above are stated for a generic implementation of FS-RAM. However, for particular FS-RAM memory layouts one can save space by storing more than one FS-RAM bit per RAM register, or by replacing table $\mathcal{R}$ with a constant time calculation of appearance set indices from the indices of FS-RAM registers and bits within registers (and adjusting fs-ram_read and fs-ram_write appropriately).

5.2.2 Constant Time Priority Queue

Brodnik et al. [2005] use the Yggdrasil FS-RAM memory layout to implement priority queue operations in constant time using $3M$ bits of space ($2M$ of ordinary memory and M of FS-RAM memory), where M is the size of the universe. This problem has non-constant lower bound for several models, including an $\Omega\left(\min\left(\frac{\lg\lg M}{\lg\lg\lg M}, \sqrt{\frac{\lg N}{\lg\lg N}}\right)\right)$ lower bound in the RAM model when the memory is restricted to $N^{O(1)}$, where N is the number of elements in the set to be maintained [Beame and Fich, 2002].

For a universe of size $M = 2^m$, for some m, the Yggdrasil FS-RAM layout consists of $r = M/2$ registers of $b = \log M$ bits each, and $B = M - 1$ distinct appearance sets (see Figure 5.3 for an example with $M = 16$). Thus, applying Theorem 5.1 we obtain the following result:

Corollary 5.1 *The discrete extended priority queue problem can be solved in the UW-RAM model in $O(1)$ worst case time per operation using $2M + ((M+1)/2)\log Mw + (M-1)w$ bits, and thus in $O(M \log M)$ words of RAM.*

5.2.3 Constant Time Dynamic Prefix Sums

Brodnik et al. [2006] use a modified version of the Yggdrasil FS-RAM to solve the dynamic prefix sums problem in constant time. The dynamic prefix sums problem consists of maintaining an array A of size N, supporting the operations $update(j, d)$ which sets $A[j]$ to $A[j] \oplus d$, and $retrieve(j)$, which returns $\oplus_{i=0}^{j} A[i]$ [Fredman, 1982; Brodnik et al., 2006], where $\oplus$ is any associative binary operation. This FS-RAM implementation sidesteps various lower bounds for dynamic prefix sums on different models: there is an $\Omega(\log N)$ algebraic complexity lower bound [Fredman, 1982] as well as under the semi-group model of computation [Hampapuram and Fredman, 1998], and a $\Omega(\log N / \log \log N)$ information-theoretic lower bound [Fredman, 1982].

The result of Brodnik et al. [2006] uses a complete binary tree on top of array A as leaves. The tree is similar to the one used in the priority queue problem, but it differs in that only internal nodes store any information, and that there are $m = \lceil \log M \rceil$ bits stored in each node, where M is the size of the universe. This tree is stored in a variant of the Yggdrasil memory called m-Yggdrasil, in which each register correspond again to a path from a leaf to the root, but this time each node stores not only one bit but the m bits containing the sum of all leaves in the left subtree of that node [Brodnik et al., 2006]. It is assumed that $nm \leq w$, where $n = \lceil \log N \rceil$ and w is the size of the word in bits. Thus, an entire path from leaf to root fits in a word and can be accessed in constant time. An update or retrieve operation consists of retrieving the values along a path in the tree and processing them in constant time using bit-parallelism and table lookup operations. The space used by the lookup table can be reduced at the expense of an increased time for the retrieve operation. In general, both operations can be supported in time $O(\iota + 1)$ with $(N-1)m$ bits of m-Yggdrasil memory and $O(M^{n/2^\iota} \cdot m + m)$ bits of RAM [Brodnik et al., 2006].

In order to represent the m-Yggdrasil memory in our model, we treat each bit of a node in the tree as a separate FS-RAM bit. Thus, the FS-RAM memory has $r = N$ registers of $b = nm$ bits each, and there are $B = (N-1)m$ distinct bits to be stored. Hence, by Theorem 5.1 we have:

Corollary 5.2 *The operations update and retrieve of the dynamic prefix sums problem can be supported in the UW-RAM model in $O(\iota + 1)$ time with $O(M^{n/2^\iota} \cdot m + Nmnw)$ bits of RAM. For constant time operations ($\iota = 1$) the space is dominated by the first term, i.e., the space is $O(M^{\sqrt{\log N}})$ bits. For $\iota = \log \log N$, the time is $O(\log \log N)$ and the space is $O(Nmnw)$ bits.*

5.3 Dynamic Programming

In this section we show how to speed up various dynamic programming algorithms in the UW-RAM. We show that an existing word-RAM algorithm for the subset sum problem can be directly translated to the UW-RAM. We also show how to adapt an existing algorithm for the knapsack problem. We note that these problems have many generalizations that can be solved using the same techniques. Based on similar techniques, we describe a word-RAM algorithm (and UW-RAM implementation) for the longest common subsequence (LCS) problem. The implementation for subset sum as well as the first solution to LCS are examples of pure bit parallelism, while the knapsack implementation and the second algorithm for LCS use the parallel lookup power of the UW-RAM.

5.3.1 Subset Sum

Given a set $S = \{x_1, x_2, \ldots, x_n\}$ of nonnegative integers (weights) and an integer t (capacity), the subset sum problem is to find $S' \subseteq S$ such that $\sum_{a_i \in S'} a_i = t$. The optimization version asks for the solution of maximum weight which does not exceed t [Cormen et al., 2001]. This problem is NP-hard, but it can solved in pseudopolynomial time via dynamic programming in $O(nt)$ time, using the following recursion by Bellman [1957]: for each $0 \leq i \leq n$ and $0 \leq j \leq t$, $C_{i,j}$ is true if there is a subset of elements $\{a_1, \ldots, a_i\}$ that adds up to j. Thus, $C_{0,0}$ is true, $C_{0,j}$ is false for all $j > 0$, and value $C_{i,j}$ is true if $C_{i-1,j}$ is true or $C_{i-1,j-a_i}$ is true ($C_{i,j}$ is false for any $j < 0$). The problem admits a solution if $C_{n,t}$ is true.

Pisinger [2003] gives an algorithm that implements this recursion in the word-RAM model with word size w by representing up to w values of a row of C. Using bit parallelism, w bits of a row can be updated simultaneously in constant time from the values of the previous row: row C_i is updated by computing $C_i = (C_{i-1} \mid (C_{i-1} >> a_i))$ (which might require shifting words containing C_{i-1} first by $\lfloor a_i/w \rfloor$ words and then by $a_i - \lfloor a_i/w \rfloor$) [Pisinger, 2003]. Assuming $w = \Theta(\log t)$, this approach leads to an $O(nt/\log t)$ time solution in $O(t \log t)$ space. The actual values in S' that compose the solution can be then recovered with the same space and time bounds with a recursive technique by Pferschy [1999].

Pisinger's algorithm can be implemented directly in the UW-RAM: entries of a row C_i are stored contiguously in memory; thus, we can load and operate on w^2 bits simultaneously when updating each row. Hence, UW-RAM implementation runs in $O(nt/\log^2 t)$ time using the same $O(t \log t)$ space.

5.3.2 Knapsack

Given a set S of n elements with weights and values, the knapsack problem asks for a subset of S of maximum value such that the total weight is below a given capacity bound b. Let $S = \{(w_i, v_i)\}_{i=1}^{n}$ where w_i and v_i are the weight and value of the i-th element. Just like the subset sum problem, this problem is NP-hard but can be solved in pseudopolynomial time using the following recurrence by Bellman [1957]. Let $C_{i,j}$ be the maximum value of a solution containing elements in the subset $S_i = \{(w_k, v_k)\}_{k=1}^{i}$ with maximum capacity j. Then, $C_{0,j} = 0$ for all $0 \leq j \leq b$, and $C_{i,j} = \max\{C_{i-1,j}, C_{i-1,j-w_i} + v_i\}$. The value of the optimal solution is $C_{n,b}$. This leads to a dynamic program that runs in $O(nb)$ time.

The word-RAM algorithm by Pisinger [2003] represents partial solutions of the dynamic programming table with two binary tables g and h and operates on the $O(w)$ entries at a time. More specifically, $g_{i,w} = 1$ and $h_{i,v} = 1$ if and only if there is a solution with weight w and value v that is not dominated by another solution in $C_{i,*}$ (i.e., there is no entry $C_{i,w'}$ such that $w' < w$ and $C_{i,w'} \geq v$). Pisinger shows how to update each entry of g and h with a constant time procedure, which can be encoded as a constant size lookup table. By composing this table $\alpha = w/10$ times, α entries of the tables can be computed in constant time, so an entire row can be computed in $O(m/w)$ time and $O(m/\log m)$ space, where m is the maximum of the capacity b and the value of the optimal solution[6]. The optimal solution can then be computed in $O(nm/w)$ time [Pisinger, 2003].

Compared to the subset sum algorithm, which relies mainly on bit-parallel operations, this word-RAM algorithm for knapsack relies on table precomputation and lookup to achieve a w-fold speedup. In this sense, the UW-RAM implementation of the knapsack algorithm is a good example of the parallel lookup power of the architecture. While we cannot precompute a composition of $\Theta(w^2)$ lookup tables to compute $\Theta(w^2)$ entries of g and h at a time, we can use the same tables with $\alpha = w/10$ as in Pisinger's algorithm and use the block of the wide word to make w simultaneous lookups to the table. Since the values in a row i of h and g depend only on row $i-1$ of these tables, then there are no dependencies between values in the same row.

One difficulty, however, is that in order to compute the values in row i in parallel we must first preprocess row $i-1$ in both h and g, such that we can return the number of one bits in both $g_{i-1,0}, ..., g_{i-1,j}$ and $h_{i-1,0}, ..., h_{i-1,j}$ in $O(1)$ time for any column $j \in \{0, m-1\}$. That is, the prefix sums of the one bits in the row $i-1$ up to column j. Note that since we are only required to compute the prefix sums of row $i-1$ one time, this is *not* the same as the dynamic problem described in Section 5.2.3, i.e., it is a static problem. Furthermore, since the algorithm is the same for both g and h, we describe the computation for g alone[7].

[6] This value is not known in advance, though an upper bound of at most twice the optimal value can be used [Pisinger, 2003; Dantzig, 1957].

[7] We note that it is possible to simulate the standard parallel prefix sums algorithm (for w processors in this

Static Prefix Sums: We divide g_{i-1} in blocks of w contiguous bits and compute the number of ones in each block $g_{i-1,k}, ..., g_{i-1,k+w-1}$ for $k \in \{0, w, 2w, ..., \lfloor m/w \rfloor w\}$ using a lookup table. We store the results in an array $\mathcal{A}$ of length $\lceil m/w \rceil$. Next, we compute the prefix sums $\mathcal{A}'$ of $\mathcal{A}$ in two steps. We divide $\mathcal{A}$ in sets of w consecutive entries. The first step is to compute the prefix sums of each set by computing

$$\mathcal{A}'[k] = \mathcal{A}[\lfloor k/w \rfloor w] + \mathcal{A}[\lfloor k/w \rfloor w + 1] + ... + \mathcal{A}[k] \ ,$$

for each $k \in \{0, ..., \lfloor m/w \rfloor\}$. Using the w blocks of a wide word, this can be done for w array indices at a time, $k_0, ..., k_{w-1}$, where $k_\ell = k' + \ell w$, and k' iterates over the sequence

$$0, 1, ..., w-1, w^2, w^2+1, ..., w^2+w-1, 2w^2, 2w^2+2, ... \ .$$

Note that we can compute all w array entries in constant time, for each k' in the previous sequence, since

$$\mathcal{A}'[k' + \ell w] = \begin{cases} \mathcal{A}'[k' + \ell w - 1] + \mathcal{A}[k' + \ell w] & \text{if } k \neq 0 \mod w, \\ \mathcal{A}[k' + \ell w] & \text{otherwise.} \end{cases}$$

The second step is to update each set of $\mathcal{A}'$ from left to right by adding to each entry in the set the last entry of the previous set. I.e., we set $\mathcal{A}'[k] = \mathcal{A}'[k] + \mathcal{A}'[\lfloor k/w \rfloor w - 1]$, for $k \in \{w, w+1, ..., \lfloor m/w \rfloor\}$. This can also be done for w values at once, $k_0, ..., k_{w-1}$, where $k_\ell = k' + \ell$, and $k' \in \{w, 2w, ..., \lfloor m/w \rfloor\}$. At this point, $\mathcal{A}'$ contains the prefix sums of $\mathcal{A}$, and took $O(|\mathcal{A}|/w) = O(m/w^2)$ time to compute, by exploiting the parallel read and write operations of the UW-RAM.

Let f be the number of ones in $g_{i-1,\lfloor j/w \rfloor}, ..., g_{i-1,j}$, which can be computed using the lookup table. To compute $g_{i-1,0}, ..., g_{i-1,j}$ we return $f + \mathcal{A}'[\lfloor j/w \rfloor]$. Since each row of g and h requires $O(m/w^2)$ to compute, and there are n rows, the total time to compute g and h (and thus to compute the optimal solution) on the UW-RAM is $O(nm/w^2)$. This achieves a w-fold speedup over Pisinger's word-RAM solution.

5.3.3 Generalizations of Subset Sum and Knapsack Problems

Pisinger [2003] uses the techniques of the word-RAM algorithm for subset sum and knapsack to obtain a word-RAM algorithm for computing a path in a layered network: given a graph $G = (V, E)$, a source $s \in V$ and a terminal $t \in V$, and a weight for each edge, is there a path of weight b from s to t? Again, this algorithm translates directly to a UW-RAM algorithm, thus yielding a w-fold speedup over the word-RAM algorithm. Pisinger further uses the algorithms for the above problems to implement word-RAM solutions for other generalizations of subset

case) [JáJá, 1992] using the UW-RAM, though we believe the algorithm described in what follows to be more straightforward.

sum and knapsack problems, such as: the bounded subset sum and knapsack problems (each element can be chosen a bounded number of times), the multiple choice subset sum and knapsack problems (the set of numbers is divided in classes and the target sum must be matched with one number of each class), the unbounded subset sum and knapsack problems (each element can be chosen an arbitrary number of times), the change-making problem, and, finally, the two-partition problem. UW-RAM implementations for all these generalizations are direct and yield a w-fold speedup over the word-RAM algorithms (recall that $w = \Omega(\log n)$).

5.3.4 Longest Common Subsequence

The final dynamic programming problem we examine is that of computing the Longest Common Subsequence (LCS) of two string sequences (see Definition 2.11 in Section 2.8.2). Let Σ be a finite alphabet of symbols, where $\sigma = |\Sigma|$. Given two sequences $X = x_1x_2 \dots x_m$ and $Y = y_1y_2 \dots y_n$, where $x_i, y_j \in \Sigma$, the LCS problem asks for a sequence $Z = z_1z_2 \dots z_k$ of maximum length such that Z is a subsequence of both X and Y. This problem can be solved via a classic dynamic programming algorithm in $O(nm)$ time [Cormen et al., 2001]. In what follows, we show how to combine techniques used for subset sum and knapsack, as well as the Four Russians technique, in order to achieve further speedups in the UW-RAM model. The first algorithm presented runs in $O(\frac{nm}{w^2} \log \sigma + m + n)$ time, while the second is more involved and runs in time $O(n^2 \log^2 \sigma/w^3 + n \log \sigma/w)$, assuming $m = n$ for simplicity.

Let $c_{i,j}$ denote the length of the LCS of $X[1..i] = x_1x_2 \dots x_i$ and $Y[1..j] = y_1y_2 \dots y_j$, then the following recurrence allows us to compute the length of the LCS of X and Y [Cormen et al., 2001]:

$$c_{i,j} = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ c_{i-1,j-1} + 1, & \text{if } x_i = y_j \\ \max\{c_{i,j-1}, c_{i-1,j}\}, & \text{otherwise} \end{cases} \tag{5.3.1}$$

The length of the LCS is $c_{m,n}$, which can be computed in $O(mn)$ time. Consider an $(m + 1) \times (n + 1)$ table C storing the values $c_{i,j}$. The idea of the UW-RAM algorithm is to compute various entries of this table in parallel. We assume $w = \Theta(\max\{\log n, \log m\})$.

Let d_k denote the values in the k-th diagonal of table C, this is $d_k = \{c_{i,j} | i + j = k\}$. Since a value in a cell $i, j > 0$ depends only on the values of cells $(i - 1, j)$, $(i - 1, j - 1)$ and $(i, j - 1)$, all values in the same diagonal d_k are independent of each other and can be computed in parallel. Thus, we use the wide word to compute various entries of a diagonal in constant time. Since each value in the cell might use up to $\min\{\log n, \log m\}$ bits, each value might use up to an entire block of the wide word (if $\log m = \Theta(\log n)$); thus, w cells can be computed in parallel. Since the total number of cells is $O(mn)$ and the critical path of the table is $m + n + 2$ cells, this

approach requires $O(mn/w + m + n)$ parallel time, resulting in a speedup of w. However, we can obtain better speedups by using fewer bits per entry of the table, which enables us to operate on more values in parallel. For this sake, instead of storing the actual values of the partial longest common subsequences, we store differences between consecutive values as described in [Masek and Paterson, 1980] for the related string edit distance problem.

Let V and H denote the tables of vertical and horizontal differences of values in C, respectively. Entries in these tables are defined as $V_{i,j} = c_{i,j} - c_{i-1,j}$ and $H_{i,j} = c_{i,j} - c_{i,j-1}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$. Figure 5.4 shows the tables C, V, and H for an example pair of input sequences. We adapt Corollary 1 in [Masek and Paterson, 1980] for the computation of V and H:

Proposition 5.1 *Let* $[x_i = y_j] = 1$ *if* $x_i = y_j$ *and 0 otherwise. Then,* $V_{i,j} = \max\{[x_i = y_j] - H_{i-1,j}, 0, V_{i,j-1} - H_{i-1,j}\}$ *and* $H_{i,j} = \max\{[x_i = y_j] - V_{i,j-1}, 0, H_{i-1,j} - V_{i,j-1}\}$.

Proof: Directly from Recurrence 5.3.1 we obtain $V_{i,j} = 1 - H_{i-1,j}$ if $x_i = y_j$ and $V_{i,j} = \max\{0, V_{i,j-1} - H_{i-1,j}\}$ otherwise. Similarly, $H_{i,j} = 1 - V_{i,j-1}$ if $x_i = y_j$ and $H_{i,j} = \max\{0, H_{i-1,j} - V_{i,j-1}\}$ otherwise. It is easy to verify from the definition of longest common subsequence and Recurrence 5.3.1 that $0 \leq H_{i,j} \leq 1$ and $0 \leq V_{i,j} \leq 1$ for all i, j, which implies that the maximum in $\max\{[x_i = y_j] - H_{i-1,j}, 0, V_{i,j-1} - H_{i-1,j}\}$ and $\max\{[x_i = y_j] - V_{i,j-1}, 0, H_{i-1,j} - V_{i,j-1}\}$ is equal to the first term if $x_i = y_j$ and to the second or third terms otherwise. ∎

We compute tables H and V according to Proposition 5.1 diagonal by diagonal using bit parallelism in the wide word. Assume an alphabet $\Sigma = \{0, 1, 2, \ldots, \sigma - 1\}$ with $\lceil \log \sigma \rceil \leq w - 1$. Although all entries in tables H and V are either 0 or 1, we will use fields of $O(\log \sigma)$ bits to store these values, since we can only compare at most $w^2 / \log \sigma$ symbols simultaneously in the wide word. We divide the wide word W in f-bit fields with $f = \max(\lceil \log \sigma \rceil, 2) + 1$. Each field will be used to store both symbols and intermediate results for the computation of the diagonals of H and V, plus an additional bit to serve as a test bit in order to implement fieldwise comparisons as described in Section 2.9. We require at least 3 bits because although all entries in tables H and V use one bit, intermediate results in calculations can result in values of -1. Thus, we require 2 bits to represent values -1, 0, and 1, and a test or sentinel bit to prevent carry bits resulting from subtractions to interfere with neighbouring fields. We represent -1 in two's complement. It is not hard to extend the techniques for comparisons and maxima to the case of positive and negative numbers [Hagerup, 1998].

Let H_k and V_k denote the k-th diagonal of H and V, respectively, i.e., $H_k = \{H_{i,j} | i + j = k\}$ and $V_k = \{V_{i,j} | i + j = k\}$. Consider table H. We will operate with each diagonal H_k using $\lceil |H_k| / \ell \rceil$ words, where $\ell = w^2 / f$. Let $f_0, \ldots, f_{\ell-1}$ denote the fields within W, in increasing order of bit significance within W. In each wide word, cells of H_k will be stored in increasing order of column, i.e., if $H_{i,j}$ is stored in field f_r, then f_{r+1} stores $H_{i-1,j+1}$. In order to compute each diagonal we must compare the relevant entries of strings X and Y. We assume that each symbol

of X and Y is stored using $\lceil \log \sigma \rceil + 1$ bits (including the test bit) and that X is stored in reverse order. X and Y can be preprocessed in $O(m+n)$ to arrange this representation, which will allow us to do constant-time parallel comparisons of symbols for each diagonal loading contiguous words of memory in wide words.

Consider a diagonal H_k. Assume that the entire diagonal fits in a word W. This will not be the case for most diagonals, but we describe the former case for simplicity. The latter case is implemented as a sequence of steps updating portions of the diagonal that fit in a wide word. We update the entries of H_k as follows:

1. We load the symbols of the relevant substrings of X and Y into words W_X and W_Y, with the substring of X in reverse order. More specifically, for a diagonal k, $W_Y = y_{j_1} y_{j_1+1} \dots y_{j_2}$, where $j_1 = k - \min(|X|, k-1)$ and $j_2 = \min(|Y|, k)$, and $W_X = x_{i_2} x_{i_2-1} \dots x_{i_1}$ with $i_2 = k - j_1$ and $i_1 = k - j_2$. We subtract W_Y from W_X, mask out all non-zero results and write a 1 in each field that resulted in 0. We store the resulting word in W_{eq}, where each field corresponding to a cell (i, j) stores a 1 if $x_i = y_j$ and a 0 otherwise (this can be implemented through comparisons as described in Section 2.9).

2. We load V_{k-1} into a word W_V and subtract it from W_{eq} to obtain $[a_i = b_j] - V_{i,j-1}$ for all i, j in H_k simultaneously and store the result in W_1.

3. We load H_{k-1} into a word W_H and subtract W_V to it to obtain $H_{i-1,j} - V_{i,j-1}$ for all i, j in H_k, storing the result in W_2.

4. Finally, using fieldwise comparisons, we obtain the fieldwise maximum of W_1, W_2 and the word $\vec{0}$. The resulting word is H_k.

All the operations described above can be implemented in constant time. The procedure to compute V_k is analogous. Note that the entries corresponding to base cases in the first row and column in the LCS table correspond to the base cases of the horizontal and vertical vectors, respectively. When computing diagonals H_k with $k \leq n+1$ and V_k with $k \leq m+1$, the entries corresponding to base cases are not computed from previous diagonals but should be added appropriately at the end of H_k and beginning of V_k.

Example 5.1 *Let $X = abbab$ and $Y = aabbba$ be two strings. Figure 5.4 shows the entries of the dynamic programming table for computing the LCS of X and Y, as well as the values of horizontal and vertical differences.*

In this example $\sigma = 2$, thus we use one bit for each symbol ('a'=0, 'b'=1), but we use $f = 3$ bits per field. Consider the diagonal H_6 in table H (in dark gray). We now illustrate how to obtain H_6 from H_5 and V_5 (in light gray). In what follows we represent the number in each field in decimal and do not include the details of fieldwise comparison and maxima.

		j	1	2	3	4	5	6
LCS			a	a	b	b	b	a
i		0	0	0	0	0	0	0
1	a	0	1	1	1	1	1	1
2	b	0	1	1	2	2	2	2
3	b	0	1	1	2	3	3	3
4	a	0	1	2	2	3	3	4
5	b	0	1	2	3	3	4	4

		j	1	2	3	4	5	6
H			a	a	b	b	b	a
i		0	0	0	0	0	0	0
1	a	1	1	0	0	0	0	0
2	b	1	0	1	0	0	0	0
3	b	1	0	1	1	0	0	0
4	a	1	1	0	1	0	1	0
5	b	1	1	1	0	1	0	0

		j	1	2	3	4	5	6
V			a	a	b	b	b	a
i								
1	a	0	1	1	1	1	1	1
2	b	0	0	0	1	1	1	1
3	b	0	0	0	0	1	1	1
4	a	0	0	1	0	0	0	1
5	b	0	0	0	1	0	1	0

Figure 5.4: Dynamic programming tables for the longest common subsequence and vector differences for $X = abbab$ and $Y = aabbba$.

W_X	=	1	0	1	1	0	$(=x_5x_4x_3x_2x_1)$
W_Y	=	0	0	1	1	1	$(=y_1y_2y_3y_4y_5)$
W_{eq}	=	0	1	1	1	0	$(W_{eq}[f \cdot (j-1)] = 1 \Leftrightarrow x_{\lvert H_5\rvert - j} = y_j)$
V_5	=	0	0	0	1	1	
$W_1 = W_{eq} - V_5$	=	0	0	1	0	-1	
H_5	=	1	0	1	0	0	
$W_2 = H_5 - V_5$	=	1	0	1	-1	-1	
$\max\{W_1, W_2, \vec{0}\}$	=	1	1	1	0	0	
H_6	=	1	1	1	0	0	0 (last 0 is the base case)

Once all diagonals are computed, the final length of the longest common subsequence of X and Y can be simply computed by (sequentially) adding the values of the last row of H or the values of last column of V (which can be done while computing H and V). The entire procedure is described in Algorithm 5.5 and leads to the following theorem.

Theorem 5.2 *Let Σ be an alphabet of size σ. Given two strings X and Y over Σ of lengths m and n, respectively, the length of the longest common subsequence of X and Y can be computed in the UW-RAM in $O(\frac{nm}{w^2}\log\sigma + m + n)$ time and $O(\min(n,m)w/\log\sigma)$ memory words in addition to the input.*

Proof: A diagonal of H and V of length ℓ entries can be computed in time $O(\ell\log\sigma/w^2 + 1)$. Adding this time over all $m+n$ diagonals yields the total time. For the space, each diagonal is represented in $\lceil \ell f/w^2 \rceil$ wide words, where $f = O(\log\sigma)$ is the number of bits per field. Since we can compute each diagonal H_k and V_k using only H_{k-1} and V_{k-1}, we only need to store 4 diagonals at any given time. Since the maximum length of a diagonal is $\min(n,m)+1$ and each wide word can be stored in w regular words of memory, the result follows. ∎

Algorithm 5.5 LCS-length$(X, Y, m = |X|, n = |Y|, \sigma)$

```
1: f ← max(⌈log σ⌉, 2) + 1 {field length in bits}
2: H_1^1 ← 0⃗ {H_{0,1} = 0}
3: V_1^1 ← 0⃗ {V_{1,0} = 0}
4: length ← 0 {length of longest common subsequence}
5: for k = 2 to m + n do
6:    ℓ ← min(n, k − 1) + min(m, k − 1) − k + 1 {length of diagonal}
7:    j_1 ← k − min(m, k − 1) {indices of relevant substrings of X and Y}
8:    j_2 ← min(n, k)
9:    i_2 ← k − j_1
10:   i_1 ← k − j_2
11:   j ← j_1
12:   i ← i_2
13:   s ← ⌈ℓf/w^2⌉ {number of wide words per diagonal}
14:   for t = 1 to s do
15:      j' ← min(j + s − 1, j_2)
16:      i' ← max(i + s − 1, i_1)
17:      W_Y ← Y[j..j']
18:      W_X ← X[i..i'] {substring of X is in reverse order}
19:      W_eq ←equal(W_X, W_Y)
20:      W_1 ← W_eq − V_{k−1}^t
21:      W_2 ← H_{k−1}^t − V_{k−1}^t
22:      H_k^t ← max(W_1, W_2, 0⃗) {base case is implicitly added at rightmost field}
23:      W_1 ← W_eq − H_{k−1}^t
24:      W_2 ← V_{k−1}^t − H_{k−1}^t
25:      V_k^t ← max(W_1, W_2, 0⃗)
26:      if t = 1 AND k ≤ m + 1 then
27:         V_k^t ← V_k^t >> f {add 0 in the first field for the base case}
28:      i ← i' + 1
29:      j ← j' + 1
30:      if t = 1 AND k ≥ m + 1 then
31:         length ← length + H_k^1[0..f − 1] {length = length + H_{m,k−m}}
32: return length
```

Recovering a Longest Common Subsequence

It is known that given a dynamic programming table storing the values of the LCS between strings X and Y, one can recover the actual subsequence by starting from $c_{m,n}$ and following the path through the cells corresponding to the values used when computing each value $c_{i,j}$ according to Recurrence (5.3.1): if $x_i = y_j$, then we add x_i to the LCS and continue with cell $(i-1, j-1)$; otherwise the path follows the cell corresponding to the maximum of $c_{i-1,j}$ or $c_{i,j-1}$. Although Algorithm 5.5 does not compute the actual LCS table, a path of an LCS can be easily computed

using tables H and V. The path starts at cell (m, n) (of either table). Then, to continue from a cell (i, j), if $x_i = y_j$, then x_i is part of the LCS, and we continue with cell $(i-1, j-1)$; otherwise, if $H_{i,j} = 1$ and $V_{i,j} = 0$, then we continue with cell $(i-1, j)$, and if $H_{i,j} = 0$ and $V_{i,j} = 1$, we continue with cell $(i, j-1)$ (and with any of the two if $H_{i,j} = V_{i,j} = 0$). This can be easily done in $O(m+n)$ time if all diagonals of tables V and H are kept in memory while computing the LCS length in Algorithm 5.5. This would require Algorithm 5.5 to use $O(nmw/\log\sigma)$ words of memory to store all diagonals.

Four Russians Technique

The computation of the longest common subsequence in the UW-RAM can be made even faster by combining the diagonal-by-diagonal order of computation described above with the Four Russians technique. The Four Russians technique [Arlazarov et al., 1970] was used by Masek and Paterson to speedup the computation of the string edit problem (and also the LCS) in a RAM with indirect addressing [Masek and Paterson, 1980]. The technique consists of dividing the dynamic programming table in blocks of size $t \times t$ cells. In a precomputation phase, all possible blocks are computed and stored as a data structure indexed by the first row and column of each block. The LCS can be then computed by looking up relevant values of the table one block at a time using the data structure. In a RAM with indirect addressing and under a suitable value of t, the last row and column of a block can be obtained by looking up the entry corresponding to the first row and column of that block in constant time. This technique yields a speedup of $O(t^2)$ with respect to computing all cells in the table, for a total time of $O(n^2/t^2)$ (for two strings of length n) plus the time for the precomputation of all blocks. By setting $t = O(\log n)$ and encoding the table with difference vectors, the precomputation time can be absorbed by the time to compute the main table (see [Masek and Paterson, 1980; Gusfield, 1997] for a more detailed description of the technique).

We can use the power of parallel memory accesses of the UW-RAM to speedup the computation of the LCS even further by looking up blocks in parallel, in a similar fashion to the diagonal-by-diagonal approach described above. For simplicity, assume $m = n$. Using the same encoding for H and V, we first precompute all possible blocks of H and V of size $t \times t$. Since a block is completely determined by its first column and row, whose values are in $\{0, 1\}$, and the two substrings of length t (over an alphabet of size σ), there are $O((2\sigma)^{2t})$ possible blocks. Note that we can encode each cell now with one bit, since we do not need to do symbol comparisons in parallel. Each block can be computed in $O(t^2)$ time with the standard sequential algorithm, so the precomputation time is $O((2\sigma)^{2t}t^2)$. We set $t = \log_{2\sigma} n/2$, and thus the precomputation time is $O(n\log^2 n)$ [Gusfield, 1997]. Since $t \leq w/2$, we can use each block of the wide word to lookup the entry for each block by using a parallel lookup operation. Thus, as described previously, we can compute tables H and V in diagonals of blocks, computing $\min(\ell, w)$ blocks simultaneously in a diagonal of length ℓ blocks. There are $(n/t)^2$ blocks to compute and the critical path of the

table has length n/t blocks. Therefore, the computation of H and V can be carried out in time $O(n^2/(t^2w) + n/t) = O(n^2 \log^2 \sigma/w^3 + n \log \sigma/w)$, since $t = \Theta(w/\log \sigma)$. We summarize this result in the following theorem:

Theorem 5.3 *Let Σ be an alphabet of size σ. Given two strings X and Y of length n over Σ, the length of the longest common subsequence of X and Y can be computed in the UW-RAM in $O(n^2 \log^2 \sigma/w^3 + n \log \sigma/w)$ time. For a constant alphabet size and $w = \Theta(\log n)$ this time is $O(n^2/\log^3 n)$.*

5.4 String Searching

Another example of a problem where a large class of algorithms can be sped up in the UW-RAM is string searching. Given a text T of length n and a pattern P of length m, both over an alphabet Σ, the string matching problem consists of reporting all the occurrences of P in T. We focus here on on-line searching, this is, with no preprocessing of the text (though preprocessing of the pattern is allowed), and we assume in general that $n \gg m$. We use two classic algorithms for this problem to illustrate different ways of obtaining speedups via parallel operations in the wide word. More specifically, we obtain speedups of $w = \Omega(\log n)$ for UW-RAM implementations of the Shift-And and Shift-Or algorithms [Baeza-Yates and Gonnet, 1992; Wu and Manber, 1992], and the Boyer-Moore-Horspool algorithm [Horspool, 1980]. For a string S, let $S[i]$ denote the i-th character of S, and let $S[i..j]$ denote the substring of S starting at position i and ending at position j. Indices start at 1.

5.4.1 Shift-And and Shift-Or

The Shift-And and Shift-Or algorithms keep a sliding window of length m over the text T. On a window at positions at substring $T[i-m+1..i]$, the algorithms keep track of all prefixes of P that match a suffix of $T[i-m+1..i]$. Thus, if at any time there is one such prefix of length $|P|$, then an occurrence is reported at $T[i-m+1]$. This is equivalent to running the $(m+1)$-state non-deterministic automaton that recognizes P starting from every position of T. For a window $T[i-m+1..i]$ in T, the j-th state of the automaton is active if and only if $P[0..j] = T[i-j+1..i]$. These algorithms represent the automaton as a bit vector and update the active states using bit-parallelism. More specifically, the Shift-And algorithm keeps a bit vector $\vec{v} = b_0 b_1 \ldots b_{m-1}$, where $b_j = 1$ whenever the j-th state is active. If $\vec{v}_i$ represents the automaton for the window ending at $T[i]$, then $\vec{v}_{i+1} = ((\vec{v}_i >> 1) \mid 1) \;\&\; Y[T[i]]$, where $Y[\sigma]$ is a bit vector with set bits in the positions of the occurrences of σ in P. The OR with a 1 corresponds to the first state always being active to allow a match to start at any position. The Shift-Or algorithm is similar but it saves this operation by representing active states with zeros instead of ones.

Algorithm 5.6 Shift-And$(T, P, n = |T|, m = |P|, \Sigma)$

```
1: {Preprocessing}
2: for each σ ∈ Σ do
3:     Y[σ] ← 0⃗
4: for j = 1 to m do
5:     Y[P[j]] ← Y[P[j]] | (1 >> (j − 1))
6: {Search}
7: V ← 0⃗
8: C ← 1 >> (m − 1)
9: for i = 1 to n do
10:    V = ((V >> 1) | 1) & Y[T[i]]
11:    if V & C ≠ 0 then
12:        report an occurrence at i − m + 1
```

We describe two UW-RAM algorithms for Shift-And that illustrate different techniques, noting that the UW-RAM implementation of Shift-Or is analogous. The running times of the UW-RAM algorithms are $O(nm/w^2 + n)$ and $O(nm/w^2 + n/w)$, which are $O(n)$ and $O(n/w)$, respectively, for $m = O(w^2)$.

w^2-bit Automata

The straightforward way of taking advantage of the wide word when implementing Shift-And is to use the entire wide word for bit vectors. We first compute the mask array $Y[\sigma]$ for each $\sigma \in \Sigma$ and store each w^2-bit vector in contiguous words of memory starting at address $Y + \sigma$. Then the code of the UW-RAM is essentially the same as the original code, replacing all references to the array Y with memory access operations for the wide word: assuming $m \leq w^2$, reading and writing to $Y[\sigma]$ implemented with by read_word$(W, Y + \sigma)$ and write_word$(W, Y + \sigma)$, for some word W. Otherwise, bit vectors are represented in $\lceil m/w^2 \rceil$ wide words (and stored in memory in $\lceil m/w^2 \rceil w$ words). The rest of the operations are done on registers, and constants are part of the precomputation. The pseudocode for this algorithm is shown in Algorithm 5.6, which assumes $m \leq w^2$ and is based on the pseudocode for Shift-And given in [Navarro and Raffinot, 2002, Chapter 2.2.2]. Since we can now update $\vec{v}$ in $O(m/w^2 + 1)$ time, the running time of Algorithm 5.6 is $O(nm/w^2 + n)$. Thus, compared to the original algorithm, the UW-RAM algorithm achieves a speedup of w when $m \geq w^2$, and a speedup of $\lceil m/w \rceil$ otherwise (no speedup is achieved for $m \leq w$).

Lemma 5.1 *When implemented in the UW-RAM, the Shift-And and Shift-Or algorithms for searching a pattern of length m in a text of length n have a running time of $O(nm/w^2 + n)$, achieving a w-fold speedup over word-RAM implementations when $m \geq w^2$.*

Algorithm 5.7 Parallel Shift-And$(T, P, n = |T|, m = |P|, \Sigma)$. For technical reasons, assume that $T[n+j] = \$$ for $j = 1, \ldots, m-1$, with $\$ \notin \Sigma$, and that $w \geq \log(n+m)$. In order to report matches at each step in time proportional to the number of matches (and not the number of blocks), we move directly to blocks with matching positions by using a function that for every word of length w returns an array A with the positions of set bits. For example, for $w = 5$ and $x = 01011$, $A = \{1, 3, 4\}$. We do this by table look up to a table with $(w/2)$-bit entries, whose space is $O(2^{w/2}w)$ words, which for $w = \log n$ is $O(\sqrt{n \log n})$.

{Preprocessing}
for each $\sigma \in \Sigma$ **do**
    $Y[\sigma] \leftarrow 0$ $\{|Y[\sigma]| = w\}$
for $j = 1$ to m **do**
    $Y[P[j]] \leftarrow Y[P[j]] \mid (1 >> (j-1))$
    $Y[\$] \leftarrow 0$
$V \leftarrow \vec{0}$
ONES $\leftarrow \frac{2^{w^2}-1}{2^w-1}$ $\{\text{ONES}_j = 1$ for all $j\}$
$C \leftarrow$ ONES $>> (w-1)$ $\{C_j = 2^{w-1}$ for all $j\}$
{Search}
$n' \leftarrow n/w$
POSNS $\leftarrow \vec{0}$ {current positions in text}
for $j = 0$ to w **do**
    POSNS $\leftarrow$ POSNS $\mid ((jn'+1) >> wj)$
for $i = 1$ to $n' + m - 1$ **do**
    $V1 \leftarrow (V >> 1) \mid$ ONES
    $V2 \leftarrow$ POSNS
    read_content$(V2, T)$ {load characters in each position $(V2_j = T[\text{POSNS}_j])$}
    read_content$(V2, Y)$ {lookup masks in array Y $(V2_j = Y[T[\text{POSNS}_j]])$}
    $V \leftarrow V1$ & $V2$
    $W \leftarrow V$ & C {check for matches at each block}
    $W \leftarrow$ transpose$(W << w-1)$
    matches $\leftarrow W_0$ {matches$[j] = 1$ if there was a match at block j}
    write_word(POSNS, matching_positions) {write all current positions in array matching_positions}
    $A \leftarrow$ lookup(matches) {position in T of k-th matching block is at matching_positions$[A[k]]$}
    for $k = 1$ to $|A|$ **do**
        report match at matching_positions$[A[k]]$
    $V \leftarrow V$ & $\sim C$ {clear most significant bit in each block}
    POSNS $\leftarrow$ POSNS $+$ ONES {update positions in T ($\text{POSNS}_j \leq n+m-1$ for all j, thus there is no carry across blocks)}

w-bit Parallel Automata

Another way of using the wide word to speedup the Shift-And algorithm is to take advantage of the parallel memory access operations of the UW-RAM to perform w parallel searches on disjoint portions of the text. This is done by using each block of a wide word to represent the automata

in each search: block j is used to search P in $T[jn/w..(j+1)n/w-1]$, for $0 \leq j \leq w-1$ (we assume w divides n). Since the operations involved in updating the automata are the same across blocks, an update to all w automata can be done with a constant number of single wide word operations. All bit vectors of the precomputed table Y are now again w-bit long, as in the original algorithm. In each step of the search, w entries of Y are read in parallel to each block according to the current character in T in the search in each portion. The pseudocode for this procedure is shown in Algorithm 5.7. The code assumes $m \leq w$, though it is straightforward to modify it for the $m > w$ case. The running time of this algorithm is now $O(nm/w^2 + n/w + occ)$, where occ is the number of occurrences found. This is always faster than the first version above, and it leads to the following theorem:

Theorem 5.4 *Given a text T of length n and a pattern P of length m, we can find the occ occurrences of P in T in the UW-RAM in time $O(nm/w^2 + n/w + occ)$.*

5.4.2 Boyer-Moore-Horspool (BMH)

We give one more example of how to use the wide word to speed up string searching by describing a UW-RAM implementation of the BMH [Horspool, 1980] algorithm. BMH keeps a sliding window of length m over the text T and searches backwards in the window for matching suffixes of both the window and the pattern. More specifically, for a window $T[i..i+m-1]$, the algorithm checks if $T[i+j-1] = P[j]$ starting with $j = m$ and decrementing j until either $j = 0$ (there is a match) or a mismatch is found. Either way, the window is then shifted so that $T[i+m-1]$ is aligned with the last occurrence of this character in P (not counting $P[m]$). The worst case running time of BMH is $O(nm)$ (when the entire window is checked for all window positions) but on average the window can be shifted by more than one character, making the running time $O(n)$ [Baeza-Yates and Régnier, 1992]. In the UW-RAM, we can take advantage of the wide word to make several character comparisons in parallel, thus achieving a w-fold speedup over the worst case behaviour of the standard algorithm.

First, we divide each wide word in f-bit fields so that each field contains one character, thus $f = \lceil \log \sigma \rceil$. At each position of the window, we do a field-wise comparison between a wide word containing the characters of the text and one containing the characters of the pattern. We do this simply by subtracting both words. Since we only care if all symbols in the words match, we only need to check if the result is zero, without having to worry about carries crossing fields (and hence we do not need a test bit). We shift the window to the next position if the result is not zero. Note that this check can be done in constant time, and it is quite simple as we do not need to identify where there was a mismatch. Thus in each window we can compare up to w^2/f symbols in parallel, and hence the running time in the worst case becomes $O(mn \log \sigma / w^2 + 1)$. We show the pseudocode in Algorithm 5.8 which, again, is based on the pseudocode of this algorithm presented in [Navarro and Raffinot, 2002, Chapter 2.3.2]. Note that for a given input the distance of the

Algorithm 5.8 BMH($T, P, n = |T|, m = |P|, \Sigma$). For simplicity, we assume that w divides $m \log \sigma$. We assume also that T and P are represented with $\log \sigma$ bits per symbol. We still use $T[i]$ to denote one character, which can be easily obtained from the packed representation in constant time (the same applies to the actual address of starting characters of substrings).

```
 1: {Preprocessing}
 2: for each σ ∈ Σ do
 3:    jump[σ] ← m
 4: for j = 1 to m − 1 do
 5:    jump[P[j]] ← m − j
 6: m′ ← w²/ log σ {characters per wide word}
 7: {Search}
 8: i = 0
 9: while i ≤ n − m do
10:    k ← m′/m {number of window segment}
11:    while k > 0 do
12:       W ← T[i + (k − 1)m′ + 1..i + km′] {W contains the substring of T of k-th window segment}
13:       V ← P[(k − 1)m′ + 1..km′] {V contains the substring of P of k-th window segment}
14:       if W − V ≠ 0 then
15:          break
16:       else if k = 1 then
17:          report occurrence at i + 1
18:       k ← k − 1
19:    i ← i + jump[T[i + m]]
```

shifts is exactly the same as in the original version of the algorithm, and therefore the average running time remains the same. Note as well that the average running time can be reduced by using each block to search in disjoint parts of the text at the expense of increasing the worst case time to $O(mn \log \sigma/w + 1)$ due to the reduction in the number of characters that can be compared simultaneously.

Theorem 5.5 *Given a text T of length n and a pattern P of length m, both over an alphabet of size σ, we can find the occurrences of P in T with a UW-RAM implementation of Boyer-Moore-Horspool in $O(mn \log \sigma/w^2 + 1)$ time in the worst-case and $O(n)$ time on average.*

5.5 Conclusions

We have introduced the Ultra-Wide Word architecture and model, and showed that several classes of algorithms can be readily implemented in this model to achieve a speedup of $\Omega(\log n)$ over traditional word-RAM algorithms. The examples described in this chapter already show the potential of this model to enable parallel implementations of existing algorithms with speedups

comparable to those of multi-core computations. We believe that this architecture could serve as well to simplify many existing word-RAM algorithms that in practice do not perform well due to large constant factors. We conjecture as well that this model will lead to new efficient algorithms and data structures that can sidestep existing lower bounds.

For future work, it would be interesting to extend the algorithmic techniques that can be used in this model as well as to describe UW-RAM implementations of other word-RAM algorithms and for other classes of problems. Finally, another interesting research direction would be to formalize the relation between the UW-RAM and other models of computation in terms of simulations and their efficiency.

Chapter 6

Paging and Online Algorithms

Consider a team of Internet service provider (ISP) technicians visiting the residences of customers that need technical support. Imagine that the quality of service standards of the ISP require that every request for technical support be served the day after the call is made. Thus, once all calls for one day are in, the manager of this team must decide which worker to send to each residence the next day and at what time. Due to the high cost of gas, she would like to come up with a schedule that minimizes the total time travelled by her workers. She then computes a schedule and makes it available to the rest of the team. Contrast this scenario with one in which the quality of service standard is higher, and the ISP guarantees that every request will be served during the same business day, provided that the call is made at least two hours before closing hours. Moreover, while still on the phone, the customer receives confirmation of an accurate estimation of the time in which a technician will arrive. Now, the manager, still trying to minimize travel time, must decide which worker to send to each residence based on the requests that she has received so far during the day, and without knowing either when she will receive the next request or where the service should be delivered. Furthermore, once a commitment with a customer has been made, it cannot be modified.

It is clear that achieving the goal of serving all requests with minimum travel time is a different challenge in each scenario. While the first scenario presents a typical optimization problem which can be solved with an algorithm that has full information of the entire input before it attempts a solution, the latter scenario requires a strategy in which decisions must be made as the input is revealed; it is a simple example of an online problem.

Online computation deals with problems in which decisions must be made with incomplete information. An online algorithm must process a sequence of events, after each of which it must take an action based only on the information provided by past events. One of the most notable representatives of online problems is paging, a problem that models the management of data in computer memory hierarchies. In Chapters 7 and 8 we study variants of the paging problem

that are relevant in multi-core computation. This chapter provides the necessary background on online algorithms and paging as well as relevant related work.

6.1 Online Algorithms

Let $\mathcal{R} = \{r_1, r_2, r_3, \ldots, r_n\}$ be a sequence of events or requests which is the input to an online problem P. An online algorithm solving P must produce a sequence of responses or actions $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$, where each s_i depends only on $\{r_1, r_2, \ldots, r_i\}$. In other words, each action s_i is produced after receiving request r_i and depends only on requests seen so far, and cannot be changed. Furthermore, the algorithm has no knowledge of even the number of total requests in the input. In the example given at the beginning of this section, each request is the address of each residence, and the action is the decision of which worker should go to that address. This problem is an application of the k-server problem, in which servers must be assigned to requests in an online fashion, with the goal of minimizing a cost measure.

Measuring the performance of an online algorithm is not straightforward, and much of the research in online algorithms has dealt with the problem of finding the right measure of performance. It should be easy to see that a simple worst-case performance analysis would not be very useful to measure the quality of online solutions, as adversarial sequences based on each algorithm's decision would result in quite negative results for any possible algorithm.

6.1.1 Competitive Analysis

The traditional way of assessing the quality of online algorithms is to compare their performance to that of an algorithm that knows the future, known as the optimal offline. This way of analyzing online computation is called competitive analysis and was introduced in 1985 by Sleator and Tarjan [1985]. In competitive analysis, the quality of an algorithm is expressed in terms of their competitive ratio, defined as follows.

Definition 6.1 (Competitive ratio [Borodin and El-Yaniv, 1998]) *Let A be an online algorithm for a minimization problem Π and let OPT denote an optimal offline algorithm for this problem. Let $A(\mathcal{R})$ and $OPT(\mathcal{R})$ denote the cost of A and OPT on a request sequence $\mathcal{R}$. For any $c \geq 1$, we say that A is c-competitive if for all request sequences $\mathcal{R}$,*

$$A(\mathcal{R}) \leq c \cdot OPT(\mathcal{R}) + \beta, \tag{6.1.1}$$

where β is a constant that does not depend on $\mathcal{R}$. We say that A is strictly c-competitive if $\beta = 0$. The competitive ratio of A is defined as the infimum of all such values of c.

If Π is a maximization problem, then the definition is analogous, with Inequality (6.1.1) becoming $A(\mathcal{R}) \geq c \cdot OPT(\mathcal{R}) - \beta$. The competitive ratio c is effectively an approximation ratio. Note, however, that online algorithms settle for approximations to overcome not knowing the entire input and not in order to achieve a better running time performance than the optimal algorithm. In fact, the running time of an online algorithm is normally not a concern, and the focus is on its competitiveness. The running time performance is obviously important in practical applications. In addition, unlike the study of approximation algorithms, which allows non-constant approximation ratios, the competitive ratio is usually thought of as a constant. Thus, the term *competitive* online algorithm is used to refer to an algorithm with constant competitive ratio.

While competitive analysis is arguably the most utilized measure of performance for online algorithms, it has often been criticized for being too pessimistic and for not reflecting the influence of important practical considerations in algorithms and input sequences. There exist other measures of performance, some of them tailored to particular problems. Next, we review the online paging problem and mention other measures of performance in that context.

6.2 Paging

A computer memory system is designed as a hierarchy of storage devices which higher capacity but slower access cost as the distance to the processors increases. After processors' registers, the fastest and smallest storage devices are caches, of which there are normally three levels, and that are usually on the processor chip, although sometimes the last level is outside the chip. The next levels in the memory hierarchy are the computer's RAM, one or more hard drives or disks, and external or network accessed storage.

Efficiently managing data across this hierarchy is fundamental to the performance of modern computers, and it is a problem of active research across several areas of Computer Science. An important aspect of this problem is to decide what data is maintained in the faster lower levels of the hierarchy. The paging problem addresses this issue by modeling a two-level memory system consisting of a slow memory of infinite size and a fast memory of size k, usually known as the cache. The input to an instance of paging is a sequence of page requests. Upon a request to a page σ, if σ is in the cache, known as a *hit*, no action is required. Otherwise, a *page fault* has occurred, and σ must be brought from slow to fast memory, possibly requiring the eviction of another page in the cache. A paging algorithm must decide which pages to maintain in the cache at all times so as to minimize the number of faults. This cost model is known as the *page fault cost model*, in which a page fault has cost of one, and a hit has no cost.

The name paging stems from the virtual memory management systems implemented in modern operating systems. While all processes running simultaneously on a computer share the physical memory space, each of them operates on its own virtual address space, which is divided

in contiguous blocks of addresses known as pages. When a process accesses a memory location, this corresponds to a request to the page that contains this location, which can either reside in the physical memory, or have been swapped to disk. Thus, the paging problem refers to the management of pages in memory and disk. The name paging is also used, however, to model the same process in any level of the hierarchy, although sometimes it is referred to as *caching* when explicitly being considered for the management of data between caches and main memory. In the latter case, the relevant unit of data storage is known as a cache *block* or *line*, and the equivalent of a page fault is known as a cache *miss*. In the paging problems studied in this thesis, we focus on the management of cache vs. memory systems, however, the results are general and applicable to any two-level memory system. We use the generic terminology related to the paging problem; thus, we refer to pages as the relevant data organization unit and as hits and faults for the aforementioned events. We refer to the fast memory as the cache, and to the slow memory simply as memory.

6.2.1 Paging Algorithms

Given a sequence of page requests, a paging algorithm must decide, upon each request that results in a page fault when the cache is full, which page to evict from the cache in order to make space for the new page. In this sense, paging algorithms are also known as *cache eviction policies*. In general, the decision should not only be which pages must be evicted from the cache, but which pages to keep in cache at all times. In the classic cost model, in which we are interested in minimizing the number of faults, both are equivalent. In other words, there is no disadvantage in keeping the cache full at all times and only evicting a page when it is required. An algorithm with this property is known as a *demand paging* or *lazy* algorithm. More specifically,

Definition 6.2 (Demand paging algorithm [Borodin and El-Yaniv, 1998]) *An algorithm is said to be demand paging if it never evicts a page unless there is a page fault and the cache is full.*

Any paging algorithm can be made demand paging without increasing its cost [Borodin and El-Yaniv, 1998]. While this is true in the traditional page fault model, we shall see that for other costs models it might be desirable for algorithms to evict pages even if there is no need to make space for a new page in the cache (see Chapter 8).

The following well known paging algorithms are defined by the actions they take when there is a page fault and the cache is full [Borodin and El-Yaniv, 1998]:

- Least-Recently-Used (LRU): evict the page in the cache whose last access is the least recent.
- First-In-First-Out (FIFO): evict the page in the cache that was brought into the cache earliest.

- Flush-When-Full (FWF): evicts all pages in the cache.
- Least-Recently-Used-2 (LRU-2): evict the page whose second to last access is the least recent; evict the least recently used page if all pages in the cache have only been requested once.
- Last-In-First-Out (LIFO): evict the page in the cache that was last brought into the cache.
- Least-Frequently-Used (LFU): evict the page in the cache that has been requested the least since it was brought into the cache.
- Clock: associates a bit with each page that is set every time the page is accessed. Pages in the cache are kept in a circular list, with a pointer to the most recently added page. When an eviction is required, the list is traversed from the pointer unsetting the bits of paging it visits and stopping at and evicting the first encountered page with an unset bit.
- Furthest-In-The-Future (FITF): evict the page in the cache whose next request will be furthest in the future.

The last policy in the list above differs from the rest in that it needs knowledge of future requests to make decisions. This strategy, known also as Longest-Forward-Distance (LFD) or Belady's algorithm [Belady, 1966], is an optimal offline strategy.

From the strategies above, LRU, FIFO, FWF, and Clock are k-competitive. This is optimal, as no deterministic online algorithm can have a competitive ratio better than k [Borodin and El-Yaniv, 1998]. The competitive ratio of LRU-2 is $2k$ [Boyar et al., 2006], while LIFO and LFU are not competitive (i.e., their competitive ratio is not bounded by a constant independent of the length of the sequence). LRU and FWF are instances of a more general class of algorithms called *marking* algorithms.

Definition 6.3 (Marking algorithm) *A marking algorithm associates a mark with each page in its cache (either explicitly or implicitly) and marks a page when it is brought to cache or if it is unmarked and requested. Upon a fault with a full cache, it only evicts unmarked pages if there are any, and unmarks all pages in cache otherwise.*

The execution of a marking algorithm over a sequence $\mathcal{R}$ defines a partition of the sequence in phases: a new phase starts on the request which requires unmarking all pages in the cache. Since the definition is dependent on the cache size, we denote it as k-phase partition. The following is an equivalent definition of a k-phase partition.

Definition 6.4 (k-phase partition) *The k-phase partition of a sequence $\mathcal{R}$ is a partition of $\mathcal{R}$ into contiguous subsequences of requests or phases such that the first phase starts with the first*

request, and each subsequent phase starts when $(k+1)$ *different pages have been requested since the beginning of the previous phase.*

In the k-phase partition of a sequence, every phase but the last one has requests for exactly k distinct pages, while the last phase might have requests for fewer distinct pages. The following fundamental property of marking algorithms will be repeatedly used in the proofs in Chapters 7 and 8.

Proposition 6.1 ([Borodin and El-Yaniv, 1998]) *A marking algorithm incurs at most* k *faults within a phase.*

All marking algorithms are k-competitive, which can be shown using the property above. This property can be generalized to any consecutive subsequence of k distinct pages, which yields the definition of *conservative* algorithms.

Definition 6.5 (Conservative algorithms [Borodin and El-Yaniv, 1998]) *An algorithm is conservative if it incurs at most* k *faults on any consecutive subsequence of requests that contains at most* k *distinct pages.*

Conservative algorithms are also k-competitive. LRU, FIFO, and Clock are examples of conservative algorithms [Borodin and El-Yaniv, 1998].

Randomized Paging Algorithms

The lower bound of k on the competitiveness of any deterministic algorithm can be bypassed using randomization, though it is important to note that the randomized competitive ratio of an algorithm is most closely related to the average competitive ratio rather than its worst case. The following algorithms are well-known randomized paging algorithms:

- Random: evict a page chosen uniformly at random.
- Mark: mark pages when accessed as in the definition of marking algorithms (Definition 6.3), and evict a page chosen uniformly at random among all unmarked pages.

In the context of randomized online algorithms, special attention must be placed into the definition of the adversary used in the analysis of algorithms' competitiveness. An *oblivious* adversary can only choose the request sequence in advance (and not depending on the online algorithm's random decisions) and serve the sequence offline. In the other extreme, an *adaptive offline* adversary can choose each request after the decision of the online algorithm and can serve

the sequence offline. In between the two, the *adaptive online* adversary can adapt the sequence to the actions of the online algorithm but must also serve the sequence online, i.e., it must serve each request right after presenting it to the online algorithm. If an algorithm is c_1-competitive against an oblivious algorithm, c_2-competitive against an adaptive online algorithm, and c_3-competitive against an adaptive offline algorithm, then $c_1 \leq c_2 \leq c_3$ [Borodin and El-Yaniv, 1998].

The simple Random algorithm is still k-competitive against an adaptive online algorithm, while Mark is $2H_k$ competitive against an oblivious adversary, where H_k is the k-th Harmonic number. H_k is defined as $H_k = \sum_{i=1}^{k} 1/i$, which is $\Theta(\log k)$. H_k is actually a lower bound for randomized algorithms against an oblivious adversary, and thus it carries to more powerful adversaries [Borodin and El-Yaniv, 1998]. This competitive ratio is attained by the optimal algorithms Partition [McGeoch and Sleator, 1991] and Equitable [Achlioptas et al., 2000].

6.2.2 Other Cost Models

While in the classic model all pages have the same size and cost, models motivated by web-caching consider pages of varying sizes. In the *fault* model [Irani, 1997] pages have varying sizes but uniform fault cost; in the *bit* model [Irani, 1997] the fault cost of each page equals its size, thus the problem of minimizing fault cost is equivalent to minimizing the amount of data brought into the cache. In these models, and assuming pages can bypass the cache if desired, LRU is $(k+1)$-competitive. The *cost* model or *weighted caching* problem [Chrobak et al., 1991] considers pages with varying fault costs but uniform page sizes. Finally, a general model allows arbitrary sizes and costs [Young, 1998], for which k-competitive deterministic algorithms are known [Cao and Irani, 1997; Young, 1998]. Bansal et al. [2008] showed an $O(\log^2 k)$-competitive randomized algorithm for the general model and an $O(\log k)$-competitive randomized algorithms for the fault and bit models. While a polynomial time algorithm for the offline weighted cache problem exists [Chrobak et al., 1991], the offline problem in the bit, fault, and general models was recently shown to be strongly NP-complete [Chrobak et al., 2012].

Unlike these models, which consider only the cost of faults, the *full access cost* model [Torng, 1998] charges a cost of 1 for a hit, and a cost of $s \geq 1$ for a fault. In this model, marking algorithms achieve a competitive ratio of $1 + \frac{(k-1)s}{L+s}$, where L is the average phase length in the k-phase partition of a sequence. In the worst case, $L = k$ and the ratio is $k(s+1)/(k+s)$, which is optimal. The model coincides with the classic model when $s \to \infty$, but can yield competitive ratios that are significantly smaller if s is small or if a sequence has high locality [Borodin and El-Yaniv, 1998]. As we shall see, the model of paging with cache usage that we present in Chapter 8 is also amenable to an analysis of algorithm performance in terms of locality of reference, yielding small competitive ratios for sequences with high locality for the proposed online algorithms.

6.2.3 Alternative Performance Measures

The competitive analysis framework has been criticized for not properly reflecting the practical performance of algorithms in some problems, one of which is paging. For example, according to competitive analysis LRU, FIFO and FWF are all optimal. While it is well known that in practice LRU outperforms FIFO and FWF, pure competitive analysis cannot distinguish between the performance of these algorithms. It also fails to reflect performance gains in the presence of lookahead. A paging algorithm with lookahead of ℓ is allowed to make decisions based on past requests as well as on ℓ requests to the future. Intuitively, this should give such an algorithm an effective theoretical advantage over algorithms without lookahead. Unfortunately, competitive analysis cannot distinguish between the two classes of algorithms, and the lower bound of k applies as well to any algorithm with finite lookahead.

To overcome the shortcomings of competitive analysis, several alternative measures of performance for online algorithms have been proposed. Some of these are generic to all online algorithms while others are specific to some problems. Among the former, there are several measures that shift away from comparisons to an offline optimal algorithm and compare the performance of two online algorithms directly. Examples of these measures are bijective and average analysis [Angelopoulos et al., 2007], and relative interval analysis [Dorrigiv et al., 2009]. In relative interval analysis, two online algorithms A and B are compared directly as follows. Let $A(\mathcal{R})$ denote the cost of A on a request sequence $\mathcal{R}$ of a minimization problem, and let $Min(A,B) = \liminf_{n\to\infty}(\min_{|\mathcal{R}|=n}\{A(\mathcal{R}) - B(\mathcal{R})\})$ and $Max(A,B) = \limsup_{n\to\infty}(\max_{|\mathcal{R}|=n}\{A(\mathcal{R}) - B(\mathcal{R})\})$. Thus $Min(A,B)$ and $Max(A,B)$ represent the minimum and maximum difference in cost between the two algorithms, respectively. Then the relative interval of A and B is defined as

$$\mathcal{I}(A,B) = [Min(A,B), Max(A,B)].$$

We say that an interval $[\alpha,\beta]$ approximates the relative interval of A and B, denoted as $[\alpha,\beta] \subseteq \mathcal{I}(A,B)$ if $Min(A,B) \leq \alpha$ and $\beta \leq Max(A,B)$. The relative interval of A and B represents the maximum and minimum difference in performance between A and B. Thus, if $\mathcal{I}(A,B) \subseteq [0,\beta > 0]$ we say that B dominates A, since on any sequence B is no worse than A, and there is at least one sequence for which B is better than A. In general, B has better performance than A in this model if $Max(A,B) > |Min(A,B)|$. It can be shown that under this performance measure LRU and FIFO have better performance than FWF, and that the model reflects the advantage of lookahead [Dorrigiv et al., 2009].

In Chapter 8 we use a variant of relative interval analysis to compare two online algorithms by considering the cost ratios instead of differences. We also take a similar approach in Chapter 7 to directly compare the performance of various strategies to manage shared caches.

There exist several other ways of measuring the performance of online algorithms. We refer the reader to [Dorrigiv and López-Ortiz, 2005, 2009; Dorrigiv, 2010] for a survey and discussion.

6.2.4 Paging with Multiple Request Sequences

Various models have been proposed to analyze the performance of paging algorithms in the presence of multiple request sequences, either modeling multiple applications or multiple threads. In what follows, we briefly review some of these models, which differ mainly in the assumptions they make with respect to the knowledge of future requests by each process and the abilities of the paging algorithm to schedule page requests. In the following, p denotes the number of request sequences and k the size of the cache.

Multipointer Paging in the Access Graph Model

Fiat and Karlin [1995] study paging algorithms in the access graph model, in which request sequences are restricted to paths in a given graph [Borodin et al., 1995]. They study the multipointer case, in which several paths through an access graph might be performed simultaneously, modeling both different applications (the graph could be disconnected) or multithreaded computations (having several paths in one same connected component). They describe a deterministic algorithm in this model and show that it is optimal up to constant factors.

Application-Controlled Caching

Cao et al. [1994] study a scenario in which various applications share a cache. A global kernel manages a dynamic allocation of cache blocks to each application, while each application is responsible for managing its assigned blocks. It is assumed for the latter that each application has knowledge of its cache access patterns, i.e., each application knows its entire sequence of requests. However, the global allocation policy has no knowledge of the future. This work presents a two-level block replacement approach which experimentally achieves an improvement in global performance over a global LRU policy. Barve et al. [2000] showed that the competitive ratio of the algorithm of Cao et al. is $2p + 2$, where p is the number of applications. They name the problem *multiapplication caching* and show that no deterministic online algorithm can have a competitive ratio better than $p + 1$ for this problem. They propose a randomized strategy that achieves a competitive ratio of $2H_p + 2$, which is approximately $2 \ln p$, and show that $\min\{H_{p-1}, H_k\}$ is a lower bound for any application-controlled algorithm. The bounds are given with respect to a worst possible interleaving of the request sequences. Note that the upper bounds do not depend on the cache size and might be smaller than the lower bound H_k for traditional paging, which is due to the fact that each application knows future page requests.

Recently, Katti and Ramachandran [2012] extended the results on the application-controlled caching setting, which they term *full knowledge* model. Unlike the work in [Cao et al., 1994; Barve et al., 2000], Katti and Ramachandran consider the case in which pages might be shared between

processes. They show a lower bound of $\frac{p}{2}\log\frac{4(k+1)}{3p}$ for any deterministic online algorithm, and a lower bound of $\frac{\log(k+1)}{2}$ for randomized algorithms. The latter establishes that Partition [McGeoch and Sleator, 1991] is optimal in this case (up to constant factors). They describe a deterministic algorithm that achieves a competitive ratio of $2(p\ln(ek/p)+1)$. For the disjoint case studied in [Cao et al., 1994; Barve et al., 2000], they present a deterministic online algorithm with competitive ratio $\max(10, p+1)$, which is optimal for $p \geq 9$ (recall that $p+1$ is a lower bound in this setting). This work also considers paging in memory hierarchies, establishing upper and lower bounds on the competitive ratio of algorithms for the last level of cache.

Multi-Threaded Paging

Feuerstein and Strejilevich de Loma [2002] introduced Multi-Threaded Paging (MTP). In this problem, given a set of page requests, an algorithm must decide at each step which request to serve next and how to serve it. They study the settings with finite and infinite request sequences, and both with and without fairness restrictions, providing lower and upper bounds for online algorithms. In the case when no fairness restrictions are imposed, they show that there exist algorithms with competitive ratio pk for the infinite and finite settings. On the other hand, they show that when general fairness restrictions are imposed, there are no competitive algorithms. Results in this model were further extended in [Strejilevich de Loma, 1998] and [Seiden, 1999]. In this model, a paging algorithm has the capability to schedule requests, and thus the order in which requests are served is algorithm dependent.

Cache Replacements for Multi-Cores

Hassidim [2010] introduced a model for cache replacement policies specific to multi-core caches in which requests are served in parallel. This model considers the fetching time of pages from memory: if there is a fault on a request of one sequence, the rest of the sequences continue to be served while the faulting sequence's page is fetched from memory. This work studies the performance of algorithms in the competitive analysis framework with makespan as the performance measure, and not the number of faults. It shows that the competitive ratio of LRU with a cache of size k is $\Omega(\tau/\alpha)$, where τ is the ratio between miss and hit times, and the offline optimal has a cache of size k/α. For a constant resource augmentation factor α, LRU has competitive ratio $\Omega(\tau)$. This result is significant since a competitive ratio of τ can be achieved by an algorithm that does not use the cache at all: the makespan of a strategy that incurs faults only is at most a factor of τ larger than that one of the optimal offline. Hassidim also shows that computing the optimal offline schedule is NP-complete and presents a Polynomial Time Approximation Scheme (PTAS) for constants p and τ.

Note that in both the MTP model of Feuerstein and Strejilevich de Loma and Hassidim's model the order of requests depends on the decisions of the algorithms, while in previous models

for multi-application caching [Cao et al., 1994; Barve et al., 2000] and multi-pointer paging [Fiat and Karlin, 1995] the order of requests is the same for all algorithms.

As in the MTP model, Hassidim's model assumes that the paging strategy can choose to serve requests of some sequences and delay others. In particular, the offline strategy is able to modify the schedule of requests, and hence is more powerful than a regular cache eviction algorithm. Building on Hassidim's model, in Chapter 7 we describe a model for multi-core paging that discards this possibility, assuming that the order in which requests of different processors arrive to the cache is given by a scheduler over which the caching strategy has no influence. Hence, our model is different from previous models in that we assume no explicit scheduling capabilities of the paging strategy, while at the same time faults introduce delays in sequences, thus changing the order of requests.

Chapter 7

Paging for Multi-Core Shared Caches

Multi-core processors are equipped with at least two and sometimes three cache levels, with at least one cache level shared by some or all cores. The performance of multi-core programs depends crucially on how efficiently cores can access data in their caches. In fact, many of the multi-core models and algorithms that have been proposed place cache performance at the same level or above parallel computation times (see Section 2.8.2). The efficient management of data in caches during the execution of a multi-core program, either explicitly by an algorithm or by the operating system, becomes a key aspect of the computation in these architectures. In this context, a primary component of cache data management is cache eviction policies.

Cache eviction policies have been widely studied both in theory and in practice for sequential processors. From a theoretical perspective, the efficient management of data in a cache is modeled by the paging problem: given a fast memory of size k and a sequence of page requests, the goal is to serve the sequence while minimizing the number of cache misses or faults (see Chapter 6). In reality it is often the case that the fast memory is a shared resource among multiple processes or multiple threads of the same process. A few models were proposed to reflect this scenario (prior to multi-cores), extending the classical paging problem to a setting in which several sequences must be served simultaneously with a shared fast memory (see Section 6.2.4). The case of multiple threads sharing a cache in multi-cores falls under this scenario as well. However, multi-core threads execute concurrently in parallel, and hence requests can be served in parallel. This implies that when a sequence suffers a cache miss, other sequences can continue to be served.

In order to reflect the parallel execution of threads, Hassidim [2010] proposed a model specific to multi-core shared caches (see Section 6.2.4). In a multi-core system with p cores, a shared cache might receive up to p page requests simultaneously. Hassidim's model is somewhat unconventional in that paging strategies not only decide which pages to evict from the cache but can also schedule the execution of threads. While in principle there is no reason why this cannot be so, historically, the operating system has kept the scheduling of execution and the paging tasks separate. Within

the operating system, the scheduler concentrates on fairness and throughput considerations to determine which task should be executed while the paging algorithm focuses on which of the pages currently in the cache should be evicted upon a fault.

In this chapter we propose a more conservative and conventional model for multi-core paging, in which paging algorithms are not allowed to make any scheduling decisions but must serve all active requests. In this model, a paging strategy serves a set $\mathcal{R}$ of p request sequences using a shared cache of size k. Requests can be served in parallel, thus various pages can be read from cache or fetched from memory simultaneously, and a page fault delays the remaining requests of the corresponding sequence by τ units of time. We define as FINAL-TOTAL-FAULTS (FTF) the problem of minimizing the total number of faults, and as PARTIAL-INDIVIDUAL-FAULTS (PIF) the problem of deciding, given a request sequence $\mathcal{R}$, a time t, and a bound vector $\vec{b} \in \mathbb{N}^p$, whether $\mathcal{R}$ can be served such that at time t the number of faults on each of the sequences R_i is at most b_i.

Our results[1] Without loss of generality, we define a paging strategy as a combination of a possible partition policy and an eviction policy and compare the performance of natural strategies for FTF within this framework. We show that when restricted to static partition strategies, the choice of the partition has larger impact than the choice of an eviction policy. We also show, however, that partition strategies cannot be competitive with respect to shared strategies if they do not update the partition sufficiently often, even for disjoint request sequences. We show as well that shared strategies with traditional eviction policies (LRU, FIFO, Clock, and FWF) have competitive ratios that are arbitrarily large in the worst case.

We then study the offline cache problem and show properties of optimal offline algorithms. An algorithm that knows future requests can benefit from delaying one or more sequences with respect to others in order to balance the demands of different sequences. This could be achieved by evicting pages even if not triggered by a fault in order to force faults on pages that would otherwise result in hits. We show, however, that forcing faults in this manner is not beneficial. More specifically, we show that any offline algorithm for FTF that forces faults can be transformed into a *lazy* algorithm that never evicts a page unless triggered by a fault, and which incurs no more faults than the original algorithm. In particular, there exists an optimal offline algorithm that does not force faults. We show hardness results for PIF, showing that PIF is NP-complete and that a natural optimization version is APX-hard. Interestingly, PIF is NP-complete even in a simplified model with $\tau = 0$, i.e., when faults do not delay the remaining requests of a sequence. This result immediately implies that this offline problem is hard in the multiapplication caching model [Barve et al., 2000; Cao et al., 1994]. Finally, we present optimal offline algorithms for both FTF and PIF that run in polynomial time in the length of the sequences (and exponential in the number of processors, which for multi-cores is much smaller than the problem size n (see

[1] Results in this chapter appeared in [López-Ortiz and Salinger, 2011; López-Ortiz and Salinger, 2012].

Chapters 3 and 4 for arguments about assumptions on the number of processors as a function of the input size)).

This chapter is organized as follows. In Section 7.1 we describe the multi-core cache model and formally define the problems we address. In Section 7.2 we derive bounds on the performance of natural strategies to minimize the number of faults. We study the offline problem in Section 7.3. We provide concluding remarks and future directions of research in Section 7.4.

7.1 The Cache Model

The model we use in this work is broadly based on Hassidim's model [Hassidim, 2010] (see Section 6.2.4 for a description of this and other models for paging for multiple sequences). We have a multi-core processor with p cores $\{1, \ldots, p\}$ and a shared cache of size k pages. The input is a multiset of request sequences $\mathcal{R} = \{R_1, \ldots, R_p\}$, where $R_j = \sigma^j_{s_1} \ldots \sigma^j_{s_{n_j}}$ is the request sequence of core j of length n_j. $\sigma^j_{s_i}$ is the page identifier of the i-th request in the sequence, with $1 \leq s_i \leq N$, where N is the size of the universe of pages. The total number of page requests is $n = \sum_{j=1}^{p} n_j$. We assume $k \gg p$ and $n_j \gg k$, for all $1 \leq j \leq p$. In particular, we assume that $k \geq p^2$, which can be regarded as a multi-core variant of the tall cache assumption[2]. We say that a request $\mathcal{R}$ is *disjoint* if $\forall i, j, i \neq j, R_i \cap R_j = \emptyset$ and *non-disjoint* otherwise. In practice, a single instruction of a core can involve more than one page. We treat each request as a request for one page, which models the case of separate data and instruction caches on a RISC architecture.

Page requests arrive at discrete timesteps. At any timestep, the cache might receive up to p page requests, each one from a different core, and these are served in one parallel step. This assumes that requested pages from different cores can be read in parallel from the cache. We assume as well that fetching can be done in parallel, i.e., pages from memory corresponding to requests of different cores can be brought simultaneously from memory to the cache.

Serving a request that results in a hit takes one timestep, while serving a request from memory takes $\tau + 1$ steps[3]. A fault delays the remaining requests of the corresponding processor by an additive term τ. In other words, if a request $\sigma^j_{s_{i^*}}$ is a fault, then for all $i > i^*$, the earliest time at which $\sigma^j_{s_i}$ can be served increases by τ. In terms of real world latencies, $\tau + 1$ corresponds to the ratio between miss and hit times of the shared cache. For instance, for the Intel Pentium M processor, the ratio between memory and L2 cache latencies is estimated to be 17 [Hassidim, 2010; Drepper, 2007].

In our model, when a page request arrives, it must be serviced. The only choice the paging algorithm has is in which page to evict shall the request be a fault. To be consistent with

[2] A cache of size Z words with lines of L words is said to be tall if $Z = \Omega(L^2)$ [Frigo et al., 1999].

[3] Note that in Hassidim's model the time to serve a fault is τ.

[Hassidim, 2010], we adhere to the convention that when a page needs to be evicted to make space, first the page is evicted and the cache cell is unused until the fetching of the new page is finished. For non-disjoint requests we use the convention that when there is a request by processor j of a page that is currently in the process of being fetched, then the sequence of processor j is only delayed until the page is fetched into the cache τ units of time after the initial request. We also assume that cache coherency is provided at no cost to the algorithms. Finally, we adopt the convention that simultaneous requests are served logically in a fixed order (e.g., by increasing number of processor).

Under this model, various natural choices of objective functions may be considered. We define and address the following problems:

Definition 7.1 FINAL-TOTAL-FAULTS (FTF) *Given a set of requests $\mathcal{R} = \{R_1, \ldots, R_p\}$, a cache size k, and an integer $\tau \geq 0$, minimize the total number of faults when serving $\mathcal{R}$ with a cache of size k.*

Definition 7.2 PARTIAL-INDIVIDUAL-FAULTS (PIF) *Given a set of requests $\mathcal{R} = \{R_1, \ldots, R_p\}$, a cache size k, a time t, an integer $\tau \geq 0$, and $\vec{b} \in \mathbb{N}^p$, can $\mathcal{R}$ be served with a cache of size k such that at time t the number of faults on each sequence R_i is at most b_i?*

Intuitively, the decision problem PIF is harder than the optimization problem FTF, since the former poses more restrictions on feasible solutions. Posing a bound on individual faults might be required to ensure fairness, and furthermore, doing so at arbitrary times can be used to ensure fairness throughout the execution of an algorithm.

7.2 Bounds of Online Strategies for Minimizing Faults

Natural strategies to manage the cache in the multi-core cache model can be classified in two families: shared and partitioned. In the first one, the entire cache is shared by all processors, and a cache cell can hold a page corresponding to any processor. In the second one, the cache is partitioned in p parts, with each part destined exclusively to store pages of requests from one processor, throughout. A partitioned strategy is static if the sizes of all parts remain constant during an execution and dynamic otherwise.

Both shared and partitioned strategies are accompanied by an eviction policy A. We use S_A to denote the algorithm that uses a shared cache with eviction policy A, and P_A^B to denote a partitioned strategy with partition function B and eviction policy A in each part. A partition function B is *static* if the size of all parts remain constant during an execution and is *dynamic* otherwise. For the latter, when the reduction of the size of a part involves page evictions, these are carried out according to the eviction policy. We make the restriction that all partitions must

assign at least one unit of cache to all processors whose requests are active. For example, S_{LRU} evicts the least recently used page in the entire cache and P_{LRU}^{OPT} performs LRU on each part of the partition, which is determined offline so as to minimize the total number of faults. Figure 7.1 shows an example of the execution of a shared strategy and a static partition strategy.

In the remainder of this section we compare the performance of partitioned and shared strategies for FTF. We denote the number of faults of a strategy Alg on a sequence $\mathcal{R}$ as $Alg(\mathcal{R})$. Also, for a sequence of page requests $\sigma = \sigma_{s_1} \dots \sigma_{s_m}$, we use $(\sigma)^\ell$ to denote the concatenation of σ with itself ℓ times.

Partitioning the cache may be desirable to avoid costs of managing concurrency issues that arise when different threads access a shared page. In addition, a static partition allows for the execution of regular paging algorithms in each part, oblivious to the presence of other threads, and thus provides performance guarantees based on individual threads or processes. In fact, if we restrict paging strategies to a fixed static partition, any marking or conservative algorithm (e.g., LRU, FIFO) has a competitive ratio of at most k, as in the sequential setting (see Section 6.2.1 for the definitions of marking and conservative algorithms). Formally:

Lemma 7.1 (Online vs. offline eviction policies with a fixed static partition) *Let A be any deterministic online eviction algorithm and let $B = \{k_1, k_2, ..., k_p\}$ be any online static partition. There exists a sequence $\mathcal{R}$ such that $P_A^B(\mathcal{R})/P_{OPT}^B(\mathcal{R}) = \Omega(\max_j\{k_j\})$. When A is any marking or conservative algorithm (e.g., LRU), there is a matching upper bound, i.e., $\forall\ \mathcal{R}$, $P_A^B(\mathcal{R})/P_{OPT}^B(\mathcal{R}) \leq \max_j\{k_j\}$.*

Proof: Lower bound. Let $j^* = argmax_j\{k_j\}$. The sequence $\mathcal{R}$ is such that for $j \neq j^*$, $R_j = (\sigma_1^j)^{n/p}$, i.e., the same page is requested n/p times, while R_j consists of requesting, among pages $\{\sigma_1, \sigma_2, ..., \sigma_{k_{j^*}+1}\}$, the page just evicted by A, where $\sigma_{i_1} \neq \sigma_{i_2}$ for $i_1 \neq i_2$, and all sequences are disjoint. $P_A^B(\mathcal{R}) = n/p + p - 1$, since it faults on every request of R_{j^*} and once on each of the other sequences. On the other hand, since P_{OPT}^B only evicts a page of sequence R_{j^*} if it is not requested in the following k_{j^*} requests, we have $P_{OPT}^B(\mathcal{R}) \leq (n/p)/k_{j^*} + p - 1$ and the first part of the lemma follows.

Upper bound. Divide sequence R_j in phases such that a new phase starts every time there is a request for the (k_j+1)-th distinct page since the beginning of the previous phase, and the first phase begins at the first page of R_j. P_{LRU}^B faults at most k_j times in each phase of R_j, while any algorithm must fault at least once in each phase. Let ϕ_j denote the number of phases of sequence R_j, then $P_{LRU}^B(\mathcal{R}) \leq \sum_{j=1}^p \phi_j k_j \leq \max_j\{k_j\} \sum_{j=1}^p \phi_j$. On the other hand, $P_{OPT}^B(\mathcal{R}) \geq \sum_{j=1}^p \phi_j$, and thus $P_{LRU}^B(\mathcal{R})/P_{OPT}^B(\mathcal{R}) \leq \max_j\{k_j\}$. ∎

Using a recent result of Peserico [2013], we can derive a similar upper bound to the one in

S_{LRU}

t	Remaining sequence	Cache
0	$\sigma_1\sigma_2\sigma_4\sigma_2\sigma_3\sigma_4$ $\sigma_5\sigma_6\sigma_2\sigma_4\sigma_5\sigma_5$	$\sigma_1\sigma_2\sigma_3\sigma_5\sigma_6$
1	$\sigma_2\sigma_4\sigma_2\sigma_3\sigma_4$ $\sigma_6\sigma_2\sigma_4\sigma_5\sigma_5$	$\sigma_5\sigma_1\sigma_2\sigma_3\sigma_6$
2	$\sigma_{\mathbf{4}}\sigma_2\sigma_3\sigma_4$ $\sigma_2\sigma_4\sigma_5\sigma_5$	$\sigma_6\sigma_2\sigma_5\sigma_1\sigma_3$
3	$\_\_\,\sigma_2\sigma_3\sigma_4$ $\sigma_{\mathbf{4}}\sigma_5\sigma_5$	$\sigma_2\sigma_6\sigma_5\sigma_1*$
4	$\_\,\sigma_2\sigma_3\sigma_4$ $\_\,\sigma_5\sigma_5$	$\sigma_2\sigma_6\sigma_5\sigma_1*$
5	$\sigma_2\sigma_3\sigma_4$ $\sigma_5\sigma_5$	$\sigma_4\sigma_2\sigma_6\sigma_5\sigma_1$
6	$\sigma_{\mathbf{3}}\sigma_4$ σ_5	$\sigma_5\sigma_2\sigma_4\sigma_6\sigma_1$
7	$\_\_\,\sigma_4$	$\sigma_5\sigma_2\sigma_4\sigma_6*$
8	$\_\,\sigma_4$	$\sigma_5\sigma_2\sigma_4\sigma_6*$
9	σ_4	$\sigma_3\sigma_5\sigma_2\sigma_4\sigma_6$
10		$\sigma_4\sigma_3\sigma_5\sigma_2\sigma_6$
11		
12		

P^B_{LRU}

Remaining sequence	Cache
$\sigma_1\sigma_2\sigma_4\sigma_2\sigma_3\sigma_4$	$\sigma_1\sigma_2\sigma_3$
$\sigma_5\sigma_6\sigma_2\sigma_4\sigma_5\sigma_5$	$\sigma_5\sigma_6$
$\sigma_2\sigma_4\sigma_2\sigma_3\sigma_4$	$\sigma_1\sigma_2\sigma_3$
$\sigma_6\sigma_2\sigma_4\sigma_5\sigma_5$	$\sigma_5\sigma_6$
$\sigma_{\mathbf{4}}\sigma_2\sigma_3\sigma_4$	$\sigma_2\sigma_1\sigma_3$
$\sigma_{\mathbf{2}}\sigma_4\sigma_5\sigma_5$	$\sigma_6\sigma_5$
$\_\_\,\sigma_2\sigma_3\sigma_4$	$\sigma_2\sigma_1*$
$\_\_\,\sigma_4\sigma_5\sigma_5$	σ_6*
$\_\,\sigma_2\sigma_3\sigma_4$	$\sigma_2\sigma_1*$
$\_\,\sigma_4\sigma_5\sigma_5$	σ_6*
$\sigma_2\sigma_3\sigma_4$	$\sigma_4\sigma_2\sigma_1$
$\sigma_{\mathbf{4}}\sigma_5\sigma_5$	$\sigma_2\sigma_6$
$\sigma_{\mathbf{3}}\sigma_4$	$\sigma_2\sigma_4\sigma_1$
$\_\_\,\sigma_5\sigma_5$	σ_2*
$\_\_\,\sigma_4$	$\sigma_2\sigma_4*$
$\_\,\sigma_5\sigma_5$	σ_2*
$\_\,\sigma_4$	$\sigma_2\sigma_4*$
$\sigma_{\mathbf{5}}\sigma_5$	$\sigma_4\sigma_2$
σ_4	$\sigma_3\sigma_2\sigma_4$
$\_\_\,\sigma_5$	σ_4*
	$\sigma_4\sigma_3\sigma_2$
$\_\,\sigma_5$	σ_4*
	$\sigma_4\sigma_3\sigma_2$
σ_5	$\sigma_5\sigma_4$
	$\sigma_4\sigma_3\sigma_2$
	$\sigma_5\sigma_4$

Figure 7.1: Example of execution of shared LRU and a static partition strategy with LRU and partition $B = \{3, 2\}$ on the input $\mathcal{R} = \{R_1, R_2\}$, with $R_1 = \sigma_1\sigma_2\sigma_4\sigma_2\sigma_3\sigma_4$ and $R_2 = \sigma_5\sigma_6\sigma_2\sigma_4\sigma_5\sigma_5$. The cache size is $k = 5$ and $\tau = 2$. Pages in bold denote faults, and a '$\_\_$' indicates a timestep in which a page is being fetched. Pages in the caches are shown from left to right in order of most recent use, and a '$*$' in the cache indicates that the cell will be used by a page currently being fetched. The number of faults incurred by S_{LRU} and P^B_{LRU} are 3 and 5, respectively.

Lemma 7.1 for marking and *dynamically conservative* algorithms[4] for dynamic partitions as well. Peserico studies the performance of cache eviction policies (for one request sequence) when the cache size can vary from 1 to a maximum size k. In this setting, he shows that marking and dynamically conservative algorithms are at most k competitive [Peserico, 2013]. This result leads to the following lemma:

Lemma 7.2 (Online vs. offline eviction policies with a fixed dynamic partition) *Let D be any online dynamic partition and let A be any marking or dynamically conservative algorithm. Then, $\forall\ \mathcal{R}$, $P_A^D(\mathcal{R})/P_{OPT}^D(\mathcal{R}) \leq pk$.*

Proof: The dynamic partition D defines p instances of paging with varying cache size in which the online and offline eviction policies have equal cache sizes at each timestep. Let $P_A^D(R_i)$ denote the number of faults of A on sequence R_i under partition D. By Theorem 4 in [Peserico, 2013], $P_A^D(R_i) \leq kP_{OPT}^D(R_i)$ for all $1 \leq i \leq p$. Adding the number of faults in all sequences yields the lemma. ∎

Lemmas 7.1 and 7.2 show that if the cache partition is determined externally and independently of the eviction policy (e.g., by a scheduler or the operating system), then traditional eviction policies that have are competitive in the sequential setting are also competitive in the multi-core scenario.

We now consider both partition and eviction policy as part of the cache management strategy. In this setting, we show that the choice of a good partition has more influence on the performance of static partition strategies than the eviction policy. In fact, if the partition function can be computed offline, then no online static partition strategy is competitive, even with an offline eviction policy.

Lemma 7.3 (Online static partition strategies are not competitive) *Let $B = \{k_1, ..., k_p\}$ be any online static partition. Let $k^* = \min_j\{k_j | k_j \geq 2\}$. Then there exists a sequence $\mathcal{R}$ such that for all eviction policies A, $P_A^B(\mathcal{R})/P_{LRU}^{OPT}(\mathcal{R}) \geq \min\{k^* - 1, p - 1\} \cdot \frac{n}{k^2 p} = \Omega(n)$.*

Proof: Consider first A=LRU. Let $j^* = \text{argmin}_j\{k_j | k_j \geq 2\}$ (i.e., $k_{j^*} = k^*$). Let P denote the set of the first $(k_{j^*} - 1)$ processors, not including j^*, in decreasing order by size of part of the cache according to B. Note that if $k_{j^*} \geq p$, then $P \cup \{j^*\}$ is equal to the set of all processors. Let $R_j = (\sigma_1^j \sigma_2^j ... \sigma_{k_j+1}^j)^{x_j}$ with x_j such that $x_j(k_j + 1) = n/p$ for all $j \in P$, and let $R_j = (\sigma_1^j \sigma_2^j ... \sigma_{k_j}^j)^{x_j}$ $j \notin P$ and $j \neq j^*$, where x_j is such that $x_j k_j = n/p$. Let $R_{j^*} = (\sigma_1^{j^*})^{n/p}$.

[4]Dynamically conservative algorithms form a subset of conservative algorithms (see Definition 6.5) that still includes LRU, FIFO, and Clock [Peserico, 2013].

P^B_{LRU} faults on every request of $|P|$ processors and faults only on the first request of processor j^*. Hence, $P^B_{LRU}(\mathcal{R}) \geq \min\{k_{j^*} - 1, p-1\} \cdot n/p$.

On the other hand, an optimal partition for $\mathcal{R}$ would be one such that all different pages of each request R_j fit in the cache. Intuitively, an optimal partition takes units of cache from j^* and assigns them to other processors. Let k_j^{OPT} denote the size of the cache for processor j according to the optimal partition, then $k_j^{OPT} = k_j + 1$ if $j \in P$, $k_j^{OPT} = \min\{1, k_j - (p-1)\}$ for $j = j^*$, and $k_j^{OPT} = k_j$ otherwise. The number of faults of P^{OPT}_{LRU} on $\mathcal{R}$ is k, since it only faults on the first request to each different page. Hence, $P^B_{LRU}(\mathcal{R})/P^{OPT}_{LRU}(\mathcal{R}) \geq \min\{k_{j^*} - 1, p-1\} \cdot n/(kp)$. We have shown a bound for an online static partition using LRU. Now, by Lemma 7.1, for any eviction policy A, $P^B_A(\mathcal{R}) \geq \frac{P^B_{LRU}(\mathcal{R})}{k}$, and the lemma follows. ∎

Although strategies that partition the cache only once or even allow a small number of changes during the execution might be simpler to manage as compared to general strategies, the performance of these strategies is not competitive when shared strategies are allowed. While this is perhaps to be expected for non-disjoint sequences, interestingly, this holds even for disjoint sequences. Theorem 7.1 shows that shared strategies are preferable over static partitions —even if the partition is computed offline— as well as dynamic partitions that do not change often.

Theorem 7.1 *Let A be any deterministic online cache eviction policy, let $sOPT$ be an optimal static partition, and let D be any online dynamic partition strategy that changes the sizes of the parts $o(n)$ times. The following statements hold:*

1. *There exists a sequence $\mathcal{R}$ such that $\frac{P^{sOPT}_{OPT}(\mathcal{R})}{S_{LRU}(\mathcal{R})} = \Omega(n)$.*
2. *For all sequences $\mathcal{R}$, $S_{LRU}(\mathcal{R})/P^{sOPT}_{OPT}(\mathcal{R}) \leq k$.*
3. *There exists a sequence $\mathcal{R}$ such that $P^D_A(\mathcal{R})/S_{LRU}(\mathcal{R}) = \omega(1)$. Furthermore, if D varies the partition a constant number of times, $P^D_A(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n)$.*

Proof: 1. Consider a sequence of requests $\mathcal{R}$, in which processor j requests the following pages, for all j simultaneously:

$$(\sigma^j_1)^{\alpha_j}(\sigma^j_1\sigma^j_2 \ldots \sigma^j_{k/p+1})^x(\sigma^j_1)^{\beta_j}$$

where $\sigma^j_{i_1} \neq \sigma^j_{i_2}$ for all $i_1 \neq i_2$, $\alpha_j = (j-1)(k/p+1)(\tau+x)$, $\beta_j = (k+p-j(k/p+1))(\tau+x)$, and x is a parameter. In other words, processor j requests the same page for a while, then repeatedly requests $k/p+1$ distinct pages (call this the *distinct period*), and then goes back to requesting the same page again. All processors do the same, taking turns

to be the processor currently in the distinct period: when one processor is in the distinct period, all other processors request repeatedly the same page. Given the request sequence, an optimal partition assigns $k/p+1$ units of cache to $p-1$ processors, and the rest to one processor: assigning more than $k/p+1$ units of cache to any processor does not result in fewer faults, and assigning less than $k/p+1$ to more than one processor increases the number of faults. Let j^* be the processor whose partition is $k_{j^*} = k/p-(p-1)$. Consider the distinct period of this processor. Let A be any eviction policy. No matter what the eviction policy A is, even the optimal offline, P_A^{sOPT} will fault at least once every k_{j^*} requests. Hence, $P_A^{sOPT}(\mathcal{R}) \geq x(k/p+1)/k_{j^*}$. On the other hand, $S_{LRU}(\mathcal{R})$ faults only on the first $k/p+1$ requests of the distinct period of each processor, for a total of $k+p$ faults. Hence, $P_A^{sOPT}(\mathcal{R})/S_{LRU}(\mathcal{R}) \geq x/(pk_{j^*})$. x can be made arbitrarily large, in fact, $n = \tau(k+p)(p-1)+xp(k+p)$, and thus $x = n/(p(k+p))+\tau(p-1)/p$. Hence, $x/(pk_{j^*}) = \Omega(n)$.

2. Divide a sequence R_j of processor j in phases such that in a sequential traversal of pages, a new phase begins either on the first page or at the (k_j+1)-th different page since the beginning of the current phase, where k_j is the size of the cache assigned by $sOPT$ to processor j. Define a shared phase equivalently for the cache size k and the sequence $\mathcal{R}'$ containing the pages of $\mathcal{R}$ in the order in which they are requested during the execution of S_{LRU} (with simultaneous requests sorted by increasing number of processor). We claim that a shared phase cannot start and end without at least one sequence changing phase. In other words, the phase of at least one sequence must end before the end of a shared phase. If this was not the case, within the shared phase, the number of different pages in the sequence of each processor j would be at most k_j, and therefore the total number of different pages in the shared phase would be at most k, which is a contradiction. Let ϕ denote the number of shared phases of sequence $\mathcal{R}$ and ϕ_j denote the number of phases of sequence R_j. The above claim implies that $\phi \leq \sum_{j=1}^{p} \phi_j$. Assuming that S_{LRU} timestamps each page at the moment of request, it is not difficult to see that the fact that S_{LRU} faults at most k times per phase in the sequential setting extends to the above definition of shared phases. Since any cache eviction algorithm must fault at least once per phase, it follows that $S_{LRU}(\mathcal{R}) \leq k\phi \leq k\sum_{j=1}^{p} \phi_j \leq kP_{OPT}^{sOPT}(\mathcal{R})$.

3. Let a stage of D denote a period in which the sizes of the partition are constant. If the number of stages of D is $o(n)$, then at least one stage has non-constant length $\ell = \omega(1)$ (in number of parallel page requests). We then apply the same argument as in the proof of statement 1. Let $\mathcal{R}$ in this stage consist of a sequence in the form of the sequence in that proof: each processor's sequence has three periods: (1) only one page σ_1^j is requested repeatedly, (2) the page requested is any page not in the cache of processor j (the *distinct period*), and (3) again only one page σ_1^j is requested. The length of period (2) is m pages, and each processor takes turns to be in the distinct period. Hence, the total number of

requests in the stage is $mp^2 = \ell p$. Let t be the time where the long stage begins. During the distinct period of processor j, R_j consists of repeatedly requesting the page not in j's cache, among pages $\{\sigma_1^j, ..., \sigma_{k(j,t)+1}\}$, where $k(j,t)$ is the size of the cache of processor j at time t. P_A^D faults on every request of the distinct period of all processors, and hence in this stage $P_A^D(\mathcal{R}) = pm = \ell$. On the other hand, in this stage, S_{LRU} faults only on the first request to a distinct page in the distinct period of each processor, and thus in this stage $S_{LRU}(\mathcal{R}) = k+p$ (recall we assume $k_j \geq 1$ for all $1 \leq j \leq p$ at all times). Let the rest of $\mathcal{R}$ be such that neither algorithm faults. Then $P_A^D(\mathcal{R})/S_{LRU}(\mathcal{R}) \geq \ell/(k+p) = \omega(1)$. Note that if partitions are allowed only a constant number of stages, then $P_A^D(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n)$. ∎

Theorem 7.1 suggests that competitive strategies must either be shared or have a partition that changes often. In fact, both types of strategies are equivalent for disjoint sequences. Although a dynamic partition strategy executes an eviction policy in each part separately, if the variation in the partition can be determined globally, then shared strategies can be simulated by a dynamic partition on disjoint sequences by reducing the part of the cache holding the page to be evicted. This implies that a dynamic partition strategy can be as effective as any other strategy. In fact, Hassidim [2010] showed in his model that there exists an optimal dynamic partition strategy that upon a fault reduces the part of some processor, evicting the page that is furthest in the future in that processor's sequence. We show in Section 7.3 that the same result holds in our model.

Multi-core paging differs from sequential paging in that the actions of algorithms modify the order of future requests. Hence, paging strategies must decide which page to evict not only with the goal of delaying further faults, but at the same time trying to properly align the demand periods of future requests. An online strategy, however, is oblivious to future requests and hence in general it cannot work toward the second goal. In this sense, in multi-core paging an optimal offline strategy has significantly more advantages over an online strategy than in sequential paging. While in the latter setting any online marking algorithm has a bounded competitive ratio of $\frac{k}{k-h+1}$ when the offline algorithm has a cache of size $h \leq k$ [Karlin et al., 1988; Torng, 1998], in multi-core paging the competitive ratio of traditional algorithms such as LRU, FIFO, Clock, and FWF can be arbitrarily large. Although an optimal offline strategy cannot explicitly schedule requests, it can increase the fault rate of a process thus effectively delaying that sequence in order to align periods of high demand with periods of low demands of other processors. The following theorem shows that the competitive ratio of S_{LRU} can grow proportionally to the square root of the length of the sequences, even when the offline strategy has a cache of about half the size. The theorem also applies to FIFO, Clock, and FWF.

Theorem 7.2 Lower bound on the competitive ratio of LRU. *Assume $p \geq 4$ and $\tau > 0$. There exists a sequence $\mathcal{R}$ and an offline eviction policy A such that $S_{LRU}(\mathcal{R})/S_A(\mathcal{R}) = \Omega(\sqrt{n\tau/k})$ when A's cache size is $h \geq k/2 + 3p/2$.*

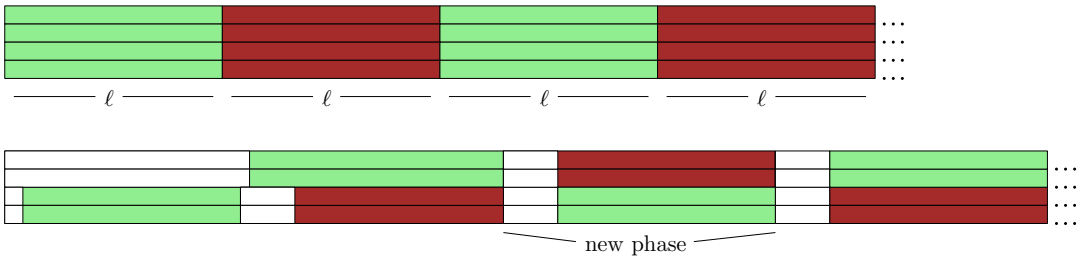


Figure 7.2: Top: A request with 4 sequences that alternates phases of low and high demand. For each sequence, green (light) phases consist of alternating requests to two pages, while red (dark) phases are consecutive requests of $k/p+1$ pages. Bottom: sequence after being served by the offline strategy S_A. White periods denote faults. After the initial faults each new phase has at most h distinct pages, and $k+p$ faults are necessary in each new phase to keep future alignments.

Proof: The request sequence $\mathcal{R}$ consists of two alternating phases which we call easy and hard, respectively (see Figure 7.2, top). In an easy phase, each sequence requests only 2 different pages, while in a hard phase, the total number of different pages requested is greater than k. More specifically, let

$$R_j = \left((\sigma_a^j \sigma_b^j)^{\ell/2} (\sigma_1^j \sigma_2^j \dots \sigma_{k/p+1}^j)^{\ell/(k/p+1)}\right)^{\phi}$$

The length of each phase is ℓ pages and there are 2ϕ phases. The number of pages in a hard phase is $k+p$, and since every request is to the least recently used page, S_{LRU} faults on every request of a hard phase. Thus, $S_{LRU}(\mathcal{R}) \geq n/2$, where n is the total number of pages.

An offline algorithm A can benefit from aligning hard phases of some sequences with easy phases of others in order to keep the total number of requested pages below h in the aligned periods. A does this by initially assigning only one cache cell to each sequence in a group $\mathcal{R}_1 = \{R_1, \dots, R_{p/2}\}$. Hence, every request in these sequences is a fault. Since A will fault only on the first two requests of the rest of the sequences $\mathcal{R}_2 = \{R_{p/2+1}, \dots, R_p\}$, sequences in $\mathcal{R}_1$ will be delayed with respect to sequences in $\mathcal{R}_2$. A will delay sequences in $\mathcal{R}_1$ so that the last page of their easy phase is aligned with the last page of the hard phase of sequences in $\mathcal{R}_2$. Consider a sequence $R_i \in \mathcal{R}_2$. The total number of pages requested during the first hard phase of R_i is $(p/2)(k/p+1)+p \leq h$, and thus A faults only on the first $k/p+1$ requests of R_i in its first hard phase. Since A also faults twice in the first easy phase of R_i, the first hard phase of R_i is completed at time $t = 2\tau + \ell + \tau(k/p+1) + \ell = 2\ell + \tau(3+k/p)$. Therefore, A needs to incur $\ell/\tau + k/p + 3$ faults in each of the sequences in $\mathcal{R}_1$ in order for their ℓ-th request to be served at time t.

After the initial faults, easy phases of sequences in $\mathcal{R}_1$ are aligned with hard phases of sequences in $\mathcal{R}_2$ and vice versa (see Figure 7.2, bottom). Call each of these ℓ-page phases a new

phase. Since each new phase has at most h different pages, A faults only at the beginning of these phases. A keeps the alignment of sequences by partitioning the cache so that the first $k/p+1$ requests of all sequences are faults. Since any one sequence has at most $k/p+1$ distinct pages in a new phase, $p(k/p+1) \leq 2k$ faults are enough to maintain the alignment. These faults, plus the initial faults to arrange the alignment, add up to $A(\mathcal{R}) \leq 4k\phi + (p/2)(\ell/\tau + k/p + 3)$. The total number of pages is $n = \phi\ell p$, and thus $\ell = n/(\phi p)$. Substituting ℓ in the number of faults and minimizing for ϕ yields $\phi = \sqrt{n/(8k\tau)}$, and hence $A(\mathcal{R}) \leq 4\sqrt{nk/3\tau} + k/2 + 3p/2 = O(\sqrt{nk/\tau})$. Since $S_{LRU}(\mathcal{R}) \geq n/2$, it follows that $S_{LRU}(\mathcal{R})/S_A(\mathcal{R}) = \Omega(\sqrt{n\tau/k})$. ∎

Note that if the offline algorithm has a cache of size k, the lower bound in Theorem 7.2 can be made even larger: the offline algorithm delays only two sequences, and the competitive ratio becomes $\Omega(\sqrt{n\tau p/k})$. The proof of Theorem 7.2 can be used to show that not only Furthest-In-The-Future (FITF) [Belady, 1966] —an optimal algorithm in classical paging— is not optimal in multi-core paging, but that it performs badly compared to the true optimal.

Corollary 7.1 *Assume $p \geq 4$ and $\tau > 0$. There exists a sequence $\mathcal{R}$ and an offline eviction policy A such that $S_{FITF}(\mathcal{R})/S_A(\mathcal{R}) = \Omega(\sqrt{n\tau}/k^{3/2})$ when A's cache size is $h \geq k/2 + 3p/2$.*

Proof: Let $\mathcal{R}$ be the sequence given in the proof of Theorem 7.2. The number of faults of FITF is at least $n/2k$, and thus $S_{FITF}(\mathcal{R})/S_{OPT}(\mathcal{R}) = \Omega(\sqrt{n\tau}/k^{3/2})$. ∎

7.3 The Offline Problem

In reality, requests sequences are not known in advance, and thus paging is an online problem. In general, however, in any online problem setting deriving efficient optimal offline solutions is both of theoretical interest as well as useful in evaluating online algorithms in practice in the competitive analysis framework. Furthermore, an online solution can be designed based on properties of the offline solution. For example, in traditional paging, LRU approximates FITF using the past as the best approximation of the future. An example of an inherently online problem for which the offline problem has been extensively studied is the list update problem (see, e.g., [Reingold and Westbrook, 1996; Munro, 2000; Ambühl, 2000; Hagerup, 2007]).

In this section we study the offline version of the multi-core paging problem. We first show hardness results for PARTIAL-INDIVIDUAL-FAULTS and its optimization version. We then show properties of optimal offline algorithms for FINAL-TOTAL-FAULTS. Finally, we present polynomial time algorithms for both PIF and FTF when the number of sequences and cache size are constants.

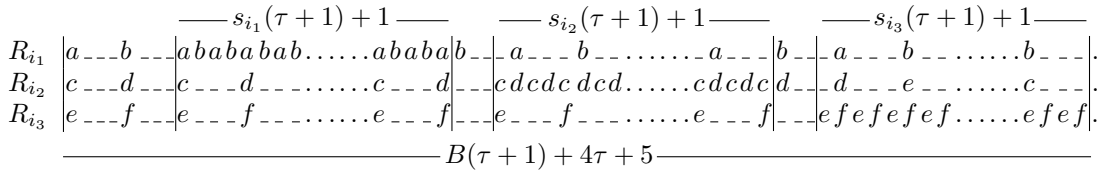


Figure 7.3: Sequences $\mathcal{R}_{i_1}, R_{i_2}, R_{i_3}$ share 4 cells of the cache. In the example, a, b, c, d, e, f are all different pages, $\tau = 3$, and the fetching period is represented by τ consecutive underscore characters (_). Each sequence R_i holds the extra cell continuously so that it incurs $s_i(\tau+1)$ hits. The number of faults of each R_i in the group is $B - s_i + 4$ at time $t = B(\tau+1) + 4\tau + 5$ if and only if $s_{i_1} + s_{i_2} + s_{i_3} = B$.

7.3.1 Hardness of Multi-Core Paging

In this section we show that even if the sequence of requests is known in advance multi-core paging is hard when we are required to satisfy sequences' individual fault bounds. More specifically, we show that PARTIAL-INDIVIDUAL-FAULTS (PIF, see Definition 7.2) is NP-complete and that there is no Polynomial Time Approximation Scheme (PTAS) for its optimization version. This is in contrast to sequential paging: if there is only one processor, both FTF and PIF are solvable by FITF. As in the proof of Hassidim's makespan problem [Hassidim, 2010], the proof of NP-completeness of PIF uses a reduction from 3-PARTITION. However, since our model disallows explicit scheduling, the reduction is quite different.

Definition 7.3 (3-PARTITION [**Garey and Johnson, 1979**]) *Given a set of integers $S = \{s_1, \ldots, s_n\}$, and a bound B, such that $B/4 < s_i < B/2$ for all $1 \leq i \leq n$, the problem is to determine if S can be partitioned into $n/3$ sets $A_1, \ldots, A_{n/3}$ such that for all $1 \leq j \leq n/3$, $\sum_{i \in A_j} s_i = B$. The restrictions of the problem imply that each subset A_i must have exactly 3 elements.*

3-PARTITION is NP-complete in the strong sense; it remains NP-complete if all integers in S are bounded by a polynomial in the size of the instance or, equivalently, if the input is encoded in unary [Garey and Johnson, 1978].

Theorem 7.3 *PARTIAL-INDIVIDUAL-FAULTS is NP-complete.*

Proof: It is easy to see that the problem is in NP: given an instance $\mathcal{I} = \{\mathcal{R}, k, t, \tau, \vec{b}\}$ with a "yes" answer and a certificate consisting of the pages to be evicted after each fault, it can be verified in time $O(tp)$ that the number of faults in each sequence R_i is at most b_i. Note that $t \leq \max_i\{|R_i|\}(\tau+1)$, and τ is a constant, therefore, the verifier runs in time polynomial in the size of the input.

In order to show that the problem is NP-complete, we build a reduction from 3-PARTITION (using the terminology in Definition 7.3). Let $\mathcal{J} = \{S, B\}$ be an instance of 3-PARTITION. We build an instance $\mathcal{I}$ of PARTIAL-INDIVIDUAL-FAULTS as follows. There are $p = |S|$ sequences. Each sequence R_i consists of alternating requests to 2 pages α^i and β^i, where $\alpha^i \neq \beta^i$, and $\alpha^i \neq \alpha^j$ and $\beta^i \neq \beta^j$ for all $i \neq j$, and $\alpha^i \neq \beta^j$ for all i, j. In other words, $R_i = \alpha^i\beta^i\alpha^i\beta^i \ldots$, and all sequences are disjoint. The length of R_i is $|R_i| = B(\tau+1) + 4\tau + 5$, where $\tau \geq 0$ is any integer. The size of the cache is $k = (4/3)p$, and we want to know if the number of faults in each sequence R_i is at most $b_i = B - s_i + 4$ at time $t = B(\tau+1) + 4\tau + 5$. Note that since 3-PARTITION is strongly NP-complete, the reduction can be done from an instance encoded in unary, and hence it can be done in time polynomial in the size of $\mathcal{J}$.

We show now that there exists a solution for $\mathcal{J}$ if and only if we can serve each R_i with at most b_i faults.

($\Rightarrow$) We show first that if $\mathcal{J}$ admits a solution, then we can serve each R_i with at most b_i faults. Let $A_1, \ldots, A_{n/3}$ be the partition for $\mathcal{J}$. Divide the sequences in groups according to the partition, so that the sequences corresponding to A_j will share a group of 4 cells of the cache. Let R_{i_1}, R_{i_2}, and R_{i_3} be the sequences in group j. Each of these sequences will be assigned one cell for some time and two at other times. In other words, the three sequences will have one dedicated cell at least until time t and will share the extra cell of the group. Sequence R_i will use the extra cell continuously for enough time so it incurs exactly $h_i = s_i(\tau+1) + 1$ hits (see Figure 7.3).

Say R_{i_1}, R_{i_2}, and R_{i_3} use the extra cell in that order. The first request to each sequence results in a fault, and it is fetched to the dedicated cell of the corresponding sequence. The second request of R_{i_1} (also a fault) is fetched to the extra cell. Now, both pages of R_{i_1} are in the cache, and they are kept there for the next h_{i_1} requests of R_{i_1}. Meanwhile, every request of R_{i_2} and R_{i_3} results in a fault and the eviction of the page in their corresponding dedicated cell. The last hit of R_{i_1} occurs at time $(2 + s_{i_1})(\tau+1) + 1$, which coincides with a new request σ for R_{i_2}, since all pages have been faults for R_{i_2}. Instead of fetching σ to this sequence's dedicated cell, σ is fetched into the extra cell or R_{i_1}'s dedicated cell, depending on which page can be evicted at the time (if σ is fetched into R_{i_1}'s dedicated cell, then this cell becomes the shared cell, and the former shared cell becomes R_{i_1}'s dedicated cell). Now, R_{i_2} has the extra cell, and the remaining requests of R_{i_1} will result in faults. After h_{i_2} hits of R_{i_2}, the extra cell is now passed to R_{i_3} (again the last hit of R_{i_2} coincides with a request of R_{i_3}), and this sequences keeps this cell until it completes h_{i_3} hits. At this point, the time elapsed is the sum of the hits of each sequence, plus 2τ for the transitions of the extra cell from R_{i_1} to R_{i_2} and from R_{i_2} to R_{i_3}, plus the initial $2(\tau+1)$ time corresponding to the first 2 faults of the three sequences. Hence, the time is $t = h_{i_1} + h_{i_2} + h_{i_3} + 4\tau + 2 = (s_{i_1} + s_{i_2} + s_{i_3})(\tau+1) + 4\tau + 5 = B(\tau+1) + 4\tau + 5$. The same strategy is used for each group in the partition, and the number of faults of sequence R_i is exactly $(t - h_i)/(\tau+1) = B - s_i + 4$. Thus, if there is a solution to $\mathcal{J}$, $\mathcal{R}$ can be served such that at time $t = B(\tau+1) + 4\tau + 5$, each sequence has incurred $b_i = B - s_i + 4$ faults.

($\Leftarrow$) We show now that a solution to the instance $\mathcal{I}$ of PIF gives a solution to the instance $\mathcal{J}$ of 3-PARTITION. If $\mathcal{R}$ can be served so that each sequence faults at most $B - s_i + 4$ times by time t, then at least $h_i = s_i(\tau + 1) + 1$ of R_i's requests must be hits. Note that for all i, $|R_i| = t$, and hence each sequence uses at least one cell until time at least t. A request can only be a hit if a sequence has two cells of the cache for consecutive timesteps. Since there are only $(4/3)p$ cells, and each sequence uses at least one cell, there are only $p/3$ extra cells which can be used to store the second page of a sequence, and hence there can be at most $p/3$ hits in one timestep.

In addition, any change in the partition that removes one cell from a sequence that had 2 cells and gives one more cell to another sequence implies at least τ time without hits for the sequences involved. To see this, let R_i and R_j be two sequences such that $k(i, t_1) = 2$ and $k(j, t_1) = 1$, respectively, and $k(i, t_1 + 1) = 1$ and $k(j, t_1 + 1) = 2$ (where $k(i, t_1)$ is the size of the cache of processor i at time t_1), for some $t_1 < t$. Say page α^i was a hit in R_i. Then, at $t_1 + 1$, β^i and α^j are requested in R_i and R_j, respectively[6]. α^i is evicted from the cache, and α^j is fetched in to the cell previously used by α^i. Sequence R_j must wait τ more timesteps before having its first hit, while sequence R_j's next request (α^i) will result in a fault. Hence, there are no hits in these two sequences in the period $[t_1 + 2, t_1 + 1 + \tau]$, i.e., τ timesteps without hits.

Let C denote the number of all such changes in partition between a pair of sequences. C does not count the initial assignment of 2 cells to a sequence when starting to serve $\mathcal{R}$, but only when a sequence acquires an extra cell that held a page of another sequence. Note that no hits can happen until time $2(\tau + 1) + 1$, and hence the total number of possible hits before time t can be at most $H_1 = (p/3)(t - 2(\tau + 1)) - C\tau = (p/3)((B - 2)(\tau + 1) + 4\tau + 5) - C\tau$. On the other hand, the minimum number of required hits is $H_2 = \sum_{i=1}^{p} h_i = \sum_{i=1}^{p} s_i(\tau + 1) + 1 = (p/3)B(\tau + 1) + p$. We require $H_1 \geq H_2$, which implies that $C \leq 2p/3$, i.e., we can have at most $2p/3$ changes in partitions. Note that since the total number of cells is $4p/3$, and every sequence is assigned at least one cell, initially at most $p/3$ sequences can be assigned two cells. Therefore, in order for each sequence to be assigned two cells at least once, every sequence must have 2 cells during a continuous period of time. We call this period the hit period of a sequence.

Consider a group of sequences whose hit periods are consecutive, i.e., a group $I = \{i_1, i_2, ...i_\ell\}$ of the sequences such that a request in sequence i_{j+1} evicts a page from sequence i_j to start its hit period. These sequences can be served using a minimum of $\ell + 1$ cells: one cell is dedicated for each sequence, while the other extra cell is used to assign 2 cells to some sequence during its hit period (this extra cell need not to be the same one during the entire execution). We claim that $\ell \leq 3$. Too see this, assume $\ell > 3$; then, since hit periods have no interruptions, the total time to serve these sequences is at least the total number of hits, plus the time for each change in partition, plus 2 initial faults, for a total of $T = \left(\sum_{i \in I} h_i\right) + (\ell - 1)\tau + 2(\tau + 1) = \left(\sum_{i \in I} s_i(\tau + 1) + 1\right) + (\ell + 1)\tau + 2 > (\ell/4)B(\tau + 1) + \ell + (\ell + 1)\tau + 2$, since $s_i > B/4$ for all

[6]Note that R_j might be fetching a page instead, but the case when a new request comes is the one that minimizes the time that elapses from t_1 until R_j's next hit.

$1 \leq i \leq p$. Taking $\ell = 4$, $T > B(\tau + 1) + 5\tau + 6$, which is strictly greater than t, and thus one sequence will not have all its required hits before t and hence it will exceed the allowed number of faults. Therefore, $\ell \leq 3$. Furthermore, we argue that ℓ is exactly 3. Assume, otherwise, that sequences are served in groups of 1, 2, and 3 sequences. Let n_ℓ be the number of groups of ℓ sequences, for $\ell \in \{1, 2, 3\}$. In order to satisfy the minimum number of hits for each sequence, it must be the case that a group of ℓ sequences must use at least $\ell + 1$ cells. Hence, the following must be satisfied: $n_1 + 2n_2 + 3n_3 = p$, and $2n_1 + 3n_2 + 4n_3 \leq (4/3)p$, with $n_1, n_2, n_3 \geq 0$. It is not difficult to see that a feasible solution must have $n_1 = n_2 = 0$, and R must be served only with groups of 3 sequences.

Finally, it must be the case that every group of sequences $I = i_1, i_2, i_3$ must satisfy $s_{i_1} + s_{i_2} + s_{i_3} = B$. Suppose that a group I_1 is such that $\sum_{i \in I_1} s_i < B$. Then, since $B = (3/p) \sum_{i=1}^{p} s_i$, there must exist another group I_2 such that $\sum_{i \in I_2} s_i > B$. Then, since the minimum number of hits per sequence is $h_i = s_i(\tau + 1)$, the total time to serve the group would be $T = \left(\sum_{i \in I_2} h_i\right) + 2\tau + 2(\tau + 1) > B(\tau + 1) + 4\tau + 5 = t$, and thus at least one sequence in the group would have to fault more than its maximum number of allowed faults by time t. Therefore, the only possible way to serve the requests satisfying the faults requirement for each sequence is to divide them in groups of 3 sequences $I_1, \ldots, I_{p/3}$ such that $\sum_{i \in I_j} s_i = B$ for all $1 \leq j \leq p/3$, which is a solution to the instance of 3-PARTITION. ∎

Observe that PIF remains NP-complete even when $\tau = 0$, i.e., when sequences are not delayed due to faults. The case $\tau = 0$ has been referred to as the *fixed interleaving* case by Katti and Ramachandran [2012], to contrast it with the *free interleaving* case that we consider here and that was introduced by Hassidim [2010]. Fixed interleaving was also considered in the multiapplication model studied by Cao et al. [1994] and Barve et al. [2000] (see Section 6.2.4). Hence, PIF is NP-complete in this model as well. Note that this is not the case for FINAL-TOTAL-FAULTS, for which FITF is optimal when $\tau = 0$, as in this case this problem reduces to sequential paging.

Recall that τ models the difference between the latencies of a cache miss and hit, whose ratio is approximately 17 for main memory and L2 cache [Hassidim, 2010]. This difference in latencies is also present in sequential paging; in fact, the traditional page fault model considers τ as infinite. However, the fact that after a cache miss the execution of the underlying program must wait until the page is fetched does not have implications in the analysis of the paging problem abstraction, as the order of page requests is not altered. The same considerations apply for the case in which various sequences share a cache, but requests are served sequentially, thus all sequences must wait until a page of any one sequence is fetched after a cache miss, which is the case of the multiapplication cache model [Cao et al., 1994; Barve et al., 2000]. In the case of our model for multi-cores, we generally think of τ being a constant greater than zero, as otherwise the parallelism offered by the architecture would be greatly diminished if threads on all cores were paused each time a single thread incurs a cache miss.

The fact that PIF remains NP-complete even in the intuitively simpler case of $\tau = 0$, while FTF admits a simple offline algorithm, provides evidence that achieving a fair distribution of faults is more difficult than merely minimizing the number of overall faults. It remains open to determine whether FTF can be solved in polynomial time when $\tau > 0$. As we shall see in Section 7.3.3, both FTF and PIF can be solved in polynomial time in the length of the sequences when the number of sequences is constant.

We show now that PIF is also hard to approximate. We define as MAX-PARTIAL-INDIVIDUAL-FAULTS (MAX-PIF) the problem of, given an instance of PIF, maximizing the number of sequences whose number of faults at a given time is within the given bound. We show that MAX-PIF is APX-hard, i.e., there is no polynomial time algorithm that can approximate this problem within a factor of $(1 - \epsilon)$ for every ϵ, unless P = NP. In order to show this, we describe a gap-preserving reduction from MAX-4-PARTITION (shown to be APX-hard by Cieliebak et al. [2003]). Therefore, unless P = NP, there is no efficient way of serving the request sequences ensuring that an arbitrarily large part of them will fault within the allowed bounds.

Theorem 7.4 *MAX-PARTIAL-INDIVIDUAL-FAULTS is* APX*-hard.*

Proof: We describe a gap preserving reduction from MAX-4-PARTITION to MAX-PIF. Let us first define the 4-PARTITION problem. 4-PARTITION is an analog of 3-PARTITION in which the goal is to partition a set $S = \{s_1, \ldots, s_n\}$ into subsets $A_1, \ldots, A_{n/4}$ such that for all $1 \leq j \leq n/4$, $\sum_{i \in A_j} s_i = B$, where $B = (4/n) \sum_{i=1}^{n} s_i$ [Garey and Johnson, 1979]. Each element s_i satisfies $B/5 < s_i < B/3$, and thus each subset must have 4 elements. 4-PARTITION is also NP-complete [Garey and Johnson, 1979], and a reduction to PIF can be built by modifying the proof of Theorem 7.3 in a straightforward way: the cache size is now $k = (5/4)p$, the length of each sequence is $B(\tau + 1) + 5\tau + 6$, and the goal is to serve the sequences such that at time $t = B(\tau + 1) + 5\tau + 6$ sequence i has incurred at most $b_i = B - s_i + 5$. It is not hard to see that the same arguments in the proof of Theorem 7.3 apply to argue that an instance of 4-PARTITION admits a solution if and only if the instance of PIF admits a solution.

The MAX-4-PARTITION problem (as defined in [Cieliebak et al., 2003]) is: given a set S and B as in the 4-PARTITION problem, find a maximum number of disjoint subsets whose elements add up to B. This problem is APX-hard, i.e., it does not admit a PTAS (assuming P $\neq$ NP)[Cieliebak et al., 2003]. For an instance $\mathcal{J}$ ($\mathcal{I}$) of MAX-4-PARTITION (MAX-PIF), let $OPT_{4PART}(\mathcal{J})$ ($OPT_{PIF}(\mathcal{I})$) denote the value of the optimal solution to $\mathcal{J}$ ($\mathcal{I}$). Let $n = |S|$ in $\mathcal{J}$. In order to show that MAX-PIF is APX-hard, we build a reduction from a given instance $\mathcal{J}$ of MAX-4-PARTITION to an instance $\mathcal{I}$ of MAX-PIF and show:

1. $OPT_{4PART}(\mathcal{J}) \geq n/4 \Rightarrow OPT_{PIF}(\mathcal{I}) \geq n$
2. $OPT_{4PART}(\mathcal{J}) < (1 - \epsilon)n/4 \Rightarrow OPT_{PIF}(\mathcal{I}) < (1 - \epsilon/4)n$

The reduction from an instance $\mathcal{J}$ of MAX-4-PARTITION to an instance $\mathcal{I}$ of MAX-PIF is exactly the same as the reduction from 4-PARTITION described above. Since a solution to $\mathcal{J}$ gives a solution to $\mathcal{I}$, if $OPT_{4PART}(\mathcal{J}) \geq n/4$ (and thus equal to $n/4$), then all sequences of $\mathcal{I}$ can be served with a number of faults within the given bounds, proving statement (1).

For statement (2), note that in the reduction from 4-PARTITION to PIF (adapted from the proof of Theorem 7.3) the only way of serving all sequences within the fault bounds is by partitioning them in groups of 4 that share 5 cells of cache. Furthermore, the 4 sequences in a group can be served within the faults bounds if and only if the corresponding elements in S add up to B. Otherwise, at least one of the sequences will have to incur more faults than the allowed bound. Therefore, $OPT_{PIF}(\mathcal{I}) \leq 4OPT_{4PART}(\mathcal{J}) + 3(n/4 - OPT_{4PART}(\mathcal{J})) = OPT_{4PART}(\mathcal{J}) + 3n/4$. Since $OPT_{4PART}(\mathcal{J}) < (1-\epsilon)n/4$, we have $OPT_{PIF}(\mathcal{I}) < (1-\epsilon)n/4 + 3n/4 = (1-\epsilon/4)n$. Thus, the reduction is gap-preserving, proving the theorem. ∎

7.3.2 Properties of Offline Algorithms for FTF

The changes in the relative alignment of sequences can significantly affect the performance of an algorithm (see the proof of Theorem 7.2 for an example). Offline algorithms can benefit from properly aligning the demand periods of future requests by means of faults and their corresponding delays. For this purpose, an algorithm could evict a page voluntarily (i.e., not forced by a page fault) before it is requested in order to force a fault. In contrast, a demand paging or lazy algorithm only evicts pages when there is a fault and the cache is full (see Definition 6.2 in Section 6.2.1). In traditional sequential paging any paging algorithm can be made lazy without increasing its cost [Borodin and El-Yaniv, 1998]. We show an equivalent result for FINAL-TOTAL-FAULTS in our multi-core model. While in the sequential case it is easy to show that an algorithm does not benefit from evicting pages early, in the parallel case we need to consider the possible effects of changing sequence alignments.

Consider a non-lazy paging algorithm A in the sequential setting. Suppose that A evicts a page σ that was in a cache cell c that will not be immediately filled with another page. A can be made to a lazy version A' that, instead of evicting σ, marks it as "ready to evict", and delays its eviction. Suppose that before σ is requested again, there is a fault on a request for a page σ' that is fetched to cell c. If only then A' evicts σ, then clearly A' and A incurred the same cost. Moreover, if σ is requested before there is a fault on another page, then A' incurs a hit, while A incurs an extra fault.

While the argument for the case when another request leads to A' evicting σ can also be applied to the multi-core setting, the second scenario has consequences that are not present in the sequential case. In the multi-core setting, if A incurs a fault on σ, this leads to a delay in the corresponding sequence. This delay is not suffered by A', which incurs a hit on this page. Although A has incurred an extra fault, it has also changed the alignment of sequences, which

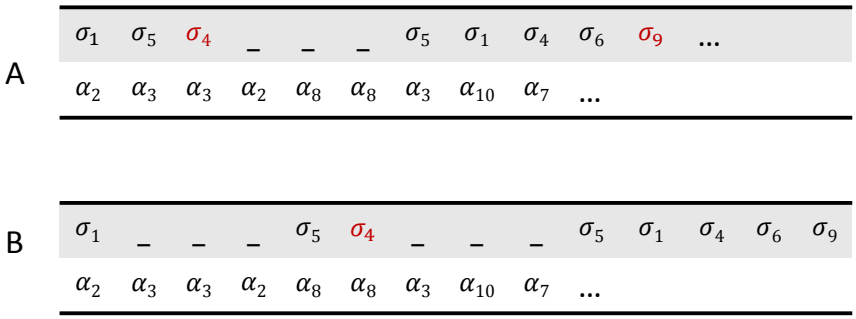


Figure 7.4: Example of forcing a fault. Algorithms A and B are serving the same sequences both starting with cache $C = \{\sigma_1, \sigma_5, \sigma_9, \alpha_2, \alpha_3, \alpha_8\}$. Algorithm B forces a fault on σ_1 and A does not. When serving σ_4 (a fault for both, in red in the figure), A must evict a page that will later result in a fault (e.g., σ_9). Algorithm B can evict α_2, which will not be requested again, thus saving a future fault (although it had to pay for the forced fault on σ_1).

could lead to future benefits. This scenario is equivalent to one in which both A and A' keep σ in their caches (as if both were lazy algorithms), but upon the next request to σ, A decides to fetch this page again from memory (thus incurring a fault and delaying the sequence). In this case, we say that A *forces a fault* on σ. We henceforth focus on this scenario. Forcing a fault and changing sequence alignments could lead to future fault savings. Figure 7.4 shows an example of this situation. We show, however, that forcing faults for the purpose of changing the alignments is not beneficial for minimizing the number of faults. We do this by showing that there exists an optimal lazy algorithm that does not force faults.

Theorem 7.5 *Let Alg be an offline optimal algorithm that forces faults. There exists an offline algorithm Alg' that is lazy such that for all disjoint requests $\mathcal{R}$, $Alg'(\mathcal{R}) = Alg(\mathcal{R})$.*

Proof: We follow an inductive argument similar to the proof of optimality of Furthest-In-The-Future in the traditional (sequential) setting [Borodin and El-Yaniv, 1998]. The proof relies on the following claim: let Alg be any paging algorithm and $\mathcal{R}$ be any request sequence. Then, for all timesteps i it is possible to construct an algorithm Alg_i such that (i) for all $t = 1, \ldots, i-1$, it behaves exactly like Alg, (ii) if Alg forces a fault on $t = i$, Alg_i does not, and (iii) $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$.

If the claim is true, then it is possible to obtain an optimal algorithm that does not force faults: for a given sequence $\mathcal{R}$, start from any optimal algorithm OPT and apply the claim with $i = 1$ to obtain OPT_1, then apply the claim with $i = 2$ to OPT_1 to obtain OPT_2, and so on

and so forth. $OPT_{t'}$ is an optimal algorithm that does not force faults, where t' is the maximum timestep of the execution of the algorithm on $\mathcal{R}$.

Let us prove the claim. Both algorithms start with an empty cache, and hence Alg_i can do exactly as Alg does up to step $i-1$. If at timestep i Alg does not force a fault, then Alg_i continues behaving like Alg until the end of the request, and the number of faults of both algorithms is the same. Now, assume that Alg_i forces a fault on $t = i$ on a page σ_1 on sequence R_s. Let $C_{Alg}(t)$ and $C_{Alg_i}(t)$ denote the caches of Alg and Alg_i right before serving $\mathcal{R}(t)$. Since both algorithms behaved exactly the same up to $t = i - 1$, we have $C_{Alg}(i) = C_{Alg_i}(i)$. Also, let $A(\mathcal{R}, t)$ denote the number of faults of algorithm A right before serving request $\mathcal{R}(t)$.

We argue that from that point on, Alg_i can be such that both algorithms will fault on exactly the same pages in the rest of the sequences (and hence keeping the same relative alignment in both algorithms), and that their caches will differ by at most one page. Furthermore, $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$ at all times. We show this by defining a set of states that describe the differences between the algorithms sequences, caches, and number of faults for each subsequent request in R_s for Alg. We define the following states at time t:

A. All sequences in both algorithms have the same alignment, $C_{Alg}(t) = C_{Alg_i}(t)$, and $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t)$.

B. All sequences other than R_s have the same alignment in both algorithms, $C_{Alg}(t) = C_{Alg_i}(t)$, and $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t) + 1$.

C. All sequences other than R_s have the same alignment in both algorithms, $C_{Alg}(t) = C_{Alg_i}(t) \cup \{\sigma\}$, and $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t)$, where σ is a page previously requested in R_s, and the remaining cell of Alg_i cache is fetching a page $\sigma' \neq \sigma$.

D. All sequences other than R_s have the same alignment in both algorithms, $C_{Alg}(t) = (C_{Alg_i}(t) \cup \{\sigma\}) \setminus \{\alpha\}$, and $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t) + 1$, where σ is a page previously requested in R_s, and α is a page from a sequence other than R_s.

E. All sequences other than R_s have the same alignment in both algorithms, $C_{Alg}(t) = C_{Alg_i}(t) \cup \{\sigma\}$, and $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t)$, where σ is a page previously requested in R_s, and the remaining cell of Alg_i cache is fetching σ.

Let $\sigma_2, \sigma_3, \ldots$ be the pages in R_s after σ_1. We define the request period of each page $\sigma_j \in R_s$ as the timesteps that include its request and possible fetching for algorithm Alg, i.e., if σ_j is requested at time t_j by Alg, then its request period is $[t_j, t_j + \tau]$ if σ_j is a fault, and just t_j if it is a hit. We will now show that the above are all the possible states that describe the algorithms in each period. We will prove this by induction on the request periods of pages in R_s.

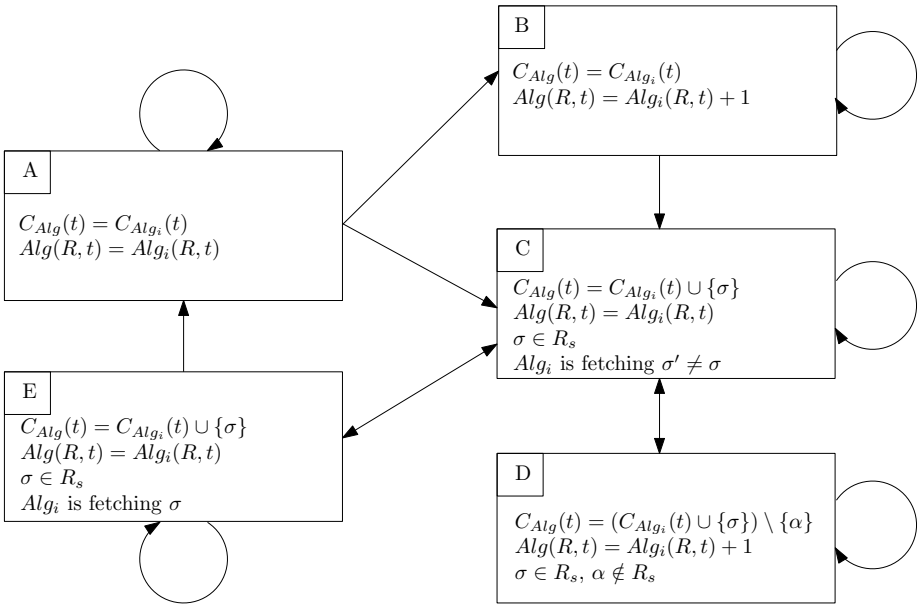


Figure 7.5: For any algorithm Alg that forces a fault on a page of a sequence R_s at time $t = i$, we can build an algorithm Alg_i that does not force faults and incurs no more faults than Alg does. The differences between Alg and Alg_i's number of faults and cache contents at each time define the five states depicted. $Alg(\mathcal{R}, t)$ and $C_{Alg}(t)$ denote, respectively, the number of faults and cache contents of Alg at time t. We show that Alg_i can maintain the execution in one of these states, and hence $Alg_i(\mathcal{R}, t) \leq Alg_i(\mathcal{R}, t)$ at all times.

Before σ_1, the algorithms are in state (A). Consider the request period for σ_1. Recall that $C_{Alg}(t_1) = C_{Alg_i}(t_1)$. Alg forces a fault on σ_1, and hence the cell corresponding to σ_1 in the cache is being used to fetch this page until the end of the period. Upon any request σ during the period, if Alg evicts α, Alg_i evicts α as well. In this period, up to τ pages $\sigma_2, ..., \sigma_{1+\tau}$ after σ_1 might be requested for Alg_i in R_s. If none of these are faults, then at the end of the period $Alg(\mathcal{R}, t_2) = Alg_i(\mathcal{R}, t_2) + 1$, and the caches of both algorithms are equal, hence the algorithms are in state (B)[7]. On the other hand, if any of the pages $\sigma_2, ..., \sigma_{1+\tau}$ is a fault for Alg_i, then Alg_i evicts any of the previous hit pages in R_s (there is a least one, σ_1). Hence, at the end of the period, the number of faults of both algorithms is the same, and both caches have the same pages, but for the evicted page by Alg_i, thus arriving at state (C).

For the request period of a page σ_j, $j > 1$, let s_j be the corresponding state of Alg and Alg_i.

[7]We assume here and for the rest of the analysis that Alg_i lets Alg run ahead at least until its next fault in R_s, so that if Alg evicts a page σ_j after the time a request for this page was served by Alg_i, Alg_i forces the fault on this page as well. In other words, if σ_j is a hit for Alg_i at time t, it will be a hit for Alg at time $t + \tau$.

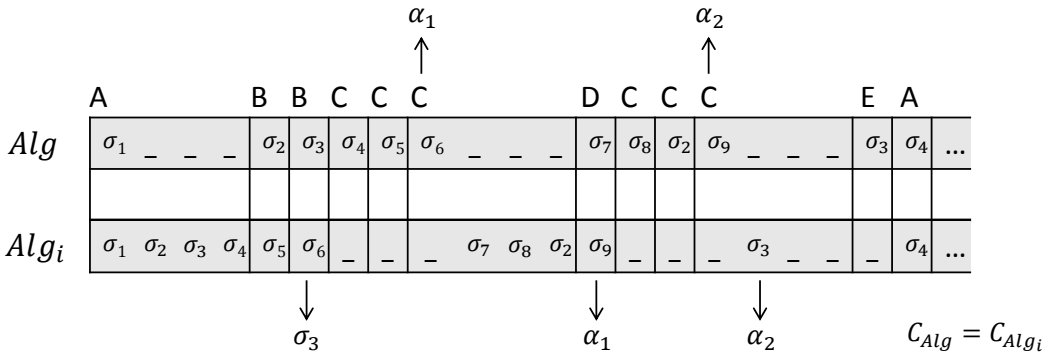


Figure 7.6: Example of an execution of Alg and Alg_i. Alg initially forces a fault on σ_1 in sequence R_s, while Alg_i does not. The figure shows the differences in alignments between the execution of both algorithms on R_s (other sequences are not depicted, since alignments are the same in both algorithms), as well as the periods of pages in Alg's execution and the states at the beginning of each period. Evicted pages for each fault are shown by arrows next to each fault. Alg_i's actions manage to bring the execution back to state A (with equal caches and number of faults), after which both algorithms behave exactly the same.

Assume that s_j is one of the states in $S = \{A, B, C, D, E\}$. We show now that Alg_i is able to reach a state $s_{j+1} \in S$ in the next period when $s_j = s$, for each $s \in S$, by describing all possible transitions between states, as depicted in Figure 7.5. Figure 7.6 shows an example of a possible sequence of state transitions, illustrating an instance of the arguments that follow.

[$s_j = A$] Suppose we have reached state (A). Since Alg_i cannot force faults only on request i, but it can do so on later requests, it just behaves exactly like Alg for the rest of the sequence, maintaining state (A).

[$s_j = B$] We can only arrive to state (B) if Alg_i had no faults for requests in R_s in the previous period. Since σ_j was requested for Alg_i in the previous period (sequences R_s in both algorithms are misaligned by τ at most), σ_j is a hit for Alg. If there is any fault on a request of another sequence, Alg_i evicts whatever Alg evicts. At time t_j the request for Alg_i is $\sigma_{j+\tau}$. If this page is a hit, then we stay in state (B). If this page is a fault, Alg_i evicts some page $\sigma_{j'}$ with $j' < j + \tau$. At least one page of R_s is in Alg_i's cache since they were all hits in the previous period (and we assume they are not evicted during this period by Alg). Hence, at the end of the period $Alg(\mathcal{R}, t_{j+1}) = Alg_i(\mathcal{R}, t_{j+1})$ and $C_{Alg}(t_{j+1}) = C_{Alg_i}(t_{j+1}) \cup \{\sigma\}$, for some $\sigma \in R_s$, arriving at state (C).

[$s_j = C$] At the beginning of this period the next page in the request ordering of Alg is σ_j and Alg_i is fetching some page $\sigma_{j'}$ that resulted in a fault in the previous period. This is the

case if we arrive from states (A), (B), and we will see that it holds if we stay in (C), or arrive from (D) or (E) as well. For all faults in sequences other than R_s, Alg_i evicts the same page that Alg evicts, unless Alg evicts σ, in which case Alg_i evicts another page in R_s. If σ_j is a hit for Alg then nothing changes and we stay in (C) as well. If σ_j is a fault, then Alg evicts a page α. Recall that $C_{Alg}(t_{j+1}) = C_{Alg_i}(t_{j+1}) \cup \{\sigma\}$. Assume first that $\alpha \neq \sigma$. Then, if all the subsequent requests of R_s in the request ordering of Alg_i are hits and α is not requested during this period, then we have one more fault for Alg and at the end of the period Alg's cache still has σ but not α, while Alg_i does not have σ but has α, hence reaching state (D). If α is requested during the period, then Alg_i forces a fault on this page and hence we remain in state (C). Now, if one of the requests from R_s for Alg_i results in a fault, say $\sigma_{j'}$, then, again if α was not requested before $\sigma_{j'}$, then Alg_i evicts α for this request. If α is requested before $\sigma_{j'}$, Alg_i forces a fault on α and evicts for $\sigma_{j'}$ whatever Alg evicted for α (or some page $\sigma' \in R_s$ if Alg evicts σ). At the end of the period the difference in number of faults remains the same. Now, if $\sigma_{j'} \neq \sigma$, then we stay in state (C). However, if $\sigma_{j'} = \sigma$, i.e., Alg_i faulted in the page that it did not have but that Alg had, then we reach state (E).

Now, we analyze the case $\alpha = \sigma$. The next page request in R_s in the ordering of Alg_i is σ_{j+1}. If this page is a fault, then Alg_i evicts another page $\sigma' \in R_s$, for example σ_j. In this case the number of faults increases by 1 for both algorithms and we remain in state (C). If on the other hand, σ_{j+1} is in Alg_i's cache (and hence in Alg's cache), Alg_i forces a fault on this page, reaching state (E).

$[s_j = D]$ We could only reach this state if σ_j is a hit. Let $\sigma_{j'}$ be the next page of R_s in the request ordering of Alg_i. Suppose α is not requested in $R(t_j)$. If $\sigma_{j'}$ is a hit, then nothing changes, and we remain in state (D). If $\sigma_{j'}$ is a fault, Alg_i evicts α and we reach state (C). If α is requested, since $\alpha \notin C_{Alg}(t_j)$, Alg evicts a page α'. Alg_i forces a fault on α. If $\sigma_{j'}$ is a hit, then we remain in state (D). If $\sigma_{j'}$ is a fault, Alg_i evicts α' if $\alpha' \neq \sigma$, or another page $\sigma' \in R_s$ if $\alpha' = \sigma$, reaching state (C).

$[s_j = E]$ This state is reached when the number of faults of both algorithms is the same and Alg_i is fetching the page σ that is missing with respect to Alg's cache. Suppose first that $\sigma_j \neq \sigma$ (i.e., this is not the timestep in which Alg_i finishes fetching σ[8]) σ_j is a hit for Alg (this is the case in the two cases that we can arrive to this state from (C), and the one from (E)). Upon any fault on another sequence Alg_i evicts whatever Alg evicts (with the assumption that Alg will not evict a page of R_s that was a hit in the previous period for Alg_i). If one of these evictions is for page σ, then Alg_i evicts another page $\sigma' \in R_s$ and we reach state (C). If σ was not evicted, then we remain in state (E). Now, if $\sigma_j = \sigma$, this page is in Alg's cache. Again, Alg_i evicts what Alg evicts for other requests. If, however, in any of these evictions the page evicted is σ[9], Alg_i evicts

[8]Since sequences R_s in the execution of Alg and Alg_i are either aligned or Alg's sequence is behind by exactly τ, if the current request to Alg is to the same page σ_j that is being fetched by Alg_i then this is the timestep in which Alg_i finishes fetching σ_j.

[9]Alg could evict σ prior to its request according to the logical order of simultaneous requests.

another page $\sigma' \in R_s$. In this case Alg faults in σ_j and we are back in state (C). On the other hand, if $\sigma_j = \sigma$ is a hit, Alg_i finishes at the same time to fetch σ, therefore both caches are equal, sequences R_s in both algorithms are aligned, and the number of faults of both algorithms is the same, reaching state (A).

Hence, Alg_i can keep the execution in these states until the end. Since $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$ in each of these states, this holds until the end of the sequence, proving the claim. ∎

Hassidim [2010] shows in his model that there is an optimal solution for minimizing the makespan that, on each fault, evicts the page that is furthest in the future for some core. In other words, if the sequence whose page should be evicted is known, the page to be evicted is the furthest in the future in that sequence. We show that the same result holds in our model for minimizing the number of faults.

Theorem 7.6 *There exists an optimal offline algorithm for* FTF *on disjoint sequences that upon each fault evicts a page $\sigma \in R_j$ whose next request time is maximal in R_j, for some j.*

Proof: As in the proof of Theorem 7.5, we claim that for any algorithm Alg there exists an algorithm Alg_i that behaves exactly like Alg until time $i-1$. If at time i Alg evicts a page from sequence R_j, Alg_i evicts the page from the same sequence that is furthest in the future, and $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$. Applying this claim on an optimal algorithm successively at each timestep gives an optimal algorithm.

We now prove the claim. Suppose that at time $t = i$ Alg evicts a page $\sigma_1 \in R_j$. Alg_i behaves exactly like Alg until $t = i-1$ but at $t = i$ it evicts a page $\sigma_2 \in R_j$, which is the page furthest in the future in this sequence. Assume $\sigma_1 \neq \sigma_2$ as otherwise the claim is trivially true. Let $C_{Alg}(t)$ and $Alg(\mathcal{R}, t)$ denote the contents of Alg's cache and the number of faults before serving $\mathcal{R}(t)$. We have $C_{Alg}(t) = C_{Alg_i}(t)$ and $C_{Alg}(t+\tau) = (C_{Alg_i}(t+\tau) \cup \{\sigma_2\}) \setminus \{\sigma_1\}$, and the sequences have the same alignment in both algorithms.

Before the request for σ_1, if upon a fault Alg evicts σ_2, Alg_i evicts σ_1, then the caches are equal and from then on Alg_i behaves exactly like Alg, thus $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$. If Alg instead evicts a page $\alpha \neq \sigma_2$, Alg_i evicts the same page.

If σ_2 was not evicted, when σ_1 is requested Alg faults and evicts a page α from some sequence. σ_1 is a hit for Alg_i and thus $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t) + 1$, $C_{Alg}(t) = (C_{Alg_i}(t) \cup \{\sigma_2\}) \setminus \{\alpha\}$, and Alg_i is ahead in R_j by τ timesteps. Assume first that σ_2 is not evicted by Alg before its request. Suppose Alg_i gets to σ_2 with this configuration (for some α). σ_2 is a fault for Alg_i and a hit for Alg. Alg_i evicts α and now $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t)$, $C_{Alg}(t) = C_{Alg_i}$ and the sequences are aligned equally. From then on both algorithms are equivalent and the claim is true. After the request for σ_1, Alg_i evicts whatever Alg evicts (assume σ_2 is not evicted by Alg during this period). If α is requested (a fault for Alg but a hit for Alg_i), Alg_i forces a fault and hence

$Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t) + 1$ and $C_{Alg}(t) = (C_{Alg_i}(t) \cup \{\sigma_2\}) \setminus \{\alpha'\}$ still holds, with α' being the page evicted by Alg. By Theorem 7.5, we can build another algorithm Alg'_i with the same number of faults that does not force faults, and hence the claim is true in this case. Now, if Alg evicts σ_2 before getting to its request Alg_i evicts α, and both caches are the same. When σ_2 is requested both algorithms will fault and $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t) + 1$ holds. However, sequences R_j in both algorithms do not have the same alignment with respect to the rest of the sequences, with Alg_i's sequence being ahead by τ timesteps. This setting is the same as the one in the proof of Theorem 7.5 after Alg has forced a fault. Applying the theorem's proof we can show that Alg_i can keep the execution within the 5 states defined in the proof, and hence $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$ at the end of the execution. Again, if at any point Alg_i forces a fault, then by the same Theorem 7.5 we can obtain an algorithm that does not force faults and that does not exceed Alg_i's faults. Since in all cases the claim is true, this proves the theorem. ∎

7.3.3 Optimal Algorithms for FTF and PIF

Theorem 7.6 implies an $O(p^n)$ time optimal algorithm for FTF that upon each fault chooses the sequence to evict from optimally by trying all possibilities. Using dynamic programming, however, we can obtain a faster algorithm that is exponential in the number of sequences, but polynomial in the length of the sequences (recall we assume $n \gg p$; in particular, we have observed in Chapter 3 that p can be effectively assumed to be $O(\log n)$). We show that this algorithm can be extended to solve PIF as well. Hence, if the number of sequences is constant, then both FTF and PIF admit polynomial time algorithms[10].

Minimizing the number of faults

We describe a dynamic program to compute the minimum number of faults to serve a request $\mathcal{R}$ of p sequences. Subproblems are defined based on the different alignments of request sequences and contents of the cache. More specifically, each subproblem consists of computing the minimum number of faults required to serve certain prefixes of the request sequences finishing with a certain set of pages in the cache. We store partial solutions in a table, and compute solutions for larger prefixes based on the combination of prefixes and cache contents that are reachable from each partial solution. Thus, the program explores all possible ways to advance in each sequence for all possible cache configurations.

We specify the prefixes of sequences using indices to requests within the sequence. In order to take into account fetching times, we extend the index space to include the periods in which

[10]Note that this does not contradict Theorem 7.3: we can think of PIF as having two input sizes, the length of the sequences n, and the number of sequences p, which is the number of cores. PIF is NP-hard when the number of sequences is not fixed, but can be solved in polynomial time in n when p and k are constants.

sequences might be waiting for pages being fetched. Let $\vec{x} = (x_1, \ldots, x_p)$, where each x_i is an index of a place in sequence i when serving it, either at a page, or at a time when a page is being fetched, with $1 \leq x_i \leq n_i(\tau+1)+1$. If x_i is of the form $x_i = (j-1)(\tau+1)+1$, then x_i is the index of the j-th page in R_i. In this case, we say x_i points to a page (denoted as $R_i(x_i)$). Otherwise, x_i points to the fetching period of page $\lceil x_i/(\tau+1)\rceil$, and $R_i(x_i)$ denotes the page being fetched. Let $\mathcal{R}(\vec{x})$ denote the set of pages p indexed by $\vec{x}$, including pages in the fetching period.

Given a request $\mathcal{R}$ we want to compute, for each possible cache configuration C and possible vector of positions $\vec{x}$, the minimum number of faults required to serve $\mathcal{R}$ up to $\vec{x}$, and arriving at a cache configuration C. If x_i is in a page, this cost does not include serving $R_i(x_i)$. If x_i points to a fetching period, this cost includes the fault on $R_i(x_i)$. We compute and store these values in a $(p+1)$-dimensional table storing the minimum cost for each configuration and position. A cell $F[C, x_1, \ldots, x_p]$ can contribute to a cell $F[C', x'_1, \ldots, x'_p]$, where C' is any configuration that can be obtained by removing $|\mathcal{R}(\vec{x}) \setminus C|$ pages from C and adding the ones in $\mathcal{R}(\vec{x})$, and $x'_i > x_i$ is the next index on sequence i. If x_i points to a page and $R_i(x_i)$ is a hit, then $x'_i = x_i + \tau + 1$, i.e., the index jumps to the next page. If $R_i(x_i)$ is being fetched or is a miss, then $x'_i = x_i + 1$, unless the page $R_i(x_i)$ finished fetching in this timestep for another sequence, in which case the index also jumps to the next page and thus $x'_i = \lceil x_i/(\tau+1)\rceil(\tau+1)+1$ (this case only applies to non-disjoint sequences). We fill the table in a bottom up fashion, updating from each cell c only the cells to which c can contribute with a lower number of faults. The total minimum number of faults is then the minimum among all cache configurations C of $F[C, n_1(\tau+1)+1, \ldots, n_p(\tau+1)+1]$. Algorithm 7.1 shows this procedure in pseudocode.

Theorem 7.7 *Given a set $\mathcal{R}$ of p sequences of total length n, a cache size k, and $\tau \geq 0$, with $p = O(1)$ and $k = O(1)$, the minimum number of faults to serve $\mathcal{R}$ can be determined in $O(n^{k+p}(\tau+1)^p)$ time.*

Proof: Let w be the total number of different pages requested in an instance. The number of possible cache configurations is $\sum_{i=0}^{k} \binom{w}{i} \leq (w+1)^k$. Hence, the total number of cells in table F is $O((w+1)^k(n(\tau+1)+1)^p)$. Counting the faults in $\mathcal{R}(\vec{x})$ for each $\vec{x}$ takes time $O(p^2)$. Since $|\mathcal{R}(\vec{x})| \leq p$, at most $\binom{k}{p} = O(k^p)$ cache configurations can be reached from any configuration C, thus the time to process each cell and update the cells that it can contribute to is $O(p^2 + k^p)$. Since $w \leq n$, when k and p are constants, the total running time is $O(n^{k+p}(\tau+1)^p)$. ∎

Deciding PARTIAL-INDIVIDUAL-FAULTS

Recall that the PIF problem asks for the feasibility of serving a request sequence $\mathcal{R}$ such that at a given checkpoint time the number of faults on each sequence R_i is at most a given bound b_i.

Algorithm 7.1 Minimum Final Total Faults($\mathcal{R}, k$)

for all configurations C **do**
  $F[C, 1, \ldots, 1] = 0$
  for each $(x_1, \ldots, x_p) \in \{2, \ldots, n_i(\tau+1)+1\}^p$ **do**
    $F[C, x_1, \ldots, x_p] = \infty$
for each $(x_1, \ldots, x_p) \in \{1, \ldots, n_i(\tau+1)\}^p$ **do**
  for all configurations C **do**
    if $F[C, x_1, \ldots, x_p] \neq \infty$ **then**
      $f = 0$ {faults in $\mathcal{R}(\vec{x})$}
      for $i = 1$ to p **do**
        if $x_i = (j-1)(\tau+1)+1$ for some j, and $R_i(x_i) \in C$ **then** {$R_i(x_i)$ is a hit}
          $x'_i = x_i + \tau + 1$
        else {$R_i(x_i)$ is being fetched or it is a fault}
          if $R_i(x_i) \notin C$ **then** {$R_i(x_i)$ is a fault}
            $f = f + 1$
          if $R_i(x_i) = R_\ell(x_\ell)$ and $x_\ell = (j'-1)(\tau+1)$ for some j' and some $\ell \neq i$ **then** {$R_i(x_i)$ was fetched for another sequence}
            $x'_i = \lceil x_i/(\tau+1) \rceil (\tau+1) + 1$
          else
            $x'_i = x_i + 1$
    for all C' s.t. $\mathcal{R}(\vec{x}) \subseteq C'$ and $(C' \setminus \mathcal{R}(\vec{x})) \subseteq C$ **do**
      if $F[C, x_1, \ldots, x_p] + f < F[C', x'_1, \ldots, x'_p]$ **then**
        $F[C', x'_1, \ldots, x'_p] = F[C, x_1, \ldots, x_p] + f$
return $\min_C\{F[C, n_1(\tau+1)+1, \ldots, n_p(\tau+1)+1]\}$

We now describe how to extend Algorithm 7.1 for FTF to solve PIF. Once again, we keep a table of partial solutions for each combination of positions within sequences and cache configurations. However, for the case of PIF not only we care about the total number of faults for each subproblem, but also about the individual number of faults in each sequence prefix and the time in which the configuration is reached. In order to keep track of this, for each cache configuration C and positions $(x_1, \ldots, x_p)$ we store a set of pairs $(\vec{f}, t)$, where $\vec{f} = (f_1, f_2, \ldots, f_p)$ specifies the faults on each sequence when reaching configuration $(C, x_1, \ldots, x_p)$ at time t. Thus, each pair $(\vec{f}, t)$ associated with $F[C, x_1, \ldots, x_p]$ represents the number of faults in each sequence for a possible way of serving sequence $\mathcal{R}$ up to time t. The algorithm proceeds similarly to the one for FTF. This time, however, we are not minimizing an objective but we are determining the feasibility of a given solution. Starting bottom-up, for each cell $F[C, x_1, \ldots, x_p]$ we adjust the indices x_i based on the contents of C and the current request, just like we did in Algorithm 7.1. We then update the fault vectors associated with the cell. For each of the current valid $(\vec{f}, t)$ pairs, we update

the number of current faults f_i in each sequence R_i based on the contents of C and the current request $R_i(x_i)$. If any of the components of $\vec{f}$ exceeds the given bound on allowed faults, then we discard the pair. Otherwise, and if we have not yet reached the checkpoint time, we include the updated pair in the cells that can be reached from $F[C, x_1, \ldots, x_p]$. If we have reached the checkpoint time or the end of all sequences with a valid vector, then we have found a feasible solution and we return true. Finally, if we have reached the end of the table with no surviving valid vector-time pairs, we return false. Algorithm 7.2 shows the algorithm in pseudocode.

Theorem 7.8 *Given a set $\mathcal{R}$ of p sequences of total length n, a cache size k, $\tau \geq 0$, a checkpoint time t, and a vector $\vec{b} = \{b_1, \ldots, b_p\}$, with $p = O(1)$ and $k = O(1)$, it can be decided if $\mathcal{R}$ can be served such that at time t each sequence R_i has incurred at most b_i faults in $O(n^{k+2p+1}(\tau+1)^{p+1})$ time.*

Proof: The number of entries of table F in Algorithm 7.2 is $O(n^{k+p}(\tau+1)^p)$ as in the algorithm for FTF. However, now each entry stores a list of pairs of fault vectors and time. Since at any time the number of faults in a sequence is at most n, the total number of different fault vectors is $O((n+1)^p)$. The time component of each pair can have at most $n(\tau+1)$ values, and hence each entry can have at most $O(n^{p+1}(\tau+1))$ pairs. For each entry in F we have to go through the list of pairs and compute the new vectors, and hence processing an entry takes $O(pn^{p+1}(\tau+1))$ time. Once the new vectors are computed, these might have to be added to at most at most $O(k^p)$ other entries. Hence the total time to process one entry is $O(pn^{p+1}(\tau+1) + k^p)$, and therefore the total time is $O(n^{k+2p+1}(\tau+1)^{p+1})$. ∎

While the running times of Algorithms 7.1 and 7.2 are not practical for realistic values of k, the results in Theorems 7.7 and 7.8 are important in that they show that the complexity of multi-core paging stems from the number of sequences and not their length, placing the problem—in both of its versions— in P when p is constant.

7.4 Conclusions

We have proposed a model for multi-core paging that extends classical paging to a setting in which various sequences must be served simultaneously with a shared cache. The presence of multiple sequences and the fact that faults delay future requests make this problem significantly more difficult than classical paging. Neither traditional online algorithms nor the optimal strategy for sequential paging are competitive for this problem. Moreover, we have shown that serving a set of requests while limiting the number of faults in each sequence is NP-complete for an unbounded number of sequences. We showed, on the other hand, that multi-core paging admits algorithms that run in polynomial time in the length of the sequences, thus the problem is in P when the

Algorithm 7.2 Partial Individual Faults($\mathcal{R}, k, \text{checkpoint}, \tau, \vec{b}$)

for all configurations C **do**
  $F[C, 1, \ldots, 1] = \{(\{0\}^p, 0)\}$
  for each $(x_1, \ldots, x_p) \in \{2, \ldots, n_i(\tau+1)+1\}^p$ **do**
    $F[C, x_1, \ldots, x_p] = \emptyset$
for each $(x_1, \ldots, x_p) \in \{1, \ldots, n_i(\tau+1)\}^p$ **do**
  for all configurations C **do**
    if $F[C, x_1, \ldots, x_p] \neq \emptyset$ **then**
      for $i = 1$ to p **do**
        if $x_i = (j-1)(\tau+1)+1$ for some j, and $R_i(x_i) \in C$ **then** {$R_i(x_i)$ is a hit}
          $x'_i = x_i + \tau + 1$
        else if $R_i(x_i) = R_\ell(x_\ell)$ and $x_\ell = (j'-1)(\tau+1)$ for some j' and some $\ell \neq i$ **then** {$R_i(x_i)$ was fetched for another sequence}
          $x'_i = \lceil x_i/(\tau+1) \rceil (\tau+1) + 1$
        else
          $x'_i = x_i + 1$
      $P = \emptyset$ {P is the list of updated fault vectors}
      for each $(\vec{f}, t)$ in $F[C, x_1, \ldots, x_p]$ **do**
        validVector = true
        for $i = 1$ to p **do**
          $f'_i = f_i$ {f_i is the i-th element of $\vec{f}$}
          if $R_i(x_i) \notin C$ **then** {$R_i(x_i)$ is a fault}
            $f'_i = f'_i + 1$
          if $f'_i > b_i$ **then** {this path exceeded the maximum faults for sequence i}
            validVector = false
        if validVector and $t + 1 \leq$ checkpoint **then**
          $P = P \cup (\vec{f}', t+1)$
          if $t + 1 =$ checkpoint or $x'_i = n_i(\tau+1)+1$ for all $i = 1..p$ **then** {we reached the checkpoint time or the end of all sequences}
            return TRUE
      for all C' s.t. $\mathcal{R}(\vec{x}) \subseteq C'$ and $(C' \setminus \mathcal{R}(\vec{x})) \subseteq C$ **do**
        $F[C', x'_1, \ldots, x'_p] = F[C', x'_1, \ldots, x'_p] \cup P$
{we reached the end without finding a feasible solution}
return FALSE

number of sequences is constant. Given the unfeasible running time of the proposed algorithms it would be desirable to obtain more efficient exact or approximate offline algorithms.

Other directions of research include determining the complexity of FINAL-TOTAL-FAULTS and

obtaining competitive online algorithms. Given the apparent excessive advantage of an offline algorithm over an online strategy that cannot do anything about future alignments, perhaps comparing online strategies to an optimal offline algorithm that can align sequences to its advantage might not lead to interesting online strategies. Hence the definition of a good evaluation framework for online strategies is open for debate.

One possibility to reduce the power of the optimal offline algorithm is to restrict algorithms to ensure that the relative progress of sequences follows a measure of fairness that considers each sequence's cache demands. For example, we could restrict the number of faults f_p incurred by a sequence on a shared cache of size k to satisfy that the ratio f_p/f_1 is bounded, where f_1 is the number of faults that the sequence would incur if served alone with a cache of size k/p. The idea of this and other similar fairness restrictions is to eliminate worst-case artificial schedules that would not be acceptable in reality.

It would be interesting to extend the model to include practical considerations such as limited parallelism in memory and cache accesses. Our multi-core model allows for all requests to be served in parallel, both when fetching pages from cache or memory. In reality, the limited bandwidth of buses together with restrictions on parallel access to memory banks limit both the number of lines that can be fetched from memory to cache and the data that can be moved from cache to cores simultaneously. Thus, it would be interesting to study a model in which only a subset of pages can be fetched from memory or cache in parallel. This can be realized by adding two parameters c and m with $1 \leq c \leq p$ and $1 \leq m \leq p$ such that at each timestep, at most c pages can be served from cache, and at most m pages can simultaneously be fetched from memory. This would result in a parameterized generalization of the original model. Finally, another interesting extension would be to include synchronization between sequences, reflecting the scenario in which threads running in multiple cores belong to the same application, and thus page requests might have dependencies.

Chapter 8

Minimizing Cache Usage in Paging

A paging algorithm must decide which pages to maintain in the cache at each time in order to minimize a defined cost measure. In the classic page fault model the cost of an algorithm is measured in terms of its number of faults and hits have no cost, reflecting the fact that an access to slow memory is orders of magnitude slower than an access to cache. As computer architectures and applications evolve, other cost models have arisen to reflect, for example, varying fetching costs and sizes in web-caches [Chrobak et al., 1991; Irani, 1997; Young, 1998], or multiple applications or threads sharing a cache (see Section 6.2.4 and Chapter 7).

In this chapter we consider a generalization of the classic page fault model whose performance objective function is a combination of both the number of faults and the amount of cache used by an algorithm. Thus in addition to the fault cost, at each step we charge a cost proportional to the number of pages in cache. In general, the model seeks algorithms with good performance in terms of number of faults while at the same time using available resources efficiently. Naturally, minimizing the number of faults and the cache usage of a paging algorithm are conflicting goals.

Paging strategies that minimize cache usage are relevant in multi-core architectures where multiple cores share some level of cache. In this context, multiple request sequences compete for the use of this shared resource. While traditional models of paging encourage algorithms to use the entire cache so as to minimize the faults incurred, a model that charges for cache usage can make a paging algorithm in a shared cache scenario be "context aware". Varying the parameters of the model for each sequence can be used to achieve a cooperative global strategy with better overall performance. Another example of a scenario in which the cache is shared is in the context of cloud-based applications. In many cloud infrastructures a scalable in-memory cache layer provides low latency for data access for various concurrent client applications [Wolfe Gordon and Lu, 2011], possibly involving a cost for customers based on cache usage. The model we present can thus be useful for the design of paging strategies that trade-off performance and monetary cost.

The cache usage model can also be used as an energy efficient paging model. Several applications use caches implemented with Content-Addressable Memories (CAMs), most notably networking routers and switches, and Translation Lookaside Buffers (TLBs). CAMs provide a single clock cycle throughput, making them faster than other hardware alternatives [Pagiamtzis and Sheikholeslami, 2006]. However, speed comes at a cost of increased power consumption, mainly due to the comparison circuitry. Reducing this power without sacrificing capacity or speed is an important goal of research in circuit design [Pagiamtzis and Sheikholeslami, 2006]. Power consumption could be reduced if inactive cache lines are turned off, thus our model can provide a framework for paging strategies that achieve good performance in terms of faults while contributing to energy savings.

Our results[1] We introduce the minimum cache usage problem, a generalization to the paging problem with a cost model that combines both the number of faults and the amount of cache that algorithms use.

We show that traditional optimal paging algorithms are still k competitive in this model, but that this ratio does not adapt to the differences in relative costs of faults and cache. We then define a family of online algorithms that combine the eviction policies of traditional marking or conservative algorithms with cache saving policies. We show that algorithms in this family achieve a competitive ratio of 2 if $\alpha < k$, where $\alpha = f/c$ is the ratio between fault and cache cost, and $\min\left\{k, \frac{\alpha(k+1)}{\alpha+k-1}\right\}$ if $\alpha \geq k$, thus matching the performance of classical algorithms when $f \gg c$. We further parametrize the analysis by considering the locality of reference of the sequence, and show that for sequences with high locality of reference the competitive ratio of our algorithms is at most 2. Simulations on real-world input sequences show that our algorithms are close to optimal in practice.

We also present a polynomial time optimal offline algorithm for the minimum cache usage problem, which we obtain via a reduction to Weighted Interval Scheduling on Identical Machines.

In the next section we introduce the minimum cache usage model and problem. We mention some of its applications, as well as related cost models. We then present an optimal offline algorithm for this problem in Section 8.2. In Section 8.3 we study and analyze online algorithms, while we present the results of our simulations in Section 8.4. We present concluding remarks in Section 8.5.

8.1 Paging with Cache Usage

The paging model we consider in this work extends classic paging to a model in which the cost of a paging algorithm on a request sequence is a weighted function of the number of faults and

[1] Results in this chapter appeared in [López-Ortiz and Salinger, 2012].

the total amount of cache used by the algorithm. An instance of paging with minimum cache consists of a sequence $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ of page requests and a maximum cache size k. Each request r_i is associated with a page σ_j, for $1 \leq j \leq N$, where N is the size of the universe of pages that can be requested. We denote by $page(r_i)$ the page associated with request r_i. Thus, if the i-th request of $\mathcal{R}$ is for page σ_j, then $page(r_i) = \sigma_j$. A paging algorithm can hold at most k pages in its cache, but can also choose to hold fewer pages, in order to reduce its cache usage.

Definition 8.1 (Total cache usage) *Let A be a paging algorithm and $\mathcal{R}$ a request sequence. Let $k(i) \leq k$ denote the number of pages in A's cache immediately before request r_i, where k is the maximum cache size. The total cache usage of A when serving $\mathcal{R}$ is defined as $C_A(\mathcal{R}) = \sum_i k(i)$.*

Definition 8.2 (Minimum Cache Usage Problem) *Given a request sequence $\mathcal{R}$ and maximum cache size k, the cost of an algorithm A on $\mathcal{R}$ is defined as $A(\mathcal{R}) = f \cdot F_A(\mathcal{R}) + c \cdot C_A(\mathcal{R})$, where $F_A(\mathcal{R})$ and $C_A(\mathcal{R})$ are the number of faults and total cache usage of A when serving $\mathcal{R}$, respectively, and $f \geq 0$ and $c \geq 0$ are parameters. The Minimum Cache Usage problem is then the problem of serving a request sequence with minimum cost.*

In reality a request sequence is revealed in an online fashion, thus our focus is on the performance of online algorithms in terms of their competitive ratio (see Section 6.1.1). An online algorithm has competitive ratio r if, given a maximum cache size k, and parameters f and c, for all request sequences $A(\mathcal{R}) \leq r \cdot OPT(\mathcal{R}) + \beta$, where OPT is the optimal offline, r is a function of k, f and c, and β is a constant that does not depend on $\mathcal{R}$. As in classic paging, the steps involved in serving a request r_i are as follows: the page associated with the request is revealed to the algorithm, after which the algorithm acts by possibly evicting one or more pages, and finally the request is served. Thus, all pages evicted in cache in step i were held in cache up to time $i - 1$. Recall from Section 6.2.1 that a paging algorithm is said to be *lazy* or *demand paging* if it only evicts a page when a page fault occurs. Observe that unlike classic paging, in which any algorithm can be made demand paging without sacrificing performance [Borodin and El-Yaniv, 1998], in our model algorithms can benefit from evicting pages even when there is no page fault.

The relation between the parameters f and c can vary according to the application to emphasize the importance of minimizing faults or using the cache efficiently, or a combination of both. Naturally, an instance with $c = 0$ and $f > 0$ is an instance of the classical model, in which the cost of an algorithm is its number of faults. On the other hand, if $f < c$ then the problem is trivial: an optimal algorithm always evicts the page of each request immediately after serving it. We assume in general that $f \geq c > 0$.

8.1.1 Applications

The cost model described above provides incentives for an eviction policy to be efficient not only in terms of its faults but also with respect to the use of the resources that are available to it. Thus, the model can be used in any environment where the latter has significance. We mention the following applications.

Shared Cache Multiprocessors Multi-core processors are equipped with both private and shared caches, with threads running in each core usually competing for the latter type. While there are schedulers that seek to achieve cooperative use of a shared cache, in general paging strategies for individual threads do not act cooperatively but use as much of the available cache as possible. The cost model we propose provides incentives for paging algorithms to balance their own benefits—a fast execution due to a small number of faults—and the benefits they can provide to concurrently running threads. Depending on the values of f and c, an algorithm will favour one or the other.

Energy Efficient Caching Content Addressable Memories (CAMs) are used in many applications that require high speed searches, and whose primary applications are in network routers [Pagiamtzis and Sheikholeslami, 2006]. CAMs are indexed by stored data words instead of memory addresses, as it is the case in regular caches. Each cell has a matchline that indicates if the stored word in the cell and the searched word match. A search for an input data word first precharges all matchlines, then each cell compares its bits against the searched bits, and matchlines corresponding to non-matching entries are discharged. The overall missing matchline dynamic power consumption for a system with w matchlines can be modeled as $P = wCV^2f$, where C is the matchline capacitance, V is the supply of a matchline and f is the frequency of misses (the power associated with a matchline in a match is small and can be neglected) [Pagiamtzis and Sheikholeslami, 2006]. The power involved in this operation can be therefore reduced if matchline precharging is controlled based on the valid bit status of each entry [Miyatake et al., 2001]: on a search, only valid entries require the precharging of matchlines, thus the power cost of a search can be proportional to the number of valid entries in the cache. In this scenario, a paging algorithm that uses its cache efficiently will contribute to power savings.

Caching in Cloud-Based Applications Cloud computing is a form of computing that makes use of a shared pool of computing resources that are accessed on-demand through a network [Mell and Grance, 2009]. The services provided by a cloud infrastructure might be the use of a provider's application (Software as a Service), the deployment of client's applications that use languages, libraries, and services supported by the provider (Platform as a Service), or the deployment of arbitrary software (e.g, operating systems and applications) that use resources such as storage

and networks (Infrastructure as a Service) [Mell and Grance, 2009]. In general, cloud computing enables the scaling of services by automatic allocation of resources based on demands [Wolfe Gordon and Lu, 2011].

Many web applications are implemented in the cloud and are backed by a data-store such as databases or NoSQL storage engines [Wolfe Gordon and Lu, 2011]. Applications usually use a caching layer in order to reduce the latency of data access. An important example of a cache layer implementation in cloud computing is memcached [Fitzpatrick, 2004]. Memcached is implemented in a distributed way with the main memories of servers combined acting as the fast memory, thus effectively increasing the total capacity of the cache compared to application-level caching [Wolfe Gordon and Lu, 2011]. In this way, various applications and processes within applications can share this cache. A provider can charge a previously agreed cost for the amount of cache that each application uses. In this context, applications using paging algorithms that take into account the amount of cache used can trade performance for cost and vice versa.

8.1.2 Related Cost Models

The performance of paging algorithms has been traditionally measured using competitive analysis [Sleator and Tarjan, 1985] and the most used cost model is the page fault model, where a fault has unit cost and hits have no cost (see Section 6.2.2 for a brief description of other cost models with applications in web caching). A related paging model that also includes the amount of cache used in the cost of algorithms was proposed by Csirik et al. [2001]. In this model an algorithm can purchase cache slots, and the overall cost of the algorithm is the number of faults plus the cost of purchased cache. As cache may only be bought, the cache size can only increase (with no bound on the maximum size). In our model, however, an algorithm is charged for the number of pages it has in the cache at every step, which can both increase or decrease. In this sense our model charges algorithms for renting cache, while keeping the upper bound k on the maximum cache available. We note as well that the idea of memory renting for reducing RAM power consumption was previously mentioned in [Chrobak, 2010].

8.2 Offline Optimum

In the next section we describe a simple family of online algorithms for the cache usage problem and analyze their competitiveness. In order to provide a better intuition for that analysis we first describe a solution to the offline problem. We recast a paging instance as an instance of weighted interval scheduling on identical machines, and use an algorithm for this problem to obtain an optimal polynomial time paging algorithm.

An instance of weighted interval scheduling on identical machines consists of a set J of jobs and a number m of available identical machines. Each job has a starting time, a duration, and a

weight. In order to be processed, a job must be assigned to a machine immediately after its start time and cannot be interrupted. A machine can process only one job at a time. The goal is to process a subset $J' \subseteq J$ of jobs such that the total weight of jobs in J' is maximized. Equivalently, each job corresponds to an interval in the real line, and we seek to schedule the maximum weight subset of intervals such that at most m intervals overlap at any time. This problem can be solved in polynomial time by reduction to Minimum Cost Flow [Arkin and Silverberg, 1987; Bouzina and Emmons, 1996].

An instance of the paging problem can be regarded as an instance of interval scheduling on identical machines: each pair of consecutive requests to the same page defines an interval whose start and end times are the times of the requests. In each pair of requests, the second request results in a hit if and only if the corresponding page is kept in the cache since the previous request, or equivalently, if the interval is scheduled.

The connection between interval scheduling and paging has been noted before in [Wagner, 2001; Brehob et al., 2004] where it is used to study cache policies in non-standard caches. It is assumed, however, that the reduction applies only when bypassing is allowed. More recently, Chrobak et al. [2012] used this connection to show that offline paging in the fault and bit models is NP-hard by reducing interval packing problems to paging. Unlike our model, these models consider pages (and hence intervals) of different sizes. The reduction we introduce in this work is from paging to interval scheduling, and it is defined as follows.

Definition 8.3 (Interval representation of a sequence) *An interval representation of a request $\mathcal{R}$ of length n is a set of intervals $\mathcal{I}(\mathcal{R}) = \{I_1, I_2, \ldots, I_n\}$ where each interval I_i corresponds to request r_i in $\mathcal{R}$. The starting time of each interval I_i is $s(I_i) = i + 1$ and the end time is $e(I_i) = j - 1$, where $j > i$ is the smallest index such that $page(r_j) = page(r_i)$, or $e(I_i) = n$ if no such j exists. We say that an interval I_i is* feasible *if $e(I_i) < n$ and* unfinished *otherwise. Thus, the length of interval I_i is $|I_i| = e(I_i) - s(I_i) + 1$.*

An example of a sequence and its interval representation is shown in Figure 8.1. Intuitively, an interval corresponding to request r_j represents the time interval in which $page(r_j)$ must reside in the cache in order for the next request to this page to result in a hit. Note that each first request to a page has no preceding interval and thus cannot be a hit. Similarly, a page that is requested for the last time in a sequence can be held in cache, but as the interval does not finish in the corresponding page, it cannot result in a hit. Note that intervals do not overlap with the times in which their corresponding pages are requested, thus using this reduction there is no need to assume that bypassing is allowed. All requests are served, but only the ones whose interval was scheduled will result in hits. Note as well that two consecutive requests to the same page define an interval of length 0 that does not overlap any other interval, and thus it is always scheduled. The following lemma formalizes the reduction.

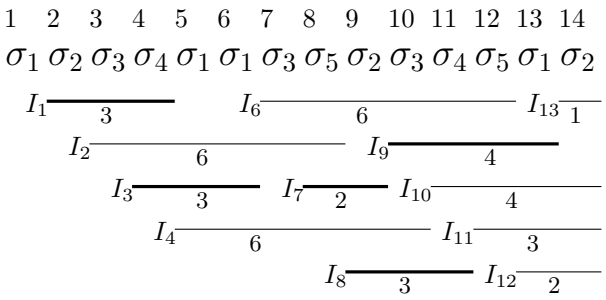


Figure 8.1: A request sequence and its interval representation. The length of each interval is shown below the interval (I_5 of length 0 is not shown). Feasible intervals are $\{I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9\}$ while $\{I_{10}, I_{11}, I_{12}, I_{13}\}$ are unfinished. The request can be served with a cache of size 3 with 8 faults and a cache usage of 29 by scheduling intervals $\{I_1, I_3, I_5, I_7, I_8, I_9\}$ on 2 machines (shown as thick lines), and thus requests 5, 6, 7, 10, 12, 14 are hits and the rest are faults. This yields the optimal cache cost for the minimum number of faults.

Lemma 8.1 *Let $\mathcal{R}$ be a request sequence. Let $\mathcal{I}' = \mathcal{I}(\mathcal{R}) \setminus \{I_i : I_i \text{ is unfinished}\}$. Let $S \subset \mathcal{I}'$ be a feasible schedule of $\mathcal{I}'$ on $k-1$ machines. Then $\mathcal{R}$ can be served with a cache of size k such that all requests r_j with $I_i \in S$ and $j = i + |I_i| + 1$ are hits, with a total cache usage of $|\mathcal{R}| + \sum_{I_i \in S} |I_i|$.*

Proof: Given a feasible schedule S, $\mathcal{R}$ can be served, at each request r_i, by bringing $page(r_i)$ to the cache, and evicting it before serving request r_{i+1} if and only if $I_i \notin S$. Thus, a page σ requested in r_i that is not evicted before serving r_{i+1} will remain in cache at least until the next request to σ, resulting in a hit. Since S is a valid schedule on $k-1$ machines, there are at most $k-1$ overlapping intervals at any time, and thus there are enough cells to keep the corresponding pages in cache. Moreover, since there are k cells in cache and at most $k-1$ pages being held in cache, there is always one cell to store the page of the current request at each step, including all requests that result in faults. Note as well that S does not contain any unfinished intervals and thus for all $I_i \in S, page(r_{e(I_i)+1})$ exists and is equal to $page(r_i)$. Since all pages that correspond to requests that result in hits are kept in cache for the duration of their intervals, and an extra cell is used for the page of each request, the cache usage is $|\mathcal{R}| + \sum_{I_i \in S} |I_i|$. ∎

In light of Lemma 8.1, when describing the actions of an algorithm while serving a request $\mathcal{R}$, we sometimes use the terminology related to interval scheduling. Thus we say that an algorithm schedules an interval I_i to mean that it keeps a page $page(r_i)$ in cache until request r_j with $j = i + |I_i| + 1$ (and $page(r_j) = page(r_i)$). We define the *cache cost* of a request r_j as the number of requests that $page(r_j)$ was kept in cache for after r_i, which equals $|I_i|$ if r_j is a hit, and is smaller otherwise.

If we are only interested in minimizing faults then the problem corresponds to Maximal Interval Scheduling. This problem can be solved by sorting intervals in increasing order of end time, and then greedily scheduling intervals while there are available machines. Minimizing the number of faults while at the same time using the least possible cache can be solved instead by computing the maximum weight schedule in the corresponding interval representation. Weighted interval scheduling on identical machines can in turn be solved by formulating the problem as a minimum cost flow problem [Arkin and Silverberg, 1987; Bouzina and Emmons, 1996]. Since we are interested in minimizing cache usage (equivalently, minimizing processing time in the interval schedule), for a given instance $\mathcal{R}$ we assign weights to intervals using the following corollary from [Bouzina and Emmons, 1996]:

Corollary 8.1 *[Bouzina and Emmons, 1996, Corollary 2] For each interval $I_j \in I(\mathcal{R})$ with processing time $|I_j|$, define a weight $w_j = M - |I_j| + 1$ [2], where M is a positive real number such that $M \geq \sum |I_j|$. Then a solution to maximum weight interval scheduling gives an optimal solution to maximal interval scheduling with minimum total processing time.*

Using the above weight assignment and a maximum weight scheduling algorithm we obtain a way of serving request $\mathcal{R}$ with the minimum number of faults, and with minimum cache usage. Recall that in general we seek to minimize the total cost of serving a sequence $\mathcal{R}$, defined as $fF(\mathcal{R}) + cC(\mathcal{R})$, which does not necessarily imply minimizing the number of faults F. However, we can still use the same reduction to interval scheduling and subsequently to minimum cost flow by first eliminating from $\mathcal{I}(\mathcal{R})$ all intervals whose length multiplied by c is greater than the fault cost. It is easy to see that any solution that includes an interval I_i such that $c|I_i| > f$ could be modified to obtain a smaller cost by not scheduling that interval and paying for the fault instead. Hence, an optimal algorithm does not schedule any interval whose cost is higher than that of the fault cost. The resulting optimal offline algorithm is shown in Algorithm 8.1, where MaxWeightSchedule is an algorithm for maximum weight interval scheduling. Clearly, computing the interval representation of a request $\mathcal{R}$ of n pages (lines 2-17) takes $O(n)$ time, while MaxWeightSchedule takes time $O(m^2 \log m)$ [Arkin and Silverberg, 1987], where m is the number of intervals of the weighted interval scheduling problem. Naturally, $m = O(n)$, which yields an $O(n^2 \log n)$ total running time. However, in general m might be much smaller than n, depending on the number of different pages in $\mathcal{R}$ and the number of intervals whose length is greater than f/c.

Theorem 8.1 *Given a request sequence $\mathcal{R}$ of length n, a cache size k, and constants $f \geq 0$ and $c > 0$, we can compute in $O(n^2 \log n)$ time a way of serving $\mathcal{R}$ with $F(\mathcal{R})$ faults and cache usage $C(\mathcal{R})$, such that $f \cdot F(\mathcal{R}) + c \cdot C(\mathcal{R})$ is minimized.*

[2] We add 1 to the weight of each interval so that intervals have non-zero weight if all intervals have length 0.

Algorithm 8.1 Minimum Cache Usage Cost($\mathcal{R}, k, f, c$)

1: {Compute interval representation of $\mathcal{R}$ without unfinished intervals}
2: $\mathcal{I} = \emptyset$ {$\mathcal{I}$ is the set of unfinished intervals}
3: $M \leftarrow 0$ {M is the positive number such that $M \geq \sum_{I_j \in \mathcal{I}} |I_j|$ that is used to define the weights of each interval in line 17}
4: **for** $j = 1$ to $|\mathcal{R}|$ **do**
5: lastRequest[$page(r_j)$] $= -1$ {lastRequest keeps track of the last time each page was requested}
6: **for** $j = 1$ to $|\mathcal{R}|$ **do**
7: $\sigma \leftarrow page(r_j)$
8: $i \leftarrow$lastRequest[σ]
9: **if** $i \neq -1$ **then** {This is not the first request to σ}
10: $s(I_i) \leftarrow i + 1$ {Set the start and end of the interval. Recall that $|I_i| = e(I_i) - s(I_i) + 1$}
11: $e(I_i) \leftarrow j - 1$
12: **if** $c \cdot |I_i| \leq f$ **then** {Only consider intervals that are not too costly}
13: add I_i to $\mathcal{I}$
14: $M \leftarrow M + |I_i|$
15: lastRequest[σ] $= j$ {Set the last request to σ to the current request}
16: **for** $i = 1$ to $|\mathcal{I}|$ **do**
17: $w(I_i) = M - |I_i| + 1$ {Set the weight of each interval}
18: $S \leftarrow$MaxWeightSchedule($\mathcal{I}, k - 1$) {Compute the maximum weight schedule of $\mathcal{I}$ on $k - 1$ identical machines}
19: **return** $f(|\mathcal{R}| - |S|) + c(\sum_{I_i \in S} |I_i| + |\mathcal{R}|)$ {Requests in $\mathcal{R}$ corresponding to intervals in S are hits}

8.3 Online Algorithms

In this section we present a family of online algorithms that adapt to the relative cost of a fault versus the cache cost. These algorithms are k-competitive in the worst case (when $f \gg c$), but can achieve significant cache savings and smaller cost when the cache cost is closer to the fault cost. As a warm-up, we show that while classical optimal paging algorithms are also k-competitive, this ratio does not improve when the cache cost is high relative to the fault cost.

Lemma 8.2 *Let A be any marking or conservative paging algorithm. The competitive ratio of A is at most k.*

Proof: Let $\mathcal{R}$ be any sequence and consider its k-phase partition. Since A is marking or conservative, it faults at most k times per phase. In addition, for a sequence of n requests any algorithm has a cache cost of at most cnk. On the other hand, any algorithm must fault at least once per phase and must pay at least cn for the cache. Let ϕ be the number of phases in $\mathcal{R}$. Then, $A(\mathcal{R})/OPT(\mathcal{R}) \leq (fk\phi + cnk)/(f\phi + cn) = k$. ∎

Lemma 8.3 *Let A be any lazy paging algorithm. Then the competitive ratio of A is at least k.*

Proof: Let $\alpha = f/c$ (recall we assume $c > 0$). Suppose that α is finite. Let $\mathcal{R}$ be the request sequence $\{\sigma_1, \sigma_2, \ldots, \sigma_{k-1}, (\sigma_k)^x\}$, with $\sigma_i \neq \sigma_j$ for all $i \neq j$, where $(\sigma)^x$ denotes a sequence of x consecutive requests to σ. Since A is a lazy algorithm, it will not evict any page from the cache, thus only faulting in the first k requests but using the entire cache until the end of the sequence. Hence, $A(\mathcal{R}) \geq fk + xkc$. An optimal algorithm can use only one cell of cache for a cost of $OPT(\mathcal{R}) = fk + xc$. Since x can be made arbitrarily large and f/c is bounded, the result follows. In the case of an unbounded α, the same sequence used in the classic lower bound of k applies: request the page in $\{\sigma_1, \ldots, \sigma_{k+1}\}$ not currently in the cache. Thus, $A(\mathcal{R}) \geq n(f + c)$ and $OPT(\mathcal{R}) \leq (n/k)f + nkc$ and the ratio approaches k as $\alpha \to \infty$. ∎

8.3.1 A Family of Cost-Sensitive Online Algorithms

Definition 8.4 *For any online paging algorithm A, we define A_d as the algorithm that acts like A, except that for each r_i, it evicts page(r_i) at time $i + d$ if this page has not been requested by that time and is still in the cache. In this case, we say that page(r_i) expires at time $i + d$. We say that a page suffers an early eviction if it is evicted as a result of a capacity miss, according to A's eviction policy. Thus, if page(r_i) is not requested or evicted early within $[i, i + d]$, it will reside in cache for $d + 1$ requests.*

We restrict our choice of online algorithms in the definition above to marking and conservative algorithms and set $d = \lfloor \alpha \rfloor = \lfloor f/c \rfloor$. Consider A=LRU. For some instances LRU could have a better cost than LRU_α[3]. We now show, however, that the cost of LRU_α is always at most twice the cost of LRU, while there exists a sequence for which the cost of LRU is k times worse than the cost of LRU_α, which is the worst possible ratio for a marking algorithm. This direct comparison of two algorithms can be seen as a variant of *relative interval analysis* [Dorrigiv et al., 2009] (see Section 6.2.3) that uses the cost ratio instead of the cost difference.

Definition 8.5 (Relative ratio interval) *Let A,B be two online algorithms and let $A(\mathcal{R})$ and $B(\mathcal{R})$ denote their cost on a request $\mathcal{R}$ of a minimization problem, respectively. Let $Min(A, B) = \liminf_{n\to\infty}(\min_{|\mathcal{R}|=n}\{A(\mathcal{R})/B(\mathcal{R})\})$ and $Max(A, B) = \limsup_{n\to\infty}(\max_{|\mathcal{R}|=n}\{A(\mathcal{R})/B(\mathcal{R})\})$. Then the relative ratio interval of A and B is*

$$\mathcal{I}(A, B) = [Min(A, B), Max(A, B)].$$

An interval $[\gamma, \delta]$ approximates $\mathcal{I}(A, B)$, denoted as $\mathcal{I}(A, B) \subseteq [\gamma, \delta]$, if $\gamma \leq Min(A, B)$ and $Max(A, B) \leq \delta$.

Thus, if $\mathcal{I}(A, B) \subseteq [\gamma \geq 1, \delta > 1]$ we say that B dominates A, since on any sequence B is no worse than A and there is at least one sequence for which B is better than A. Lemma 8.5

[3]To keep notation simple, we refer to $A_{\lfloor\alpha\rfloor}$ as A_α.

and Theorem 8.2 show that $\mathcal{I}(\text{LRU}, \text{LRU}_\alpha) \subseteq [1/2, k]$. Thus, although LRU does not properly dominate LRU_α, the latter is generally preferable to the former.

Throughout the proofs in this section we use the following lemma:

Lemma 8.4 *[Panagiotou and Souza, 2006, Corollary 11] Let two vectors $\vec{x} = (x_1, \ldots, x_n) \geq \vec{0}$ and $\vec{y} = (y_1, \ldots, y_n) > \vec{0}$ be given. Let π denote a permutation of $(1, \ldots, n)$. Then*

$$\frac{\sum_{i=1}^{n} x_i}{\sum_{i=1}^{n} y_i} \leq \min_{\pi} \max \left\{ \frac{x_i}{y_{\pi(i)}} : 1 \leq i \leq n \right\} \leq \max \left\{ \frac{x_i}{y_{\pi(i)}} : 1 \leq i \leq n, \text{ and fixed } \pi \right\}$$

Lemma 8.5 *Let $\alpha = f/c$ be finite. Then $Max(LRU, LRU_\alpha) = k$.*

Proof: Note that since LRU is k-competitive, then for all $\mathcal{R}$, $LRU(\mathcal{R})/LRU_\alpha(\mathcal{R}) \leq k$. Consider the sequence $\mathcal{R} = \{\sigma_1, \sigma_2, \ldots, \sigma_{k-1}, (\sigma_k)^x\}$ used in the proof of Lemma 8.3. The cost of LRU is at least $fk + xkc$, while LRU_α keeps the first $k-1$ pages only for $\lfloor \alpha \rfloor$ requests, incurring a cost of $fk + c(\lfloor \alpha \rfloor (k-1) + x)$. Since x can be made arbitrarily large and $\alpha = f/c$ is bounded, $LRU(\mathcal{R})/LRU_\alpha(\mathcal{R}) \geq k$, and the results follows. ∎

Theorem 8.2 *Assume $k \geq 2$. Then, for all request sequences $\mathcal{R}$, $LRU_\alpha(\mathcal{R}) \leq 2 \cdot LRU(\mathcal{R})$, and therefore $Min(LRU, LRU_\alpha) \geq 1/2$.*

Proof: Let $\mathcal{R}$ be any sequence. Let F and C denote the faults and cache cost of LRU on $\mathcal{R}$ and let F_α and C_α denote the corresponding costs for LRU_α. Let $C_\alpha = C_{fh} + C_{hh} + C_{ff} + C_{hf} + \gamma$, where C_{fh} is the cache cost of requests that are faults for LRU_α and hits for LRU, and C_{ff}, C_{hh}, and C_{hf} are defined analogously. γ is the cost of keeping unfinished intervals. We will use the following properties: (1) every page of a request sequence is kept in LRU's cache for at least as long as in LRU_α's cache; and (2) any request that is a fault for LRU_α and is a hit for LRU corresponds to a page that expired in LRU_α's cache.

To see that Property (1) holds, note that if LRU evicts a page σ upon request r_i, then either σ has expired in LRU_α's cache, or it is evicted at this point on request r_i as well. The latter holds because if σ was evicted from LRU's cache, then there are k distinct requests since the last request to σ, and since σ has not expired in LRU_α's cache, there are $k-1$ pages in LRU_α's cache that have not expired either and are younger than σ. Hence upon request r_i, LRU_α evicts σ as well. Property (1) implies that every request that is a hit for LRU_α is a hit for LRU, and thus $F_\alpha \geq F$ and $C_{hf} = 0$. Property (2) follows from the fact if LRU_α evicts a page σ due a capacity miss, then its cache is full and since all pages stay longer in LRU's cache, then LRU's cache holds the same pages and evicts σ as well, hence the next request to σ is also a fault for LRU.

Property (1) implies as well that LRU's cache cost is $C \geq C_{fh} + F_\alpha - F + C_{ff} + C_{hh} + \gamma$. Moreover, both properties imply that $C_{fh} = \lfloor\alpha\rfloor(F_\alpha - F)$. Hence,

$$\begin{aligned}
\frac{LRU_\alpha(\mathcal{R})}{LRU(\mathcal{R})} &\leq \frac{fF_\alpha + c(\lfloor\alpha\rfloor(F_\alpha - F) + C_{hh} + C_{ff} + \gamma)}{fF + c(\lfloor\alpha\rfloor(F_\alpha - F) + F_\alpha - F + C_{ff} + C_{hh} + \gamma)} \\
&\leq \frac{fF_\alpha + c\lfloor\alpha\rfloor(F_\alpha - F)}{fF + c(\lfloor\alpha\rfloor(F_\alpha - F) + F_\alpha - F)} \qquad \text{(by Lemma 8.4)} \\
&= \frac{\alpha F_\alpha + \lfloor\alpha\rfloor(F_\alpha - F)}{\alpha F + \lfloor\alpha\rfloor(F_\alpha - F) + F_\alpha - F}
\end{aligned}$$

We now show that the above expression is bounded by 2. Let $\delta = \alpha - \lfloor\alpha\rfloor < 1$. Since $F_\alpha \geq F$,

$$\begin{aligned}
0 &\leq \alpha F + (F_\alpha - F)(2 - \delta) \\
0 &\leq \alpha F + 2(F_\alpha - F) + \lfloor\alpha\rfloor(F_\alpha - F) + \alpha(F - F_\alpha) \\
\alpha F_\alpha + \lfloor\alpha\rfloor(F_\alpha - F) &\leq 2\alpha F + 2\lfloor\alpha\rfloor(F_\alpha - F) + 2(F_\alpha - F) \\
\frac{\alpha F_\alpha + \lfloor\alpha\rfloor(F_\alpha - F)}{\alpha F + \lfloor\alpha\rfloor(F_\alpha - F) + F_\alpha - F} &\leq 2 \qquad \blacksquare
\end{aligned}$$

8.3.2 Bounds on the Competitive Ratio of A_α

We now show that for any marking or conservative algorithm A, the competitive ratio of A_α adapts to the relative costs of faults and hits, being at most 2 when the cost of faults is relatively small, and matching the competitiveness of traditional paging algorithms when the cost is negligible.

Theorem 8.3 *Let A be any marking or conservative algorithm and let $\alpha = f/c$. Assume $k \geq 2$. The competitive ratio of A_α is at most $2 - \frac{1+\alpha-\lfloor\alpha\rfloor}{\alpha+1}$ if $\alpha < k$ and $\min\left\{k, \frac{\alpha(k+1)}{k+\alpha-1}\right\}$ if $\alpha \geq k$.*

Proof: Let $\mathcal{R}$ be any request sequence. Let us assume first that $\alpha < k$. Since A_α keeps each page for no more than $\lfloor\alpha\rfloor + 1 \leq k$ requests (including the request to the page itself), there can be at most k pages in the cache at any given time. Thus, A_α does not incur early evictions. Let $\mathcal{I}(\mathcal{R})$ be the interval representation of $\mathcal{R}$. A_α will incur a hit on every request following an interval $I_i \in \mathcal{I}(\mathcal{R})$ such that $|I_i| \leq \lfloor\alpha\rfloor$, and it will fault on any request following a longer interval. Let w be the number of distinct pages in $\mathcal{R}$. A_α will incur an extra cost of at most $w(f + c\lfloor\alpha\rfloor)$, since it will fault on each first request to a page, and will hold the page corresponding to their last request for at most $\lfloor\alpha\rfloor$ requests. Thus, $A_\alpha(\mathcal{R}) \leq c(\sum_{|I_i|\leq\alpha} |I_i| + n) + (F + w)(f + c\lfloor\alpha\rfloor)$,

where F is the number of feasible intervals that are longer than α. Since $f < ck$, OPT can schedule all intervals of length at most f/c and will not schedule longer intervals. In addition, OPT will fault in every first request to a page but will not schedule any unfinished intervals. Thus, $OPT(\mathcal{R}) \geq c(\sum_{|I_i| \leq \alpha} |I_i| + n) + f(F + w)$. Therefore,

$$\frac{A_\alpha(\mathcal{R})}{OPT(\mathcal{R})} \leq \frac{c(\sum_{|I_i| \leq \alpha} |I_i| + n) + (F + w)(f + c\lfloor \alpha \rfloor)}{c(\sum_{|I_i| \leq \alpha} |I_i| + n) + f(F + w)}$$

The above ratio is maximum when $F = n - w$, and thus $\frac{A_\alpha(\mathcal{R})}{OPT(\mathcal{R})} \leq \frac{\alpha + \lfloor \alpha \rfloor + 1}{\alpha + 1} = 2 - \frac{1 + \alpha - \lfloor \alpha \rfloor}{\alpha + 1} \leq 2 - \frac{1}{\alpha + 1}$.

Assume now $\alpha \geq k$. Consider a phase j in the k-phase partition of $\mathcal{R}$ that is not the last phase. Although A is marking or conservative, A_α might incur more than k faults in a phase, since it might evict a marked page during the phase. However, a marked page will be evicted only when it expires, and not due to capacity misses. Let m be the number of requests in the phase. Call a request *external* if it is the first request to a page in the phase (i.e., this is the first request to this page in $\mathcal{R}$, or the request is the end of an interval that started in an earlier phase), and *internal* otherwise (i.e., the interval ending in the request started in phase j). Let E and I denote the sets of intervals ending in external and internal requests, respectively, and let F_I and F_E be the number of faults of A_α on internal and external requests in phase j, respectively. Let $H_I = \sum_{I_i \in I, |I_i| \leq \alpha} |I_i|$ be the cache cost of hits on internal requests. Let $H_E = \sum_{I_i \in E'} |I_i|$, where $E' \subseteq E$ is the set of intervals ending in external requests that result in hits and let $G = \sum_{I_i \in E \setminus E'} |I_i|$. The total cost of A_α in phase j for these requests is

$$A_\alpha = f(F_I + F_E) + c(H_I + H_E + m + G + \lfloor \alpha \rfloor F_I)$$

Note that we must add the cost of unfinished intervals to the total cost of A_α. These correspond to last requests to certain pages in $\mathcal{R}$. Instead of charging the cost of each of these requests to the phase when they are requested, we charge it to the phase in which the corresponding page was first requested in $\mathcal{R}$. Let u_j be the number of first requests to a page during phase j. A_α pays at most $u_j \alpha$ for keeping these pages in the cache after they are last requested. The faults on these u_j requests are included in F_E. Thus, the cost of A_α in phase j becomes

$$A_\alpha^j = f(F_I + F_E) + c(H_I + H_E + m + G + \lfloor \alpha \rfloor (F_I + u_j))$$

On internal requests, an optimal algorithm will fault exactly on the same requests as A_α, since OPT does not schedule intervals longer than α and it can certainly schedule all other internal intervals. Note that this implies an upper bound of k on the cost ratio in each phase: the total cache cost of A_α in the phase is at most mk, and the number of external faults is $F_E \leq k$. Since

OPT incurs at least one external fault in the phase and incurs a cache cost of at least m, we have $A^j_\alpha/OPT_j \leq (f(F_I + F_E) + cmk)/(f(F_I + 1) + cm) \leq k$.

We can provide a more refined analysis by taking into account the actual use of cache of A_α. Since OPT faults on the same pages as A_α on internal requests, OPT pays $fF_I + cH_I$ for these requests. However, OPT might incur hits on external requests that resulted in faults for A_α. Observe first that all external requests that result in hits for A_α are also hits for OPT. To see this, note that any hits for A_α on external requests must be to pages that were requested in phase $j-1$. Otherwise, the page should have been evicted due to a capacity miss or it expired during phase $j-1$. In particular, the first request in phase j must be a fault for A_α. Hence, the cache cost of every external hit for A_α is smaller than the cache cost of this first request. Therefore, OPT can schedule all intervals corresponding to A_α's external hits (because there are at most $k-1$ such intervals) and it will schedule them because their length is at most α. The cache cost of these hits for OPT is then cH_E.

Consider now the external requests that result in faults for A_α. The cost of these intervals for A_α is $G = E_h + E_f$, where E_h and E_f are the cache costs paid by A_α on external intervals that end in requests resulting in hits and faults for OPT, respectively. Let h_{OPT} be the number of such hits. The cost of these hits for OPT is at least $E_h + h_{OPT}$. In addition, the cost for external faults is $F_{OPT}f$, where $F_{OPT} = k - h_{OPT} - k + F_E = F_E - h_{OPT} \geq u'_j$, where $u'_j = \max\{u_j, 1\}$, since OPT faults in any first request during the phase and it must incur at least one external fault in the phase. Thus, the cost of OPT in phase j is at least

$$OPT^j \geq f(F_I + F_{OPT}) + c(H_I + H_E + m + E_h + h_{OPT})$$

Therefore,

$$\begin{aligned}
\frac{A^j_\alpha}{OPT^j} &\leq \frac{f(F_I + F_E) + c(H_I + H_E + m + G + \lfloor\alpha\rfloor(F_I + u_j))}{f(F_I + F_{OPT}) + c(H_I + H_E + m + E_h + h_{OPT})} && (8.3.1)\\
&\leq \frac{\alpha(2F_I + F_E + u_j) + H_I + H_E + m + E_h + E_f}{\alpha(F_I + F_{OPT}) + H_I + H_E + m + E_h + h_{OPT}} && (8.3.2)\\
&\leq \frac{\alpha(2F_I + F_E + F_{OPT}) + H_I + H_E + m + E_h}{\alpha(F_I + F_{OPT}) + H_I + H_E + m + E_h + h_{OPT}} \quad (\text{as } E_f \leq \alpha(F_{OPT} - u_j)) && (8.3.3)\\
&\leq \max\left\{2, \frac{\alpha(F_E + F_{OPT})}{\alpha F_{OPT} + h_{OPT}}\right\} \quad (\text{by Lemma 8.4}) && (8.3.4)\\
&= \max\left\{2, \frac{\alpha(F_E + F_{OPT})}{\alpha F_{OPT} + F_E - F_{OPT}}\right\} && (8.3.5)
\end{aligned}$$

Note that the second expression in (8.3.5) is an increasing function in F_E and decreases with

F_{OPT} (since $\alpha \geq k \geq 2$ and $F_{OPT} \geq u'_j$), and since $u'_j \leq F_{OPT} \leq F_E \leq k$,

$$\frac{A^j_\alpha}{OPT^j} \leq \max\left\{2, \frac{\alpha(k+u'_j)}{u'_j(\alpha-1)+k}\right\}$$

Finally, the right expression above is maximum when u'_j is minimum, hence

$$\begin{aligned}\frac{A^j_\alpha}{OPT^j} &\leq \max\left\{2, \frac{\alpha(k+1)}{k+\alpha-1}\right\} \\ &= \frac{\alpha(k+1)}{k+\alpha-1} \qquad \text{(since } \alpha \geq 2\text{)}\end{aligned}$$

Thus, the cost ratio of each phase is $r = \min\left\{k, \frac{\alpha(k+1)}{k+\alpha-1}\right\}$. Adding up all the costs for all phases but the last one we have $\sum_j A^j_\alpha / \sum_j OPT^j \leq r$, and since the cost of A_α in the last phase is at most $2fk$, we have $A(\mathcal{R}) \leq rOPT(\mathcal{R}) + 2fk$. ∎

Lemma 8.6 gives a lower bound on the competitive ratio for A_α, which matches the upper bound for $\alpha < k-1$. For larger values of α the gap between upper and lower bounds is reduced as α grows. Lemma 8.7 gives a straightforward smaller lower bound for any online algorithm.

Lemma 8.6 *For A marking or conservative, the competitive ratio of A_α is at least $2 - \frac{1+\alpha-\lfloor\alpha\rfloor}{\alpha+1}$ if $\alpha < k-1$ and $\frac{\alpha k+k^2/2}{\alpha+k^2}$ otherwise.*

Proof: Assume that $\alpha < k-1$ and let $\mathcal{R}$ be the sequence such that each request is for a page not in A_α's cache among pages $\{\sigma_1, \sigma_2, \ldots, \sigma_{\lfloor\alpha\rfloor+2}\}$. A page not in cache always exists since A_α keeps each page for at most $\lfloor\alpha\rfloor$ requests and thus there are at most $\lfloor\alpha\rfloor + 1 < k$ pages in the cache at any give time. Since the cache is never full A_α keeps each page for exactly $\lfloor\alpha\rfloor$ requests and thus it pays $f + c(\lfloor\alpha\rfloor + 1)$ per request. Since the interval length of each request is $\lfloor\alpha\rfloor + 1 > \alpha$, an optimal strategy will evict each page after it is served, and thus its cost per request is $f + c$. Therefore, $A_\alpha(\mathcal{R})/OPT(\mathcal{R}) \geq (\alpha + \lfloor\alpha\rfloor + 1)/(\alpha+1) = 2 - \frac{1+\alpha-\lfloor\alpha\rfloor}{\alpha+1} > 2 - \frac{2}{\alpha+1}$.

For the case $\alpha \geq k-1$, let $\mathcal{R}$ again be the sequence that requests the page not in A_α's cache, but now among $\{\sigma_1, \sigma_2, \ldots, \sigma_{k+1}\}$. Consider a phase in the k-phase partition of $\mathcal{R}$. Since A_α is marking or conservative, it does not evict any pages that are requested in the same phase unless they expire. However, since $\lfloor\alpha\rfloor \geq k-1$, and each phase has length k, a page could expire only after the last request of the phase. Thus, no pages that were requested during the phase are evicted and at the end of the phase the cache will necessary be full. Hence, in each phase A_α incurs k faults and uses at least $\sum_{i=1}^{k} i = k(k+1)/2$ cache, for a cost of $fk + ck(k+1)/2$. In turn, and optimal algorithm can keep k pages in cache at all times an incur at most one fault in the phase, for a cost of at most $f + ck^2$. Thus, $A_\alpha(\mathcal{R})/OPT(\mathcal{R}) \geq (\alpha k + k^2/2)/(\alpha + k^2)$. ∎

Lemma 8.7 *The competitive ratio of any online deterministic algorithm is at least* $\frac{k(\alpha+1)}{\alpha+k^2}$.

Proof: Let $\mathcal{R}$ be the sequence that request the page not in A's cache among $\{\sigma_1, \sigma_2, \ldots, \sigma_{k+1}\}$. Since A faults on every request and uses at least one cell per request $A(\mathcal{R}) \geq fn + cn$. OPT can fault at most once every k requests and uses at most kn cache. Thus $A(\mathcal{R})/OPT(\mathcal{R}) \geq n(f+c)/((n/k)f + knc) = k(\alpha+1)/(\alpha+k^2)$. ∎

Locality of Reference

The classic paging cost model has been criticized for not being able to capture the benefit of online algorithms on sequences with high locality of reference [Borodin and El-Yaniv, 1998]. Various studies have analyzed the competitiveness of paging algorithms in a parameterized manner, attempting to capture relevant characteristics of sequences such as, for example, locality and typical memory accesses [Panagiotou and Souza, 2006], and attack rate [Moruz and Negoescu, 2012]. Similarly to the full access cost model [Torng, 1998] (see Section 6.2.2), our cache usage model is amenable to the analysis of algorithms in terms of the locality of reference of input sequences. We now give a parameterized competitive ratio for A_α that varies with the locality of reference of the input sequence, for which we use the definition in terms of the average phase length in its k-phase partition.

Theorem 8.4 *Let A be any marking or conservative algorithm, let $\alpha = f/c$, and let $k \geq 2$. Let $\mathcal{R}$ be any request sequence and let ϕ be the number of phases in $\mathcal{R}$'s k-phase partition. Let $L(\mathcal{R}) = |\mathcal{R}|/\phi$. Then $A_\alpha(\mathcal{R})/OPT(\mathcal{R}) \leq 2$ if $L(\mathcal{R}) > k\alpha(\alpha-2)$, and $\frac{A_\alpha(\mathcal{R})}{OPT(\mathcal{R})} \leq 1 + \frac{\alpha k+1-\alpha}{\alpha+k-1+L(\mathcal{R})}$ otherwise.*

Proof: From the proof of Theorem 8.3, Equation (8.3.3), the cost of A_α in the j-th phase is $A_\alpha^j \leq c(\alpha(2F_I + F_E + F_{OPT}) + H_I + H_E + m + E_h)$, where F_I and F_E are the number of internal and external faults, respectively, H_I and H_E the cache cost of intervals that result in internal and external hits, respectively, m is the number of requests in the phase, and E_h is the cache cost of requests that are faults for LRU_α and hits for OPT. The cost of OPT in this phase is at least $OPT^j \geq f(F_I + F_{OPT}) + c(H_I + H_E + m + E_h + h_{OPT})$. Summing over j we obtain

$$\frac{\sum A_\alpha^j}{\sum OPT^j} \leq \frac{\alpha(2\sum F_I + \sum F_E + \sum F_{OPT}) + n + \sum H_I + \sum H_E + \sum E_h}{\alpha(F_I + \sum F_{OPT}) + \sum H_I + \sum H_E + n + \sum E_h + \sum h_{OPT}} \tag{8.3.6}$$

$$\leq \max\left\{2, \frac{\alpha(\sum F_E + \sum F_{OPT}) + n}{(\alpha-1)\sum F_{OPT} + \sum F_E + n}\right\} \tag{8.3.7}$$

Suppose $\alpha \geq 2$. Then the above expression is an increasing function of F_E, and thus the ratio is maximal for $\sum F_E = \phi k$. Let $L = L(\mathcal{R}) = n/\phi$. Suppose that $L \leq k(\alpha^2 - 2\alpha)$. Then it is not

hard to verify that the above expression decreases with F_{OPT}. Since $\sum F_{OPT} \geq \phi$, and $\phi = n/L$,

$$\begin{aligned}
\frac{A_\alpha(\mathcal{R})}{OPT(\mathcal{R})} &\leq \max\left\{2, \frac{\alpha((n/L)k + (n/L)) + n}{(\alpha-1)(n/L) + k(n/L) + n}\right\} \\
&= \max\left\{2, \frac{\alpha(k+1) + L}{\alpha + k - 1 + L}\right\} \\
&= \max\left\{2, 1 + \frac{k(\alpha-1) + 1}{\alpha + k - 1 + L}\right\}
\end{aligned}$$

Now, if $L > k(\alpha^2 - 2\alpha)$, then the ratio is maximized when $\sum F_{OPT} = \phi k$. Substituting $\sum F_{OPT}$ in (8.3.7) above we obtain $\frac{A_\alpha(\mathcal{R})}{OPT(\mathcal{R})} \leq \max\left\{2, 1 + \frac{\alpha k}{\alpha k + L}\right\} = 2$. Finally, by Theorem 8.3, if $\alpha < 2$ the upper bound is 2 as well. ∎

8.4 Real World Sequences

We measured the performance of various algorithms on real world cache traces collected from 4 applications using VMTrace (for Linux) and the Etch tool (on Windows NT) [Kaplan et al., 2003]. We obtained the traces from [Kaplan] and truncated them to 3×10^6 entries. Table 8.1 shows a description of the sequences. We simulated LRU, LRU$_\alpha$, FWF, FWF$_\alpha$, FIFO, FIFO$_\alpha$, and OPT on these sequences. For each sequence, we used the size of cache that would yield a fault rate of 1% and 0.1% for LRU. Figures 8.2, 8.3, 8.4, and 8.5 show the cost ratio compared to OPT, fault rate, and average cache usage for the test input sequences for two cache sizes. For the total cost we set $c = 1$ and $f = \alpha$. We implemented the optimal offline (Algorithm 8.1) using the reduction to minimum cost flow in [Bouzina and Emmons, 1996], and solved the minimum cost flow instances using the implementation of the cost scaling algorithm in the LEMON C++ library [LEMON]. Results in these practical instances show that the cost of A_α algorithms adapt nicely to the value of α, and that their fault rate and cache usage approaches those ones of the optimal offline. In fact, the ratio A_α/OPT is never more than 2 and in most cases is close to 1. As suggested by Theorem 8.4, the cost ratio of A_α algorithms improves for sequences with higher locality. Note as well that as α grows, the performance of the traditional marking algorithms gets closer to that of its cost-sensitive counterpart, which is more noticeable for instances with smaller caches.

8.5 Conclusions

We introduced a model for paging with minimum cache usage and presented a cost-sensitive family of online algorithms whose performance adapts to the relative costs of cache and faults.

Application	Description	Length	Avg. phase length	
acroread (Windows NT)	Acrobat Reader	3×10^6	722 (k=15)	19108 (k=20)
espresso (Linux)	circuit simulator	3×10^6	196 (k=5)	1502 (k=7)
gs (Linux)	GhostScript 3.33	3×10^6	542 (k=16)	18405 (k=40)
grobner (Linux)	Grobner basis functions	3×10^6	330 (k=8)	9918 (k=22)

Table 8.1: Description of input sequences used in simulations.

The cost model that we propose is able to capture locality of reference, yielding a competitive ratio of at most 2 for inputs with high locality. Experiments on request sequences collected from actual programs agree with the theoretical results.

It would be interesting to show a better lower bound for online algorithms, and to propose and analyze other online algorithms, including randomized ones. A natural direction of research would be to evaluate the model in an application, either in theory or in practice. For example, it would be interesting to study and design a global shared caching strategy that varies the relative cache and fault cost for various threads so that the cooperative execution leads to an advantage in overall performance.

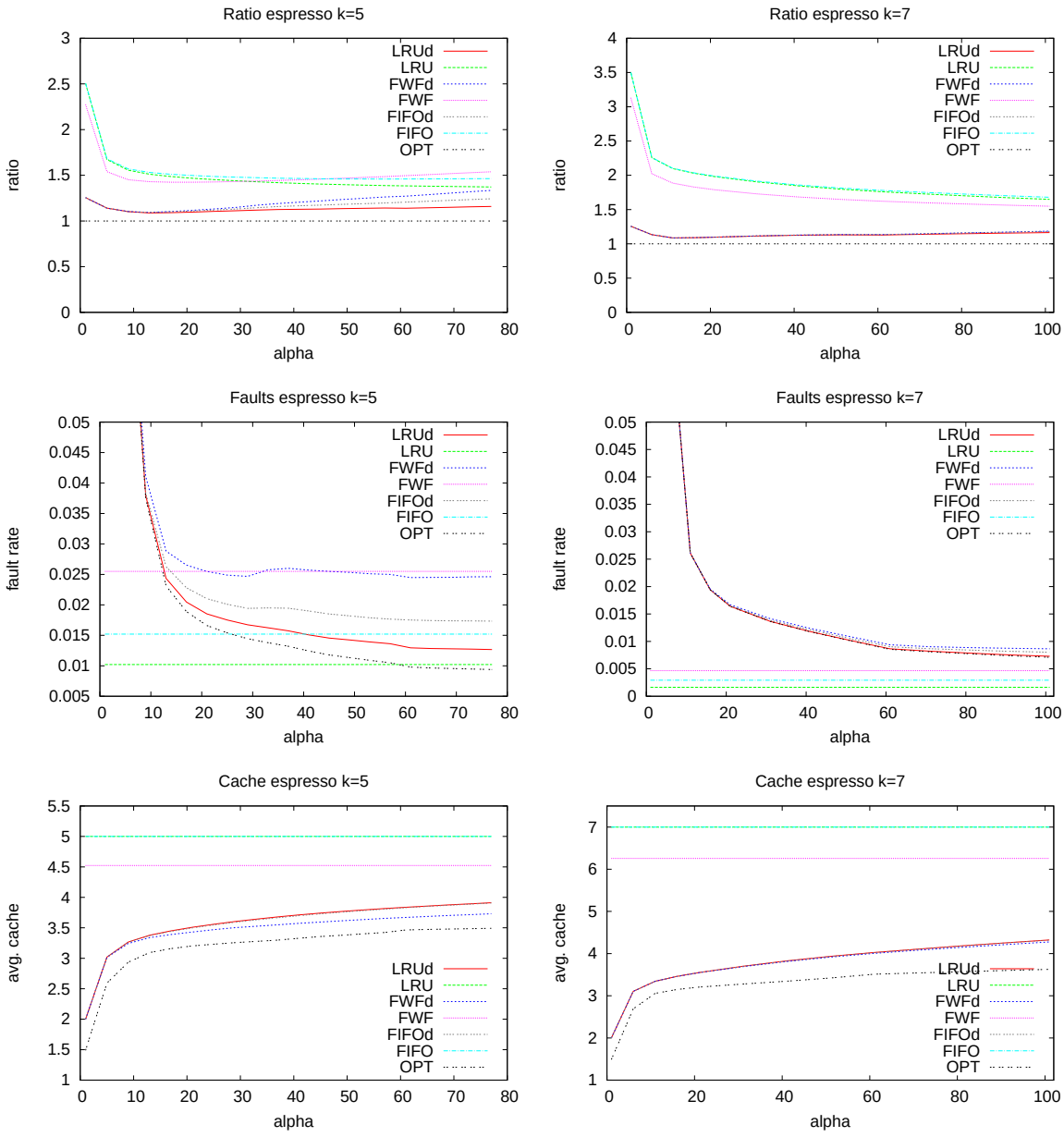


Figure 8.2: Cost ratio, fault rate, and average cache used by LRU, LRUd, FWF, FWFd, FIFO, FIFOd, and OPT (with $d = \alpha$) on sequence "espresso" of length 3×10^6 with cache sizes $k = 5$ (average phase length 196) and $k = 7$ (average phase length 1502).

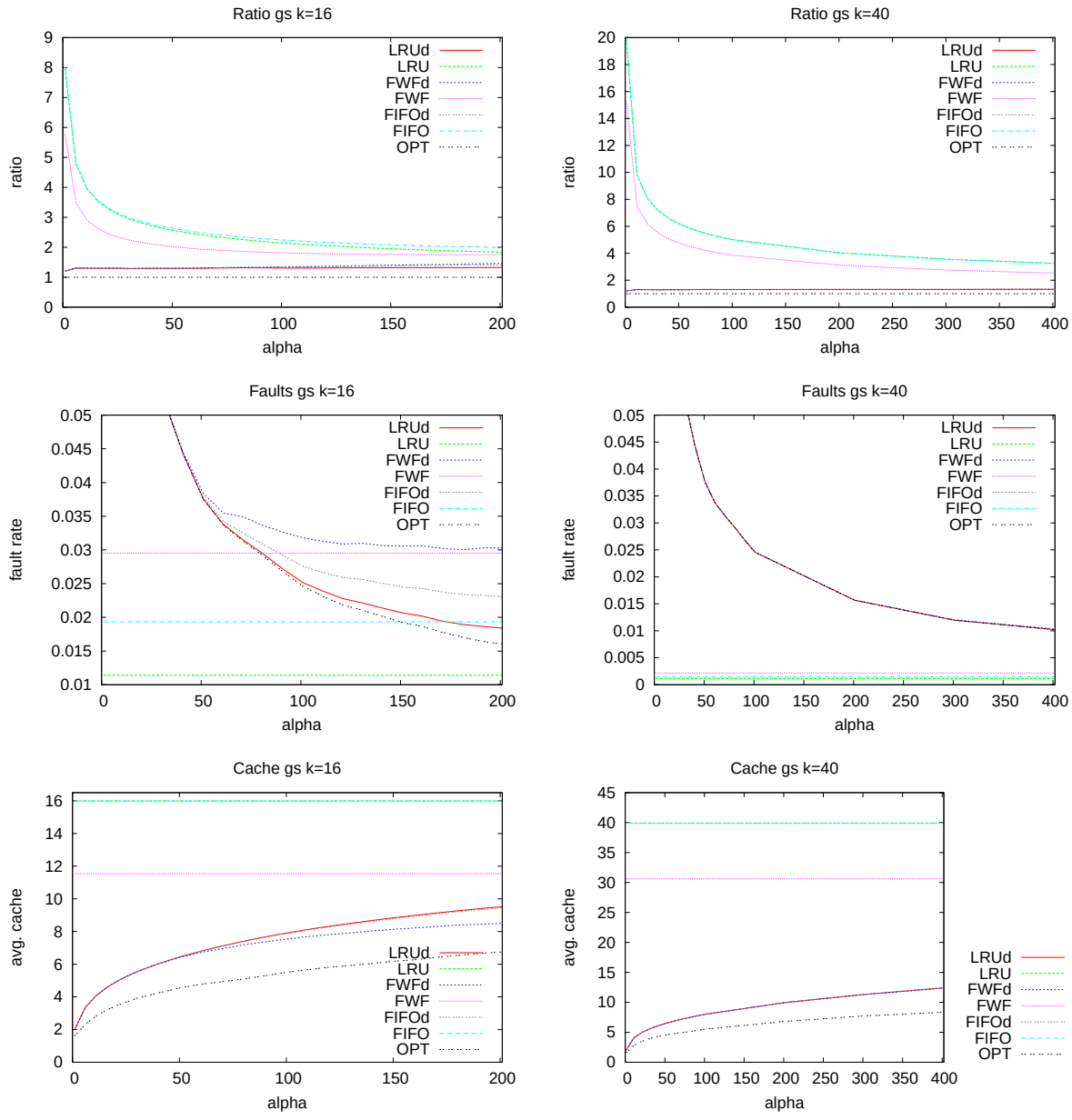


Figure 8.3: Cost ratio, fault rate, and average cache used by LRU, LRUd, FWF, FWFd, FIFO, FIFOd, and OPT (with $d = \alpha$) on sequence "gs" of length 3×10^6 with cache sizes $k = 16$ (average phase length 542) and $k = 40$ (average phase length 18405).

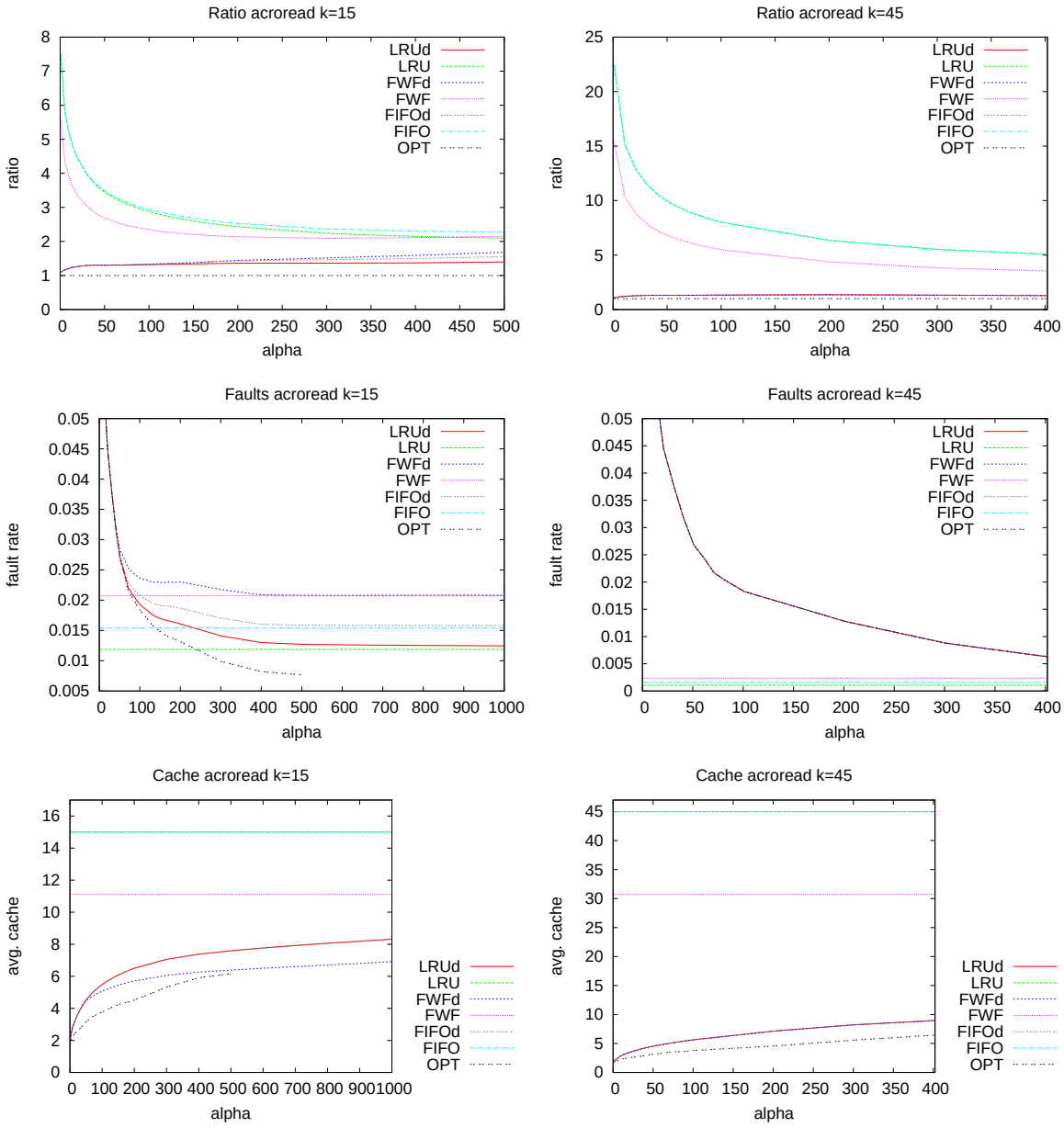


Figure 8.4: Cost ratio, fault rate, and average cache used by LRU, LRUd, FWF, FWFd, FIFO, FIFOd, and OPT (with $d = \alpha$) on sequence "acroread" of length 3×10^6 with cache sizes $k = 15$ (average phase length 722) and $k = 20$ (average phase length 19108).

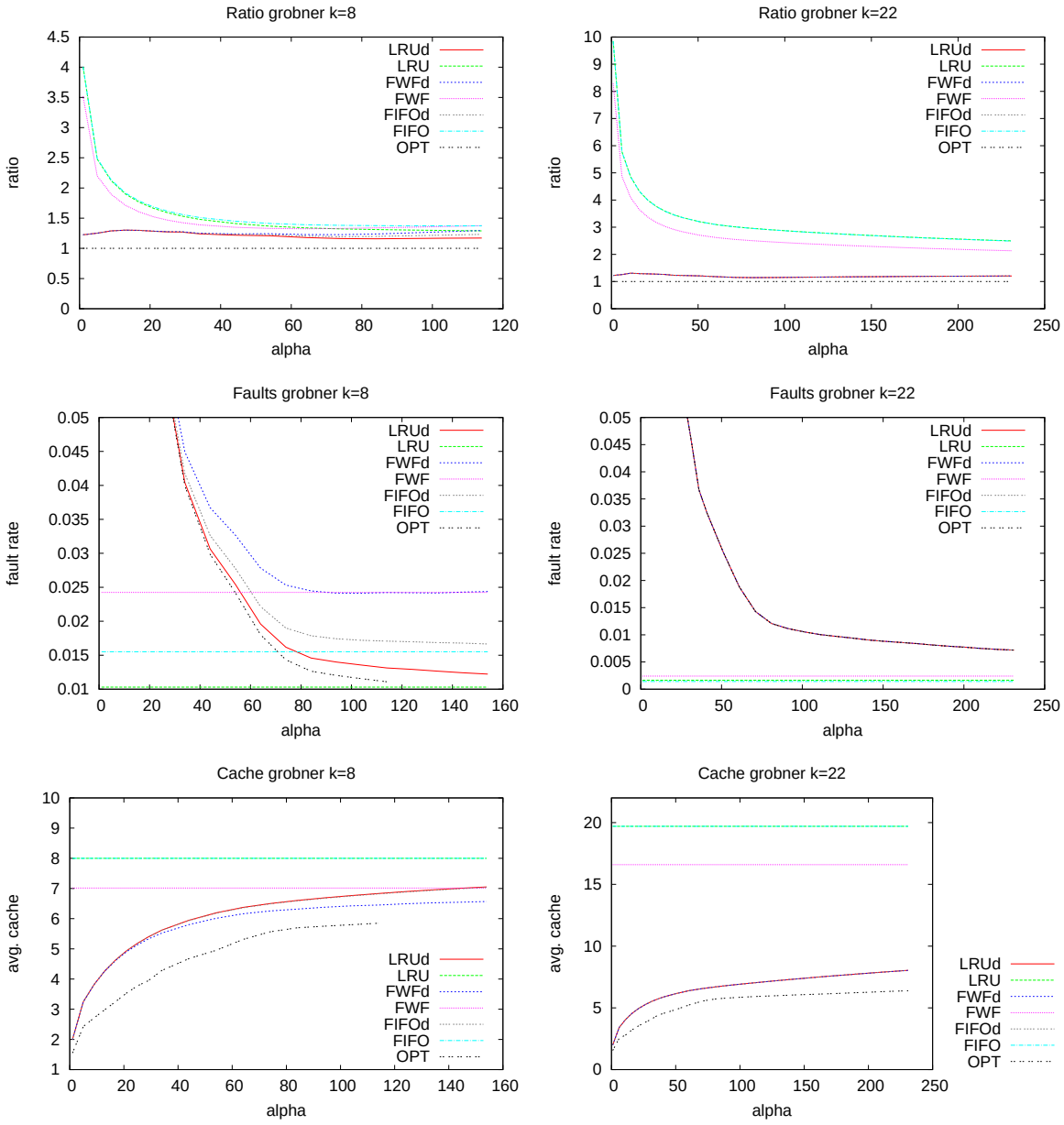


Figure 8.5: Cost ratio, fault rate, and average cache used by LRU, LRUd, FWF, FWFd, FIFO, FIFOd, and OPT (with $d = \alpha$) on sequence "grobner" of length 3×10^6 with cache sizes $k = 8$ (average phase length 330) and $k = 22$ (average phase length 9118).

Chapter 9

Toward a Generic Hybrid CPU-GPU Parallelization of Divide-and-Conquer Algorithms

In this chapter we address the design of algorithms for heterogeneous architectures that feature multi-core processors together with Graphic Processor Units (GPUs) (see Chapter 2.8.3 for a description of the GPU architecture and programming model). We propose a model for hybrid computation in these architectures and describe a generic strategy to translate sequential implementations of divide-and-conquer algorithms to a program that runs both in CPU and GPU, with a schedule that balances the workload of both processing units.

Since the appearance of multi-core architectures we have witnessed an increase in algorithms and applications designed to take advantage of the parallel processing capabilities of these now ubiquitous processors. At the same time, there exists a vast collection of graphic applications for optimized performance on GPUs. Originally designed as specialized processors for graphic operations, the development of accessible programming languages such as CUDA and OpenCL has enabled the use of GPUs for general purpose programming, known as General Purpose computing on GPUs (GPGPU). Consequently, researchers and practitioners have developed algorithms for this architecture for various classes of problems, most notably for problems that allow for data-parallel algorithms, many of which fall under the category of the so-called *embarrassingly parallel* problems. In the last few years, the increasing power and low cost of GPUs has transformed common computers into heterogeneous architectures with tremendous computing power. While for most applications capable of parallel execution there exist implementations either for multi-cores or GPUs, the vast majority of existing applications and algorithms do not yet take full advantage of the available computing power, thus leaving the current processing resources of even middle-end commodity computers largely underutilized. This scenario has motivated the

development of projects in several areas, from the proposal of new operating systems [Baumann et al., 2009; Nightingale et al., 2009] to the continuous development of tools and languages to enable an easy transition from traditional CPU code to heterogeneous platforms [Surhone et al., 2010; McCool et al., 2006; Nickolls et al., 2008; Stone et al., 2010].

Since the vector processing nature of GPUs is suitable for problems that allow efficient data-parallel algorithms, many of the existing GPU algorithms fall into this category. However, many problems allow for parallel algorithms that are task-parallel, or a combination of both task-parallel and data-parallel [Chakrabarti et al., 1995]. Thus, such problems can benefit from the use of CPU cores for non-data-parallel tasks. In fact, plans for the convergence of CPUs and GPUs into one platform by the largest microprocessor manufacturers are becoming a reality [Seiler et al., 2009; AMD, 2008], which confirms the relevance of algorithms that can be sped-up using the power of both architectures together [Barlas et al., 2011]. Many algorithms for hybrid CPU-GPU architectures have been designed in the last years, most notably for fundamental linear algebra problems [Tomov et al., 2009; Humphrey et al., 2010; Ezzatti et al., 2011], among many others. The design of efficient hybrid algorithms encompasses many challenges. A careful task division must be done so that each portion of the algorithm can run on the platform that suits best its characteristics. In addition, it is desirable that algorithms and schedulers adapt to the characteristics and current availability of the computing devices.

Our results[1] We describe a generic approach to develop algorithms for a hybrid CPU-GPU architecture, which we term Hybrid Processing Unit or HPU. We focus on algorithms for a large class of problems suitable for divide-and-conquer solutions. Starting from a sequential recursive implementation of a divide-and-conquer algorithm, we translate this implementation to parallel code that is suitable for running on both CPU and GPU, with a generic translation that can be applied with little knowledge of the particular algorithm. We propose a model for the HPU platform and analyze the optimal division of work for parallel divide-and-conquer under this model. While the analysis presented applies to divide-and-conquer algorithms, the ideas behind it are applicable to other classes of algorithms with structured dependencies between a large number of independent tasks. We then present a case study for the application of our framework using mergesort as a sample algorithm. The simplicity of our implementations confirms the practicality of our approach, while at the same time leading to significant improvements in performance over sequential implementations.

The rest of this chapter is organized as follows. We review related work in Section 9.1. We then describe the hybrid CPU-GPU model in Section 9.2. In Section 9.3 we describe the generic parallelization of divide-and-conquer algorithms for hybrid computation, and we analyze the work division and scheduling in Section 9.4. We then present a case study of our model using mergesort

[1] Results in this chapter appeared in [López-Ortiz et al., 2013] and are joint work with Alejandro López-Ortiz and Robert Suderman.

in Section 9.5 together with experimental results in Section 9.5.4. We conclude the chapter in Section 9.6 with conclusions and future research directions.

9.1 Related Work

Several researchers have developed algorithms for heterogeneous architectures. An important set of hybrid algorithm implementations are grouped in the Matrix Algebra on GPU and Multicore Architectures (MAGMA) library [Tomov et al., 2010a, 2009]. MAGMA provides hybrid implementations of several linear algebra algorithms to enable execution in both multi-core and GPUs, thus extending and adapting the LAPACK [Anderson et al., 1992] and ScaLAPACK [Blackford et al., 1997] libraries to heterogeneous architectures. Examples of these algorithms are Cholesky [Agullo et al., 2010], LU [Agullo et al., 2011a; Baboulin et al., 2012], and QR factorizations [Agullo et al., 2011b; Kurzak et al., 2012], and Hessenberg reduction [Tomov et al., 2010b]. In general, the approach for implementing hybrid CPU-GPU code in MAGMA is to schedule tasks in each computing unit according to their nature: tasks which exhibit small parallelism and that are often on the critical path are scheduled on the CPU while sets of independent tasks are scheduled in the GPU [Tomov et al., 2010a, 2009]. A recent work implements a hybrid divide-and-conquer strategy for dense symmetric and Hermitian eigenproblems [Vömel et al., 2012]. The divide-and-conquer approach is specific to these problems, in contrast to our generic approach.

Another library of high-performance linear algebra CPU-GPU hybrid implementations is CULA [Humphrey et al., 2010], which divides computation to enable the CPU and GPU to execute the tasks that each is best suited for, while at the same time carefully overlapping operations in both units.

Hybrid algorithms have been recently developed for other types of problems such as ray tracing [Budge et al., 2008], encryption and decryption of block cyphers [Barlas et al., 2011], and stencil computations [Venkatasubramanian et al., 2009], as well as to accelerate domain decomposition methods [Papadrakakis et al., 2011].

An important part of computation on heterogeneous architectures is the proper load balancing between computing units. While some implementations follow analytically determined static schedules [Barlas et al., 2011], others rely on dynamic schedules by runtime systems. StarPU was proposed as a runtime layer to facilitate the dynamic scheduling of parallel tasks in heterogeneous architectures [Augonnet et al., 2011]. The programmer is responsible for the implementation of tasks for each computing unit and to declare data dependencies between them, while the runtime system is responsible for handling data movements and efficient scheduling of tasks. The programmer can provide hints for the latter, and thus StarPU provides a high-level framework for the design of scheduling policies. StarPU has been incorporated into MAGMA for dynamic scheduling of routines on multi-gpu environments [Agullo et al., 2011a,b]. Other

runtime systems for heterogeneous CPU-GPU architectures are Anthill [Teodoro et al., 2009] and SuperMatrix [Quintana-Ortí et al., 2009].

In this work we target problems with known dependencies, and thus a tailored static division of work suits our purpose. In addition, unlike most of the works described above, which target High Performance Computing applications and solutions, the primary focus of our framework is on generality and ease of programming, and secondly on performance. While in many cases it is possible to design and implement parallel algorithms for specific problems that will likely exhibit better performance than the general solutions provided by our approach, we emphasize that our main goal is to provide a simple strategy to develop algorithms that can take advantage of the computing power available in commodity computers, rather than extracting the last ounce of performance from a given hardware architecture.

9.2 A Hybrid CPU-GPU Model

We propose a hybrid CPU-GPU model which we term Hybrid Processing Unit (HPU) and describe a balancing scheme which shares the load optimally in a near automatic fashion for certain well known families of problems.

The HPU consists of a multi-core CPU processor with p cores and a GPU device with g processing elements, which for simplicity we call GPU cores. Given that the true parallelism provided by a GPU varies significantly depending on execution aspects such as scheduling and memory accesses, we do not think about the number of GPU cores as exactly matching the number of physical processing elements but rather as a measure of the empiric degree of parallelism observed when running a suitable test program[2]. In turn, in general the number of CPU cores p might not be equal to the number of physical cores but to a parameter indicating the number of cores available for processing tasks (e.g., one or more cores can handle thread launching or other scheduling tasks).

To account for the different characteristics (and in particular speed) of CPU and GPU cores, we denote as γ_c and γ_g the number of operations per unit of time that a single CPU and GPU core can complete, respectively, with $\gamma_c > \gamma_g$. For simplicity, we normalize these factors, setting $\gamma_c = 1$ and $\gamma_g = \gamma < 1$. We assume that these architectures are balanced in the sense that the ratio γ of operations per unit hold for any kind of operations (logic or memory access), similarly to what is assumed, for example, in [Rosenberg and Chiang, 2010]. We also assume that $\gamma g > p$, and thus the raw computational power of the GPU is higher than that of the CPU.

The focus in this work is on the most common scenario of one multi-core CPU unit along with one GPU card (which we call *processing units*), although the model could easily be extended to the case of multiple GPU cards.

[2]We defer the details about how to estimate g to Section 9.5.4.

In terms of communication, transmitting w words between CPU and GPU takes time $\lambda + \delta w$, where λ is a fixed latency cost, and δ is the variable cost per word. For the kind of application that we consider in this work we do not explicitly take this cost into account, but we limit the number of data transfers between processing units to the minimum possible. Similarly, we do not explicitly consider scheduling costs, as preliminary experiments showed that the overhead was negligible.

9.3 Generic Divide-and-Conquer Parallelization

The standard approach to a divide-and-conquer (DC) algorithm involves dividing the problem into smaller subproblems, recursively solving these subproblems, and combining the solutions of the subproblems into a final solution (see Algorithm 9.1). We consider DC algorithms whose time complexity can be expressed by:

$$T(n) = aT(n/b) + f(n), \quad T(1) = \Theta(1)$$

Naturally, the algorithm works by dividing the problem in a subproblems of size n/b each and combining their solutions to obtain the final solution to the problem. The division and combination portion of the algorithm takes time $f(n)$. A DC algorithm can be parallelized in a straightforward manner by executing recursive calls in parallel, leading to a simple thread-based implementation suitable for multi-cores. Nevertheless, practical issues such as the efficient use of private and shared caches might hinder the ideal parallel performance of such implementation, with the chosen thread schedule playing a major role in this aspect.

In general, a strategy in which each recursive call launches a thread does not suit the GPU multi-processing model, since at least in some architectures GPU threads are unable to launch additional threads during execution[3]. Instead, they are designed to run hundreds of parallel threads executing one same kernel launched by the CPU host. In this sense, a breadth first execution of a DC algorithm can suit this execution mode better: the independent tasks on one level of the recursion tree can be seen as the same task being executed on different parts of the input. Given enough independent tasks, one kernel can be launched to execute an entire level of the tree in parallel.

[3]CUDA 5 does implement dynamic parallelism, enabling kernels to launch other kernels without involvement of the host. However, currently the depth of the nested computation can be at most 24 and is limited by the availability of resources in the GPU [NVIDIA, 2012]. While this might change in the future allowing for a more suitable strategy for divide-and-conquer implementations, our breadth-first strategy remains valid and applicable to a wider range of architectures which are not CUDA-enabled.

Algorithm 9.1 Generic divide-and-conquer implementation

Recursive(param)

```
1: if endCondition(param) then
2:     return BaseCase(param)
3: {param_j} ← Divide(param)
4: for j = 1 to |{param_j}| do
5:     S_j ← Recursive(param_j)
6: S ← Combine({S_j}, param)
7: return S
```

Algorithm 9.2 Breadth-first divide-and-conquer

BreadthFirst(params)

```
1: split params into basecases and recursions
2: if recursions is empty then
3:     for each param in basecases do
4:         BaseCase(param)
5:     return
6: add basecases to next_params
7: for each param in recursions do
8:     {param_j} ← Divide(param)
9:     for j = 1 to |{param_j}| do
10:        add param_j to next_params
11: BreadthFirst(next_params)
12: for each param in recursions do
13:    Combine(param)
```

9.3.1 Breadth-First Structure

The first step of our strategy to obtain a hybrid implementation of a DC algorithm is to convert the sequential code from the form in Algorithm 9.1 to one that will execute in breadth-first order. We do this by replacing multiple recursive calls with one recursive call that represents multiple subproblems, encoded in the parameters of the recursive call. Algorithm 9.2 shows the modified pseudocode. At each level of the recursion, parameters for all subproblems are encoded in *params*, some of which correspond to base cases. The rest of the parameters (recursions) are divided according to the algorithm's divide procedure (line 7) and grouped in one list of parameters (next_params) to be passed on to the one recursive call in line 11. After the recursive call, the results of subproblems in each group in the level are combined according to the combine procedure (line 12). Note that at each level subproblems corresponding to base cases are passed on to the next recursive call, and their execution is delayed until no more recursive calls remain.

Algorithm 9.3 Pseudo-code for functionGPU

functionGPU(parameters, base)

```
1: id ← get_global_id()
2: param ← parameters[id]
3: memory = base + fn(id, param)
4: thread_function(param, memory)
```

Algorithm 9.4 Pseudo-code for Sum

sum(array, size)

```
1: if size > 1 then
2:     sum(array, size/2)
3:     sum(array + size/2, size/2)
4:     array[0] ← array[0] + array[size/2] {array[0] stores the result}
```

9.3.2 Conversion to GPU Code

In order to modify the existing code for GPU execution, the thread launching system and the base-case, division, and combination steps must be suitably adapted. Each subproblem will have a separate thread. As GPU threads have a relatively small overhead for launching more threads than available cores, the advantage of processing as many tasks in parallel will take priority over launching only as many threads as can be run in parallel. Then, during execution, each GPU thread is provided a unique thread id that can be used to load its unique set of parameters and to determine any (previously allocated) memory blocks on which it will operate. This generic description is shown in Algorithm 9.3, where fn is a function on the thread's id and parameters that determines the thread's relevant memory blocks, and **thread_function** denotes the operations performed by the GPU thread.

9.3.3 Example: Divide-and-Conquer Sum

We show an example of the code translation described above for a simple divide-and-conquer procedure that computes the sum of elements in an array (see Algorithm 9.4). The pseudocode of the resulting GPU program is shown in Algorithm 9.5. This program is executed at each level of the recursion, with numSubProblems indicating the number of subproblems at the current level. For a level with b subproblems, the i-th thread computes the sum of elements i and $i + b$ in the array. The final result is stored in $array[0]$. In this case, the relevant parameter to the thread is only numSubProblems, and the relevant memory blocks for the thread are given by its id and numSubProblems. Lines 2-3 in Algorithm 9.5 correspond to **thread_function** in Algorithm 9.3.

Algorithm 9.5 Pseudo-code for GPU Sum

sum(numSubProblems, array)

```
1: id ← get_global_id()
2: if id < numSubProblems then
3:     array[id] ← array[id] + array[id + numSubProblems]
```

9.4 Work Division and Scheduling Strategies

A proper work division and scheduling is key to an efficient implementation. In a heterogeneous architecture, tasks in an algorithm should be assigned to each processing unit according to the tasks' nature and dependencies. Implementations on CPU-GPU architectures generally assign to the CPU tasks with many dependencies [Ezzatti et al., 2011] or tasks in the critical path [Tomov et al., 2010a]. In the case of the regular DC algorithms that we consider in this work, what we regard as *tasks* are the division and combination portions of the algorithm. Hence, all tasks are similar to each other in nature. On the other hand, in regular DC algorithms all paths from the root of the tree to a leaf have approximately equal lengths, and in this sense there are several critical paths, thus they cannot all be assigned to CPU cores. Instead, we assign tasks to the CPU or GPU according to the availability of parallel independent tasks at each level of the recursion tree. Note that although at each level of the recursion there are several subproblems whose division and combination functions are independent and can execute in parallel, the division and combine functions of each subproblem remain sequential. In other words, the only source of parallelism is in the recursive calls, and we do not consider parallelizations of divide and combine functions of particular algorithms. Recall that our main goal is to provide a generic approach for translating recursive DC algorithms to hybrid code with little effort.

It is desirable that the task division involves minimal communication costs. With this goal in mind, we design two division and scheduling strategies, which we call basic and advanced. We explain next these strategies and analyze the conditions for scheduling tasks in each computing unit for each strategy.

9.4.1 Basic Hybrid Work Division

Consider the recursion tree of a DC algorithm as depicted in Figure 9.1. Each level of the recursion tree consists of division and combination tasks that are independent of each other and can thus be executed in parallel. The basic work division strategy schedules each level on the CPU or GPU depending on where it is more efficient to execute the entire level. There is an advantage to schedule a level in the GPU when the number of independent subproblems allows the use of enough cores to overcome their comparatively slower speed.

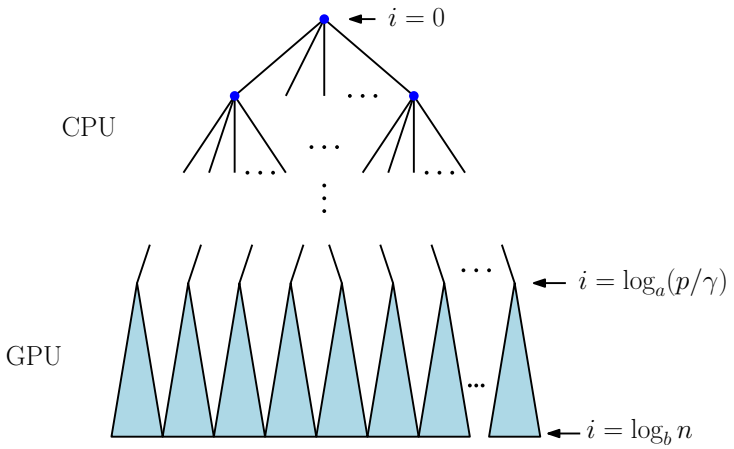


Figure 9.1: Basic hybrid work division. Each level of the recursion tree is executed in the platform in which it runs faster according to the characteristics of the architectures and the divide-and-conquer problem. Thus, levels above $i = \log_a(p/\gamma)$ are executed in the CPU (while the GPU is idle), whereas levels below i are executed in the GPU (while the CPU is idle).

Consider the execution time of each processing unit for a given level i in the recursion tree, where $0 \leq i \leq \log_b(n) - 1$, the 0-th level being the top of the tree. Recall that the DC algorithm solves a subproblems of size b and that the division and combination steps take time $f(n)$.

1. $0 \leq i < \log_a(p)$: $T_{CPU}(n, i) = f(n/b^i)$, $T_{GPU}(n, i) = f(n/b^i)/\gamma$. Since $\gamma < 1$, it is faster to run the level on the CPU.

2. $\log_a(p) \leq i < \log_a(g)$: $T_{CPU}(n, i) = (a^i/p)f(n/b^i)$, $T_{GPU}(n, i) = f(n/b^i)/\gamma$. It becomes faster to run the level on the GPU when $a_i/p \geq 1/\gamma$, i.e., $i \geq \log_a(p/\gamma)$.

3. $\log_a(g) \leq i < \log_b(n)$: $T_{CPU}(n, i) = (a^i/p)f(n/b^i)$, $T_{GPU}(n, i) = (a^i/(\gamma g))f(n/b^i)$. Since we assume $g\gamma \geq p$, it is faster to run the level on the GPU.

4. leaves: $T_{CPU}(n) = n^{\log_b a}/p$, $T_{GPU}(n) = n^{\log_b a}/(\gamma g)$. Again it is faster to run the leaves on the GPU.

Hence there is only one transfer at level $i = \log_a(p/\gamma)$. Note that if $g\gamma < p$ then at every level it is faster to execute on the CPU and there is no transfer to the GPU at any point with this strategy. As only a single data synchronization point occurs when work is transferred from the CPU to the GPU and back, transfer time is minimized and accounts for only a small part of the overall processing time.

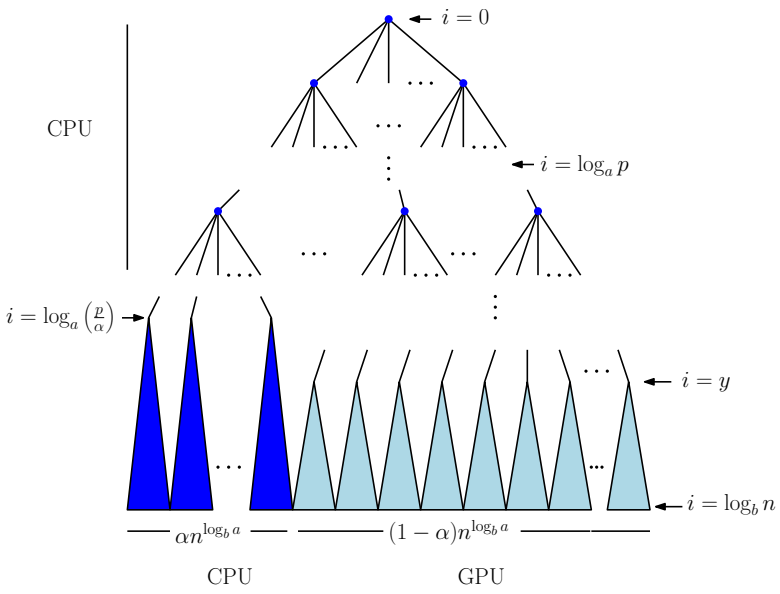


Figure 9.2: Advanced hybrid work division. The GPU will execute so long as the CPU has enough tasks to keep all cores busy (until it reaches level $\log_a(p/\alpha)$ in a bottom up execution of the left part of the tree in the figure), while keeping only one transfer between processing units. The goal is to find the value of α that maximizes the total work executed by the GPU. In the figure, dark blue and light blue subproblems are executed by the CPU and GPU, respectively.

A drawback of this strategy is that at any point only one of the computing units (i.e., CPU or GPU) is active. We now describe a strategy that builds on this basic strategy while minimizing idle periods.

9.4.2 Advanced Hybrid Work Division

The new strategy that we propose for dividing the work among CPU and GPU aims to minimize idle periods and communication between units. If all tasks could be executed in parallel the division would be straightforward: we should divide the work so that both processing units take the same time in their assigned portions. However, in general in DC algorithms there is not enough parallelism at all levels of the trees to maintain a uniform division of work. For example, when the number of available subproblems is less than p, any fraction of them assigned to the GPU will leave at least one CPU core idle. Since CPU cores are faster than GPU cores, it is preferable to execute these serial tasks in the CPU. Therefore, all tasks at levels of the recursion tree where there is at most p subproblems ($0 \leq i \leq \log_a p$) are assigned to the CPU (see Figure 9.2).

For lower level of the trees, we can execute some tasks on the GPU while keeping all CPU cores

busy. Consider the recursion tree depicted in Figure 9.2. When there are more than p subproblems, there is an advantage in running some subproblems in the CPU and some in the GPU. The idea is to run subproblems in the GPU so long as no CPU core is idle. At the same time, we want to keep communication and synchronization costs low. Toward this end, we restrict the number of data transfer between CPU and GPU to two points during the execution. Let y be a parameter denoting the level in the tree (from the top) at which we offload any computation to the GPU, and let α be the parameter denoting the fraction of subproblems that are assigned to the CPU. Thus, at level y, the CPU is assigned αa^y subproblems, while the GPU is assigned $(1-\alpha)a^y$. Note that once a level has been divided in this way, lower levels of the tree will keep the same fraction of subproblems for each processing unit, and hence no further synchronization or data transfer is required until the GPU has solved all subproblems. The choice of y and α determines the amount of work that the GPU will do. Our strategy maximizes the work that the GPU does with two data transfers while avoiding idle CPU cores. In the rest of this section we show how to determine the parameters y and α that maximize GPU work.

Parameter Optimization

For the sake of analysis, consider a bottom up execution of the recursion tree in Figure 9.2. Starting from the bottom level, both CPU and GPU execute with a work ratio of α. Since we want to avoid idle CPU cores, we run both CPU and GPU until the CPU portion reaches p subproblems. We only consider $\alpha \geq p/n$, so that the CPU starts at the bottom level with at least p tasks. Then, the CPU portion reaches p subproblems at level $\log_a(p/\alpha)$. At this point, we stop the GPU execution and let the CPU compute all the unfinished portions of the tree. The time that it takes to the CPU to reach level $\log_a(p/\alpha)$ from the bottom is

$$T_c(n) = \frac{\alpha}{p}\left(n^{\log_b a} + \sum_{i=\log_a(p/\alpha)}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right)\right)$$

During this time, the GPU executes bottom up and reaches level y in the tree. The value of y can be determined by making the GPU and CPU times equal. We have 3 cases, depending on whether the GPU is never saturated or always saturated throughout, or a combination of both. Let $T_g^{max}(n)$ denote the maximum time the GPU can execute while using all its cores (i.e., before it reaches g subproblems). Thus,

$$T_g^{max}(n) = \frac{(1-\alpha)}{\gamma g}\left(n^{\log_b a} + \sum_{i=\log_a(g/(1-\alpha))}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right)\right)$$

(i) $(1-\alpha)n^{\log_b a} < g$. In this case, the GPU is never saturated and thus $T_g(n) = (1/\gamma)(1 + \sum_{i=y}^{\log_b(n)-1} f(n/b^i))$.

(ii) $T_c(n) \leq T_g^{max}(n)$. In this case, the GPU is always saturated and thus $T_g(n) = ((1-\alpha)/\gamma g)(n^{\log_b a} + \sum_{i=y}^{\log_b(n)-1} a^i f(n/b^i))$.

(iii) $T_c(n) > T_g^{max}(n)$. In this case, $T_g(n) = T_g^{max}(n) + (1/\gamma)\sum_{i=y}^{\log_a(g/(1-\alpha))-1} f(n/b^i)$.

The goal is to determine the value of α that maximizes the work done by the GPU from the bottom until level y. We first determine y by solving the equation $T_g(n) = T_c(n)$ for each of the 3 cases above. This yields a piecewise function $y = y(\alpha)$. The work done by the GPU in this period is given by

$$W_g(n) = (1-\alpha)\left(n^{\log_b a} + \sum_{i=y(\alpha)}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right)\right).$$

Maximizing for α yields the optimal work ratio value.

After the GPU reaches level y, it transfers the results back to the CPU, which finishes the computation. Note that it could still be advantageous to continue execution on the GPU for levels above y. However, this would invariably imply either having idle CPU cores or a new work ratio α, which would in turn imply further synchronization and data transfer between processing units.

Example

We illustrate this procedure using the characteristics of a sample architecture and a divide-and-conquer algorithm whose division and combination function takes time $\Theta(n^{\log_b a})$ (and thus $T(n) = \Theta(n^{\log_b a} \log n)$). Mergesort is an example of such algorithm. We assume that the implementation of the combination and division function is the same both in the CPU and GPU, and thus the constants hidden in the complexities are the same and will cancel out when solving for the level y.

The time that the CPU takes to reach p problems from the bottom is

$$T_c(n) = \frac{\alpha n^{\log_b a}}{p}\left(\log_b n - \log_a \frac{p}{\alpha} + 1\right).$$

The maximum time the GPU can be fully saturated is

$$T_g^{max}(n) = \frac{(1-\alpha)n^{\log_b a}}{\gamma g}\left(\log_b n - \log_a \frac{g}{1-\alpha} + 1\right).$$

Algorithm 9.6 Pseudo-code for Mergesort

```
mergesort(array, size)
1: if size > 1 then
2:     mergesort(array, size/2)
3:     mergesort(array + size/2, size/2)
4:     merge(array, array + size/2, size/2)
```

Thus, we have the following function for the GPU time:

$$T_g(n) = \begin{cases} (1/\gamma)(n^{\log_b a}\frac{a}{a-1}a^{-y} - \frac{1}{a-1}), & \text{if } (1-\alpha)n^{\log_b a} < g \\ \frac{(1-\alpha)n^{\log_b a}}{\gamma g}(\log_b n - y + 1), & \text{if } (1-\alpha)n^{\log_b a} \geq g \text{ and } T_g^{max}(n) \geq T_c(n) \\ T_g^{max}(n) + n^{\log_b a}\frac{a}{\gamma(a-1)}\left(a^{-y} - \frac{1-\alpha}{g}\right), & \text{if } (1-\alpha)n^{\log_b a} \geq g \text{ and } T_g^{max}(n) < T_c(n) \end{cases}$$

We now solve $T_c(n) = T_g(n)$ for y for each of the cases above, obtaining a piecewise function $y(\alpha)$ that depends on each case. The work done by the GPU is then

$$W_g(n) = (1-\alpha)n^{\log_b a}(\log_b n - y(\alpha) + 1).$$

By replacing in this equation the parameters of a divide-and-conquer algorithm, of a particular architecture, and the input size, we can maximize the work using numeric methods. For example, using mergesort as the divide-and-conquer algorithm and the parameters of one of our architectures[4] (i.e., $a = b = 2$, $f(n) = \Theta(n)$, $p = 4$, $g = 2^{12}$, $\gamma = 1/160$) and an input size $n = 2^{24}$, we obtain the y function and the fraction of GPU work over total work function depicted in Figure 9.3. In this case the total work is $n^{\log_b a}(\log_b n + 1)$. The work ratio that maximizes the GPU work is $\alpha^* \approx 0.16$, for which the GPU does approximately 52% of the total work. The level reached by the GPU with α^* is approximately 10. Since $\log_2 g = 12$, this means that for the GPU is both saturated and non-saturated during its execution for $\alpha = \alpha^*$. Figure 9.4 depicts the work division for this example.

9.5 Case Study: Mergesort

The ideas of our method are applicable in general to algorithms whose parallel structure can be specified by directed acyclic graphs. In this section we use mergesort as a test case for the gains of our general framework for divide-and-conquer algorithms. We particularly chose mergesort as an example of a task-parallel algorithm that is not readily made for execution on a GPU, but that nevertheless is amenable to the kind of hybrid parallelization that we propose.

[4]The architectures and parameters are described in Section 9.5.4.

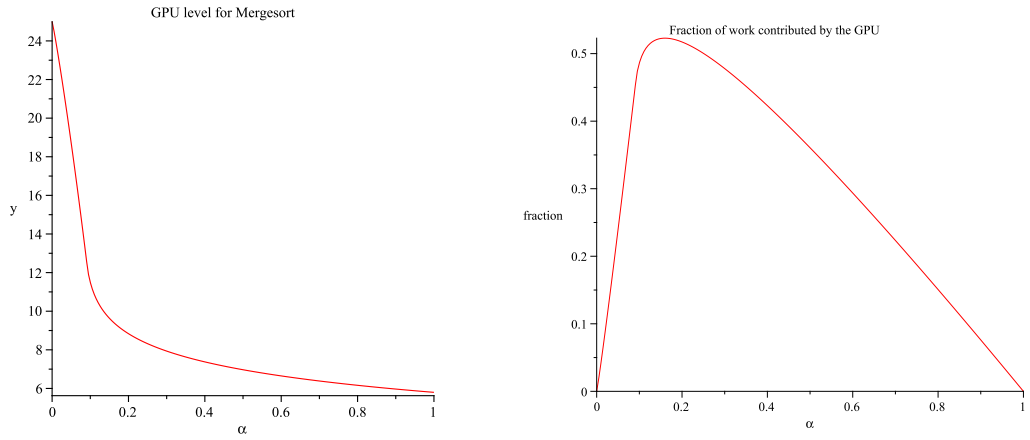


Figure 9.3: For mergesort ($a = b = 2$, $f(n) = \Theta(n)$) and parameters $p = 4$, $g = 2^{12}$, $\gamma^{-1} = 160$ and $n = 2^{24}$, (left) level reached by the GPU while the CPU has at least p tasks at the same level as a function of the work ratio α, and (right) the percentage of work done by the GPU as a function of α.

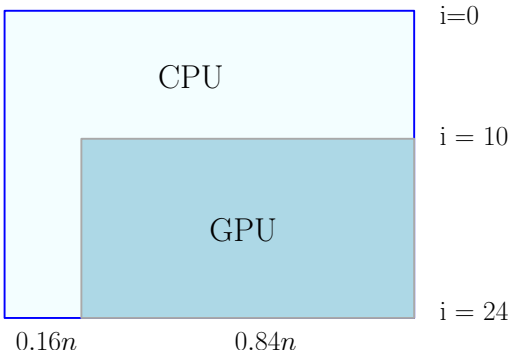


Figure 9.4: Advanced hybrid work division for mergesort. The figure represents the recursion tree shown in Figure 9.2 with the height and width of the rectangles representing the height and work of the computation. For the parameters in the example ($p = 4$, $g = 2^{12}$, $\gamma^{-1} = 160$ and $n = 2^{24}$) the work ratio that maximizes the GPU work is $\alpha \approx 0.16$ and the transfer level is 10.

Algorithm 9.7 Pseudo-code for Breadth-first Mergesort

mergesort_bf(array, totalSize, size, numSublists)

```
1: if size > 1 then
2:     mergesort_bf(array, totalSize, size/2, 2 · numSublists)
3:     for i = 0 to numSublists − 1 do
4:         offset ← i · size
5:         merge(array + offset, array + offset + size/2, size)
```

Consider the classic recursive mergesort implementation as shown in Algorithm 9.6. As described in Section 9.3.1, we first convert the recursive divide-and-conquer implementation to a breadth-first one. Compared to the pseudocode in Algorithm 9.2, a breadth-first execution of mergesort is somewhat simplified as the division into subproblems and condition for basecases are data independent. Thus, a single recursion is performed with parameters indicating the sublists to be sorted. A sublist can be specified by an offset with respect to the beginning of the entire list and the size of the sublist. The offset for the i-th sublist is simply $offset = i \cdot size$. This limits the parameters to the array being sorted, the total array size, and the number of sublists, which are the same for each sublist. Furthermore, determining when only base cases remains becomes trivial: once the number of current sublists equals or exceeds the total length of the array, the maximum number of elements in a sublist is one, and therefore only base cases remain. To finish the conversion, the base-case, division, and combination steps must be performed for each sublist. For mergesort, as no division and base case exist, these parts are removed entirely and only the combine step must be performed, which corresponds to merging pair of sublists. Algorithm 9.7 shows the breadth-first mergesort implementation[5].

9.5.1 Basic Hybrid Implementation

The merge operations for each sublist in line 5 in Algorithm 9.7 are independent of each other and can potentially be executed in parallel. For the basic hybrid work division as described in Section 9.4.1, these operations are executed either on the GPU or on one or more CPU cores. For each recursion level in which merge operations are executed on the GPU, a program running in the host launches a GPU kernel with parameters indicating the size and number of sublists to be merged. Based on its id, each GPU thread identifies the sublist on which it will operate. Similarly, when merge operations are executed on the CPU, depending on the recursion and number of cores available, multiple threads are created to merge sublists in parallel (with each thread merging a pair of sublists sequentially).

[5]In order to keep the description of the approach simpler, we assume that the input size is a power of 2. The same general approach is applicable in general, although some adjustments to the implementation are required.

Algorithm 9.8 Pseudo-code for Advanced Hybrid Mergesort

HybridMergesort(array, totalSize, size, numSublists)

```
1: if size > 1 then
2:    if numSublists > threshold then
3:       cpuLists ← α · numSublists
4:       gpuLists ← numSublists − cpuLists
5:       mergesort_bf_cpu(array, size · cpuLists, size, cpuLists)
6:       mergesort_bf_hybrid(array + size · cpuLists, size · gpuLists, size, gpuLists)
7:    else
8:       HybridMergesort(array, totalSize, size/2, 2 · numSublists)
9:    for i = 0 to numSublists − 1 do
10:      offset ← i · size
11:      merge(array + offset, array + offset + size/2, size)
```

9.5.2 Advanced Hybrid Implementation

The advanced work-division strategy as described in Section 9.4.2 is implemented by, when reaching certain threshold level in the recursion tree, launching two simultaneous CPU threads, one that executes the basic hybrid strategy, and another one that executes a CPU implementation (see Figure 9.2). For these threads, the input is divided according to the division ratio α. Since this ratio remains constant across levels, when the hybrid thread switches to execution on the GPU (level y in Figure 9.2), the number of subproblems executed in each processing unit will respect the chosen ratio. This implementation is shown in Algorithm 9.8. In this algorithm, the methods **mergesort_bf_cpu** and **mergesort_bf_hybrid** correspond, respectively, to implementations of Algorithm 9.7 to be executed exclusively on the CPU, and in a hybrid fashion according to the basic model.

9.5.3 GPU Optimizations

So far, the hybrid implementation of mergesort described above is oblivious to the characteristics of the particular divide and combine functions. In the case of the merge method, certain optimizations to the implementation are possible and have a significant impact on performance. In order to achieve coalesced memory accesses, prior to executing a parallel merge operation on the GPU, we permute the input so that the set of i-th elements in all sublists are in contiguous locations. Thus, various parallel threads operating on different sublists will access contiguous memory segments. To adapt the GPU kernel to use this method, sublists are iterated using the thread id as the initial position, and increasing this value by the total number of sublists. As the CPU cache benefits from reading from sequential blocks, before transferring the array to the CPU, the array is permuted back to the original arrangement. Thus this optimization is transparent to the CPU implementation. We note that by incorporating this optimization, which is specific

Platform	CPU	GPU
HPU1	Intel® Core™ 2 Extreme CPU Q6850	ATI Radeon™ HD 5970
HPU2	AMD A6 3650	ATI Radeon™ HD 6530D

Table 9.1: Specification of hybrid platforms used in experiments.

Platform	p	g	γ^{-1}
HPU1	4	4096	160
HPU2	4	1200	65

Table 9.2: Platforms parameters (p: number of CPU cores, g: number of GPU cores, γ: CPU-GPU scalar performance ratio).

to the application, we have chosen to improve performance at the cost of some minor generality. Similar considerations could be applied to other applications, and one can choose whether to incorporate them or not, depending on their difficulty of implementation and the performance gains they may lead to.

9.5.4 Experimental Results

We implemented the hybrid mergesort algorithm and tested its performance on two OpenCL platforms: an Intel® Core™ 2 Extreme CPU Q6850 (4 cores at 3.00 GHz, 8 Mb cache) with an ATI Radeon™ HD5970[6] GPU card (which we call HPU1), and an AMD Accelerated Processing Unit A6 3650 (4 cores at 2.6 GHz, 4 Mb Cache) with an integrated ATI Radeon™ HD 6530D card (called HPU2) (see Table 9.1). The algorithms were implemented with OpenCL 1.1 ATI-Stream-v2.3 in Ubuntu 10.04.4 64-bit (HPU1), and OpenCL 1.2 AMD-APP in Ubuntu 12.04 64-bit (HPU2).

We estimated the parameters γ (ratio between CPU and GPU scalar performance) and g (number of GPU cores) for each platform as shown in Table 9.2. Recall that g does not actually correspond to the physical number of cores or processing elements of the GPU but rather to an approximation of the number of threads that fully saturates the device when running a suitable procedure. In this case, in order to estimate g, we ran an implementation of an elementwise sum of two arrays in which all threads worked in consecutive array segments. We measured the

[6]The HD5970 is a Dual GPU card, but only one card was used in the experiments, as the parallelism available in the application could only saturate both cards at the lowest levels of the recursion tree, not justifying the overhead of additional data transfers.

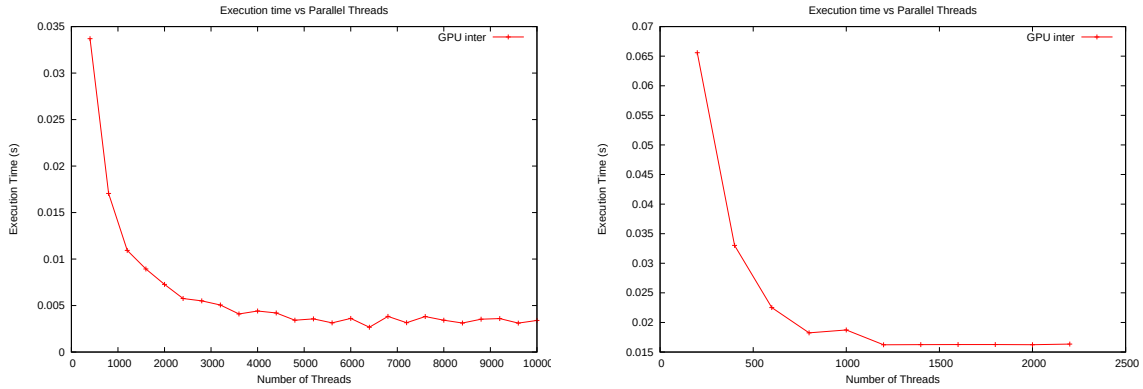


Figure 9.5: Running time as a function of the number of GPU threads used in an elementwise sum of two arrays for platform HPU1 (left) and HPU2 (right). The size of each array is 2^{24}.

running time as the number of threads used increased, and set g to the value after which no improvement in performance was detected[7]. Figure 9.5 shows the running times as a function of the number of threads for each platform. The parameters were set to $g = 4096$ for HPU1 and $g = 1200$ for HPU2. In order to estimate γ, a 1-thread merge operation over two lists was executed on both CPU and GPU. Figure 9.6 shows the running time for different input sizes. As expected, the time ratio remains relatively constant. These parameters were set to $\gamma^{-1} = 160$ for HPU1 and $\gamma^{-1} = 65$ for HPU2.

We measured the performance of the advanced hybrid mergesort implementations for various transfer levels and ratios. Figure 9.7 shows the speedups of the hybrid implementation on HPU1 (using 4-CPU cores) with respect to a 1-core CPU recursive implementation, as a function of the ratio α for various transfer levels for an input of size $n = 2^{24}$ (elements in all input sequences were chosen uniformly at random between 0 and $2n-1$). Recall from the example in Section 9.4.2 that for this input size the estimated optimal ratio and transfer levels were $\alpha \approx 0.16$ and $y = 10$. We observe that the speedups do not differ too much across transfer levels, although speedups increase from level 7 and start decreasing with level 11, in accordance with the estimation. Similarly, the performance is slightly better for transfer ratios that are close to the estimated one.

Figure 9.8 shows the speedups obtained in both platforms with the values of transfer level and ratio that resulted in the best speedups (in red). The green lines in the figures show the speedup

[7]The sum of two arrays used to estimate g shares the characteristics of the merge with optimization for memory coalesced access of our application, and thus it provides a good approximation of the degree of parallelism of the architecture in this case. Another option would have been to use the same merge function, and in general, the same divide or combine function of the application can be used for this purpose. Note that the parameter estimation is done only once.

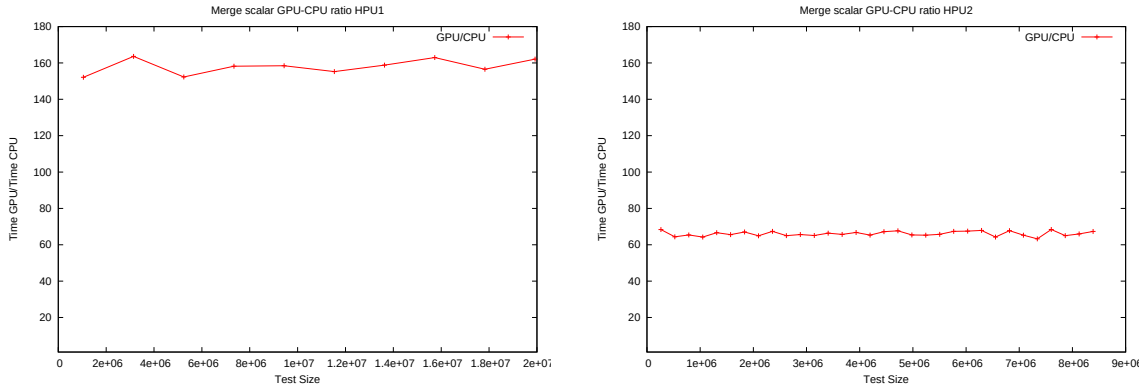


Figure 9.6: Ratio between scalar performance of single GPU and CPU cores when executing a merge operation on HPU1 (left) and HPU2 (right).

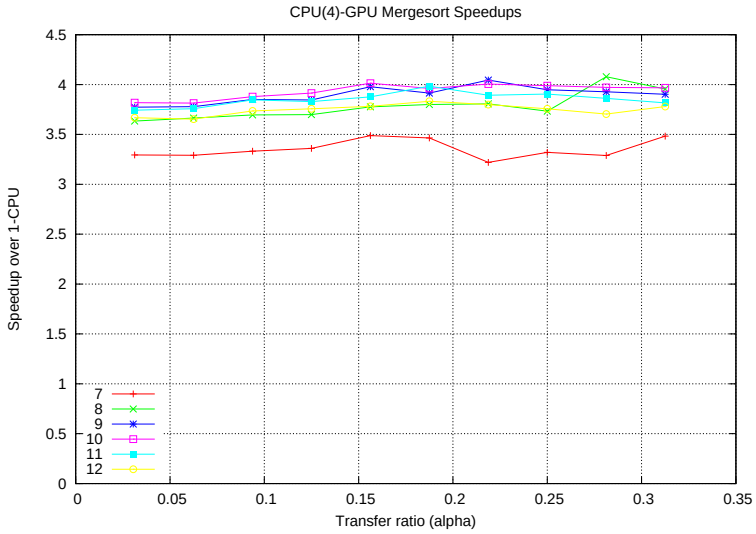


Figure 9.7: Speedup of hybrid mergesort implementation on HPU1 with an instance of size $n = 2^{24}$ as a function of the transfer ratio α. Each curve corresponds to a different transfer level between processing units (parameter y in Figure 9.2).

estimated in the advanced model analysis for the parameters of the platforms. The maximum speedups achieved were 4.54x for HPU1 and 4.35x for HPU2, which are close to the estimated 5.47x and 5.7x by the analysis for the corresponding input size, respectively. Recall that the

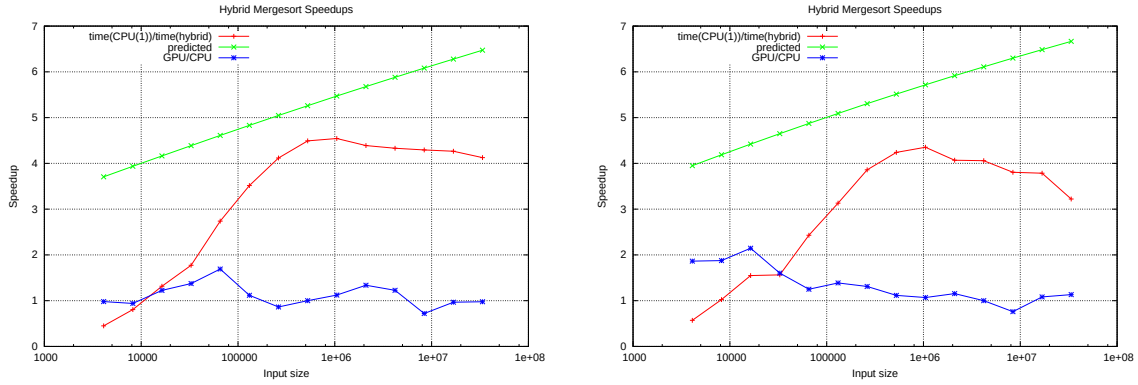


Figure 9.8: Speedup of hybrid mergesort implementation (red) as a function of the input size for HPU1 (left) and HPU2 (right). The green line depicts the estimated speedups in the analytical model. The blue line shows the ratio between the time of execution of GPU and the time while the CPU is fully utilized (see Section 9.4.2).

overall gains in performance are limited by the sequential execution of the merge methods on large input sizes at the top levels of the recursion tree, which also limits the performance of a similar multi-core only implementation to 2.5x-3x speedups on 4 cores (see Section 3.3). In this sense, the performance gains obtained through the hybrid implementation are considerable, as we should take into account that according to the analysis the GPU does about 50% of the total work, which in the best case could lead to a 2x speedup over a 4-core execution. Recall as well from the advanced hybrid model description in Section 9.4.2 that the GPU should execute so long as the CPU has enough tasks to keep cores busy (as shown in the bottom of the triangle in Figure 9.2). The blue line in Figure 9.8 shows the ratio between these parallel GPU and CPU times. Observe that the ratio is in general close to one and that the best speedup points coincide with the instances in which this ratio is closest to one.

We observe as well that as the input size grows, the obtained speedups (red) decrease and do not keep up with estimated ones (green). We believe that as the input size increases, poor cache utilization hurts the performance of the multi-core portion of the execution. Speedups start to decrease around an input size of $n = 2^{20}$. The space used by the algorithm is roughly $2n \cdot \text{sizeof(int)}$, i.e., $2^{23} = 8$ Mb. The sizes of the last level CPU caches in HPU1 and HPU2 are 8 Mb and 4 Mb, respectively. Thus, for larger input sizes multiple cores will compete for cache use.

For the sake of comparison with a fully parallel solution, we show the times and speedups of a mergesort GPU implementation that implements a parallel algorithm for the merge phase. Like the implementation with sequential merge, the parallel GPU implementation executes the

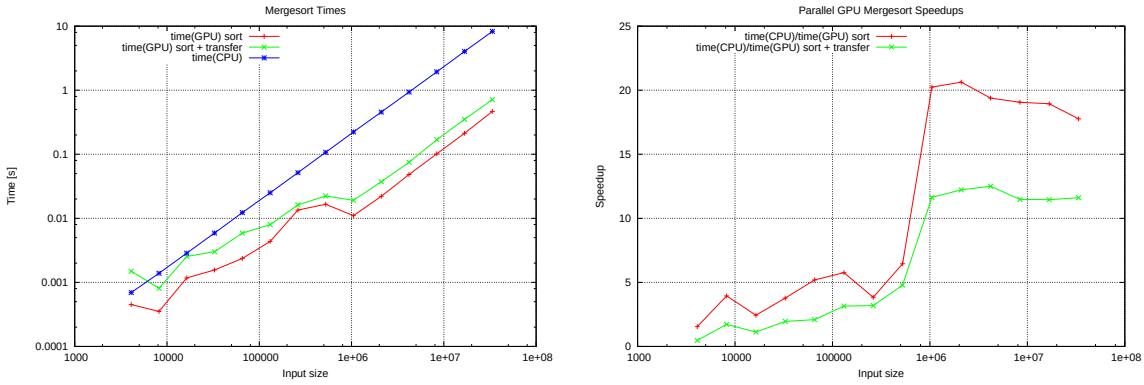


Figure 9.9: Times (left) and speedups (right) of a GPU only implementation of mergesort with parallel merge compared to a sequential CPU recursive implementation as a function of the input size running on HPU1. Red lines correspond to the times and speedups for sorting only on the GPU while the green lines include the time of data transfers.

recursion tree in breadth-first order as well, merging pairs of sublists in each level. Merging two sublist is implemented by performing a binary search for each element in parallel in order to find its position in the merged list. Figure 9.9 shows the times and speedups compared to a recursive divide-and-conquer execution on one CPU core on HPU1. We observe that speedups are only significantly larger than those of our solution for large input sizes, reaching 18x-20x speedups for sorting only and being reduced to about 12x when considering the overhead of data transfers.

Finally, to see how the resulting best parameters compare to the predicted ones by the advanced hybrid model, Figure 9.10 shows the ratio α and transfer level y that resulted in the smallest running times for each input size compared to the ones predicted by the model for HPU1. Note that resulting parameter values are closer to the predicted ones as the input size grows, which coincides with higher speedups. Observe as well that in the case of the optimal transfer level, the obtained values essentially coincide with the predicted ones for larger values of the input size, as the fractional numbers shown in the figure can only take integer values in an actual execution.

9.6 Conclusions

In this chapter we presented the Hybrid Processing Unit, a model for hybrid computation on heterogeneous CPU-GPU architectures. In this model, we describe a generic framework to implement hybrid divide-and-conquer algorithms and provide a work-division strategy that minimizes idle times and communication between processing units.

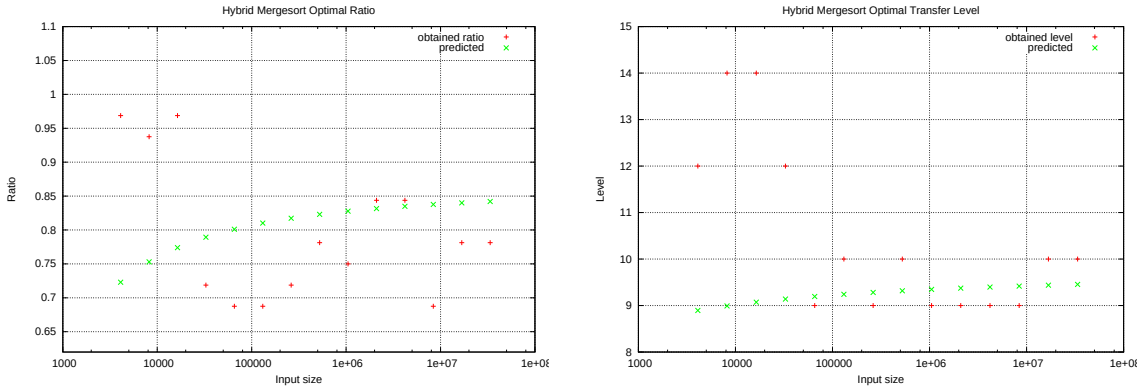


Figure 9.10: Red points show the work ratio α (left) and transfer levels y (right) between CPU and GPU that resulted in the smallest running times for each input size (for HPU1). Green points correspond to the optimal values as predicted by the model.

Experimental results on a mergesort example confirm the accuracy of the model at predicting speedups and parameters that yield the best performance, thus suggesting that a model based on traditional approaches to the design and analysis of parallel computation can be useful in a heterogeneous scenario.

For future work, we plan to refine the model by considering cache, communication, and scheduling costs explicitly, as well as to extend its applicability to other classes of problems that are suitable for obtaining performance gains in heterogeneous architectures.

Chapter 10

Conclusions

The advent of multi-core processors has changed the landscape of computing, bringing parallel computation to the foreground of the field for both researchers and practitioners. Parallelism is no longer exclusive of high performance computing but is a pervasive feature in desktops, laptops, and embedded systems. While multi-cores provide a moderate level of parallelism, highly parallel Graphic Processing Units (GPUs) have converted commodity computers into powerful computational devices. This renewed relevance of parallelism has prompted researchers in theoretical Computer Science to revisit the models and algorithms that were developed for both practical and theoretical parallel machines in the past decades. As a consequence, new and adapted solutions for the reality of multiple resource-sharing processors have started to emerge.

This thesis is an example of theoretical developments for modern parallelism. We have studied several problems related to the modeling of computation in the mentioned architectures. Within multi-cores, we have focused in the implications of the reality of a small number of processors in the aid of the analysis and design of algorithms. In Chapter 3 we develop a model for multi-core computing in which we assume that the number of processors is bounded by a logarithmic function on the input size. We have shown that this assumption simplifies the design of a large class of divide-and-conquer and dynamic programming algorithms, while attaining optimal parallel performance with respect to the corresponding sequential algorithms. The assumption of a small number of processors is key to the optimality results. In Chapter 4 we argue that, in general, parallel systems with a small number of processors (logarithmic or sublinear in the input size) exhibit significant differences compared to systems with a linear number of processors, as it was often assumed in the past. These differences appear in several aspects related to parallel computation such as memory access conflicts, the feasible size of communication networks, and the attainable cache performance in multi-threaded computations, among others. Furthermore, we have defined complexity classes to capture how efficiently problems can be solved in parallel and have shown that the class of problems that can be optimally sped up with a logarithmic

number of processors strictly contains the analogous polynomial processor class. The results in these two chapters are evidence of the advantages that parallel architectures and algorithms can achieve when designed explicitly towards a moderate level of parallelism.

Chapter 5 presents the Ultra-Wide Word model and architecture (UW-RAM). This model explores an alternative form of parallelism in the form of a wide-word ALU that can operate on thousands of bits in parallel. We describe algorithms for this architecture for various problems solvable by dynamic programming as well as for text searching. We also show how the Ultra-Wide Word can simulate a non-standard memory architecture, thus enabling the implementation of efficient data structures for priority queues and dynamic prefix sums. Many of the algorithms described are simple modifications to word-RAM algorithms. In fact, the goal of this model is to enable word-RAM algorithms to achieve speedups compared to those of multi-threaded computations, while avoiding the more difficult aspects of parallel programming, thus retaining the simplicity of sequential algorithm design.

Chapters 7 and 8 study the paging problem in shared cache environments. In Chapter 7 we propose a model for paging in multi-core shared caches and analyze the performance of various strategies. We show that traditional paging algorithms such as LRU and FIFO are not competitive in the multi-core setting, while we argue that any competitive strategy must dynamically adjust a cache partition among cores. We also study the offline paging problem and show that achieving a fair distribution of faults is NP-complete and hard to approximate. We show, however, that this problem admits a polynomial time algorithm when the number of sequences is constant, thus suggesting that the difficulty of multi-core paging stems from the number of sequences rather than their length.

In Chapter 8 we propose a model for paging in which algorithms must account for the amount of cache they use, in addition to the number of faults. This model seeks paging strategies that efficiently use the available cache, and it can be used in settings in which the cache is a shared resource —such as multi-cores or caching in the cloud—, as well as in scenarios in which partial use of the cache can lead to energy savings —such as in caching with Content Addressable Memories (CAMs). We present a family of cache-aware online algorithms that achieve a competitive ratio that adapts to the relative costs of cache and faults, and that are 2-competitive for sequences with high locality of reference. We experimentally show that these algorithms are nearly optimal for real-world memory traces.

Finally, we turn our attention to computation in heterogeneous architectures with multi-cores and GPUs. The vector nature of GPUs makes it suitable for data-parallel algorithms. We argue that many parallel solutions allow for a task-parallel component, which can be handled by scalar CPUs, leading to a hybrid solution. Chapter 9 presents a model for heterogeneous computing in an CPU-GPU architecture. We focus on the design of divide-and-conquer algorithms for this architecture, describing a generic translation of a recursive sequential implementation to a parallel algorithm that is scheduled automatically to execute in both the GPU and CPU. The

main advantage of our approach is the simplicity of the implementation, which is independent of the parameters of the architecture. The scheduler is responsible for an efficient task partition among computing units, which is achieved based on a simple model of the underlying architecture.

Each of the developments described above left interesting open questions. We mention several specific directions of future work in the conclusions of each chapter. In general, it would be interesting to extend the applicability of the models we propose, thus expanding the set of algorithms that can be effortlessly translated into implementations that take advantage of the parallelism provided by multi-cores and GPUs, the classes of problems that can be parallelized with bit-parallelism in the UW-RAM, as well as the classes of problems that can be sped up optimally with a small number of processors. It would be particularly interesting to exactly characterize the class of problems that can be parallelized optimally with a logarithmic number of processors. Furthermore, showing that this class contains most, if not all problems in P, would be extremely interesting, and it would be a breakthrough to do so by devising a generic strategy to parallelize any algorithm with a logarithmic number of processors, even if allowing a sublogarithmic inefficiency factor.

Multi-core computing has brought attention not only to the design of algorithms that can perform as many of their operations as possible in parallel, but it has also emphasized the importance of data locality in computation. Multi-core shared caches provide a low-latency, high-bandwidth communication medium for cores which enables the efficient collaboration of threads in a tightly coupled implementation. At the same time, however, the competition for this shared resource can hurt the performance of otherwise efficient programs. The management of data in multi-cores caches, whether it is from the point of view of the algorithm, the schedule, or cache eviction policies, is of utmost relevance for the performance of applications in these architectures.

For sequential computation the RAM model provides, in most cases, a sufficiently appropriate abstraction of the underlying architecture without considering the memory hierarchy. Hence, traditional algorithm design is focused primarily on the number of operations performed. In contrast, in parallel multi-core computation the cache complexity of algorithms and execution schedules is crucial, and is thus becoming a primary performance measure in theoretical models. Given the prevalence of parallel architectures in computing, we can only expect that both parallelism and locality will become even more relevant considerations in the design of algorithms, and more generally in our notion of computational efficiency.

Given the myriad of parallel architectures designs and paradigms, it seems unlikely that one model of computation will ever become the standard model under which be base our notion of efficiency. Instead, several models will be used when most appropriate. Incorporating the most relevant characteristics of the underlying hardware for the intended scope of each model, while providing a simple framework for the design and analysis of algorithms with accurate performance predictions, remains as one of the most important challenges in theoretical parallel computation.

References

U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory Comput. Syst.*, 35(3):321–347, 2002. 36, 46, 48

D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1 - 2):203 – 218, 2000. ISSN 0304-3975. doi: 10.1016/S0304-3975(98)00116-9. URL http://www.sciencedirect.com/science/article/pii/S0304397598001169. 133

A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, Sept. 1988. ISSN 0001-0782. doi: 10.1145/48529.48535. URL http://doi.acm.org/10.1145/48529.48535. 44

A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *STOC '87: Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing*, pages 305–314, New York, NY, USA, 1987a. ACM. ISBN 0-89791-221-7. doi: http://doi.acm.org/10.1145/28395.28428. 31

A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science*, pages 204–216, Los Angeles, California, October 1987b. 31

A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. In *SPAA '89: Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, New York, NY, USA, 1989. ACM. ISBN 0-89791-323-X. doi: http://doi.acm.org/10.1145S/72935.72937. 30

A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theor. Comput. Sci.*, 71(1):3–28, 1990. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(90)90188-N. 29, 30

E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, cheaper, better a hybridization methodology to develop linear algebra software for, 2010. 193

E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. LU factorization for accelerator-based systems. In *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*, pages 217 –224, dec. 2011a. doi: 10.1109/AICCSA.2011.6126599. 193

E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR factorization on a multicore node enhanced with multiple GPU accelerators. In *25th IEEE International Symposium on Parallel and Distributed Processing*, IPDPS'11, pages 932 –943, may 2011b. doi: 10.1109/IPDPS.2011.90. 193

D. Ajwani, N. Sitchinava, and N. Zeh. Geometric algorithms for private-cache chip multiprocessors. In M. de Berg and U. Meyer, editors, *Proceedings of the 18th Annual European Symposium (ESA 2010)*, volume 6347 of *LNCS*, pages 75–86. Springer, 2010. ISBN 978-3-642-15780-6. 94

D. Ajwani, N. Sitchinava, and N. Zeh. I/O-optimal distribution sweeping on private-cache chip multiprocessors. In *25th IEEE International Symposium on Parallel and Distributed Processing*, pages 1114–1123. IEEE, 2011. ISBN 978-1-61284-372-8. 45, 94

S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput.*, 36(11):1367–1369, Nov. 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.5009478. URL http://dx.doi.org/10.1109/TC.1987.5009478. 50, 73

A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model –one step closer towards a realistic model for parallel computation. In *SPAA '95: Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, New York, NY, USA, 1995. ACM. ISBN 0-89791-717-0. doi: http://doi.acm.org/10.1145/215399.215427. 32

B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *In Proc. Programming Models for Massively Parallel Computers*, pages 116–123. IEEE Computer Society Press, 1993. 57

C. Ambühl. Offline list update is NP-hard. In *Proceedings of the 8th Annual European Symposium (ESA 2000), volume 1879 of LNCS*, pages 42–51. Springer, 2000. 150

AMD. The industry-changing impact of accelerated computing. AMD Whitepaper, Advance Micro Devices, 2008. URL http://sites.amd.com/jp/Documents/AMD_fusion_Whitepaper.pdf. 09/04/2011. 192

E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992. ISBN 0-89871-294-7. 193

A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54 (3):13, 2007. 102

S. Angelopoulos, R. Dorrigiv, and A. López-Ortiz. On the separation and equivalence of paging strategies. In N. Bansal, K. Pruhs, and C. Stein, editors, *SODA*, pages 229–237. SIAM, 2007. ISBN 978-0-898716-24-5. 134

A. Apostolico, M. J. Atallah, L. L. Larmore, and S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988, 1990. ISSN 0097-5397. doi: http://dx.doi.org/10.1137/0219066. 74

L. Arge, M. T. Goodrich, M. J. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 197–206. ACM, 2008. ISBN 978-1-59593-973-9. 44, 45, 94

L. Arge, M. T. Goodrich, and N. Sitchinava. Parallel external memory graph algorithms. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2010. 45, 94

E. M. Arkin and E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Appl. Math.*, 18(1):1–8, 1987. ISSN 0166-218X. doi: 10.1016/0166-218X(87)90037-0. 174, 176

V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economic construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR*, 194:487–488, 1970. (In Russian). English translation in Soviet Math. Dokl., 11,1209-1210, 1975. 64, 96, 120

S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009. ISBN 0521424267, 9780521424264. 9, 25

C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. ISSN 1532-0634. doi: 10.1002/cpe.1631. URL http://dx.doi.org/10.1002/cpe.1631. 193

M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. *Procedia CS*, 9:17–26, 2012. 193

R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10): 74–82, Oct. 1992. ISSN 0001-0782. doi: 10.1145/135239.135243. URL http://doi.acm.org/10.1145/135239.135243. 102, 121

R. A. Baeza-Yates and M. Régnier. Average running time of the Boyer-Moore-Horspool algorithm. *Theoretical Computer Science*, 92(1):19 – 31, 1992. ISSN 0304-3975. doi: 10.1016/0304-3975(92)90133-Z. URL http://www.sciencedirect.com/science/article/pii/030439759290133Z. 124

N. Bansal, N. Buchbinder, and J. S. Naor. Randomized competitive algorithms for generalized caching. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, STOC '08, pages 235–244, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-047-0. doi: 10.1145/1374376.1374412. URL http://doi.acm.org/10.1145/1374376.1374412. 133

G. Barlas, A. Hassan, and Y. A. Jundi. An analytical approach to the design of parallel block cipher encryption/decryption: A CPU/GPU case study. In *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '11, pages 247–251, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4328-4. 192, 193

R. D. Barve, E. F. Grove, and J. S. Vitter. Application-controlled paging for a shared cache. *SIAM J. Comput.*, 29:1290–1303, 2000. ISSN 0097-5397. 135, 136, 137, 140, 154

A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: http://doi.acm.org/10.1145/1629575.1629579. URL http://doi.acm.org/10.1145/1629575.1629579. 192

P. Beame and F. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65:2002, 2002. 110

L. A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. 131, 150

R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957. 112, 113

M. A. Bender and C. A. Phillips. Scheduling DAGs on asynchronous processors. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 35–45, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-667-7. doi: 10.1145/1248377.1248384. URL http://doi.acm.org/10.1145/1248377.1248384. 85

L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. ISBN 0-89871-397-8. 193

G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Computers*, 38(11):1526–1538, 1989. 37

G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, Mar. 1996. ISSN 0001-0782. doi: 10.1145/227234.227246. URL http://doi.acm.org/10.1145/227234.227246. 36

G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA 2004: Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244, New York, NY, USA, 2004. ACM. ISBN 1-58113-840-7. doi: http://doi.acm.org/10.1145/1007912.1007948. 38, 48, 52, 94

G. E. Blelloch, P. B. Gibbons, G. J. Narlikar, and Y. Matias. Space-efficient scheduling of parallelism with synchronization variables. In *SPAA*, pages 12–23, 1997. 37

G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46:281–321, March 1999. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/301970.301974. URL http://doi.acm.org/10.1145/301970.301974. 19, 20, 34, 37, 94

G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandrana, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the 2008 ACM-SIAM Symposium on Discrete Algorithms*, January 2008. 49, 50, 51, 52, 56, 73, 93, 94

G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 189–199. ACM, 2010. ISBN 978-1-4503-0079-7. 48, 49

G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. doi: 10.1145/1989493.1989553. URL http://doi.acm.org/10.1145/1989493.1989553. 57, 58, 59

R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 362–371, New York, NY, USA, 1993. ACM. ISBN 0-89791-591-7. doi: 10.1145/167088.167196. URL http://doi.acm.org/10.1145/167088.167196. 20, 35

R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999. ISSN 0004-5411. doi: 10.1145/324133.324234. URL http://doi.acm.org/10.1145/324133.324234. 20, 35, 36

R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 132–141, Washington, DC, USA, 1996a. IEEE Computer Society. ISBN 0-8186-7255-2. URL http://dl.acm.org/citation.cfm?id=645606.661333. 36, 46, 47, 49

R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *SPAA 1996: Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, 1996b. 35, 36

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37 (1):55–69, 1996c. 36

A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-56392-5. 128, 130, 131, 132, 133, 156, 157, 171, 184

A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *J. Comput. Syst. Sci.*, 50:244–258, April 1995. ISSN 0022-0000. doi: 10.1006/jcss.1995.1021. URL http://portal.acm.org/citation.cfm?id=207371.207379. 135

P. Bose, E. Y. Chen, M. He, A. Maheshwari, and P. Morin. Succinct geometric indexes supporting point location queries. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 635–644, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics. URL http://dl.acm.org/citation.cfm?id=1496770.1496840. 88

K. I. Bouzina and H. Emmons. Interval scheduling on identical machines. *Journal of Global Optimization*, 9:379–393, 1996. ISSN 0925-5001. 174, 176, 185

J. Boyar, M. R. Ehmsen, and K. S. Larsen. Theoretical evidence for the superiority of LRU-2 over LRU for the paging problem. In *Proceedings of the 4th International Conference on Approximation and Online Algorithms*, WAOA'06, pages 95–107, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-69513-3, 978-3-540-69513-4. doi: 10.1007/11970125_8. URL http://dx.doi.org/10.1007/11970125_8. 131

P. G. Bradford. Parallel dynamic programming. Technical Report #352, Indiana University, Department of Computer Science, 1994. URL citeseer.ist.psu.edu/bradford94parallel.html. 74

M. Brehob, S. Wagner, E. Torng, and R. J. Enbody. Optimal replacement is NP-hard for non-standard caches. *IEEE Trans. Computers*, 53(1):73–76, 2004. 174

R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321812.321815. 14, 65, 85, 95

A. Brodnik. *Searching in Constant Time and Minimum Space*. PhD thesis, University of Waterloo, 1995. Also available as Technical Report CS-95-41. 106, 107, 108

A. Brodnik, S. Carlsson, M. L. Fredman, J. Karlsson, and J. I. Munro. Worst case constant time priority queue. *Journal of Systems and Software*, 78(3):249 – 256, 2005. ISSN 0164-1212. doi: 10.1016/j.jss.2004.09.002. URL http://www.sciencedirect.com/science/article/pii/S0164121204002018. 102, 108, 109, 110

A. Brodnik, J. Karlsson, J. I. Munro, and A. Nilsson. An O(1) solution to the prefix sum problem on a specialized memory architecture. In G. Navarro, L. E. Bertossi, and Y. Kohayakawa, editors, *IFIP TCS*, volume 209 of *IFIP*, pages 103–114. Springer, 2006. ISBN 0-387-34633-3. 102, 111

B. C. Budge, J. C. Anderson, C. Garth, and K. I. Joy. A hybrid CPU-GPU implementation for interactive ray-tracing of dynamic scenes. Technical Report CSE-2008-9, University of California, Davis Computer Science, 2008. 193

F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM. ISBN 0-89791-060-5. doi: http://doi.acm.org/10.1145/800223.806778. 35, 69, 92

P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, USITS'97, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267279.1267297. 133

P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *Proceedings of the USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 11–11, Berkeley, CA, USA, 1994. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1267257.1267268. 135, 136, 137, 140, 154

S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 74–83, New York, NY, USA, 1995. ACM. ISBN 0-89791-717-0. doi: 10.1145/215399.215423. URL http://doi.acm.org/10.1145/215399.215423. 192

T. M. Chan. Point location in o(log n) time, Voronoi diagrams in o(n log n) time, and other transdichotomous results in computational geometry. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, pages 333–344. IEEE Computer Society, 2006. ISBN 0-7695-2720-5. 64, 102

T. M. Chan and M. Patrascu. Transdichotomous results in computational geometry, i: Point location in sublogarithmic time. *SIAM J. Comput.*, 39(2):703–729, 2009. 64

B. Chazelle. Triangulating a simple polygon in linear time. *Disc. and Comp. Geometry*, 6: 485–524, 1991. 74

R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete algorithm*, SODA '06, pages 591–600, New York, NY, USA, 2006. ACM. ISBN 0-89871-605-5. doi: 10.1145/1109557.1109622. URL http://doi.acm.org/10.1145/1109557.1109622. 53, 58

R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 207–216, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: 10.1145/1378533.1378574. URL http://doi.acm.org/10.1145/1378533.1378574. 52, 53, 54, 56, 94

R. A. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. *Theory Comput. Syst.*, 47 (4):878–919, 2010. 47

R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE, 2010. 56, 57, 58

M. Chrobak. SIGACT news online algorithms column 17. *SIGACT News*, 41(4):114–121, 2010. ISSN 0163-5700. doi: 10.1145/1907450.1907547. 173

M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New results on server problems. *SIAM J. Discret. Math.*, 4(2):172–181, 1991. ISSN 0895-4801. doi: 10.1137/0404017. 133, 169

M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu. Caching is hard - even in the fault model. *Algorithmica*, 63(4):781–794, 2012. 133, 174

M. Cieliebak, S. Eidenbenz, and G. Woeginger. Double digest revisited: Complexity and approximability in the presence of noisy data. In T. Warnow and B. Zhu, editors, *Computing and Combinatorics*, volume 2697 of *Lecture Notes in Computer Science*, pages 519–527. Springer Berlin / Heidelberg, 2003. URL http://dx.doi.org/10.1007/3-540-45071-8_52. 10.1007/3-540-45071-8_52. 155

R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988. ISSN 0097-5397. doi: http://dx.doi.org/10.1137/0217049. 66

R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. In S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, and P. G. Spirakis, editors, *ICALP (1)*, volume 6198 of *Lecture Notes in Computer Science*, pages 226–237. Springer, 2010. ISBN 978-3-642-14164-5. 47

R. Cole and V. Ramachandran. Analysis of randomized work stealing with false sharing. *CoRR*, abs/1103.4142, 2011. 47

R. Cole and V. Ramachandran. Efficient resource oblivious algorithms for multicores with false sharing. In *26th IEEE International Parallel and Distributed Processing Symposium*, pages 201–214. IEEE Computer Society, 2012. ISBN 978-1-4673-0975-2. 47, 48

R. Cole and O. Zajicek. The APRAM: incorporating asynchrony into the PRAM model. In *SPAA '89: Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, New York, NY, USA, 1989. ACM. ISBN 0-89791-323-X. doi: http://doi.acm.org/10.1145/72935.72954. 30

S. A. Cook. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathdmatique*, 27:99–124, 1981. 22

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001. 18, 53, 66, 70, 75, 79, 92, 112, 115

J. Csirik, C. Imreh, J. Noga, S. S. Seiden, and G. J. Woeginger. Buying a constant competitive ratio for paging. In F. Meyer auf der Heide, editor, *ESA*, volume 2161 of *LNCS*, pages 98–108. Springer, 2001. ISBN 3-540-42493-8. 173

D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *PPOPP '93: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-589-5. doi: http://doi.acm.org/10.1145/155332.155333. 32

G. B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):pp. 266–277, 1957. ISSN 0030364X. URL http://www.jstor.org/stable/167356. 113

E. W. Dijkstra. Over seinpalen (in Dutch). Circulated privately, 1974. URL http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF. 33

R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51: 161–166, 1950. 76

R. Dorrigiv. *Alternative Measures for the Analysis of Online Algorithms*. PhD thesis, University of Waterloo, 2010. 134

R. Dorrigiv and A. López-Ortiz. A survey of performance measures for on-line algorithms. *SIGACT News*, 36(3):67–81, 2005. 134

R. Dorrigiv and A. López-Ortiz. On developing new models, with paging as a case study. *SIGACT News*, 40(4):98–123, 2009. 134

R. Dorrigiv, A. López-Ortiz, and A. Salinger. Optimal speedup on a low-degree multi-core parallel architecture (LoPRAM). In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 185–187, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: 10.1145/1378533.1378568. URL http://doi.acm.org/10.1145/1378533.1378568. 66

R. Dorrigiv, A. López-Ortiz, and J. I. Munro. On the relative dominance of paging algorithms. *Theor. Comput. Sci.*, 410(38-40):3694–3701, 2009. 134, 178

U. Drepper. What every programmer should know about memory, 2007. 141

R. A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2:137–151, 1987. 74

P. W. Dymond and M. Tompa. Speedups of deterministic machines by synchronous parallel machines. In *STOC '83: Proceedings of the Fifteenth annual ACM Symposium on Theory of Computing*, pages 336–343, New York, NY, USA, 1983. ACM. ISBN 0-89791-099-0. doi: http://doi.acm.org/10.1145/800061.808763. 96

D. Eckstein. Simultaneous memory access. Technical report, Computer Science Dept., Iowa State Univ., 1979. 12

P. Ezzatti, E. Quintana-Ortí and, and A. Remon. High performance matrix inversion on a multi-core platform with several GPUs. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 87 –93, feb. 2011. doi: 10.1109/PDP.2011.66. 192, 198

W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, January 1968. ISBN 0471257087. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20{&}path=ASIN/0471257087. 89

E. Feuerstein and A. Strejilevich de Loma. On-line multi-threaded paging. *Algorithmica*, 32(1):36–60, 2002. 136

A. Fiat and A. R. Karlin. Randomized and multipointer paging with locality of reference. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, STOC '95, pages 626–634, New York, NY, USA, 1995. ACM. ISBN 0-89791-718-9. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/225058.225280. URL http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/225058.225280. 135, 137

F. E. Fich, P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17(3):606–627, 1988. 12, 79

J. A. Fisher. Very long instruction word architectures and the ELI-512. *SIGARCH Comput. Archit. News*, 11:140–150, June 1983. ISSN 0163-5964. URL http://portal.acm.org/citation.cfm?id=1067651.801649. 101

B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=1012889.1012894. 173

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9): 948–960, Sept. 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071. URL http://dx.doi.org/10.1109/TC.1972.5009071. 10

M. J. Flynn and K. W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1):67–70, Mar. 1996. ISSN 0360-0300. doi: 10.1145/234313.234345. URL http://doi.acm.org/10.1145/234313.234345. 11

S. Fortune and J. Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, New York, NY, USA, 1978. ACM. doi: http://doi.acm.org/10.1145/800133.804339. 12, 65

M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 345–354, New York, NY, USA, 1989. ACM. ISBN 0-89791-307-8. doi: 10.1145/73007.73040. URL http://doi.acm.org/10.1145/73007.73040. 103, 107

M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. 64

M. L. Fredman. The complexity of maintaining an array and computing its partial sums. *J. ACM*, 29(1):250–260, Jan. 1982. ISSN 0004-5411. doi: 10.1145/322290.322305. URL http://doi.acm.org/10.1145/322290.322305. 111

M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA '06: Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 271–280, New York, NY, USA, 2006. ACM. ISBN 1-59593-452-9. doi: http://doi.acm.org/10.1145/1148109.1148157. 46, 47, 49, 52

M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0409-4. URL http://dl.acm.org/citation.cfm?id=795665.796479. 46, 47, 50, 57, 141

A. Fujiwara, M. Inoue, and T. Masuzawa. Parallelizability of some P-complete problems. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 116–122, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67442-X. URL http://dl.acm.org/citation.cfm?id=645612.663168. 96

Z. Galil and K. Park. Parallel dynamic programming. Technical Report CUCS-040-91, Columbia University, Computer Science Dept., 1991. URL citeseer.ist.psu.edu/galil92parallel.html. 74

M. R. Garey and D. S. Johnson. "Strong" NP-completeness results: Motivation, examples, and implications. *J. ACM*, 25(3):499–508, July 1978. ISSN 0004-5411. doi: 10.1145/322077.322090. URL http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/322077.322090. 151

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0-7167-1044-7. 151, 155

M. Garland. NVIDIA GPU. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1339–1345. Springer US, 2011. ISBN 978-0-387-09766-4. URL http://dx.doi.org/10.1007/978-0-387-09766-4_276. 10.1007/978-0-387-09766-4_276. 59, 60, 61

A. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, New York, NY, USA, 1988. ISBN 0-521-34585-5. 67

P. B. Gibbons. A more practical PRAM model. In *SPAA '89: Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, New York, NY, USA, 1989. ACM Press. ISBN 0-89791-323-X. doi: http://doi.acm.org/10.1145/72935.72953. 12, 30

P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: accounting for contention in parallel algorithms. In *SODA '94: Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 638–648, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics. ISBN 0-89871-329-3. 29

L. M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT News*, 9(2):25–29, 1977. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/1008354.1008356. 25

A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.*, 5(2):164–189, Apr. 1983. ISSN 0164-0925. doi: 10.1145/69624.357206. URL http://doi.acm.org/10.1145/69624.357206. 37

R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, Inc., New York, NY, USA, 1995. ISBN 0-19-508591-4. 21, 22, 26, 85, 87

R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003. 64

D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997. ISBN 0-521-58519-8. 120

J. L. Gustafson. Moore's law. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1177–1184. Springer, 2011. ISBN 978-0-387-09765-7. 41

T. Hagerup. Sorting and searching on the word RAM. In M. Morvan, C. Meinel, and D. Krob, editors, *STACS 98*, volume 1373 of *LNCS*, pages 366–398. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-64230-5. URL http://dx.doi.org/10.1007/BFb0028575. 10.1007/BFb0028575. 62, 63, 64, 102, 103, 107, 116

T. Hagerup. Online and offline access to short lists. In *Proceedings of the 32nd International Symposium on Mathematical Foundations of Computer Science*, volume 4708 of *Lecture Notes in Computer Science*, pages 691–702. Springer, 2007. ISBN 978-3-540-74455-9. 150

H. Hampapuram and M. L. Fredman. Optimal biweighted binary trees and the complexity of maintaining partial sums. *SIAM J. Comput.*, 28(1):1–9, 1998. 111

Y. Han. Deterministic sorting in O(nlog logn) time and linear space. *J. Algorithms*, 50:96–105, January 2004. ISSN 0196-6774. doi: 10.1016/j.jalgor.2003.09.001. URL http://portal.acm.org/citation.cfm?id=975978.975984. 64, 102

Y. Han and M. Thorup. Integer sorting in O(n sqrt (log log n)) expected time and linear space. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 135–144, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1822-2. URL http://dl.acm.org/citation.cfm?id=645413.652131. 64

J. Hartmanis and J. Simon. On the power of multiplication in random access machines. In *Switching and Automata Theory, 1974., IEEE Conference Record of 15th Annual Symposium on*, pages 13 –23, oct. 1974. doi: 10.1109/SWAT.1974.20. 10

A. Hassidim. Cache replacement policies for multicore processors. In A. C.-C. Yao, editor, *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 5-7, 2010. Proceedings*, pages 501–509. Tsinghua University Press, 2010. ISBN 978-7-302-21752-7. doi: http://conference.itcs.tsinghua.edu.cn/ICS2010/content/papers/39.html. 5, 136, 139, 141, 142, 148, 151, 154, 162

J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)*. Morgan Kaufmann, 2007. ISBN 978-0-12-370490-0. 1, 26, 42, 43

J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728. 2, 42, 60, 61

J. E. Hopcroft, W. J. Paul, and L. G. Valiant. On time versus space and related problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, FOCS, pages 57–64. IEEE, 1975. 96

R. N. Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6): 501–506, 1980. ISSN 1097-024X. doi: 10.1002/spe.4380100608. URL http://dx.doi.org/10.1002/spe.4380100608. 103, 121, 124

J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108 –109, feb. 2010. doi: 10.1109/ISSCC.2010.5434077. 43

T. Hu and M. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2):362–373, 1982. doi: 10.1137/0211028. URL http://epubs.siam.org/doi/abs/10.1137/0211028. 79

T. Hu and M. Shing. Computation of matrix chain products. Part II. *SIAM J. Comput.*, 13(2): 228–251, May 1984. ISSN 0097-5397. doi: 10.1137/0213017. URL http://dx.doi.org/10.1137/0213017. 79

J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. Cula: hybrid GPU accelerated linear algebra routines. In *Proc. of SPIE Defense and Security Symposium (DSS)*, April 2010. doi: 10.1117/12.850538. 192, 193

O. H. Ibarra and N. Q. Trân. On the parallel complexity of solving recurrence equations. In *ISAAC '94: Proceedings of the 5th International Symposium on Algorithms and Computation*, pages 469–477, London, UK, 1994. Springer-Verlag. ISBN 3-540-58325-4. 75, 76

IntelTurboBoost. Intel(R) Turbo Boost Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html. 41

IntelXeonPhi. Intel delivers new architecture for discovery with Intel(R) Xeon Phi(TM) co-processors. http://newsroom.intel.com/community/intel_newsroom/blog/2012/11/12/intel-delivers-new-architecture-for-discovery-with-intel-xeon-phi-coprocessors?cid=rss-258152-c1-278335, 12 2012. Retrieved 2012-11-27. 43

S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 701–710, New York, NY, USA, 1997. ACM. ISBN 0-89791-888-6. doi: 10.1145/258533.258666. URL http://doi.acm.org/10.1145/258533.258666. 133, 169

G. Jacobson. Space-efficient static trees and graphs. *Foundations of Computer Science, IEEE Annual Symposium on*, pages 549–554, 1989. 64

J. JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992. ISBN 0-201-54856-9. 11, 12, 13, 14, 15, 16, 18, 19, 25, 38, 39, 40, 41, 67, 114

M.-Y. Kao and P. N. Klein. Towards overcoming the transitive-closure bottleneck: efficient parallel algorithms for planar digraphs. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 181–192, New York, NY, USA, 1990. ACM. ISBN 0-89791-361-2. doi: 10.1145/100216.100237. URL http://doi.acm.org/10.1145/100216.100237. 66

S. F. Kaplan. Trace reduction for virtual memory simulation. http://www.cs.amherst.edu/~sfkaplan/research/trace-reduction/index.html. 185

S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Flexible reference trace reduction for VM simulations. *ACM Trans. Model. Comput. Simul.*, 13(1):1–38, 2003. ISSN 1049-3301. doi: 10.1145/778553.778554. 185

A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:77–119, 1988. 148

R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 869–942. MIT Press, Cambridge, MA, USA, 1990. 12, 20, 21, 22, 23, 25, 26, 66

A. K. Katti and V. Ramachandran. Competitive cache replacement strategies for shared cache environments. *Parallel and Distributed Processing Symposium, International*, 0:215–226, 2012. ISSN 1530-2075. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2012.29. 135, 154

Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2.19*, 14 November 2012. URL http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf. 60, 61, 62

P. M. Kogge. Parallel solution of recurrence problems. *IBM J. Res. Develop*, 18:138–148, 1974. 76

C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theor. Comput. Sci.*, 71(1):95–132, Mar. 1990. ISSN 0304-3975. doi: 10.1016/0304-3975(90)90192-K. URL http://dx.doi.org/10.1016/0304-3975(90)90192-K. 23, 87, 95, 96, 100

L. Kucera. Parallel computation and conflicts in memory access. *Inf. Process. Lett.*, 14:93–96, 1982. 12

J. Kurzak, R. Nath, P. Du, and J. Dongarra. An implementation of the tile QR factorization for a GPU and multiple CPUs. In *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*, PARA'10, pages 248–257, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28144-0. doi: 10.1007/978-3-642-28145-7_25. URL http://dx.doi.org/10.1007/978-3-642-28145-7_25. 193

F. T. Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. ISBN 1-55860-117-1. 17, 18

C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988. URL citeseer.ist.psu.edu/leiserson88communicationefficient.html. 31

LEMON. Lemon graph library. http://lemon.cs.elte.hu/trac/lemon. 185

T. Lengauer and K. W. Wagner. The binary network flow problem is logspace complete for P. *Theor. Comput. Sci.*, 75(3):357–363, Oct. 1990. ISSN 0304-3975. doi: 10.1016/0304-3975(90)90101-M. URL http://dx.doi.org/10.1016/0304-3975(90)90101-M. 26

A. López-Ortiz and A. Salinger. Brief announcement: paging for multicore processors. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 137–138. ACM, 2011. ISBN 978-1-4503-0743-7. 140

A. López-Ortiz and A. Salinger. Paging for multi-core shared caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 113–127, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1115-1. doi: 10.1145/2090236.2090246. URL http://doi.acm.org/10.1145/2090236.2090246. 140

A. López-Ortiz and A. Salinger. Minimizing cache usage in paging. In T. Erlebach and G. Persiano, editors, *10th International Workshop, WAOA 2012, Ljubljana, Slovenia, September 13-14, 2012, Revised Selected Papers*, volume 7846 of *Lecture Notes in Computer Science*. Springer, 2012. 170

A. López-Ortiz and A. Salinger. On the sublinear processor gap for parallel architectures. In T.-H. Chan, L. Lau, and L. Trevisan, editors, *Theory and Applications of Models of Computation*, volume 7876 of *Lecture Notes in Computer Science*, pages 193–204. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38235-2. doi: 10.1007/978-3-642-38236-9_18. URL http://dx.doi.org/10.1007/978-3-642-38236-9_18. 85

A. López-Ortiz, A. Salinger, and R. Suderman. Toward a generic hybrid CPU-GPU parallelization of divide-and-conquer algorithms. In *Proceedings of the 15th Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*, 2013. To appear. 192

B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, volume 2, pages 61–70, 1995. URL citeseer.ist.psu.edu/maggs95models.html. 29, 32

Y. Mansour, N. Nisan, and U. Vishkin. Trade-offs between communication throughput and parallel time. In *STOC '94: Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, pages 372–381, New York, NY, USA, 1994. ACM. ISBN 0-89791-663-8. doi: http://doi.acm.org/10.1145/195058.195199. 31

W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980. 79, 116, 120

M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin. Performance evaluation of GPUs using the RapidMind development platform. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. 192

L. McGeoch and D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991. ISSN 0178-4617. URL http://dx.doi.org/10.1007/BF01759073. 10.1007/BF01759073. 133, 136

K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984. 29

P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, Information Technology Laboratory, July 2009. URL http://www.csrc.nist.gov/groups/SNS/cloud-computing/. 172, 173

L. Mirsky. A dual of Dilworth's decomposition theorem. *The American Mathematical Monthly*, 78(8):876–877, 1971. 76

H. Miyatake, M. Tanaka, and Y. Mori. A design for high-speed low-power CMOS fully parallel content-addressable memory macros. *IEEE Journal of Solid-State Circuits*, 36(6):956 –968, 2001. ISSN 0018-9200. doi: 10.1109/4.924858. 172

G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. 1

G. Moruz and A. Negoescu. Outperforming LRU via competitive analysis on parametrized inputs for paging. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1669–1680. SIAM, 2012. URL http://dl.acm.org/citation.cfm?id=2095116.2095248. 184

J. I. Munro. Tables. In V. Chandru and V. Vinay, editors, *FSTTCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996. ISBN 3-540-62034-6. 64, 88

J. I. Munro. On the competitiveness of linear search. In *Proceedings of the 8th Annual European Symposium on Algorithms*, ESA '00, pages 338–345, London, UK, 2000. Springer-Verlag. ISBN 3-540-41004-X. URL http://portal.acm.org/citation.cfm?id=647910.740474. 150

J. I. Munro and E. L. Robertson. Parallel algorithms and serial data structures. In *Proceedings of the 17th Annual Allerton Conference con Communication, Control and Computing*, pages 21–26, 1979. ISBN 0732-6181. 39

G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages. 122, 124

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500. 59, 192

E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: http://doi.acm.org/10.1145/1629575.1629597. URL http://doi.acm.org/10.1145/1629575.1629597. 192

NVIDIA. Dynamic parallelism in CUDA, 2012. URL https://developer.nvidia.com/sites/default/files/akamai/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf. Retrieved on 01/19/2013. 195

K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, 2006. 170, 172

K. Panagiotou and A. Souza. On adequate performance measures for paging. In *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing*, STOC '06, pages 487–496, New York, NY, USA, 2006. ACM. ISBN 1-59593-134-1. doi: 10.1145/1132516.1132587. URL http://doi.acm.org/10.1145/1132516.1132587. 179, 184

C. M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0201530821. 22

M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU-GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13-16):1490–1508, Mar. 2011. ISSN 00457825. doi: 10.1016/j.cma.2011.01.013. URL http://dx.doi.org/10.1016/j.cma.2011.01.013. 193

R. H. Parrot. *Parallel Programming*. Addison-Wesley, 1987. 27, 28

E. Peserico. Paging with dynamic memory capacity. *CoRR*, abs/1304.6007, 2013. URL http://arxiv.org/abs/1304.6007. 143, 145

U. Pferschy. Dynamic programming revisited: Improving knapsack algorithms. *Computing*, 63 (4):419–430, 1999. 112

D. Pisinger. Dynamic programming on the word RAM. *Algorithmica*, 35:128–145, 2003. ISSN 0178-4617. URL http://dx.doi.org/10.1007/s00453-002-0989-y. 10.1007/s00453-002-0989-y. 112, 113, 114

V. R. Pratt and L. J. Stockmeyer. A characterization of the power of vector machines. *J. Comput. Syst. Sci.*, 12(2):198–221, Apr. 1976. ISSN 0022-0000. doi: 10.1016/S0022-0000(76)80037-2. URL http://dx.doi.org/10.1016/S0022-0000(76)80037-2. 10, 24

G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 121–130, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504196. URL http://doi.acm.org/10.1145/1504176.1504196. 194

M. Raab and A. Steger. "balls into bins" - a simple and tight analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '98, pages 159–170, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65142-X. URL http://dl.acm.org/citation.cfm?id=646975.711521. 92

R. Ramanathan. Intel multi-core processors: Making the move to quad-core and beyond. Intel Whitepaper, Intel Corporation, 2006. URL http://www.intel.com/technology/architecture/downloads/quad-core-06.pdf?iid=tech_mc+body_qcdoc. 02/11/2008. 41

N. Reingold and J. Westbrook. Off-line algorithms for the list update problem. *Inf. Process. Lett.*, 60(2):75–80, 1996. 150

R. Riesen and A. B. Maccabe. MIMD (Multiple Instruction, Multiple Data) machines. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1140–1149. Springer, 2011. ISBN 978-0-387-09765-7. 42, 43

A. L. Rosenberg and R. C. Chiang. Toward understanding heterogeneity in computing. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA*, pages 1–10. IEEE, 2010. 194

R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359327.359336. 101

W. L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22(3):365–383, 1981. 24

S. S. Seiden. Randomized online multi-threaded paging. *Nord. J. Comput.*, 6(2):148–161, 1999. 136

L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro*, 29(1):10–21, 2009. 192

J. Simon. On feasible numbers (preliminary version). In *STOC*, pages 195–207. ACM, 1977. 24

M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. ISBN 053494728X. 8, 9

D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985. 128, 173

L. Snyder. Type architectures, shared memory, and the corollary of modest potential. In J. F. Traub, B. J. Grosz, B. W. Lampson, and N. J. Nilsson, editors, *Annual review of computer science vol. 1, 1986*, pages 289–317. Annual Reviews Inc., Palo Alto, CA, USA, 1986. ISBN 0-8243-3201-6. URL http://dl.acm.org/citation.cfm?id=17814.17826. 32

L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984. 22, 23

J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010. ISSN 1521-9615. 59, 192

V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. ISSN 0029–599X. 47, 50, 74

A. Strejilevich de Loma. New results on fair multi-threaded paging. *Electronic Journal of SADIO*, 1(1):21–36, 1998. URL http://publicaciones.dc.uba.ar/Publications/1998/Str98. 136

L. Surhone, M. Tennoe, and S. Henssonow. *Intel Array Building Blocks*. VDM Verlag Dr. Mueller AG & Co. Kg, 2010. ISBN 9786133312159. URL http://books.google.com/books?id=OVxFYgEACAAJ. 192

G. Teodoro, R. Sachetto, O. Sertel, M. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, 31 2009-sept. 4 2009. doi: 10.1109/CLUSTR.2009.5289193. 194

S. Tomov, J. Dongarra, P. Du, and R. Nath. Magma version 0.2 user guide. MAGMA, 2009. URL http://http://icl.cs.utk.edu/magma/. 09/06/2011. 192, 193

S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.*, 36:232–240, June 2010a. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2009.12.005. URL http://dx.doi.org/10.1016/j.parco.2009.12.005. 193, 198

S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Comput.*, 36(12):645–654, Dec. 2010b. ISSN 0167-8191. doi: 10.1016/j.parco.2010.06.001. URL http://dx.doi.org/10.1016/j.parco.2010.06.001. 193

E. Torng. A unified analysis of paging and caching. *Algorithmica*, 20:194–203, 1998. 133, 148, 184

L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990a. 31, 54

L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 943–972. MIT Press, Cambridge, MA, USA, 1990b. 30

L. G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, 2011. 54, 55

P. van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. MIT Press, 1990. 9, 10

P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. 108

S. Venkatasubramanian, R. W. Vuduc, and n. none. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 244–255, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-498-0. doi: 10.1145/1542275.1542312. URL http://doi.acm.org/10.1145/1542275.1542312. 193

U. Vishkin. Implementation of simultaneous memory address access in models that forbid it. *J. Algorithms*, 4:45–50, 1983. 12

J. Vitter and R. Simons. New classes for parallel complexity: A study of unification and other complete problems for p. *Computers, IEEE Transactions on*, C-35(5):403 –418, may 1986. ISSN 0018-9340. doi: 10.1109/TC.1986.1676783. 96

C. Vömel, S. Tomov, and J. Dongarra. Divide and conquer on hybrid GPU-accelerated multicore systems. *SIAM Journal on Scientific Computing*, 34(2):C70–C82, 2012. doi: 10.1137/100806783. URL http://epubs.siam.org/doi/abs/10.1137/100806783. 193

S. Wagner. *Restricted Cache Scheduling*. PhD thesis, Michigan State University, 2001. 174

A. Wolfe Gordon and P. Lu. Low-latency caching for cloud-based web applications. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB '11)*, Athens, Greece, 2011. 169, 173

S. Wu and U. Manber. Fast text searching: allowing errors. *Commun. ACM*, 35(10):83–91, Oct. 1992. ISSN 0001-0782. doi: 10.1145/135239.135244. URL http://doi.acm.org/10.1145/135239.135244. 102, 121

N. E. Young. On-line file caching. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98, pages 82–86, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. ISBN 0-89871-410-9. URL http://dl.acm.org/citation.cfm?id=314613.314658. 133, 169