Algorithmic Foundation of Parallel Paging and Scheduling under Memory Constraints

A Dissertation presented

by

Rathish Das

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

January 2021

Stony Brook University

The Graduate School

Rathish Das

We, the dissertation committe for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Michael A. Bender - Dissertation Advisor
Professor, Department of Computer Science

Rezaul Chowdhury - Dissertation Advisor
Associate Professor, Department of Computer Science

Joseph S. B. Mitchell - Dissertation Advisor
Professor, Department of Applied Mathematics & Statistics and Computer Science

Steven Skiena - Chairperson of Defense
Professor, Department of Computer Science

Esther M. Arkin
Professor, Department of Applied Mathematics & Statistics and Computer Science

Kunal Agrawal - External Committee Member
Associate Professor, Computer Science and Engineering, Washington University in St. Louis

This dissertation is accepted by the Graduate School

Eric Wertheimer
Dean of the Graduate School

Abstract of the Dissertation

Algorithmic Foundation of Parallel Paging and Scheduling under Memory Constraints

by

Rathish Das

Doctor of Philosophy

in

Computer Science

Stony Brook University

2021

Every modern computer—be it a desktop or a supercomputer—uses hierarchical memory. One of the crucial characteristics of any memory hierarchy is that there is a small, fast memory, followed by a large, slow memory (possibly followed by more levels). The small, fast memory is a scarce resource, and efficiently managing it is crucial to high performance. Data is served to the CPU from the small, fast memory.

Sharing the small and fast memory among multiple processors gives rise to many challenges. The first challenge is to efficiently share the small and fast memory among multiple processors even if the processors access disjoint sets of pages. The second challenge is to avoid the race conditions that arise when processors access sets of pages which overlap. Race conditions occur when two or more processors access the same memory location, and at least one modifies its content. The third challenge is to efficiently synchronize the processors to minimize the parallel running time.

The contribution of this dissertation is thus three-fold. To handle the first challenge, we develop an algorithmic foundation for automated management of shared-memory in multilevel-memory systems. We consider the parallel paging problem where p processors share a memory of size k. The goal of the problem is to partition the memory among the processors over time to minimize the average completion time. We resolve this long-standing open problem by designing online algorithms with tight upper and lower bounds of $\Theta(\log p)$ competitive ratio with $O(1)$ resource augmentation.

We further extend the parallel paging problem to a new kind of multilevel-memory system where there are bandwidth differences among the memory levels. This new kind of memory is called high-bandwidth memory (HBM), and is common in new supercomputers. Systems equipped with HBM do not fit into classical memory-hierarchy models due to HBM's atypical characteristics, and hence natural and classical paging policies perform poorly. We present a simple but counterintuitive constant-competitive online algorithm for HBM management.

To handle the second challenge, we give provably good algorithms that manage memory

usage in a parallel computation to avoid race conditions and achieve optimal or near-optimal span (parallel running time). We ask the following question: given a fixed budget of extra memory for mitigating the cost of races in a parallel program, how should memory be used or scheduled in order to minimize the overall running time? We provide NP-hardness results and constant factor single and bicriteria approximation algorithms to this question.

To handle the third challenge, we present techniques that take into account synchronization costs among processors and yet achieve the optimal span at the cost of slightly increasing work. We present optimal $O(\log n)$ span algorithm for Strassen's Matrix Multiplication (MM) with only a $\Theta(\log \log n)$-factor blow-up in work as well as a near-optimal $O(\log n \log \log n)$ span algorithm with no asymptotic blow-up in work.

Dedicated to my parents Ramesh Das, Manju Das, and my sister Manasi Das.

Contents

Acknowledgments

I would like to express my deep gratitude to my advisors Profs. Michael A. Bender, Rezaul A. Chowdhury, and Joseph S. B. Mitchell, who have always been making things simple to understand. Without their deep insight into this domain and valuable time to teach me, it would not have been possible for me to move ahead properly.

I would like to thank Prof. Esther Arkin for educating me with many concepts of approximation algorithms and hardness proofs. I am indebted to Profs. Kunal Agrawal, Enoch Peserico, and Michele Squizzatto for all the engaging discussions on the parallel paging problem that became a core component of this dissertation.

I am also very grateful to Profs. Martín Farach-Colton, Jie Gao, Mayank Goswami, Rob Johnson, Matthew J. Katz, Benjamin Moseley, Enoch Peserico, Valentin Polishchuk, Michele Scquizzato, Steven Skiena, and Csaba D. Tóth for their constant guidance in the research projects that I worked on during my Ph.D. I also thank Cynthia Phillips and Jonathan Berry from Sandia National Lab for sharing their expertise in computer systems.

I also had the opportunities to work with and learn from my colleagues Zafar Ahmad, Anand Aiyer, Rory Bennett, Arghya Bhattacharya, Arani Bhattacharya, Ayon Chakraborty, Rohit Chatterjee, Abiyaz Chowdhury, Sharmila Duppala, Aaron Gregory, Mohammad Javanmard, William Kuszmaul, Tianchi Mo, Andrea Lincoln, Quanquan Liu, Sarthak Ghosh, Logan Graham, Jayson Lynch, Sam McCauley, Prashant Pandey, Sikha Singh, David Tench, Shih-Yu Tsai, Helen Xu, and Yimin Zhu. My labmates Zafar Ahmad and Aaron Gregory helped me a lot at different stages of my Ph.D.

I am especially indebted to Ayon Chakraborty, Pramod Ganapathi, and William Kuszmaul for their constant support and all the things they have taught me. They have helped me a lot to improve my presentation and writing skill. I had a very enjoyable experience with them while discussing the technicalities of the research problems that I worked on during my Ph.D.

I thank the Institute of Advanced Computation Science, Stony Brook University for supporting me with a Junior Researcher Award Fellowship. I am also thankful to the NSF for supporting my research through grants CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CNS-1938709, and CCF-1439084.

My deepest gratitude extends to my parents and my sister who were always very supportive to me. I am also very grateful to my uncle Arun Biswas who at my absence helped my parents during our difficult times.

Publications

Referred Conference Publications during My Ph.D.

- (Alphabetical) Kunal Agrawal, Michael A. Bender, **Rathish Das**, William Kuszmaul, Enoch Peserico, and Michele Scquizzato, "Tight Bounds of Parallel Paging and Green Paging." *Proceedings of the 32nd Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), Alexandria, USA (Virtual conference), pp. 3022-3041, 2021.
- (Alphabetical) Michael A. Bender, **Rathish Das**, Martín Farach-Colton, Tianchi Mo, David Tench, and Yung Ping Wang, "Mitigating false positives in filters: to adapt or to cache?" *Proceedings of the 2nd SIAM Symposium on Algorithmic Principles of Computer Systems* (APOCS), Alexandria, USA (Virtual conference), pp. 16-24, 2021.
- **Rathish Das**, Kunal Agrawal, Michael Bender, Jonathan Berry, Benjamin Moseley, and Cynthia Phillips, "How to Manage High-Bandwidth Memory Automatically." *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA), Philadelphia, USA (Virtual conference), pp. 187-199, 2020.
- (Alphabetical) Esther Arkin, **Rathish Das**, Jie Gao, Mayank Goswami, Joseph Mitchell, Valentin Polishchuk, and Csaba D. Toth, "Cutting Polygons into Small Pieces with Chords: Laser-Based Localization." *Proceedings of the 28th Annual European Symposium on Algorithms* (ESA-Track A), Pisa, Italy (Virtual Conference), pp. 7:1-7:23, 2020.
- (Alphabetical) Michael Bender, Rezaul Chowdhury, **Rathish Das**, Rob Johnson, William Kuszmaul, Andrea Lincoln, Quanquan Liu, Jayson Lynch, and Helen Xu, "Closing the Gap Between Cache-oblivious and Cache-adaptive Analysis." *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA), Philadelphia, USA (Virtual Conference), pp. 63-73, 2020.
- (Alphabetical) Michael Bender, **Rathish Das**, Martin Farach-Colton, Rob Johnson, and William Kuszmaul, "Flushing Without Cascades," *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), Salt Lake City, USA, pp. 650–669, 2020.
- **Rathish Das**, Shih-Yu Tsai, Sharmila Duppala, Jayson Lynch, Esther Arkin, Rezaul Chowdhury, Joseph Mitchell, and Steven Skiena, "Data Races and the Discrete Resource-time Tradeoff Problem with Resource Reuse over Paths," *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA), Phoenix, Arizona, USA, pp. 359–368, 2019.
- Mohammad Javanmard, Pramod Ganapathi, **Rathish Das**, Zafar Ahmad, Stephen Tschudi, and Rezaul Chowdhury, "Towards Efficient Architecture-Independent Algorithms for Dynamic Programs," *Proceedings of the 33rd International Conference on High Performance Computing* (ISC), Frankfurt. Germany, pp. 143–164, 2019.

Short Paper(s)/Workshop Paper(s)/Poster(s) during My Ph.D.

- (Alphabetical) Kunal Agrawal, Michael A. Bender, **Rathish Das**, William Kuszmaul, Enoch Peserico, and Michele Scquizzato, "Tight Bounds of Parallel Paging and Green Paging." *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA), Philadelphia, USA, pp. 493-495, 2020.
- Mohammad Javanmard, Pramod Ganapathi, **Rathish Das**, Zafar Ahmad, Stephen Tschudi, and Rezaul Chowdhury, "Towards Efficient Architecture-Independent Algorithms for Dynamic Programs:poster," *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (PPoPP), Washington D. C. USA, pp. 413–414, 2018.
- (Alphabetical) Esther Arkin, Peter Brass, **Rathish Das**, Jie Gao, Mayank Goswami, Joseph Mitchell, Valentin Polishchuk, and Csaba D. Toth, "Optimal Cutting of a Polygon by Lasers," *26th Fall Workshop of Computational Geometry* (FWCG), New York, USA, 2016.

Chapter 1

Introduction

Every modern computer, be it a desktop or a supercomputer, uses hierarchical memory. Hierarchical memory consists of multiple levels of memory—registers, followed by L1, L2, and possibly L3 caches, and then DRAM and disk—that vary in latency and size. The idea is to put the memory with the lowest latency closer to the CPU to feed the CPU at a faster rate and to increase the memory size down the hierarchy gradually.

One of the crucial characteristics of any memory hierarchy is that they have a small, fast memory, followed by a large, slow memory (possibly followed by more levels).

The small, fast memory is a scarce resource, and efficiently managing it is crucial to high performance. Data is served to the CPU from the small, fast memory. Since the small memory often cannot fit all the data required by a processor, data is transferred between the small and the large memories. Often the time spent on data transfers between the memory levels dominates the running time of programs. Hence, optimizing data transfers and efficiently managing the fast, small memory play a vital role in getting high performance.

Sharing the small and fast memory among multiple processors gives rise to many challenges. In recent times, there has been a surge in use of multiprocessor architectures. The end of Moore's law [134, 139] suggests that a processor's clock speed has almost reached its limit. Hence, to maintain steady growth in processing power, vendors have taken a different strategy—using multiple processors instead of a uniprocessor to share the same workload. This new technique has placed parallel computation at the forefront of high performance computing. At the same time, multiprocessor systems bring new challenges that are not seen when memory is used by a single processor only.

The first challenge is to efficiently share the small and fast memory among multiple processors even if the processors access disjoint sets of pages. As all processors compete for the same shared memory, efficiently sharing that memory is a significant concern to minimize the processors' parallel running time. The marginal benefit of more memory may vary across processors, and this relationship may not have a good structure. For instance, it may be that Processor 1 derives more marginal benefit from one extra page compared to Processor 2, while at the same time, Processor 2 derives more benefit from ten extra pages compared to Processor 1. Besides, this marginal benefit of extra memory can vary over time. Complicating matters even further, the actual scheduling of processors matters. For instance, running a small subset of processors—and temporarily stalling all others—may allow for better performance compared to running all processors at once. Therefore, deciding how the different processors' schedules are interleaved is an

essential part of efficiently managing any shared-memory multiprocessor system.

The second challenge is to avoid race conditions that arise when processors do not access disjoint sets of pages. Race conditions occur when two or more processors access the same memory location, and at least one modifies its content. Races are often undesirable as they can lead to nondeterministic and incorrect program behavior. Hence, designing race-free programs is very critical for shared-memory multiprocessor systems. Races can be eliminated by associating a mutual-exclusion lock with the memory location or allowing only atomic accesses to it. Such a solution, however, makes all accesses to that location serial and thus destroys all parallelism. For associative and commutative updates to a memory cell, one can instead use a reducer, which allows parallel race-free updates at the expense of using some extra space. More extra space usually leads to more parallel updates, which in turn improve the parallelism of a multiprocessor system.

The third challenge is to efficiently synchronize processors to minimize parallel running time. In the classical PRAM model, spawning p threads into p processors takes $\Theta(1)$ time, which is not very realistic. The Binary-Forking model is a relatively new model that charges $\Theta(\log p)$ span (parallel running time) to spawn (fork) p threads in parallel, thus taking into account the synchronization cost to spawn p processors. One can trivially transform an algorithm from the PRAM model to the binary-forking model by simply adding an extra $\Theta(\log n)$ span for every such fork/join operation. This increases the span by a factor of $O(\log n)$. The question looms: can an algorithm from the PRAM model be transformed to the binary-forking model with the same asymptotic bounds for span and work?

1.1 Scope and Contribution of this Dissertation

The scope of this dissertation is to tackle the three vital challenges described above that arise in shared-memory multiprocessor systems: (1) efficiently managing the memory shared by multiple processors, (2) avoiding race conditions with minimal loss in parallelism, and (3) synchronizing processors efficiently to minimize span. The contribution of this dissertation is thus three-fold. (1) To handle the first challenge, we develop an algorithmic foundation for automated management of shared-memory in multilevel-memory systems; (2) to handle the second challenge, we give provably good algorithms that schedule memory in a parallel computation to avoid race conditions and achieve optimal or near-optimal span (parallel running time); (3) to handle the third challenge, we present techniques that take into account synchronization costs among the processors and yet achieve optimal span at the cost of slightly increasing work.

1.2 Algorithmic Foundation for Automated Management of Shared Memory (Handling the First Challenge):

The problem of managing the contents of a ***cache*** (i.e., a small ***fast memory***) is crucial to achieving good performance on large machines with multi-level memory hierarchies. This problem is classically known as ***caching*** or ***paging*** [23, 104, 126, 42]. When a processor accesses a location in fast memory, the access cost is small (the access is a ***hit***); when it accesses a location that is not in fast memory, the access cost is large (the access is a ***miss*** or a ***fault***). The paging algorithm

decides which ***pages*** (or ***blocks***) remain in fast memory at any point in time, or in other words, which page(s) to evict when a new page is brought into fast memory. This problem is generally formulated as being ***online***, meaning that the paging algorithm does not know the future requests.

Sequential paging—when there is a single processor which accesses fast memory—has been studied for decades and has many positive results [23, 104, 126, 42] and lower bounds [126]. There are also many extensions with different weights, block sizes, and beyond-worst-case competitive analyses [112, 63, 14, 62, 17, 79, 141, 43, 20, 18, 70, 82, 83].

We study the problem of ***parallel paging*** where p processors share the same fast memory of some size k. Each processor runs its own program, and the set of pages accessed by different programs are disjoint.[1] The goal is to share the small memory among the processors in a way that minimizes some objective function of processors' completion times. We focus on minimizing average completion time, and we also give bounds on makespan (i.e., maximum completion time) and median completion time.

We further extend the parallel paging problem to a new kind of multilevel-memory system where there are bandwidth differences among the memory levels. This new kind of memory is called high-bandwidth memory (HBM) and is common in new supercomputers.

1.2.1 Parallel Paging

The parallel paging problem was first articulated by Fiat and Karlin [69] in 1995 and has remained open since then. The parallel paging problem introduces new challenges that are not encountered in the sequential problem.

First, multiple processors compete for the same resource (shared cache) and the paging algorithm must decide, for each processor and at each time, how many and which of its pages to keep in cache. The marginal benefit of having more memory may vary across processors and this relationship may not have good structure. For instance, it may be that Processor 1 derives more marginal benefit from one extra page compared to Processor 2 while at the same time, Processor 2 derives more benefit from ten extra pages compared to Processor 1.

In addition, this marginal benefit of extra cache can vary over time and the online paging algorithm must change the number of pages of fast memory allocated to the processors accordingly. Complicating matters even further, the actual scheduling of processors matters. For instance, running a small subset of processors—and temporarily stalling all others—may allow for better performance compared to running all processors at the same time. Therefore, how the different processors' accesses are *interleaved* is an important part of what the parallel paging algorithm must decide, and different interleavings of accesses can lead to vastly different performances.

Whereas sequential paging has been understood for decades [23, 104, 126, 42], parallel paging has largely resisted analysis. The only known upper bounds for parallel paging [47, 69, 22, 88, 94, 68, 106] consider relaxations of the problem in which the interleaving of accesses by different

[1] It would also be interesting to consider the related problem in which the processors are collectively running a *single* program in a shared address space. To model this, one would have to also capture dependencies within the parallel program, for example with a dependency DAG. Complicating matters further, for many parallel programs the dependency DAG is non-deterministic depending on data races. This work focuses on disjoint entities, that can be arbitrarily interleaved without dependencies, and aims to understand how such entities can efficiently share memory resources.

processors is fixed ahead of time—in particular, this is enforced by making it so that, whenever any processor blocks on a miss, *all of the processors block*, which in turn eliminates a large amount of the parallelism inherent to the problem. The known lower bounds [80, 102], on the other hand, focus only on the offline parallel paging problem, and on analyzing the performance of traditional paging algorithms such as LRU. Parallel paging has also been extensively studied within the systems community, particularly after multicore processors became mainstream—starting from some pioneering work on (offline and online) heuristics that dynamically adjust the sizes of the cache partitions dedicated to each processor (see, e.g., [135, 130, 132, 99, 140, 48]).

Our Contributions. We consider this long standing open problem of parallel paging in its general form when the interleavings are not fixed. As in [80, 102], we analyze this problem in a ***timed*** model where a hit takes unit time and a miss takes s time units for some parameter $s > 1$. Note that, in the traditional model used for paging [126, 42], a hit costs 0 and a miss costs 1 — we will call this the 0-1 model. The timed model is more general and captures the 0-1 model as a special case (by letting s tend towards infinity).

An important feature of the timed model is that it captures the passage of time even when a processor experiences a hit. Modeling this passage of time allows for a more fine-grained analysis of parallel paging since it allows us to compare the progress of one processor to another even if one is experiencing hits while the other is experiencing misses. This avoids the unrealistic assumption in the 0-1 model that a particular processor can have an infinite number of hits in the time that another processor has a single miss.

We give tight upper and lower bounds for the deterministic online parallel paging problem in the timed model showing that the optimal competitive ratio for average completion time is $\Theta(\log p)$, using constant resource augmentation. We also consider makespan, proving a lower bound of $\Omega(\log p)$ and an upper bound of $O(\log^2 p)$ for the competitive ratio. This result represents significant progress on a long-standing open problem, even for the 0-1 model. A remarkable feature of our algorithms is that they are oblivious to s, achieving optimal competitive ratios for all values of s simultaneously.

1.2.2 Paging in High-Bandwidth Memory (HBM)

We now extend the parallel paging problem to a new kind of multilevel-memory system where there are bandwidth differences among the memory levels. This new kind of memory is called high-bandwidth memory (HBM) and is common in new supercomputers.

In the past two decades, the number of cores per chip has grown significantly. As a result, the *relative memory capacity*, defined as memory capacity divided by available gigaflops, has decreased by more than 10x [93]. Thus, processors are becoming more starved for data. Recent innovations in 3D die-stacking technology have driven vendors in implementing a new approach for improving memory performance [109, 92], specifically, memory bandwidth. The approach is to bond memory directly to the processor package where there can be more parallel connections between the memory and the processors' private caches, enabling a higher bandwidth than can be achieved using older technologies. Throughout the rest of this dissertation, we refer to on-package 3D memory technologies as ***high-bandwidth memory*** or ***HBM***.[2]

[2]Hardware vendors use various brand names such as High-Bandwidth Memory (HBM), Hybrid Memory Cube (HMC), and MCDRAM for this technology.

HBM, with its improved ability to feed processors, provides an opportunity to overcome this memory bottleneck, if application software can use it.

HBM is not a replacement for DRAM ("main memory") since it is generally about 5 times smaller than DRAM due to constraints such as heat dissipation, as well as economic factors. For example, current HBM sizes range from 16 gigabytes per compute node (the Department of Energy's "Trinity" [61]) to 96 gigabytes per compute node (the Department of Energy's "Summit" [138]), several times smaller than the per-node sizes of DRAM on those systems (96 GB and 512 GB, respectively).

HBM management is not identical to cache management as it introduces complications that do not exist in more traditional memory hierarchies. In particular, cores compete not only for HBM capacity, but also for the more limited channel capacity between HBM and DRAM. HBM does not fit into a standard memory hierarchy model [74, 27], because in traditional hierarchies, both the latency and bandwidth improve as the levels get smaller. This is not true of HBM.

The question looms: are there provably good algorithms for automatically controlling HBM?

Our Contributions We propose a multicore model for HBM that captures the high bandwidth from the cores to HBM and the much lower bandwidth to DRAM. There are p parallel channels connecting p cores to the HBM but only a single channel connecting HBM to DRAM. This configuration captures the high (on-package) bandwidth between the p cores and HBM and the much lower (off-package) bandwidth between HBM and DRAM. Data is transferred in blocks — there are up to p parallel block transfers from the HBM to the cores, but only one block transfer at a time between DRAM and HBM. The roughly comparable latencies are captured by setting all block-transfer costs (times) to 1.

We focus on instances where the multicore's threads access disjoint sets of blocks. This emphasizes the cores' competition for HBM and the limited bandwidth between HBM and DRAM. We present the following set of results.

- *Sharing the HBM-to-DRAM channel fairly does not work.* We consider how to design block-replacement policies for the HBM coupled with the First-Come-First-Serve (FCFS) algorithm for determining the order of accesses from HBM to DRAM. We show that even though LRU is a very good block-replacement policy, if we use FCFS in the HBM-to-DRAM channel, LRU performs poorly. In particular, with any constant amount of resource augmentation the makespan of using FCFS with LRU is an $\Omega(p)$-factor away from the optimal policy in the worst case. This negative result establishes that more sophisticated management of the channel between HBM and DRAM is central to designing a good algorithm for the problem. The seemingly fair FCFS policy is bad.
- *Priority-based mechanism for managing the HBM-to-DRAM channel.* Our main contribution is to devise a $O(1)$-competitive online algorithm that minimizes the maximum running time of p processors sharing an HBM. We give a priority-based policy for managing the channel between HBM and DRAM. We impose a pecking order on the cores, so that a high-priority core never has a request to DRAM blocked by a request from a lower-priority core. Our algorithms for HBM management are built around this priority-based mechanism.
- *Minimizing the maximum running time is strongly NP-hard.*
- $O(1)$*-approximation algorithms for average completion time*

1.3 Avoiding Race Conditions with Extra Memory (Handling the Second Challenge)

We now consider the case where the processors collectively execute a single program in a shared address space. Sharing an address space often gives rise to determinacy races. A determinacy race [108, 66] occurs if two or more logically parallel instructions access the same memory location, and at least one of them modifies its content. Races can lead to nondeterministic and incorrect program behavior, and thus they are often undesirable.

A data race is a special case of a determinacy race which can be eliminated by associating a mutual-exclusion lock with the memory location in question or allowing only atomic accesses to it. Such a solution, however, makes all accesses to that location serial and thus destroys all parallelism.

Reducers [72, 41, 117] give an alternative way to eliminate data races on a shared variable without destroying parallelism when the variable's update operations are associative and commutative. Such elimination of data races is achieved by using extra space.

We model data races in a program using a directed acyclic graph, assuming that there are no cyclic read-write dependencies among the memory locations accessed by the program. Each node in the DAG represents a memory location, and a directed edge from node u to node v means that v is updated using the value stored at u. The in-degree $d_v^{(in)}$ of node v gives the number of times v is updated.

Naturally, the question emerges: given a fixed budget of extra space for mitigating the cost of races in a parallel program, which memory locations should be assigned reducers and how should the space be distributed among those reducers in order to minimize the overall running time?

We concentrate on a variation of this problem where space reuse among reducers is allowed by routing every unit of extra space along a (possibly different) source to sink path of the DAG and using it in the construction of multiple (possibly zero) reducers along the path. This problem variation is motivated by the bottleneck seen in highly parallel programs when a single global memory manager is used. Recursive fork-join programs often avoid repeated calls to an external memory manager altogether, along with the overhead of repeated memory allocations/deallocations. Instead, recursive fork-join programs allocate a large segment of memory at the beginning of the program execution and then split and merge the memory segment along the fork and join nodes, respectively.

Remarkably, our problem formulation extends to more general problems such as project scheduling, inventory management, etc.

Our Contributions We generalize our race-avoiding space-time tradeoff problem to a discrete resource-time tradeoff problem with general non-increasing duration functions and resource reuse over paths of the given DAG.

For general DAGs, we show that even if the entire DAG is available to us offline the problem is strongly NP-hard under all three types of duration functions, and we give approximation algorithms for solving the corresponding optimization problems. We also prove hardness of approximation for the general resource-time tradeoff problem and give a pseudo-polynomial time algorithm for series-parallel DAGs.

1.4 Reducing Synchronization Cost of the Processors with Extra Memory (Handling the Third Challenge)

We now turn our attention to a parallel computation model, called the binary-forking model, that better captures the performance of parallel algorithms implemented using modern multithreaded programming languages on multicore shared-memory machines. The widely studied theoretical PRAM model does not consider costs of spawning and synchronizing threads, and the costs are often non-constant in real machines. The binary-forking model incurs a cost of $\Theta(\log n)$ to spawn or synchronize n tasks or threads, and thus realistically captures the performance of modern parallel machines. As a result, algorithms achieving optimal performance bounds in the PRAM model may not be optimal in the binary-forking model. Often, algorithms need to be redesigned to achieve optimal performance bounds in the binary-forking model, and the non-constant synchronization cost makes the task challenging.

For many problems in the binary-forking model, one can achieve the same span as in PRAM at the expense of increasing the total work of an algorithm. For example, Cole's parallel merge sort achieves optimal span $\Theta(\log n)$ and does optimal total work $\Theta(n \log n)$ in the PRAM model. One could trivially retain the optimal $\Theta(\log n)$ span in the binary-forking model by increasing work to $\Theta(n^2)$—in parallel, each item finds its rank in the sorted array by comparing itself with $n - 1$ other items. So the real challenge to design algorithms in the binary forking model is to retain both the same span and work bounds as in the PRAM model.

Our contributions In this dissertation, we design efficient parallel algorithms in the binary-forking model for fundamental problems such as Strassen's (and Strassen-like) matrix multiplication (MM). Our results easily extend to many dynamic programming algorithms in the binary-forking model. We present an optimal $O(\log n)$ span algorithm for Strassen's Matrix Multiplication (MM) with only a $\Theta(\log \log n)$-factor blow-up in work as well as a near-optimal $O(\log n \log \log n)$ span algorithm with no asymptotic blow-up in work.

A remarkable feature of our algorithms is that we avoid the use of atomic instructions except possibly inside the join operations, which are implemented by the runtime system.

1.5 Organization of this Dissertation

This dissertation is organized as follows. In Chapters 2 and 3 we handle the first challenge to efficiently share the small and fast memory among multiple processors. In Chapter 2 we present tight bounds for parallel paging. In Chapter 3 we extend the parallel paging problem to a modern memory hierarchy, called high-bandwidth memory. In Chapter 4 we handle the second challenge to avoid race conditions. In Chapter 5 we present techniques to reduce synchronization costs among processors in order to achieve better span and work bounds. Finally, we give a conclusion in Chapter 6.

Chapter 2

Algorithmic Foundation of Parallel Paging

In the *parallel paging* problem, there are p processors that share a cache of size k. The goal is to partition the cache among the processors over time in order to minimize their average completion time. For this long-standing open problem, we give tight upper and lower bounds of $\Theta(\log p)$ on the competitive ratio with $O(1)$ resource augmentation.

A key idea in both our algorithms and lower bounds is to relate the problem of parallel paging to the seemingly unrelated problem of *green paging*. In green paging, there is an energy-optimized processor that can temporarily turn off one or more of its cache banks (thereby reducing power consumption), so that the cache size varies between a maximum size k and a minimum size k/p. The goal is to minimize the total energy consumed by the computation, which is proportional to the integral of the cache size over time.

We show that any efficient solution to green paging can be converted into an efficient solution to parallel paging, and that any lower bound for green paging can be converted into a lower bound for parallel paging, in both cases in a black-box fashion. We then show that, with $O(1)$ resource augmentation, the optimal competitive ratio for deterministic online green paging is $\Theta(\log p)$, which, in turn, implies the same bounds for deterministic online parallel paging.

Portions of this work are based on preliminary results contained in the thesis [122]. This work was presented as a brief announcement at SPAA 2020 [9] and as a full paper at SODA 2021 [10].

2.1 Introduction

The problem of managing the contents of a ***cache*** (i.e., a small ***fast memory***) is critical to achieving good performance on large machines with multi-level memory hierarchies. This problem is classically known as ***paging*** or ***caching*** [42]. When a processor accesses a location in fast memory, the access cost is small (the access is a ***hit***); when it accesses a location that is not in fast memory, the access cost is large (the access is a ***miss*** or a ***fault***). The paging algorithm decides which ***pages*** (or ***blocks***) remain in fast memory at any point in time or, in other words, which page(s) to evict when a new page is brought into fast memory. This problem is generally formulated as being ***online***, meaning that the paging algorithm does not know the future requests.

Sequential paging—when there is a single processor accessing the fast memory—has been

studied for decades and is a very well-understood problem [23, 104, 126, 42], including several of its extensions.

In this work, we study the problem of ***parallel paging*** where p processors share the same fast memory of some size k. Each processor runs its own program, and the set of pages accessed by different programs are disjoint. At each point in time, the paging algorithm gets to decide how much cache goes to each processor, and also gets to dictate each processor's eviction strategy. The goal is to share the small memory among the processors in a way that minimizes some objective function of processors' completion times. We focus on minimizing average completion time, and we also give bounds on makespan (i.e., maximum completion time) and median completion time.

The parallel paging problem introduces complexity that is not seen in the sequential problem. First, multiple processors compete for the same resource and the paging algorithm must decide, for each processor and at each time, how many and which of its pages to keep in cache. The marginal benefit of having more memory may vary across processors and this relationship may not have good structure. For instance, it may be that Processor 1 derives more marginal benefit from one extra page compared to Processor 2 while at the same time, Processor 2 derives more benefit from ten extra pages compared to Processor 1. In addition, this marginal benefit of extra cache can vary over time and the online paging algorithm must change the number of pages of fast memory allocated to the processors accordingly. Complicating matters even further, the actual scheduling of processors matters. For instance, running a small subset of processors—and temporarily stalling all others—may allow for better performance compared to running all processors. Therefore, how the different processor's accesses are *interleaved* is an important part of what the parallel paging algorithm must decide, and different interleavings of accesses can lead to vastly different performances.

Whereas sequential paging has been understood for decades [23, 104, 126, 42], parallel paging has largely resisted analysis. The only known upper bounds for parallel paging [47, 69, 22, 88, 94, 68, 106] consider relaxations of the problem in which the interleaving of accesses by different processors is fixed ahead of time—in particular, this is enforced by making it so that, whenever any processor blocks on a miss, *all of the processors block*, which in turn eliminates a large amount of the parallelism inherent to the problem. The known lower bounds [80, 102], on the other hand, focus only on the offline parallel paging problem, and on analyzing the performance of traditional paging algorithms such as LRU. Parallel paging has also been extensively studied within the systems community, particularly after multicore processors became mainstream—starting from some pioneering work on (offline and online) heuristics that dynamically adjust the sizes of the cache partitions dedicated to each processor (see, e.g., [135, 130, 132, 99, 140, 48]).

This Dissertation. In this work, we consider this long-standing open problem of parallel paging in its general form when the interleavings are not fixed. As in [80, 102], we analyze this problem in a ***timed*** model where a hit takes unit time and a miss takes s time units for some parameter $s \geq 1$.[1] Note that, in the traditional model used for paging [126, 42], a hit costs 0 and a miss costs 1 — we will call this the 0-1 model. The timed model is more general and captures the 0-1 model as a special case (by letting s tend towards infinity).

An important feature of the timed model is that it captures the passage of time even when a processor experiences a hit. Modeling this passage of time allows for a more fine-grained analysis

[1] We use the letter s to be consistent with the notation used in the ***full access cost model*** [42], which charges 1 for an access to fast memory and $s \geq 1$ to move a page from slow to fast memory.

of parallel paging since it allows us to compare the progress of one processor to another even if one is experiencing hits while the other is experiencing misses. This avoids the unrealistic assumption in the 0-1 model that a particular processor can have an infinite number of hits in the time that another processor has a single miss.

We give tight upper and lower bounds for the deterministic online parallel paging problem in the timed model showing that the optimal competitive ratio for average completion time is $\Theta(\log p)$, using constant resource augmentation. (Resource augmentation is perhaps the most popular refinement of competitive analysis that allows bypassing overly pessimistic worst-case bounds by endowing the online algorithm with more resources than the offline optimum it is compared to [42].) We also consider makespan, proving a lower bound of $\Omega(\log p)$ and an upper bound of $O(\log^2 p)$ for the competitive ratio. This result represents significant progress on a long-standing open problem—it was first articulated by Fiat and Karlin [69] in 1995 and has remained open even in the 0-1 model. A remarkable feature of our algorithms is that they are oblivious to s, achieving optimal competitive ratios for all values of s simultaneously.

A foundational idea in both our algorithms and lower bounds is to relate the problem of parallel paging to the (seemingly unrelated) problem of ***green paging***, which focuses on minimizing the memory usage (hence, e.g., the energy consumption) of a (single) processor's cache for a computation, and is a problem of independent interest.

Green Paging. In green paging, the fast memory consists of ***memory banks*** that can be turned on or off over time. Memory banks that are active (i.e., turned on) can store pages—and thus requests for those pages result in a hit—but also consume energy; memory banks that are inactive cannot store pages, but do not consume energy. The goal in green paging is to minimize the total energy consumption of a computation. More formally, there is a single processor that is running, and the green paging algorithm gets to control both the page-eviction policy and the size of the processor's cache over time, assigning any size between a minimum of k/p and a maximum of k, where p is a given parameter. The goal is to service the request sequence while minimizing the integral of memory capacity over time—a quantity we call ***memory impact***.[2] This is a simple model for studying the total amount of memory usage over time by a computation.

In this work, green paging serves both as a problem to be studied on its own, and as an *analytical tool* for studying parallel paging. Indeed, both our upper and lower bounds hinge on unexpected relationships between the two problems. We remark that the use of the same variable p for apparently unrelated quantities in the two problems is intentional, as it plays exactly the same role when "translating" one model into the other.

Results. We now summarize our results for deterministic online parallel and green paging. All of the results assume a constant factor of resource augmentation (which is necessary even for sequential paging [126, 73, 33]).

- **Relating parallel and green paging.** We show that green and parallel paging are tightly related. In particular, *any* green paging algorithm with k memory can be translated "black box"

[2]Note that it is not always optimal to keep the cache size at k/p since this can lead to a larger number of misses, thereby increasing the running time of the computation. As an example, say a processor accesses 4 pages in round-robin, accessing each page t times. If we give the processor a cache of size 4, it will finish in time $4t$ for a total memory impact of $16t$. On the other hand, if we give it a cache of size 2, at least half of its accesses will be misses. Therefore, the running time will be at least $2st$ for a total memory impact of $4st$. For any $s > 4$, allocating a cache of size 4 results in a smaller memory impact than allocating a smaller cache of size 2.

into a parallel paging algorithm with $O(k)$ memory capacity. If the former is online, so is the latter. If the former has a competitive ratio of β, then the latter achieves an average completion time with a competitive ratio of $O(\beta)$; additionally, the latter achieves a $(1-\varepsilon)$-completion time (i.e., the time to complete all but an ε fraction of sequences) that is $O(\beta \log(\varepsilon^{-1}))$-competitive with the optimal $(1-\varepsilon/2)$-completion time.

We also prove a relationship between lower bounds for the two problems. If we have an online lower bound construction against online algorithms for green paging, achieving a competitive ratio of $\Omega(\beta)$, then all online algorithms for parallel paging must also have competitive ratio $\Omega(\beta)$ for both mean completion time and maximum completion time.

- **Tight bounds for green paging.** For a lower bound, we show that any online deterministic algorithm has a competitive ratio of at least $\Omega(\log p)$ even with $O(1)$ resource augmentation. And for an upper bound, we give a simple memory allocation algorithm that is both online and *memoryless* (i.e., it does not depend on future or past requests) and that can be combined with LRU replacement to achieve the optimal competitive ratio of $O(\log p)$.
- **Tight bounds for parallel paging.** Using the previous two results, we obtain tight upper and lower bounds of $\Theta(\log p)$ for the parallel paging problem, both when the objective is to minimize mean completion time, and when the objective is to minimize the time spent completing a constant fraction of processes. We also arrive at an upper bound of $O(\log^2 p)$ and a lower bound of $\Omega(\log p)$ for the competitive ratio of optimizing makespan.

Bottom line: optimize memory impact. The relationship between green and parallel paging is powerful. For decades, little progress has been made on parallel paging partly because it has been unclear how to handle the interleaving and interference between different processors. The algorithms in this work give a clear lesson: rather than focusing on the interactions between processors, and rather than greedily focusing on optimizing the *running times* of processors, one should instead focus on minimizing the *memory impact* of each processor (the same value that is optimized by green paging!). This—along with basic load balancing to keep processors from getting too far ahead or behind—allows the processors to share the cache with each other in the most constructive possible way.

Related Work on Sequential Paging. As mentioned above, sequential paging has been studied for decades for a two-layer memory heirarchy in the 0-1 model. The simple algorithm LFD (Longest Forward Distance) that evicts the page accessed furthest in the future has long been known to be optimal [23, 104]. In the *online* setting, where the algorithm does not know the future, the *competitive analysis* framework is typically used to analyze algorithms. In the 0-1 model, an online paging algorithm has a ***competitive ratio*** of (no more than) β if, for every request sequence, it incurs at most β times as many faults as an optimal offline algorithm incurs with a memory of capacity $h \leq k$ (plus an additive constant independent of sequence length). The ratio $\alpha = k/h$ is called the ***resource augmentation*** factor. Many simple, deterministic algorithms including LRU, FIFO, FWF, and CLOCK have a competitive ratio of $\frac{k}{k-h+1}$ [126, 42]; and the same ratio holds for RAND [42]. This ratio is optimal for deterministic algorithms, and even for randomized ones if page requests can depend on previous choices of the paging algorithm. Since $\frac{k}{k-k/2+1} < 2$, this ratio implies that these algorithms never fare worse than the optimal offline algorithm would on a memory system with half the capacity and twice the access cost.

Related Work on Parallel Paging. The theoretical understanding of the parallel paging problem remains incomplete. Most prior positive results assume that the p request sequences are combined into a single fixed interleaved sequence, to be serviced by selecting which pages to keep in memory so as to minimize the total number of faults [47, 69, 22, 88, 94] or other metrics [106]. That is, the speed at which processors progress relatively to each other is treated as being *independent* of the number of page faults made by each processor, even though in reality a processor that incurs few faults will progress much faster than one that incurs many faults. Feuerstein and Strejilevich de Loma [68] further relax the problem so that they can choose the interleaved sequence rather than assuming that it is given.

An unfortunate consequence of the fixed-interleaving assumption is that whenever a processor incurs a fault, all other processes "freeze" until the fault is resolved. This negates much of the inherent parallelism in the problem since, when a processor encounters a fault, other processors should continue working. However, when one doesn't make this assumption, the problem becomes much more complicated since processors can advance while other processors are blocking on faults, and thus the relative rates at which processors advance is determined by when they hit or miss. This means that the *actual interleaving of request sequences of different processors depends on the paging algorithm*, since the paging algorithm determines when processors hit or miss.

Some recent works [80, 102, 55, 87] do not make the fixed-interleaving assumption; these works investigate the complexity of the offline problem and show lower bounds for traditional paging algorithms such as LRU, or consider restricted models. However, no general upper bounds or lower bounds are known, and the fully general problem of how to manage a shared fast memory among multiple processors has remained open.

Related Work on Green Paging. The last decade has seen a surge in interest for paging models where memory capacity is not static, but can instead change over time [50, 101, 79, 113, 114, 33, 31, 22]. One justification for such models is the increased popularity of virtualization/cloud services: the amount of physical memory allotted to a specific virtual machine often varies considerably over time based on the number and priority of other virtual machines supported by the same hardware. Another justification for dynamic capacity models lies in the ability of modern hardware to turn off portions of memory so as to reduce power, often with the goal of minimizing the overall energy used in a computation—this is the task we refer to as green paging.

The first work to address green paging was [50], allowing the paging algorithm to determine both the capacity and the contents of the memory on any given request, with the goal of minimizing a linear combination of the total number of faults and of the average capacity over all requests. This problem has been investigated by López-Ortiz and Salinger [101] and later, in the more general version where pages have sizes and weights, by Gupta et al. [79]. Subsequent work, and in particular the ***elastic paging*** of [113, 114], showed that one can effectively decouple page replacement from memory allocation: even if the latter is chosen adversarially, LFD is still optimal, and a number of well-known paging algorithms like LRU or FIFO are optimally competitive, with a competitive ratio that is extremely close albeit not quite equal to the classic $k/(k-h+1)$. A similar line was taken by ***adaptive caching*** [33] with a slightly different cost model. These results [113, 114, 33] imply that green paging is a problem of memory allocation: once memory is allocated, one can simply use LRU for page replacement—as its cost will be within a factor $O(1)$ of the optimal (for that memory allocation).

2.2 Technical Overview

This section gives an overview of our main results and of the techniques that we use to prove them. Recall that we will use the timed model where cache hits take time 1, and cache faults take time s for some integer $s \geq 1$. A detailed specification of the models for parallel paging and for green paging can be found in Section 2.3.

The relationship between Parallel and Green Paging. In parallel paging, the fact that processors must share a cache of size k suggests that the cache-usage per processor should be treated as a critical resource. Green paging, in turn, optimizes this resource for an *individual* processor by minimizing the total memory impact for that processor.

Green paging does not concern itself with minimizing running time directly though—for example, a green paging algorithm might choose to use a very small portion of the cache for a long time rather than a larger portion for a short time. Additionally, since green paging focuses on only a single processor, it does not say anything about the interactions between concurrent processors. These interactions (i.e., the ways in which the working sets for different processors change relative to each other over time) play a critical role in the parallel paging problem.

One of the key contributions of this work is that, in spite of these obstacles, memory impact really is the right way to think about parallel paging. Even though parallel paging involves complicated interactions between processors, we show that the problem can be decomposed in a way so that each individual processor can be optimized separately. The result is a black-box reduction from the parallel-paging problem to green paging. Remarkably, the opposite direction is also true: any online lower-bound construction for green paging can be transformed in a black-box fashion to obtain an online lower-bound construction for parallel paging.

By proving a tight relationship between green and parallel paging, and then giving tight bounds for green paging, we immediately obtain tight bounds for parallel paging as a result.

In the rest of this section, we first describe a series of simplifications that allow us to think about each individual processor's use of cache in terms of a so-called boxed memory profile. We then explain how to achieve tight bounds for green and parallel paging.

2.2.1 A Useful Tool: Box Profiles

In the green paging problem, the paging algorithm sets a ***memory profile*** $m(i)$, which dictates how much cache the processor uses at each point in time. A key insight, however, is that we need only consider profiles with a certain nice geometric structure, called box profiles.

Box profiles. A ***memory box of height h*** is a time-interval of length $\Theta(sh)$ during which a processor is allocated exactly h memory. We call a memory profile m a ***box profile*** if it can be decomposed into a sequence of memory boxes $b_1, b_2, \ldots$.

Box profiles are without-loss-of-generality in the following sense: If an online algorithm for green paging produces a memory profile m, then the algorithm can be modified (online) to instead produce a *box profile* m'. Moreover, the box profile m' will incur at most a constant-factor more memory impact than does m.

The intuition behind this transformation is the following: without loss of generality, the profile m never grows at a rate of more than 1 per s time steps, because fetching a page from the slow memory to the fast memory takes s time (although it can shrink arbitrarily fast). Thus, whenever

the memory profile m is at some height h, the profile must have already been at height $\Omega(h)$ for time at least $\Omega(sh)$. This naturally allows for one to decompose the profile into (overlapping) chunks where each chunk closely resembles a box. Making these chunks not overlap, and so that the decomposition is online, requires several additional ideas that we give in Section 2.4.

Box profiles were previously used as analytical tools for understanding cache-adaptive algorithms [33, 31], which are algorithms that exhibit optimal cache behavior in the presence of a varying-size cache. An interesting feature of our work is that box profiles play an important role not just analytically, but also *algorithmically* in our treatments of green and parallel paging.

Simplifying box profiles: compartmentalization, smooth-growth, and optimal eviction. We can also assume that the box profile m' has additional nice properties, the simplest of which are that every box has a power-of-two height 2^j, and that the height-to-width ratio of every box is always the same.

On top of these, we can assume ***compartmentalization***. This property says that, at the beginning of each box b's lifetime, the cache is flushed (i.e., the cache size briefly dips to 0). This means that each box of height h must incur h cache misses *just to populate its cache*. These cache misses can be handled by increasing the box's width by an additional sh. Since the box already had width $\Theta(sh)$, the increase does not change the asymptotic memory impact of the box. Compartmentalization plays an important role in the design of algorithms for parallel paging, since it means that consecutive boxes in a processor's profile need not be scheduled adjacently in time.

We can also assume the ***smooth-growth property***: whenever two boxes b_i, b_{i+1} come after one another, the height of the latter is at most twice that of the former. This property will be especially useful when proving lower bounds.

Finally, because each box in a box profile has a fixed height, LRU in a box of height 2^j is guaranteed to perform 2-competitively with the optimal eviction policy OPT in a box of height 2^{j-1} [126]. Up to a constant factor of resource augmentation, we can therefore assume without loss of generality that the optimal policy OPT is used within each box.

2.2.2 Tight Bounds for Green Paging

Consider a green paging instance with maximum memory k and minimum memory k/p. One can assume without loss of generality that k, p are powers of 2. In addition, we can assume that the page replacement strategy is optimal (or LRU) within each box—therefore, the algorithm needs only decide the sequence of boxes to be used. Furthermore, as discussed at the beginning of the section, one can also assume that the asymptotically optimal solution OPT is a box profile using boxes with heights $k/p, 2k/p, 4k/p, \ldots, k$.

A Universal Box Profile. In Section 2.6.2, we present the ***BLIND*** algorithm for green paging (specifically, to decide the sequence of boxes that the algorithm should use), which achieves competitive ratio $O(\log p)$ using constant resource augmentation. A remarkable property of this algorithm is that the sequence of boxes that it uses is *oblivious* to the input request sequence σ. In particular, the BLIND algorithm always uses a fixed sequence of boxes that we call the ***universal box profile*** U.

We construct the universal box profile U by performing repeated traversals of a tree $\mathcal{T}$, where each node in $\mathcal{T}$ is associated with a certain box size. The tree $\mathcal{T}$ has $1 + \log_2 p$ levels, and each

internal node in the tree has four children. For each of the levels $0, 1, 2, \ldots, \log_2 p$, starting at the leaves, the nodes in level i are boxes of height $2^i k/p$. In particular, the root node is a box of height k, and the leaves are boxes of height k/p. The key property of this tree is that each internal node's memory impact is equal to the sum of the memory impact of all its children and therefore, the sum of the *memory impacts* of the boxes at each level i is the same for every level.

The universal box profile U is constructed by performing a postorder traversal of the tree $\mathcal{T}$ (i.e., we start in the bottom left leaf, and we always visit children before visiting parents). Whenever the postorder traversal completes, it then restarts.

Analyzing the BLIND Algorithm. In Section 2.6.2, we prove the following theorem.

Theorem 1. *Using resource augmentation $\alpha = 2$, the competitive ratio of BLIND is $O(\log p)$.*

To analyze the BLIND algorithm, consider the optimal box profile OPT, which uses boxes $x_1, x_2, \ldots$, and compare it to the universal box profile U, which uses boxes $y_1, y_2, \ldots$. Let $U_{\text{prefix}} = \langle y_1, y_2, \ldots, y_j \rangle$ be the smallest prefix of U that contains OPT as a subsequence, and call a box y_i ***successfully utilized*** if it is used in the OPT subsequence.

The challenge is to bound the total memory impact of U_{prefix} by $O(\log p)$ times the total memory impact of OPT. In the rest of this overview, let us focus on only the first tree-traversal in U_{prefix}.

The key combinatorial property of the BLIND algorithm is that, for every root-to-leaf path P in the tree $\mathcal{T}$, at least one box in that path P is guaranteed to be successfully utilized. In particular, in Section 2.6.2 we show that if the path P has nodes $p_1, p_2, \ldots, p_j$ where p_j is a leaf, then once BLIND's postorder traversal reaches p_j, the next box that the algorithm successfully utilizes is guaranteed to be one of $p_1, p_2, \ldots, p_j$.

The tree $\mathcal{T}$ is designed so that each box b has exactly the same memory impact as the sum of its descendant leaves. Since every root-to-leaf path contains at least one successfully utilized box, it follows that: the sum of the memory impacts of successfully utilized boxes is at least as large as the sum of memory impacts of *all* leaves.

By design, the sum of the memory impacts of the leaves in $\mathcal{T}$ is $1/(1 + \log p)$ of the total memory impact of all boxes in $\mathcal{T}$. The consequence is that, the memory impacts of successfully utilized boxes must represent at least a $1/(1 + \log p)$ fraction of U_{prefix}'s memory impacts, as desired.

It is interesting that the BLIND algorithm achieves a competitive ratio of $O(\log p)$ while being oblivious to the input sequence σ. On the other hand, the fact that BLIND is oblivious gives hope that an even smaller competitive ratio might be achievable by an adaptive algorithm. Remarkably, this turns out not to be the case.

Lower-Bound Construction: Go Against the Flow. In Section 2.6.1, we prove the following:

Theorem 2. *Suppose $s \geq p^{1/c}$ for some constant c. Consider the green paging problem with maximum box-height k and minimum box-height k/p. Let ALG be any deterministic online algorithm for green paging, and let α be the amount of resource augmentation. Then, the competitive ratio of ALG is $\Omega\left(\frac{\log p}{\alpha}\right)$.*

For simplicity, here we focus on the case where the resource augmentation $\alpha = 1$, where $s \geq p$, and where the minimum box-height for OPT is normalized to 1 (meaning that $k = p$).

Consider an request sequence σ for ALG in which, at every step we request the element i most-recently evicted from ALG's cache. (In particular, ALG misses on every request.) Let c_{ALG} be the total memory impact of ALG on the sequence σ.

We now design (offline based on σ) an algorithm OFF that achieves total memory impact $c_{\text{OFF}} \leq O(c_{\text{ALG}}/\log p)$.

The algorithm OFF selects a threshold 2^j and always does the *opposite* of what ALG does with respect to that threshold. Namely, whenever ALG has cache-size 2^j or greater (we call these time intervals ***islands***), OFF sets its cache-size to 1. And whenever ALG has cache-size less than 2^j (we call these time intervals ***seas***), OFF sets its cache-size to 2^j.

For now, the only constraint that we will place on the threshold 2^j is that $\log p \leq 2^j \leq k/\log p$. Later, however, we will see that a careful selection of 2^j is essential to complete the proof.

Analyzing the Lower-Bound Construction. In order to analyze c_{OFF}, we begin by considering the islands. During these time intervals, both ALG and OFF miss on every request, but ALG uses a cache of size $2^j \geq \log p$ whereas OFF uses a cache of size 1. Thus the memory impact of ALG is at least a factor of $\log p$ larger than that of OFF during the islands.

Next, we consider the seas. Each of these time intervals has a ***transition cost*** for OFF, in which OFF must transition from a cache of size 1 to a cache of size 2^j, and incurs $2^j - 1$ misses in order to fill its cache. If we ignore the transition costs for a moment, then it turns out that OFF never incurs any cache misses within a sea. This is because the request sequence σ is designed to only have memory footprint $\leq 2^j$ within a given sea. Since ALG always misses, each request costs ALG at least s. On the other hand, since OFF never misses (but uses a cache of size 2^j), each request costs OFF $2^j \leq k/\log p \leq s/\log p$. Again we have that the memory impact of ALG is at least a factor of $\log p$ larger than that of OFF.

If the threshold size 2^j is not selected carefully, then the transition costs can end up dominating the other costs in OFF. Indeed, if not for the transition costs, one could select the threshold 2^j to be $\sqrt{p}$ and force a competitive ratio of $\Omega(\sqrt{p})$ (rather than $\Omega(\log p)$).

In order to minimize the transition costs, one must select the threshold size 2^j in a special way. We select 2^j to be the box-height in the range $[\log p, k/\log p]$ that contributes the least total memory impact to ALG over all such box heights. That is, for $i \in \{0, \ldots, \log k\}$ define $S(2^i)$ to be the total memory impact incurred by boxes of height 2^i in ALG, and define $j = \arg\min_{2^i=\log p}^{k/\log p} S(2^i)$. Since $\sum_i S(2^i) = c_{\text{ALG}}$, one can deduce that $S(2^j) \leq O(c_{\text{ALG}}/\log p)$.

To complete the proof, we show that the sum of the transition costs is $O(S(2^j))$. By the smooth-growth property, between every sea and island, ALG always has at least one box of height 2^j. The cost of this box for ALG is within a constant factor of the corresponding transition cost for OPT. This establishes that the total of the transition costs is $O(S(2^j)) \leq c_{\text{ALG}}/\log p$, as desired.

2.2.3 Using Green Paging to Solve Parallel Paging

In Section 2.5.1, we prove the following theorem:

Theorem 3. *Given an online algorithm for green paging with competitive ratio β, one can construct an online algorithm for parallel paging with competitive ratio $O(\beta)$ for average completion time. Moreover, if the green paging algorithm uses α resource augmentation, then the parallel paging algorithm uses $O(\alpha)$ resource augmentation.*

We also extend our analysis to show that the same algorithm achieves guarantees on the completion time for a *given number* of processors. This provides a continuous tradeoff between average-completion-time type guarantees and makespan-type guarantees, and allows for us to obtain guarantees for metrics such as *median* completion time.

Theorem 4. *Given an online algorithm for green paging with competitive ratio β, one can construct an online algorithm for parallel paging that achieves the following guarantee: For any $i \in \mathbb{N}$, the maximum completion time for all but a fraction 2^{-i} of all sequences is within a factor of $O(i\beta)$ of the optimal time to complete all but a fraction of 2^{-i-1}. Moreover, if the green paging algorithm uses α resource augmentation, then the parallel paging algorithm uses $O(\alpha)$ resource augmentation.*

Two important special cases of Theorem 4 are: the makespan is within a factor of $O(\beta \log p)$ of optimal, and the median completion is within a factor of $O(\beta)$ of the optimal time to complete at least $3/4$ of the processors.

In the rest of this section, we give an overview of the proof of Theorem 3, meaning that our focus is on minimizing average completion time. For ease of notation, we will do the reduction in the special case where $\beta = 1$ (meaning we are given an optimal algorithm for green paging) and $\alpha = \Theta(1)$. In the following discussion, let OPT denote the optimal solution to the parallel paging problem (for minimizing average-completion time).

A Warmup: The Box-Packing Algorithm. We begin by describing a simple parallel-paging algorithm which we call the Box-Packing algorithm. The Box-Packing algorithm behaves well on a certain special class of inputs (which we call uniform-impact), but will not guarantee small average completion times in general. Later in the section, we will use the algorithm as a building block to construct a different algorithm that does offer general guarantees.

During the Box-Packing algorithm, each processor runs an instance of green paging in order to produce a sequence of boxes with heights between k/p and $k/2$. The Box-Packing algorithm then greedily packs these boxes into a memory of size k over time using the following approach: if at any point in time, less than $k/2$ of the memory is being used, then the Box-Packing algorithm selects a processor q that is not currently executing (if such a processor exists), and places the next box for that processor into the cache.

An important feature of the Box-Packing algorithm is that, when picking which processor q to allocate space for in the cache, the algorithm performs a form of load balancing. Rather than giving priority to the processors q that have run for the least total time (which might seem natural), the algorithm instead selects the processor q that has incurred the *smallest total memory impact* so far, out of the processors that are idle. We call this the ***impact-balancing property***.

When the Box-Packing Algorithm Does Well: Uniform-Impact Processors. Despite not being an optimal algorithm for parallel paging in general, the Box-Packing algorithm does do well in one important special case: the case where the processors are ***uniform-impact***.

For each processor q, let I_q denote the total memory impact used by the green-paging solution for q. We call the processors $1, 2, \ldots, p$ ***uniform-impact*** if $I_1 = I_2 = \cdots = I_p = I$ for some I.

The fact that the processors are uniform-impact ensures that the cache is always close to fully utilized. In particular, a critical failure mode for the Box-Packing algorithm is if the size-k memory is under-utilized (i.e., less than $k/2$ of the memory is allotted to boxes), but there are no remaining processors to schedule (because too many processors have already finished). If

the processors are uniform-impact, however, then the Box-Packing algorithm finishes all of the processors at roughly the same time, avoiding the under-utilization failure mode.

Because the memory is close to fully utilized, the total running time is equal to the total memory impact of all processors divided by k, i.e., $\Theta(pI/k)$.

On the other hand, assuming that the processors are impact-balanced, we can show that the optimal average completion time is also $\Theta(pI/k)$. In particular, in OPT, the $p/2$ processors that finish first must together incur total memory impact at least $\Omega(pI)$,[3] thereby requiring time at least $\Omega(pI/k)$. Thus the $p/2$ processors that finish last in OPT each incur running times $\Omega(pI/k)$.

The Final Algorithm: The Phased Box-Packing Algorithm. In order to do well when the processors are not uniform-impact, we introduce the Phased Box-Packing algorithm. This is the algorithm that gives the guarantees in Theorems 3 and 4.

The algorithm consists of $1 + \log p$ phases $0, 1, 2, \ldots, \log p$, where phase i begins at the first point in time where only $p/2^i$ processors remain. During each phase i, the Phased Box-Packing algorithm runs an instance of the Box-Packing algorithm on the $p/2^i$ processors that remain, and then terminates that instance prematurely at the end of the phase. Let ΔT_i be the running time of the i-th phase, and let S_i denote the set of processors that finish during the i-th phase.

Analyzing the Phased Box-Packing Algorithm by Comparing Phases. The key to performing a competitive analysis of the algorithm is to analyze each phase based not on the average completion time of the processors S_i that finish in *that* phase, but instead based on the average completion time of the processors S_{i+1} that finish in the next phase. Note that the processors S_{i+1} represent a $1/4$ fraction of the processors that execute during phase i. By the impact-balancing property, it follows that the processors S_{i+1} incur a constant fraction of the memory impact that is incurred in phase i, and that all of the processors in S_{i+1} incur (almost) the same memory impacts as one-another in phase i. This means that, by ignoring the other processors that execute during phase i, one can treat the processors in S_{i+1} as being uniform-impact, and conclude that the average completion time in OPT for the processors in S_{i+1} is $\Omega(\Delta T_i)$.

Phase i contributes running time at most ΔT_i to at most $2|S_{i+1}|$ processors. On the other hand, the processors S_{i+1} require average running time at least $\Omega(\Delta T_i)$. Thus we can perform a charging argument where we use the running time of processors S_{i+1} in OPT to pay for the running times of all processors in phase i. This proves that the algorithm is $O(1)$-competitive with OPT.

2.2.4 Transforming Green Paging Lower Bounds into Parallel Paging Lower Bounds

We now consider how to transform an arbitrary lower-bound construction for green paging into a matching lower-bound construction for parallel paging. Section 2.5.2 proves the following theorem.

[3]Without loss of generality, we assume that OPT always allocates at least $\Omega(k/p)$ space to each processor, since up to a factor of 2 in resource augmentation we can feel free to spread half of OPT's memory equally among the processors and limit OPT's control to the other half. Note that, without this minimum-allocation height assumption, the following problem might arise: OPT could use boxes of height $o(k/p)$, possibly achieving memory impact less than I for some processor. The minimum-allocation height assumption fixes this by ensuring that all of OPT's boxes have height $\Omega(k/p)$.

Theorem 5. *Suppose there exists a green paging lower bound construction $\mathcal{L}$ that achieves competitive ratio $\Omega(\beta)$. Then all deterministic parallel paging algorithms (that use $\alpha \leq O(1)$ resource augmentation) must incur competitive ratio $\Omega(\beta)$ for both average-completion time and makespan.*

Consider a deterministic parallel paging algorithm A that has resource augmentation $\alpha = \Theta(1)$. We can assume without loss of generality that A always allocates space at least $k/(2p)$ to every processor. In particular, these minimum allocations combine to only use half of the memory, which up to a constant factor in resource augmentation can be ignored.

As A executes the p processors on their request sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$ (which we will define in a moment), each processor's request sequence σ_i is executed with some memory profile m_i. Since m_i always allocates between $k/(2p)$ and k memory, one can think of m_i as being a green-paging solution for sequence σ_i.

To construct adversarial sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$ for A, we use the lower-bound construction $\mathcal{L}$ to construct each of the sequences in parallel. We terminate each of the sequences σ_i once the corresponding memory profile m_i produced by A reaches some large memory impact R.

The fact that each of the profiles m_i have the same memory impacts R allows for us to lower bound the average completion time for algorithm A. In particular, the first $p/2$ processors to complete must incur total memory impact at least $\Omega(pR)$, thereby incurring total running time at least $\Omega(pR/k)$. It follows that the final $p/2$ processors to complete each take time more than $\Omega(pR/k)$. Thus $\Omega(pR/k)$ is a lower bound for both the average completion time and the makespan of A.

In order to complete the proof, we construct an alternative parallel-paging solution B that has makespan (and thus also average completion time) only $O(pR/k\beta)$. Note that, because algorithm A is analyzed with resource augmentation α, the maximum memory size for B is k/α.

Now we consider the optimal green-paging solution for each request sequence σ_i, where the optimal solution is restricted to have minimum box height $k/(4\alpha p)$ and maximum box height $k/(2\alpha)$ (i.e., the optimal solution is limited by a factor-of-2α resource augmentation in comparison to the solutions produced by A). Let m_i^{OPT} be the (boxed) memory profile produced by the optimal green-paging solution for σ_i. By the definition of the lower-bound construction $\mathcal{L}$, we know that the memory impact of each m_i^{OPT} is only $O(R/\beta)$.

To construct the parallel-paging solution B, we simply perform the Box-Packing algorithm from Section 2.2.3 on the box profiles $m_1^{\text{OPT}}, \ldots, m_p^{\text{OPT}}$. In particular, whenever the total memory allocated to processors is less than $\frac{k}{2\alpha}$, algorithm B selects a processor i out of those not currently executing (if there is one) and allocates space for the next box the profile m_i^{OPT}. Note that the box is guaranteed to fit into B's memory of size k/α, since the maximum box height in any profile m_i^{OPT} is only $k/(2\alpha)$.

A simple way to analyze the average running time of B is to note that (without loss of generality) the request sequences $m_1^{\text{OPT}}, m_2^{\text{OPT}}, \ldots, m_p^{\text{OPT}}$ are impact balanced, each having the same memory impact $I = \Theta(R/\beta)$. By the analysis in Section 2.2.3, the total makespan for B is only $O(pI/k) = O\left(Rp/k\beta\right)$. Since this is a factor of $\Omega(\beta)$ smaller than the average completion time and makespan of A, this completes the lower-bound transformation.

2.2.5 Putting Pieces Together

Combining the upper and lower bounds for green paging in Section 2.2.2, we arrive at the following tight bound on green paging.

Theorem 6. *Suppose $s \geq p^{1/c}$ for some constant c. Consider the green paging problem with maximum box-height k and minimum box-height k/p. Then there exists a deterministic online algorithm (with $O(1)$ resource augmentation) that achieves competitive ratio $O(\log p)$. Moreover, this competitive ratio is asymptotically optimal for deterministic online algorithms.*

Note that the assumption in Theorem 6 that $s \geq p^{1/c}$ is natural for the following reason: if any green paging algorithm ever uses a square of height more than sk/p, then the algorithm would be better off using a square of height k/p (and just incurring cache misses everywhere). Thus the natural parameter regime for green paging is when the maximum box-height k is less than sk/p, meaning that $p \leq s$.

By combining the upper and lower bounds for green paging with the reductions in Sections 2.2.3 and 2.2.4, we arrive at the following bounds for parallel paging:

Theorem 7. *There exists an online deterministic algorithm for parallel paging that achieves competitive ratio $O(\log p)$ for average completion time, using resource augmentation $O(1)$. Moreover, for any $i \in \mathbb{N}$, the maximum completion time for all but a fraction 2^{-i} of all sequences is within a factor of $O(i \log p)$ of the optimal time to complete all but a fraction of 2^{-i-1}. One consequence of this is that a competitive ratio of $O(\log^2 p)$ is achieved for makespan. Furthermore, any deterministic parallel paging algorithm must have competitive ratio $\Omega(\log p)$ for both average completion time and makespan, as long as $s \geq p^{1/c}$ for some constant c.*

2.3 The Models

Before proceeding with our results, we first take a moment to discuss the models for green paging and parallel paging in detail and, in particular, to highlight some of the unintuitive differences between these problems and classic paging.

2.3.1 The Green Paging Model

Green paging models computations in environments, such as cloud computing or low-power computing, where the amount of memory resources allocated to a given task can be changed dynamically, and the objective is to complete the task minimizing the total amount of memory resources consumed over time.

Formally, in green paging, like in standard paging, an algorithm controls a memory system with two layers: a fast memory that can hold a limited number of pages, and a slow memory of infinite capacity. Accessing a page takes one time step if that page is in fast memory. If a requested page is not currently in fast memory, it can be accessed only after copying it into fast memory; fetching a page from slow to fast memory takes $s \geq 1$ time steps. During the s time steps of a fault, the corresponding fast memory page must be available and not otherwise in use. In practice, $s \gg 1$; we assume for simplicity that s is an integer, allowing us to work with discrete time steps. Pages can be brought into fast memory only when requested, but can be discarded at any time, instantaneously and at no cost. We denote by $P_{ALG}(i)$ the set of pages kept in (or being loaded into) fast memory throughout the i-th time step by a paging algorithm ALG.

Two main differences set green paging apart from classic paging. The first is that the fast memory capacity is not fixed, but in general varies over time *under the control of the paging algorithm*,

between a maximum of k and a minimum of k/p pages. We denote by $m_{ALG}(i) \geq |P_{ALG}(i)|$ the fast memory capacity set by ALG throughout the i-th time step. We call the function $m(i)$ the ***memory profile***.[4] The second difference is that the paging algorithm must access any given sequence of page requests $\sigma = \langle r_1, \ldots, r_n \rangle$ minimizing not the total time taken $T_{ALG}(\sigma)$ (or, equivalently, the number of faults) but instead the integral, over that time interval, of the fast memory capacity, that is, $\sum_{i=0}^{T_{ALG}(\sigma)} m_{ALG}(i)$. We call this quantity the ***memory impact*** of a paging algorithm. As mentioned in Section 3.1, lower capacity does not necessarily translate into a lower memory impact, if it means more page faults and thus more processing time—think of a cycle over four pages serviced with memory capacity $m(i) = 4$ or with memory capacity $m(i) = 2$ for all i.

Solely to simplify the analysis, we also introduce a third, minor, difference: we allow the possibility of idling on any given time step following a page access (or another idle time step). The time step is counted towards the memory integral, but the request sequence does not advance and the memory contents do not change (unless $|P_{ALG}(\cdot)|$ must decrease as a consequence of a reduction of $m_{ALG}(\cdot)$). This is justified by the availability of the No OPeration (NOP) instruction in most processors.

Denote by $r_{ALG}(i)$ the request from σ serviced at time step i by a green paging algorithm ALG. Then ALG is *online* if it determines $m_{ALG}(i)$ and $P_{ALG}(i)$ based solely on $r_{ALG}(1), \ldots, r_{ALG}(i)$. Informally, we define the ***competitive ratio*** with $\alpha \geq 1$ ***resource augmentation*** by comparing the memory impact of the online algorithm with that of an optimal offline algorithm that runs on a system with α times less capacity and pays α times as much for the same capacity, i.e., one whose memory capacity during the i-th time step $m_{OPT}(i)$ lies between $\lfloor k/p\alpha \rfloor$ and $\lfloor k/\alpha \rfloor$, and that incurs a cost equal to $\sum_{i=0}^{T_{OPT}(\sigma)} \alpha \cdot m_{OPT}(i)$. In other words, the memory impact of the optimal algorithm is scaled by a factor α, so that for all j, it costs the optimal algorithm the same amount to allocate j/α memory as it does for the online algorithm to allocate j memory (i.e., allocating a *p-fraction* of memory costs the same for both algorithms). We remark that the main focus of this work is the case of $\alpha = \Theta(1)$, however, in which case this distinction is not important.

Note that in general we do not assume $k/p = 1$. This models a number of situations of practical interest. In some systems, memory can be only allocated in units significantly larger than a single block. More typically, computing systems incur other running costs (e.g., the power consumed by a motherboard or processor) that cannot be reduced below a certain threshold per unit of time if a computation is running at all, regardless of how little memory is used; in these cases k/p represents the memory capacity below which these other costs become dominant, and thus below which memory capacity reductions cannot grant significant cost reductions, per unit of time, to the system as a whole.

2.3.2 The Parallel Paging Model

Parallel paging models computations where multiple processing units, each operating concurrently and independently, share nonetheless the same fast memory—a situation that multicore

[4]This should not be confused with the memory profile function as defined in [33]. In both cases a memory profile function specifies the quantity of memory available at any given time, but (1) in [33], this quantity is set adversarially and not under the control of the algorithm, and (2) in [33] time advances with the page faults of the algorithm.

processors have made over the last two decades the standard on virtually all computing systems from supercomputers to mobile phones. As in classic paging, the goal is to choose at each point in time which pages to keep in fast memory so as to minimize some objective function of processors' completion times.

The model we adopt is essentially the one introduced by López-Ortiz and Salinger [102]. We have p processors that share the same fast memory of size k. We assume $k \geq p$. Each processor issues a sequence of page requests; hence we have p sequences of page requests $\sigma_1 = \langle r_1^1, \ldots, r_{n^1}^1 \rangle, \ldots, \sigma_p = \langle r_1^p, \ldots, r_{n^p}^p \rangle$. (Note that the number of processors is denoted by the same parameter, p, that denotes the ratio between the maximum and the minimum memory capacity in the green paging problem. This is intentional since, although the two quantities are drastically different, when showing the equivalence between green paging and parallel paging we will show that they play exactly the same role.) We assume that the sets of pages requested by different processors are disjoint, corresponding to separate processes.

Accessing a page takes one time step if that page is in fast memory (i.e., cache). If a requested page is not currently in fast memory, it can be accessed only after copying it into fast memory; fetching a page from slow to fast memory takes $s \geq 1$ time steps. During the s time steps of a fault, the corresponding fast memory page must be available and not otherwise in use. In practice, $s \gg 1$; we assume for simplicity that s is an integer, allowing us to work with discrete time steps. The goal of the paging algorithm is to choose which pages to maintain in fast memory so as to minimize the *maximum*, *average* or *median* time to service $\sigma_1, \ldots, \sigma_p$. As in green paging, we allow the possibility of idling on any given time step following a page access (or another idle time step).

The parallel paging model allows for all of the processors to make progress in parallel. That is, multiple processors can experience cache misses and cache hits concurrently. However, it may sometimes be useful for the algorithm to temporarily halt some subset of processors (i.e., those processors are simply idle) in order to make the best possible use of the fast memory (e.g., in order to fit the entire working set for some subset of processors into fast memory).

Note that the only interaction between the p processors lies in the fact that the fast memory must be partitioned (dynamically) between them; in particular, we denote by $m_{ALG}^j(i)$ the amount of fast memory allocated by a paging algorithm ALG throughout the i-th time step to the j-th processor (so that $\sum_{j=1}^p m_{ALG}^j(i) \leq k$ for all i) and by $P_{ALG}^j(i)$ the set of its pages in, or being loaded into, the fast memory at that time step. In general, note that $m_{ALG}^j(i)$ *can* be 0 if and only if the j-th processor is idle on the i-th time step. We also remark that, although the order in which each processor accesses its own page sequence is fixed, the sequences will be interleaved in different ways depending on how much memory and thus "speed" each request sequence is allocated.

Denote by $r_{ALG}^j(i)$ the request from σ_j serviced on time step i by a parallel paging algorithm ALG. Then ALG is *online* if $m_{ALG}^1(i), \ldots, m_{ALG}^p(i)$ and $P_{ALG}^1(i), \ldots, P_{ALG}^p(i)$ depend solely on $r_1^1, \ldots, r_{ALG}^1(i), \ldots, r_1^p, \ldots, r_{ALG}^p(i)$. A factor of $\alpha \geq 1$ of *resource augmentation* simply means that the optimal offline algorithm is restricted to a memory of size $\lfloor k/\alpha \rfloor$.

2.4 A Toolbox for Paging Analysis

The goal of this section is to show how imposing certain constraints on the memory allocation does not significantly degrade performance. In a nutshell, these involve rounding capacity to the next power of two, ensuring that capacity does not change "too often", and periodically lowering capacity to 0. Operating under these constraints significantly simplifies subsequent analyses.

2.4.1 Memory Expansions

Let $m_{ALG}(i)$ denote the memory capacity set by algorithm ALG throughout the i-th clock tick. In our analyses, we shall often compare two algorithms by allotting to the first "more (memory) space" or "more time". A more formal definition of this notion is that of *expansion* of a memory profile function $m(t)$—basically a new memory profile in which each clock tick with memory m is replaced by one *or more clock ticks* with memory *at least* m.

Definition 1. *Consider a memory profile function $m(t) : \mathcal{N} \to \mathcal{N}$. An* expansion *of $m(t)$ is pair of functions $\bar{t}(t) : \mathcal{N} \to \mathcal{N}$ and $\bar{m}(\bar{t}) : \mathcal{N} \to \mathcal{N}$, with the following properties:*

1. *$\bar{t}(t+1) > \bar{t}(t)$.*
2. *for each $\bar{t}$, either:*
   (a) *there exists t such that $\bar{t} = \bar{t}(t)$, and then $\bar{m}(\bar{t}) \geq m(t)$, or*
   (b) *there exist t^- and $t^+ = t^- + 1$ such that $\bar{t}(t^-) < \bar{t} < \bar{t}(t^+)$, and then $\bar{m}(\bar{t}) \geq \min(m(t^-), m(t^+))$.*

If $\forall t$ we have that $\bar{t}(t)$ and $\bar{m}(\bar{t}(t-1)+1), \ldots, \bar{m}(\bar{t}(t))$ can be determined solely on the basis of the page requests up to time t, then we say that the expansion is online.

If a paging algorithm can service a request sequence before clocktick T given $m(t)$ memory, then an *optimal offline* algorithm can obviously always service the same request sequence on any expansion of $m(t)$ before (the image of) T – essentially by "wasting" any extra memory, and idling through extra clockticks while preserving the memory contents. Note that it is not true *in general*, for all paging algorithms. Indeed, even if $m(t)$ and $\bar{m}(t)$ are constant, there are online algorithms such as FIFO that service *some* sequences *faster* if given *less* memory (a phenomenon known as Belady's anomaly [42]).

Definition 2. *The **memory impact** of a memory profile function $m(t)$ is the integral of memory (space) allotted to the profile over time, that is $\sum_t m(t)$.*

For the rest of the work, we use the memory impact and cost of a memory profile interchangeably.

2.4.2 Space and Time Normalization

To simplify memory management, we consider expansions of memory profiles $m(t)$ with some constraints. More precisely, we show how any "natural" memory profile can be expanded online

into another memory profile that satisfies some constraints and incurs a memory impact within a factor of $O(1)$ of the original profile.

With "natural" we mean having a very simple, specific property: on any tick in which allocated memory increases, it increases by at most one page and then remains constant for (at least) $s-1$ more ticks. This reflects the fact that it is pointless to increase the memory profile unless one needs to load one or more pages not currently in memory, which takes s clockticks per page. More formally:

Definition 3. *A memory profile function $m(t)$ is* natural *if, for any t such that $m(t+1) > m(t)$, $m(t+1) = \cdots = m(t+s) = m(t)+1$.*

We remark that the expansions of natural memory profiles we consider are not, in general, natural. The idea is that the expanded profiles are actually wasting space, but their definition makes them easier to handle in proofs. It would not be difficult, albeit extremely cumbersome, to consider natural expansions for which all proofs still work.

The first constraint a memory expansion must satisfy is that the allocated memory should, at any given time, be a power-of-2 multiple of the minimum allowed memory capacity k_{min}.

Definition 4. *A memory profile function $m(t)$ is* space-normalized *if, for all t, there exists some $i \in \mathcal{Z}_0^+$ such that $m(t) = 2^i k_{min}$.*

The second constraint imposes that, roughly speaking, whenever allocated memory changes to a new value m, it should remain m for a number of clockticks that is an integer multiple of $s \cdot m$. This is formally defined as follows.

Definition 5. *A memory profile function $m(t)$ is* time-normalized *if any maximal interval during which $m(\cdot)$ maintains constant value m lasts a number of clockticks equal to an integer multiple of $s \cdot m$. We call such an interval a* ***box****.*[5]

Figure 2.1 shows an example of a space- and time-normalized memory profile function. We now show how to expand online any natural memory profile function $m(t)$ into a space- and time-normalized function $\bar{m}(\bar{t})$ such that the total memory impact incurred by the latter is within a constant factor of that incurred by the former, that is, $\sum_{\bar{t}} \bar{m}(\bar{t}) = O(\sum_t m(t))$. Informally, given a memory capacity $m(t)$, we round $m(t)$ up to the nearest power-of-2 multiple of k_{min} (normalizing space) and then maintain memory constant at this normalized capacity for the minimum interval that guarantees time normalization – adding idle cycles if necessary, i.e. if $m(\cdot)$ would grow above the allotted capacity before the end of the interval.

More formally, denoting by $\bar{m}(\bar{t})$ the normalized memory profile function, we consider a set $\bar{t_0}, \ldots, \bar{t_n}$ of *upticks*, which are the only destination ticks in which $\bar{m}(\bar{t})$ *may* change (i.e. $\bar{m}(\bar{t_i}) = \cdots = \bar{m}(\bar{t_{i+1}} - 1)$), so $\bar{m}(\bar{t})$ is completely defined by its values on the upticks. We map the original clocktick 0 into $\bar{t_0} = 0$, and denote in general with t_i the original clocktick mapped into $\bar{t_i}$. Then, for any $i \geq 0$ we set $\bar{m}(\bar{t_i})$ to the lowest power-of-2 multiple of k_{min} no smaller than $m(t_i)$, i.e. $\bar{m}(\bar{t_i}) = k_{min} \cdot 2^{\lceil \log(\frac{m(t_i)}{k_{min}}) \rceil}$, and determine $\bar{t_{i+1}}$ as follows. Let t_i^+ be the earliest

[5]Recall that in the technical overview (Section 3.3), for convenience, we defined boxes to always have fixed width $\Theta(sm)$, rather than having width equal to an integer multiple of $s \cdot m$. If one breaks each box (as defined here) into multiple boxes of the form described in Section 3.3, then one can assume without loss of generality that all boxes have the form described in Section 3.3.

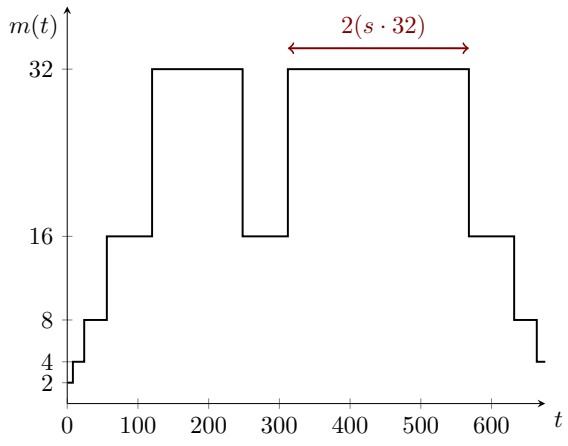


Figure 2.1: A space- and time-normalized memory profile.

original clocktick, if any, after t_i such that $m(t_i^+) > \bar{m}(\bar{t}_i)$. If there is no such t_i^+ equal to or smaller than $t_i + s \cdot \bar{m}(\bar{t}_i)$, then either we are at the very end of the request sequence, or the original memory profile remains below $\bar{m}(\bar{t}_i)$ for at least $s \cdot \bar{m}(\bar{t}_i)$ ticks – in the latter case we simply map $t_i, \ldots, t_i + s \cdot \bar{m}(\bar{t}_i)$ into $\bar{t}_i, \ldots, \bar{t}_i + s \cdot \bar{m}(\bar{t}_i)$, and set $\bar{t}_{i+1} = \bar{t}_i + s \cdot \bar{m}(\bar{t}_i)$ and thus $t_{i+1} = t_i + s \cdot \bar{m}(\bar{t}_i)$). If we are at the very end of the request sequence, we simply imagine there is a sharp rise of memory consumption after it, and we set t_i^+ to be the first clocktick *after the completion* of the request sequence with $m(t_i^+) = \infty$. Whether this is the case, or whether t_i^+ "occurred naturally", we map $t_i, \ldots, t_i^+ - 1$ into $\bar{t}_i, \ldots, \bar{t}_i + (t_i^+ - t_i) - 1$, as above, and map the single clocktick t_i^+ into the interval $\bar{t}_i + (t_i^+ - t_i), \ldots, \bar{t}_i + s \cdot \bar{m}(\bar{t}_i)$, with all clockticks in the interval save the last having memory $\bar{m}(\bar{t}_i)$ (thus achieving time-normalization). The last clocktick becomes the first clocktick $\bar{t}_{i+1}$ of the $(i+1)$-th interval, and is allotted memory equal to the lowest-power-2 multiple of k_{min} that is no smaller than $m(t_i^+)$.

By construction, the space and time normalization does not increase the memory integral of the original profile by more than a constant factor.

2.4.3 Compartmentalization

An additional useful transformation we can apply to a (space- and time-normalized) memory profile, and implicitly to a generic page replacement algorithm, is *compartmentalization*. Compartmentalization expands the memory profile by adding to each box a prefix of duration equal to s times the capacity of that box, and a suffix of identical duration at the end of which all pages are evicted from memory. Note that compartmentalization can always be carried out online.

We can easily show that if an optimal page replacement algorithm services a request sequence over a sequence of boxes, it does the same even if the boxes are compartmentalized. Very simply, one can reload at the beginning of the box any pages in memory, or being loaded in memory, in the absence of compartmentalization; at s ticks per page, this takes no longer than the added prefix. Note that the original algorithm could have started the box with a pending fault, terminating after $s' \leq s$ ticks. If so, we simply idle for s' ticks, and then reproduce the original page

replacement, ending with the memory exactly in the same state (including any pending fault). Then, we allow any pending fault to terminate (which takes at most s ticks), and idle through the rest of the suffix. The page replacement strategy above is not necessarily optimal, but it obviously provides an upper bound on the memory impact of the optimal strategy in the presence of compartmentalization. We then immediately have:

Proposition 1. *Under optimal replacement policy, compartmentalization increases the memory impact of a time-normalized memory profile by a factor of at most three.*

We stress that the above holds for the optimal replacement policy, not necessarily for *any* policy—in fact, it is not difficult to show that it does not hold for some online algorithms. But ultimately, we only care about bounds under optimal page replacement, because we know we can translate them into bounds under e.g. LRU with only $O(1)$-overhead (in terms of memory impact and resource augmentation).

The usefulness of compartmentalization lies in the fact that we can subsequently add between (expanded) boxes stretches of arbitrarily little memory without hindrance—we do so in Section 2.5.1. Note that such an addition is not necessarily an expansion, because the added intervals could sport lower capacity than both the boxes preceding them and those following them—so they could cause a loss of memory contents and a degradation of performance if they were added without compartmentalization. This is not the case with compartmentalization, since the memory contents are cleared at the end of each box anyways. Compartmentalization is a way to add these extra stretches knowing that they will at most triple the memory impact of a time-normalized memory profile. Compartmentalization also plays a crucial role in circumventing a subtle issue encountered in Section 2.6.2 that arises from the unexpected power of idling around the discontinuities of a memory profile function.

2.5 The Tight Relationship between Green Paging and Parallel Paging

Green paging and parallel paging appear to be two wildly different variants of paging, both in terms of system model (sequential processing with variable memory vs. parallel processing with constant memory) and in terms of goals (minimizing memory consumption over time vs. minimizing completion time).

In this section, we describe a black-box approach for turning online algorithms for green paging into online algorithms for parallel paging with the same competitive ratio. We then show a relationship in the other direction, which is that any lower bound construction for online green paging can be transformed in a black-box fashion into an online lower-bound construction for (deterministic) parallel paging.

For simplicity, we shall hereafter assume that p is a power of two. The same results can be obtained, with some additional complications, assuming that p is just an integer, substituting $\lceil \log p \rceil$ for $\log p$ in all formulas. All logarithms in this work are to base 2.

2.5.1 Transforming Green Paging Algorithms into Parallel Paging Algorithms

The key idea to translate an efficient algorithm for green paging algorithm into an efficient one for parallel paging is simply to generate independently an efficient green paging strategy for each of the p parallel sequences, and "pack" (expansions of) the memory profiles together into the shared memory, guaranteeing that (a) as little memory as possible is wasted in the process, including the expansion, and (b) at any given time, the integral of the memory in each profile so far is roughly the same. We remark that even though the use of normalization results in memory profiles that are sequences of "rectangular" boxes of space-time (to be packed within a large rectangle of space-time, with height equal to the amount of shared memory k, and width equal to the time taken to service the sequences), one cannot exploit standard 2D packing results: one cannot treat each profile as a single enveloping rectangle without potentially wasting $\Theta(k)$ memory, but one cannot treat each profile as a *set* of rectangles to be packed independently either, since the individual boxes of each profile must obviously be placed after each other in a specific order and cannot overlap temporally.

We now describe this packing process. We assume the p green paging memory profiles we start with are already compartmentalized and space- and time-normalized. Effectively, given the partition of each memory profile into boxes, at any given time step we execute (one box of) one or more memory profiles in parallel, adding an idle cycle at capacity zero to all the other memory profiles. Note that different consecutive boxes of the same profile potentially take place in non-contiguous intervals of time.

At any given time, we assign highest priority to the profile without any allocated memory—i.e., the processor is idle or has just completed a box—that has incurred, so far, the lowest total memory impact (with ties broken arbitrarily); any memory page that becomes available is then *reserved* for that profile, until enough pages become available that its next box can be scheduled. No other profile with no allocated memory can be executed before, not even if enough room in memory is available. Clearly, priorities change dynamically, so if a processor q that just completed a box has currently the highest priority, then the space in memory is reserved for q. (Put another way: at any time step, if two processors p and q have no allocated memory and q has had a lower memory impact than p so far, then p does not get to run unless and until q gets (enough memory) to run.) Note that this process is online, since each box is scheduled without any knowledge of future boxes.

We now show that this process completes within $3c_{max}/k_{min}$ clockticks, where c_{max} is the maximum cost of any profile.

Lemma 1. *Consider p (space- and time-normalized, and compartmentalized) green paging strategies for p request sequences $\sigma_1, \ldots, \sigma_p$ with memory capacity between k and $k_{min} = k/p$, and let c_{max} be the maximum memory impact of any profile. Then, they can be packed online to yield a parallel paging strategy that completes within time $3c_{max}/k_{min}$.*

This immediately yields the following.

Theorem 8. *Consider p (space- and time-normalized, and compartmentalized) green paging strategies for p request sequences $\sigma_1, \ldots, \sigma_p$ with memory capacity between k and $k_{min} = k/p$, and assume that each is optimal with α resource augmentation within a factor of β, and that their respective*

costs are within a factor of γ of each other. Then they can be packed to yield a parallel paging strategy that completes, with α resource augmentation, within time $6\beta\gamma$ of any parallel paging strategy for the p sequences. Furthermore, if the green paging is online, so is the parallel paging.

Proof. Let c_{min} be the minimum cost of any of the p green paging strategies, and $c_{max} \leq \gamma c_{min}$ be the maximum. Then, by Lemma 1, the p green paging strategies can be packed online into a single parallel paging strategy that completes within time $T = 3\gamma c_{min}/k_{min}$. Suppose there were a parallel paging strategy that completed all p sequences in time less than $T/6\beta\gamma$. This would automatically yield a green paging strategy for each of the p sequences, with memory capacity between k and 0 (note, not k_{min}). The least expensive of these strategies would then have cost less than $(k/p) \cdot (T/6\beta\gamma)$; raising the memory capacity to k_{min} whenever lower would increase the cost to less than

$$\frac{k}{p} \cdot \frac{T}{6\beta\gamma} + k_{min} \cdot \frac{T}{6\beta\gamma} = \frac{2k_{min}T}{6\beta\gamma} = \frac{2k_{min}}{6\beta\gamma} \cdot \frac{3\gamma c_{min}}{k_{min}} = \frac{c_{min}}{\beta},$$

against the hypothesis that the p strategies were all optimal within a factor of β for green paging with memory capacity between k and $k_{min} = k/p$. □

Analyzing Max, Average, and Median

Theorem 8 proves that, if one can come up with a green paging algorithm that is optimal within a factor of c with a certain resource augmentation, one can automatically obtain parallel paging *with the same resource augmentation* that is optimal within a factor of $O(c)$ *provided that the different sequences one is servicing incur green paging costs within a constant factor of each other.* In this case every sequence takes $\Theta(1)$ the time any other sequence takes to complete, and it is straightforward to prove that the median and average completion times are also within a factor of $O(c)$ of the optimal.

It is then also immediate to extend the analysis to a situation where one has p sequences of infinite length, and one is seeking a parallel paging algorithm ALG that minimizes, *simultaneously for every memory integral* w, the time necessary to complete all of the prefixes of the p sequences that could be completed, each in isolation, with a memory integral equal to at most w. The same strategy also obviously minimizes, within a constant factor, the average time for all these prefixes, and the median time.

The situation is different if one has *finite* sequences of drastically different cost. In this case, simply packing the corresponding green paging sequences can be a provably suboptimal choice for parallel paging, because one might be left after some time with only a few sequences uncompleted, whose green paging allocations are consuming only very little memory and "wasting" the rest—instead of using it in a way that *increases* the memory integral (potentially by $poly(p)$, as it would be easy to prove), but correspondingly *decreases* the total completion time (again potentially by $poly(p)$, as it would be easy to prove).

In this case, we show that a simple variation on the scheme above that starts with a green paging algorithm with resource augmentation α and optimal within a factor of β, that is still online if the starting scheme is online, and provides a parallel paging strategy that, still with resource augmentation α, yields:

1. An average completion time that is optimal within a factor of $O(\beta)$.

2. For any $i \in \mathbb{N}$, a maximum completion time for all but a fraction 2^{-i} of all sequences that is within a factor of $O(i\beta)$ of the optimal time to complete all but a fraction 2^{-i-1}; this means:

   (a) a maximum completion time within a factor of $O(\beta \log p)$ of the optimal, and

   (b) a median completion time (intended as the time to complete at least $\frac{1}{2}p$ schedules) within a factor of $O(\beta)$ of the optimal time to complete at least $\frac{3}{4}p$ schedules.

It's not too difficult to show that, in terms of the inverse of the fraction of uncompleted sequences 2^i a) the logarithmic loss in time and b) the $O(1)$ "quantile augmentation" are both fundamentally inevitable in an online setting. So we can't do better. More precisely, quantile augmentation is necessary if you consider a set of sequences of exponentially growing length $2^p, (2^p)^2, \ldots, (2^p)^p$—basically because by the time you need only one more sequence completed to meet your quantile target, the work you've done so far is vanishingly small compared to what you still have to do, but because you do not know which sequence you have to focus on, you have to split your work evenly among all survivors, meaning that crucial sequence will receive only little space and take more time than necessary.

The scheme is simply to pack together the green paging strategies with memory between k and k/p for the p sequences, until $p/2$ of them have completed; then, pack together green paging strategies with memory between k and $\frac{k}{p/2}$ for the suffixes of the remaining $p/2$ sequences, until $p/4$ have completed; and so on, packing at each stage with $p/2^i$ "survivors" the green paging strategies with memory between k and $\frac{k}{p/2^i}$ for those survivors, until all sequences have completed. Recall that in each stage, we pack the next box of a processor that has the highest priority (whose memory impact is minimum so far). This is obviously an online strategy as long as the green paging strategies it is based on are online.

In the scheme above denote by T_i, with $i = 0, \ldots, \log(p)+1$, the first time when no more than $\frac{p}{2^i}$ sequences remain uncompleted, and by σ_{p+1-j}, with $1 \leq j \leq p$, the j-th sequence to complete – so that σ_1 completes *last*, σ_p completes *first*, and all of $\sigma_{2^i}, \ldots, \sigma_{2^{i-1}+1}$ complete between $T_{\lg(p)-i}$ and $T_{\lg(p)-i+1}$. The key idea is that $\sigma_{2^i}, \ldots, \sigma_{2^{i-1}+1}$ remain all uncompleted at time $T_{\lg(p)-i}$, but constitute at least one quarter of those sequences uncompleted at time $T_{\lg(p)-i-1}$, so that they "occupy" at least one quarter of the "memory space" in the interval $[T_{\lg(p)-i-1}, T_{\lg(p)-i}]$ – and if their memory occupation is optimal within a factor of β, writing for brevity $(T_j - T_{j-1})$ as ΔT_j, it would then be impossible to complete them all in time less than $\frac{1}{4\beta}\Delta T_{\lg(p)-i}$. In fact, σ_1 also occupies the *entire* memory space throughout the last interval $[T_{\lg(k)}, T_{\lg(p)+1}]$ (in addition to at least one quarter – in fact, at least one half – of the space in the interval $[T_{\lg(k)-1}, T_{\lg(p)}]$), and so it would be impossible to complete in time less than $\frac{1}{4\beta}\Delta T_{\lg(p)} + \frac{1}{\beta}\Delta T_{\lg(p)+1}$.

Then under *any* algorithm the average completion time of the p sequences is at least:

$$T_{avg} = \frac{1}{\beta}\Delta T_{\lg(p)+1} + \sum_{i=1}^{\lg(p)} \frac{1}{4\beta}\frac{p}{2^i}\Delta T_i > \sum_{i=0}^{\lg(p)} \frac{1}{4\beta}\frac{p}{2^{i+1}}\Delta T_{i+1} \tag{2.1}$$

while under the scheme above the average completion time is no more than:

$$\sum_{i=0}^{\lg(p)} \frac{p}{2^i}\Delta T_{i+1} < 8\beta T_{avg}. \tag{2.2}$$

Thus we have the following theorem:

Theorem 3. *Given an online algorithm for green paging with competitive ratio β, one can construct an online algorithm for parallel paging with competitive ratio $O(\beta)$ for average completion time. Moreover, if the green paging algorithm uses α resource augmentation, then the parallel paging algorithm uses $O(\alpha)$ resource augmentation.*

In a similar fashion, note that to complete all but a fraction $2^{-(i+1)}$ of *all* p sequences, any algorithm must complete for any $j \geq \lg p - i$ all but at most $2^{-(i+1)}p$ of $\sigma_1, \ldots, \sigma_{2^j}$, thus requiring time at least $\frac{1}{4\beta}\Delta T_{\lg p - j}$, since *each* of the 2^j sequences requires at least $\frac{1}{2^j\beta}\Delta T_{\lg p - j}$ space. Then, the total time $T_{2^{-(i+1)}}$ for any algorithm to complete all but a fraction $2^{-(i+1)}$ of all sequences satisfies

$$T_{2^{-(i+1)}} \geq \max_{j \leq i} \frac{1}{4\beta}\Delta T_{\lg p - j} \geq \frac{1}{4\beta i}\sum_{j=0}^{i} \Delta T_{\lg p - j}, \tag{2.3}$$

where the last term is no more than $\frac{1}{4\beta i}$ the time our scheme takes to complete all but a fraction 2^{-i} of all sequences.

Thus we arrive at the following theorem:

Theorem 4. *Given an online algorithm for green paging with competitive ratio β, one can construct an online algorithm for parallel paging that achieves the following guarantee: For any $i \in \mathbb{N}$, the maximum completion time for all but a fraction 2^{-i} of all sequences is within a factor of $O(i\beta)$ of the optimal time to complete all but a fraction of 2^{-i-1}. Moreover, if the green paging algorithm uses α resource augmentation, then the parallel paging algorithm uses $O(\alpha)$ resource augmentation.*

2.5.2 Transforming Green Paging Lower Bounds into Parallel Paging Lower Bounds

In this section, we consider the problem of transforming an arbitrary lower-bound construction for green paging into a matching lower-bound construction for parallel paging. Throughout the rest of the section, k denotes the maximum amount of memory that can be allocated in green paging, and $k/(2p)$ the minimum amount. We also use k to represent the amount of memory available in parallel paging, and p to be the number of processors.

Defining the notion of lower-bound construction for green paging We begin by formally defining the notion of a ***green paging lower-bound algorithm*** $\mathcal{L}$. A lower-bound algorithm $\mathcal{L}$ takes as input a deterministic online green-paging algorithm A (that uses $O(1)$ resource augmentation), and produces a request sequence $\sigma(A)$ on which A performs poorly. The way in which the lower-bound algorithm $\mathcal{L}$ and the green paging algorithm A interact is that the i-th request in $\sigma(A)$ is determined based on the behavior of algorithm A while serving the first $(i-1)$ requests in $\sigma(A)$.

Each lower-bound algorithm $\mathcal{L}$ must have a ***termination size*** R. This means that the request sequence $\sigma(A)$ terminates once the total memory impact incurred by A reaches R. Note that R is independent of the algorithm A.

A lower-bound algorithm $\mathcal{L}$ is said to ***achieve competitive ratio β*** if every green-paging algorithm A with $O(1)$ resource augmentation incurs a factor of $\Omega(\beta)$ more memory impact on

$\sigma(A)$ than does the optimal green-paging algorithm[6]. That is, the optimal green paging algorithm incurs memory impact only $O(R/\beta)$.

We prove the following theorem.

Theorem 5. *Suppose there exists a green paging lower bound construction $\mathcal{L}$ that achieves competitive ratio $\Omega(\beta)$. Then all deterministic parallel paging algorithms (that use $\alpha \leq O(1)$ resource augmentation) must incur competitive ratio $\Omega(\beta)$ for both average-completion time and makespan.*

Proof. Let A be a deterministic algorithm for parallel paging that uses resource augmentation $\alpha = O(1)$. We can assume without loss of generality that A always allocates space at least $k/(2p)$ to every processor. In particular, these minimum allocations combine to only use half of the memory, which up to a constant factor in resource augmentation can be ignored.[7]

As A executes the p processors on their request sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$, each processor's request sequence σ_i is executed with some memory profile m_i. Since m_i always allocates between $k/(2p)$ and k memory, one can think of m_i as being a green-paging solution for sequence σ_i.

Given a lower-bound algorithm $\mathcal{L}$, we construct each of the request-sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$ by running parallel instances of $\mathcal{L}$, with resource augmentation 2α. The result is that each memory profile m_i incurs total memory impact R (where R is the termination size of $\mathcal{L}$ and is assumed without loss of generality to be sufficiently large). Moreover, if m_i^{OPT} is defined to be the memory profile that the optimal green-paging solution uses for request-sequence σ_i, then the total memory impact incurred by m_i^{OPT} is $O(R/\beta)$. Without loss of generality, each profile m_i^{OPT} is a box profile (i.e., it is normalized and compartmentalized). Because we are considering $\mathcal{L}$ with 2α resource augmentation (meaning that the algorithm against which $\mathcal{L}$ is competing has 2α resource augmentation), each profile m_i^{OPT} consists of boxes with heights between $\frac{k}{4\alpha p} = \Theta(k/p)$ and $\frac{k}{2\alpha} = \Theta(k)$.

We now consider the average completion time for Algorithm A. Since each processor has memory impact R, the first $p/2$ processors to complete must incur total memory impact at least $\Omega(pR)$, thereby incurring total running time at least $\Omega(pR/k)$. This means that the final $p/2$ processors to complete each take time more than $\Omega(pR/k)$. Thus $\Omega(pR/k)$ is a lower bound for both the average completion time and the makespan of A.

In order to complete the proof, we construct an alternative parallel-paging solution B that has makespan (and thus also average completion time) only $O(pR/k\beta)$. Because A has α resource augmentation, the amount of memory available to the parallel-paging algorithm B is only k/α.

We can assume without loss of generality that the profiles m_i^{OPT} are box profiles. The algorithm B performs the Box-Packing algorithm from Section 2.2.3 on the box profiles $m_1^{\text{OPT}}, \ldots, m_p^{\text{OPT}}$. In particular, whenever the total memory allocated to processors is less than $k/2\alpha$, algorithm B selects a processor i out of those not currently executing (if there is one) and allocates space for the next box the profile σ_i^{OPT}. Note that the box is guaranteed to fit into B's cache of size k/α, since the maximum box height in any profile σ_i^{OPT} is only $k/(2\alpha)$.

Whenever algorithm B is in a state where it has allocated at least $\frac{k}{2\alpha}$ memory to boxes, we call B ***saturated***, and whenever B has not allocated $\frac{k}{2\alpha}$ memory to boxes (because there are no

[6]Note that $\alpha = O(1)$ resource augmentation means that the optimal green-paging algorithm has minimum box-size $k/(2\alpha p)$ and maximum box size $k/(2\alpha)$.

[7]By allowing for an extra factor of two in resource augmentation, we can actually think of A as having $2k$ memory, k of which is pre-allocated evenly among the processors (note that giving A extra memory can only help it). By re-normalizing this new size to k, it follows that every processor has at least $k/(2p)$ memory at all times.

more processors to allocate boxes to) we call B ***unsaturated***. The makespan (i.e., running time) of B can be broken into two components, the amount of time T_1 during which B is saturated, and the amount of time T_2 during which B is unsaturated.

Since the total memory impact of profiles $\sigma_1^{\text{OPT}}, \ldots, \sigma_p^{\text{OPT}}$ is $O(pR/\beta)$, the amount of time T_1 that B can spend saturated is at most $O(pR/\beta k)$ (recall that $\alpha \leq O(1)$ so α does not appear here).

On the other hand, whenever B is unsaturated, all of the remaining processors are executing simultaneously. It follows that T_2 is upper-bounded by the makespan of the processor $i \in \{1, 2, \ldots, p\}$ with the largest makespan (the makespan of a processor is the sum of the widths of the boxes in m_i^{OPT}). Each processor i incurs total memory impact $O(R/\beta)$ and has has minimum box-height $\Omega(k/p)$. It follows that the sum of the widths of the boxes in m_i^{OPT} is $O(R/\beta)/\Omega(k/p) \leq O(pR/\beta k)$.

Combining T_1 and T_2, the total makespan of algorithm B is at most $O(pR/\beta k)$. This is a factor of $\Omega(\beta)$ smaller than the average-completion time and the makespan for algorithm A, as desired. □

2.6 Tight Bounds for Green Paging

This section proves both lower and upper bounds for green paging. These results also appear in [122]. Once these green paging results are proved, the equivalence results from the previous section immediately translate them into corresponding bounds for parallel paging.

2.6.1 Lower Bounds for Green Paging

In this section we show lower bounds on the competitive ratio of deterministic algorithms.

Theorem 2. *Suppose $s \geq p^{1/c}$ for some constant c. Consider the green paging problem with maximum box-height k and minimum box-height k/p. Let ALG be any deterministic online algorithm for green paging, and let α be the amount of resource augmentation. Then, the competitive ratio of ALG is $\Omega\left(\frac{\log p}{\alpha}\right)$.*

Proof. The proof proceeds as follows. First, we briefly recap the model parameters and some assumptions we make. Then, we consider for a generic online algorithm a specific request sequence, on which it is guaranteed to fault on every request, and that has some additional properties. Then, we show how an offline algorithm can service the same sequence paying, informally, α times the cost on a fraction at most $1/\log p$ of all requests – and $\alpha/\log p$ times the cost on all remaining requests.

Let us briefly recap the model parameters and assumptions. Let k be the maximum memory available to the online algorithm ALG, and k/p the minimum. ALG is compared to an offline algorithm OFF with memory between $h = k/\alpha$ and $k/(\alpha p)$, *that pays α times as much for the same space*: i.e., τ timesteps at memory size k/α cost OFF a total of $k\tau$, while they would cost ALG only $k\tau/\alpha$. Accessing a page costs ALG 1 timestep if the page is in memory. Otherwise, s timesteps are required to bring it into memory, and 1 more to access it.

We can assume without loss of generality that ALG is normalized and compartmentalized—since normalization and compartmentalization can be performed online, and increase the total

cost by at most a constant factor. By the same token we assume for simplicity that k, p and α are powers of 2. We can also assume without loss of generality that ALG is "smooth-growing", in the sense that no box of capacity c is ever preceded by a box of capacity less than $c/2$; again, it is trivial to verify that any online algorithm can be transformed online into a smooth-growing one by increasing the total cost by at most a factor $O(1)$ (in fact, $4/3 = 1 + 1/4 + 1/16 + \dots$).

Also, we can assume $s \geq p$: if s were smaller, we can simply modify an algorithm to never exceed a capacity threshold of more than s times the minimum – by simply substituting boxes of minimum capacity for any boxes exceeding the threshold. Then servicing the same requests in these boxes can take no more than s times longer, with a capacity that is at least s times smaller. Note that, this implicitly replaces p with s, meaning that the lower bound that we will get is $\Omega(\log s)$. Since $s \geq p^{1/c}$, this is $\Omega(\log p)$.

Given ALG, we construct an evil request sequence σ as follows. First, we request $k+1$ distinct pages $p_0, p_1, \dots, p_k$. Then, every further request is for the most recently evicted page; we extend the request sequence to include at least $(k+1)\log p/\alpha$ requests. It is immediate that ALG faults on every request (and the cost on the initial $k+1$ is a fraction at most $O(\alpha/\log p)$ of the total). Let $B(2^i)$ for $i = 1, 2, \dots, \log k$, be the total cost budget spent by ALG on boxes of capacity 2^i (i.e., 2^i times their total duration). Obviously, the cost incurred by ALG c_{ALG} equals $\sum_i B(2^i)$.

Now, let $j = \arg\min_i B(2^i)$ for $(k/p)\log p < 2^i < k/\max(\alpha, \log p)$; note that $B(2^j) = O(c_{ALG}/\log p)$. We service σ with a normalized and compartmentalized algorithm alternating between capacity 2^j and the minimum capacity $k/(\alpha p)$. In particular, whenever ALG's capacity is at least 2^j, we adopt minimum capacity $k/(\alpha p)$; we refer to any maximal interval of such requests as an *island*. Whenever ALG's capacity is less than 2^j, we adopt the larger capacity 2^j; we refer to any such maximal interval of such requests as a *sea*. We refer to the first box of a sea, and of an island, as its *shore*.

We show that we can service any one sea with at most 2^j faults on its shore. We do so by loading and holding in our memory a copy of ALG's memory, and an *eviction stack* with the pages not in ALG's memory that have been most recently evicted by ALG (in particular, with the most recently evicted on top). It is immediate that after the initial setup phase (which takes place on the sea shore and incurs at most 2^j faults), every request will be for the page on the top of the stack—which is never empty since at sea ALG's capacity is strictly less than 2^j.

Note that the total number of sea and land shores differs by at most 1; and that, by the smooth-growth property, all island shores are boxes of capacity 2^j, preceded by a box of capacity 2^{j-1} Then, the total cost we incur on sea shores is at most $\alpha O(B(2^j))$, i.e. (by the definition of j) no larger than $O(\alpha/\log p)c_{ALG}$. Once offshore, we have capacity that is at most $O(p/\log p)$ that of ALG, and since we never fault and ALG always does, we incur a cost that is at most $O(\alpha/\log p)$ of ALG's. Similarly, on land requests, our capacity is $k/(\alpha p)$ while ALG's is at least $(k/p)\log p$. Thus, as we both fault on each land request, our cost is at most $O(\alpha/\log p)$ ALG's.

Then, we can service the entire request sequence with a cost that is $O(\alpha/\log p)c_{ALG}$. □

2.6.2 $O(\log p)$-Competitive Green Paging

We now present a deterministic online algorithm, which we call BLIND, whose competitive ratio is $O(\log p)$ when provided with a factor of (at least) two of resource augmentation, that is, when $h \leq k/2$. According to Theorem 2, this is optimal to within a constant factor when a constant amount of resource augmentation is given.

Algorithm BLIND is quite simple: it implements the LRU replacement policy, running with a suitably *predetermined* sequence of capacities. That is, the maximum amount of pages that BLIND retains in its cache at a given time step is *independent of* σ. Hence, not only BLIND does not use information about future requests in order to adjust its capacity (because it is an online algorithm): it doesn't even look at the *past* requests (whence the name *blind*)! The intuition is that, roughly speaking, BLIND, for each suitably defined period of time, "divides" its incurred cost among capacities $k/p, 2k/p, 4k/p, \ldots, k$ in such a way that at least a $\log p$-th fraction of its incurred cost is spent at the "right" capacity, that is, at roughly the same capacity that OPT has on the same subsequence of requests.

The road to the specification and the analysis of BLIND is divided into three parts: first, we design an $O(1)$-approximation offline algorithm by building a "quantized" version of OPT; second, we obtain a simplified version of the above algorithm which achieves a logarithmic (in p) approximation; and lastly, we show that a non-clairvoyant version of the latter algorithm achieves the same approximation factor when provided with a cache of size at least twice. We begin with some necessary preliminaries.

Definition 6. *For integer i, an* i-phase *of an algorithm for the green paging problem is a sequence of $3s2^i$ consecutive time steps spent at capacity 2^i.*

Thus, the total memory impact incurred by an algorithm for an i-phase is $3s4^i$. We now define an ***$(i, k/p)$-universal box profile***, a key concept in the design of our algorithm. Its definition is recursive.

Definition 7. *Let i be an integer and k/p be a power of two. A $(i, k/p)$-*universal box profile *of an algorithm for the green paging problem is a $\log k/p$-phase if $i \leq \log k/p$, and the concatenation of four consecutive $(i-1, k/p)$-universal box profiles followed by an i-phase otherwise.*

The memory impact (cost) of a $(i, k/p)$-universal box profile is easily established.

Lemma 2. *The memory impact of a $(i, k/p)$-universal box profile is $3s4^i(\log \left\lceil \frac{2^i}{k/p} \right\rceil + 1)$.*

Proof. By induction on i. The base case is for $i \leq \log k/p$; in this case, an $(i, k/p)$-universal box profile is a $\log k/p$-phase, whose cost is, by definition, at most $3s(k/p)^2$. Then assume the statement holds for $i-1$, with $i > \log k/p$, and consider an $(i, k/p)$-universal box profile. By definition of $(i, k/p)$-universal box profile and by applying the inductive hypothesis, its total cost is at most

$$\begin{aligned}
4(3s4^{i-1}(\log \left\lceil \frac{2^{i-1}}{k/p} \right\rceil + 1)) + 3s4^i &= 3s4^i(\log \left\lceil \frac{2^{i-1}}{k/p} \right\rceil + 1) + 3s4^i \\
&= 3s4^i(\log \left\lceil \frac{2^{i-1}}{k/p} \right\rceil + 2) \\
&= 3s4^i(\log 4 \left\lceil \frac{2^{i-1}}{k/p} \right\rceil) \\
&= 3s4^i(\log 2 \left\lceil \frac{2^{i}}{k/p} \right\rceil) \\
&= 3s4^i(\log \left\lceil \frac{2^{i}}{k/p} \right\rceil + 1),
\end{aligned}$$

where in the second-to-last equality we have used the hypothesis that k/p is a power of two. □

We say that an algorithm A services a subsequence of consecutive requests σ_j *over an i-phase* when σ_j gets serviced in $3s2^i$ time steps with A using capacity 2^i, but where each time step t is charged for a cost of exactly 2^i, even if $P(t)$, the number of pages in cache at time t, is smaller. Broadly speaking, its cost is accounted as if its capacity were *exactly* 2^i for each time step. (This is reasonable in practice since a paging algorithm works with *boxes* of cache locations rather than single locations, and thus the "cost"—in terms of its energetic costs or in terms of space taken away from other processors—of a box should be accounted even when not all the locations of the box are used). Thus, a subsequence of consecutive requests serviced over an i-phase comes at a cost of exactly $3s4^i$. Finally, Let OPT_i, $i = 0, 1, \ldots, \log k$ be an optimal offline algorithm running with capacity at most 2^i. Obviously, $\text{OPT}_i(\sigma) \geq \text{OPT}_{i+1}(\sigma)$, since an offline algorithm does not need to use all its cache locations.

We now introduce BLOCK_i, a recursively-defined offline algorithm which well approximates OPT_i. BLOCK_i is defined as an offline algorithm that services an input sequence σ by servicing its longest possible prefix either over an i-phase (in this case we say that BLOCK_i *maximizes*) or by simulating BLOCK_{i-1} for the same cost $3s4^i$ as an i-phase (in this case we say that BLOCK_i *simulates*), whichever yields the *longest* prefix; after having serviced such a prefix, BLOCK_i flushes its memory and services the remaining suffix of σ in the same way. Note that BLOCK_i effectively partitions σ into subsequences $\sigma_1, \sigma_2, \ldots, \sigma_x$ each of them (with the possible exception of σ_x) serviced incurring cost $3s4^i$. We have the following result, which shows that BLOCK_i does not spend much more than OPT_i.

Proposition 2.

$$\text{BLOCK}_i(\sigma) \leq 64 \cdot \text{OPT}_i(\sigma).$$

Proof. In order to prove an upper bound on the approximation ratio of BLOCK_i, we shall consider the concatenations of two consecutive subsequences σ_j and σ_{j+1} and compare $2 \cdot 3s4^i$, the cost spent by BLOCK_i for them, with (a lower bound to) the cost spent by OPT_i for servicing the same pair of subsequences.

It is sufficient to prove the claim when BLOCK_i maximizes on subsequence σ_{j+1}, that is, when σ_{j+1} gets serviced over an i-phase. It is easy to verify that, by construction, OPT_i must use at least $2^{i-1} + 1$ memory locations for servicing at least one request of σ_{j+1}; in fact, if this is not the case, then BLOCK_i would have serviced σ_{j+1} by never exceeding capacity 2^{i-1}, that is, BLOCK_i would have simulated BLOCK_{i-1}, since by doing so it would have payed, for the same subsequence of requests, half the cost of maximizing, and used the remaining half of the budget $3s4^i$ to process more requests than those in σ_j (because $(3/2)s4^i$ is clearly sufficient to service at least one request with a cache of size 2^i), thus servicing a prefix longer than σ_{j+1}, a contradiction.

Let t be the first time step when OPT_i has $2^{i-1} + 1$ pages in cache while servicing requests of σ_{j+1}, and consider the $s2^{i-2}$ time instants that precede t. We are going to prove that (i) the total cost accrued by OPT_i in such time steps is at least $(3/2)s4^{i-2}$, and that the subsequence serviced in such time steps is a subsequence of the concatenation of σ_j and σ_{j+1}. If this is the case, then we have that given any subsequence of consecutive requests of σ serviced by BLOCK_i with a cost of at most $2 \cdot 3s4^i$, the cost accrued by $\text{OPT}_i(\sigma)$ for the same subsequence is at least $(3/2)s4^{i-2}$. Since $2 \cdot 3s4^i = 64(3/2)s4^{i-2}$, the theorem follows.

Recalling that every offline algorithm grows its current capacity only upon a page fault, and thus only by at most one at each requests, it follows that during all the s time steps that precede t, OPT$_i$ has at least 2^{i-1} pages in cache, during the s preceding time steps it has at least $2^{i-1}-1$ pages in cache, and so forth for 2^{i-2} times, until $s2^{i-2}$ time instants are considered. The total cost accrued by OPT$_i$ in such time steps is at least

$$\begin{aligned}
\sum_{x=0}^{2^{i-2}-1} s\left(2^{i-1}-x\right) &= s \sum_{x=2^{i-1}-2^{i-2}+1}^{2^{i-1}} x \\
&= \frac{s\left(2^{i-2}\right)\left(2^{i}-2^{i-2}+1\right)}{2} \\
&\geq s\left(2^{i-2}\right)\left(2^{i-1}-\frac{2^{i-2}}{2}\right) \\
&= s\left(2^{i-2}\right)\left(2\cdot 2^{i-2}-\frac{2^{i-2}}{2}\right) \\
&= \frac{3}{2} s\left(4^{i-2}\right),
\end{aligned}$$

and thus we have proved (i).

It remains to prove (ii), that is, that the subsequence serviced by OPT$_i$ in the $s2^{i-2}$ time steps the precede t is "not too long". To this end, observe that we can assume that BLOCK$_i$ at the beginning of subsequence σ_j loads in cache all the pages that OPT$_i$ holds at the same moment (since the latter does not need to process σ_j starting with an empty cache), since the corresponding cost is at most $s4^i$, and thus BLOCK$_i$ remains with a budget of at least $2s4^i$. This budget is clearly sufficient to process the at most $s2^{i-2}$ requests that OPT$_i$ services in the $s2^{i-2}$ time steps the precede t, since BLOCK$_i$, once loaded in cache all the pages that OPT$_i$ holds at the beginning, can simply mimic the behavior of the latter. By construction, σ_j is the longest subsequence following σ_{j-1} that BLOCK$_i$ services with cost $3s4^i$, and thus claim (ii) is proved. □

Now we give an offline algorithm, IDLE-BLIND$_i$, which is a $O(\log p)$ approximation of BLOCK$_i$. This is an intermediate step towards both the definition and the analysis of our sought online algorithm, BLIND. IDLE-BLIND$_i$ is an offline algorithm that services each such subsequence σ_j over an $(i+1, k/p)$-universal box profile (and thus with capacity 2^{i+1}), as follows. If BLOCK$_i$ maximizes over σ_j, IDLE-BLIND$_i$ idles (that is, it stops servicing requests) over the four initial $(i, k/p)$-universal box profiles of the $(i+1, k/p)$-universal box profile, services σ_j over the last $(i+1)$-phase applying a LRU replacement policy, and then possibly idles until the end of the $(i+1)$-phase. Otherwise, if BLOCK$_i$ simulates over σ_j, IDLE-BLIND$_i$ services σ_j with IDLE-BLIND$_{i-1}$ over the four initial $(i, k/p)$-universal box profiles, and then possibly idles until the end of the $(i+1)$-phase. Notice that *the capacity of* IDLE-BLIND$_i$ *is simply a sequence of* $(i+1, k/p)$*-universal box profiles, and thus is independent of the request sequence*, and in particular of the past requests. The next proposition bounds from above the performance of IDLE-BLIND$_i$.

Proposition 3.

$$\text{IDLE-BLIND}_i(\sigma) \leq 4(\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1) \cdot \text{BLOCK}_i(\sigma).$$

Proof. The key point of the proof is showing that subsequence all the requests of σ_j can actually be serviced by IDLE-BLIND$_i$ over an $(i+1, k/p)$-universal box profile. To do so, it is sufficient to prove this fact when BLOCK$_i$ maximizes over σ_j, since the other case holds by induction. If BLOCK$_i$ maximizes over σ_j, IDLE-BLIND$_i$, by construction, services requests of σ_j over an $(i+1)$-phase, that is, using capacity 2^{i+1} for $3s2^{i+1}$ time steps. This means that, for processing σ_j, IDLE-BLIND$_i$ can use twice the capacity used by BLOCK$_i$, for twice the time. By the well-known result of Sleator and Tarjan for classic paging, IDLE-BLIND$_i$, which replaces pages using LRU, incurs at most twice the faults incurred by BLOCK$_i$ on that subsequence; this means that, after $3s2^i$ time steps, IDLE-BLIND$_i$ has serviced at least half the requests of σ_j, and thus after $3s2^{i+1}$ time steps has serviced all of them.

Now we can compare the cost incurred by IDLE-BLIND$_i$ with the cost of BLOCK$_i$. By definition, IDLE-BLIND$_i$ pays the cost of a $(i+1, k/p)$-universal box profile every time that BLOCK$_i$ incurs the cost of an i-phase, that is, $3s4^i$; hence, by Lemma 2,

$$\begin{aligned}\text{IDLE-BLIND}_i(\sigma) &\leq \frac{3s4^{i+1} \cdot (\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1)}{3s4^i} \cdot \text{BLOCK}_i(\sigma) \\ &= 4(\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1) \cdot \text{BLOCK}_i(\sigma).\end{aligned}$$

□

Now observe that IDLE-BLIND$_i$ *exploits its clairvoyance only to decide when to idle.* We define BLIND$_i$ similarly to IDLE-BLIND$_i$, with the difference that BLIND$_i$ never idles. Notice that BLIND$_i$ is an online deterministic algorithm.

The unexpected power of idling around the discontinuities of a memory profile function is neutralized when memory profiles are compartmentalized. In fact, if capacity never changes there is clearly no advantage in idling; but when it changes, roughly speaking, there is still no advantage in idling since the memory contents are cleared at the end of each box anyways. Hence, thanks to compartmentalization, and leveraging properties of the LRU replacement policy adopted by both BLIND$_i$ and IDLE-BLIND$_i$, we have the following.

Proposition 4. *Assume compartmentalization. Then,*

$$\text{BLIND}_i(\sigma) \leq \text{IDLE-BLIND}_i(\sigma).$$

Proof. Let b_j be the j-th box of the memory profile (i.e., a maximal interval of time during which the capacity of the memory profile remains unchanged) of the two algorithms, and let c_j be its capacity. Let σ_{b_j} be the subsequence of consecutive requests of the request sequence σ that IDLE-BLIND$_i$ services during box b_j. It is then sufficient to show that, for any j, BLIND$_i$ has serviced all the requests of σ_{b_j} by the end of box b_j.

We show this by induction on j. The claim clearly holds for the first box b_1 of the memory profile because both algorithms start with an empty memory, both start to service requests from the beginning of σ, and since the memory capacity never changes till the end of the box there is clearly no advantage in idling.

Then assume the claim holds for each box $b_{j'}$ with $1 < j' < j$, and consider box b_j. If BLIND$_i$ has serviced all the requests of σ_{b_j} before box b_j begins, then we are done. Otherwise, by the inductive hypothesis, when box b_j begins algorithm BLIND$_i$ has to service a suffix of σ_{b_j}. Hence, we have to show that BLIND$_i$ services any suffix of σ_{b_j} by the end of box b_j or, in other words,

that LRU, starting with an empty memory (because of compartmentalization) of fixed capacity, takes more time to service a sequence of requests than to service any of its suffixes.[8]

The page requests in the suffix of σ_{b_j} serviced by BLIND$_i$ during the j-th box of the memory profile can be partitioned into (at most) two parts: those up to the c_j-th distinct page request in the suffix (if it exists), and the subsequent requests. For the first part of requests, BLIND$_i$ misses only on the first occurrence of a requested page. Observe that IDLE-BLIND$_i$ might not miss on all such requests during box b_j, since it may already have the corresponding pages in memory because they were needed to service some requests in the prefix of σ_{b_j}. However, because of compartmentalization, each of such pages must have been brought in memory by IDLE-BLIND$_i$ when first requested in the prefix of σ_{b_j}. Therefore, the time taken by BLIND$_i$ to service the first part of the suffix is no larger than the time taken by IDLE-BLIND$_i$ to service all the requests of σ_{b_j} up to the first part of the suffix.

It now remains to consider the (possible) second part of requests of the suffix of σ_{b_j} serviced by BLIND$_i$, which begins right after c_j distinct page requests appeared in the suffix. Since both BLIND$_i$ and IDLE-BLIND$_i$ use LRU, at that moment they have the same set of pages in fast memory, and the order of such pages according to their latest access is clearly the same for both sets. Therefore, from that point till the end of the suffix the two algorithms identify, yielding the claim for box b_j. □

Proposition 5. *When the LRU replacement policy is applied, compartmentalization increases the cost of a time-normalized memory profile by a factor of at most two.*

Proof. Since in a time-normalized memory profile each box lasts a number of clockticks equal to an integer multiple of s times the memory capacity, the expansion due to compartmentalization at most doubles the cost of the memory profile.

Now it remains to show that any sequence serviced without compartmentalization can also be serviced in its entirety when compartmentalization is applied. Let b_j be the j-th box of a time-normalized memory profile, and let c_j be its capacity. Let σ_{b_j} be the subsequence of consecutive requests that LRU services during box b_j. It is then sufficient to show that, for any j, all the requests of σ_{b_j} get serviced by the end of box b_j in the compartmentalized expansion of the memory profile.

We show this by induction on j. By construction, the claim clearly holds for the first box b_1 of the memory profile. Then assume the claim holds for each box $b_{j'}$ with $1 < j' < j$, and consider box b_j. If in the compartmentalized expansion all the requests of σ_{b_j} have been serviced before box b_j begins, then we are done. Otherwise, by the inductive hypothesis, when box b_j begins LRU has to service a suffix of σ_{b_j}. Hence, we have to show that LRU takes more time to service a sequence of requests than to service, starting with an empty memory but with $s \cdot c_j$ more time steps available, any of its suffixes.

The page requests in any suffix of σ_{b_j} can be partitioned into (at most) two parts: those up to the c_j-th distinct page request in the suffix (if it exists), and the subsequent requests. For the first part of requests, LRU with compartmentalization misses only on the first occurrence of a requested page, whereas without compartmentalization LRU might not miss on such requests since those pages may be already in fast memory because they were needed to service some

[8] Recall that, somewhat surprisingly, this is not true in general: for instance, it is not true for the FIFO replacement policy—see Section 2.4.3.

requests in the prefix of σ_{b_j}. However, observe that compartmentalization allows enough extra time to do the extra work of bringing those pages in fast memory.

It now remains to consider the (possible) second part of requests of the suffix of σ_{b_j}, which begins right after c_j distinct page requests appeared in the suffix. Because of the LRU replacement, at that moment the contents of the fast memory are the same in both—i.e., with and without compartmentalization—cases; moreover, the order of those pages according to their latest access is clearly the same in both cases. Therefore, from that point till the end of the suffix the two executions identify, yielding the claim for box b_j. □

Define BLIND as $\text{BLIND}_{\log k/2}$. Below we provide its straightforward pseudocode.

Algorithm 1 BLIND

1. Service σ through a sequence of $(\log k, k/p)$-universal box profiles, evicting the least recently used page(s) whenever capacity is adjusted downwards or a page fault occurs.

Putting all pieces together, we obtain the following result.

Theorem 1. *Using resource augmentation $\alpha = 2$, the competitive ratio of BLIND is $O(\log p)$.*

Proof. Combining Proposition 1 with the results of [33],[9] Proposition 2, Proposition 3, and Proposition 4, we have

$$\begin{aligned}\text{BLIND}_i(\sigma) &\leq 3 \cdot \text{IDLE-BLIND}_i(\sigma) \\ &\leq 12(\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1) \cdot \text{BLOCK}_i(\sigma) \\ &\leq 768(\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1) \cdot \text{OPT}_i(\sigma).\end{aligned}$$

Hence, on a cache of size $k = 2^{i+1}$, and with resource augmentation $\alpha = 2$, the competitive ratio of BLIND_i is at most $768(\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1)$, and since BLIND coincides with $\text{BLIND}_{\log k/2}$, the theorem follows. □

[9] When LRU in applied, the prefix of duration equal to s times the capacity of the box need not be added, and as a result the overhead is only two rather than three.

Chapter 3

How to Manage High-Bandwidth Memory Automatically

This chapter develops an algorithmic foundation for automated management of the multilevel-memory systems common to new supercomputers. In particular, the High-Bandwidth Memory (HBM) of these systems has a similar latency to that of DRAM and a smaller capacity, but it has much larger bandwidth. Systems equipped with HBM do not fit in classic memory-hierarchy models due to HBM's atypical characteristics.

Unlike caches, which are generally managed automatically by the hardware, programmers of some current HBM-equipped supercomputers can choose to explicitly manage HBM themselves. This process is problem specific and resource intensive. Vendors offer this option because there is no consensus on how to automatically manage HBM to guarantee good performance, or whether this is even possible.

We give theoretical support for automatic HBM management by developing simple algorithms that can automatically control HBM and deliver good performance on multicore systems. HBM management is starkly different from traditional caching both in terms of optimization objectives and algorithm development. Since DRAM and HBM have similar latencies, minimizing HBM misses (provably) turns out not to be the right memory-management objective. Instead, we directly focus on minimizing makespan. In addition, while cache-management algorithms must focus on what pages to keep in cache; HBM management requires answering two questions: (1) which pages to keep in HBM and (2) how to use the limited bandwidth from HBM to DRAM. It turns out that the natural approach of using LRU for the first question and FCFS (First-Come-First-Serve) for the second question is provably bad. Instead, we provide a priority based approach that is simple, efficiently implementable and $O(1)$-competitive for makespan when all multicore threads are independent.

This work is published in the proceedings of SPAA 2020 [56].

3.1 Introduction

Enabled by the recent innovations in 3D die-stacking technology, vendors have begun implementing a new approach for improving memory performance by increasing the bandwidth between on-chip cache and off-package DRAM [109, 92]. The approach is to bond memory directly to

the processor package where there can be more parallel connections between the memory and caches, enabling a higher bandwidth than can be achieved using older technologies. Throughout the rest of this work, we refer to the on-package 3D memory technologies as ***high-bandwidth memory*** or ***HBM***.[1]

The HBM cannot replace DRAM ("main memory") since it is generally about 5 times smaller than DRAM due to constraints such as heat dissipation, as well as economic factors. For example, current HBM sizes range from 16 gigabytes per compute node (the Department of Energy's "Trinity" [61]) to 96 gigabytes per compute node (the Department of Energy's "Summit" [138]), several times smaller than the per-node sizes of DRAM on those systems (96 GB and 512 GB, respectively). HBM therefore augments the existing memory hierarchy by providing memory that can be accessed with up to 5x higher bandwidth than DDR4, today's DRAM technology, when feeding a CPU [1], and up to 20x higher bandwidth when feeding a GPU [138], but with latency similar to DDR4.

As the number of cores on a chip has grown in the past two decades, the *relative memory capacity*, defined as memory capacity divided by available gigaflops, has decreased by more than 10x [93]. Thus, processors are becoming more starved for data. HBM, with its improved ability to feed processors, provides an opportunity to overcome this bottleneck, if application software can use it.

HBM has a critical limitation besides its constrained size: since it uses the same technology as DRAM, it offers little or no advantage in latency. Therefore, the HBM is not designed to accelerate memory-latency-bound applications, only memory-bandwidth-bound ones.[2]

How HBM is managed. Intel's Knights Landing processors [129] on Trinity can boot into multiple modes. In "cache mode," the HBM is system-controlled and is integrated into the memory hierarchy as the "last level of cache." In "flat mode," the programmer controls HBM by explicitly copying data in and out of HBM, trying to squeeze as much performance from the system as possible. Hybrid mode splits the HBM into one "flat" piece and one "cache" piece.

This proliferation of HBM modes exists because there is no consensus on how to automatically manage HBM efficiently, or whether this is even possible.

On the other hand, on most systems, on-chip cache is automatically managed by the hardware and works well enough that application programmers can treat the cache hierarchy a a black box. They can also assume sufficient support from libraries of cache-aware and cache-oblivious algorithms. Ideally, we would like a system controlled HBM to also work well-enough to free the programmer from the need to manage it.

However, managing the HBM is not identical to managing caches and introduces complications that do not exist in more traditional memory hierarchies. In particular, cores compete not only for HBM capacity, but also for the more limited channel capacity between HBM and DRAM. HBM does not fit into a standard memory hierarchy model [74, 27], because in traditional hierarchies, both the latency and bandwidth improve as the levels get smaller. This is not true of HBM.

The question looms: Are there provably good algorithms for automatically controlling HBM?

[1]Hardware vendors use various brand names such as High-Bandwidth Memory (HBM), Hybrid Memory Cube (HMC), and MCDRAM for this technology.

[2]HBM does not increase *off-package* DRAM bandwidth. It does not accelerate a scan of a large chunk of data in DRAM that does not fit into the HBM, because the operation is limited by the DRAM bandwidth. Therefore, HBM improves some memory-bandwidth-bound computations but does not automatically improve all.

3.1.1 Results

Multicore HBM model. We propose a multicore model for HBM that captures the high bandwidth from the cores to HBM and the much lower bandwidth to DRAM. There are p parallel channels connecting p cores to the HBM but only a single channel connecting HBM to DRAM; see Figure 3.1. This configuration captures the high (on-package) bandwidth between the p cores and HBM and the much lower (off-package) bandwidth between HBM and DRAM. Data is transferred in blocks — there are up to p parallel block transfers from the HBM to the cores, but only one block transfer at a time between DRAM and HBM. The roughly comparable latencies are captured by setting all block-transfer costs (times) to 1.

HBM management. We focus on instances where the multicore's threads access disjoint sets of blocks. This emphasizes the cores' competition for HBM and the limited bandwidth between HBM and DRAM.

What should the performance objective be? The high-level objective is to improve the performance by finishing all threads as quickly as possible — in other words, we want to minimize the makespan. In sequential caching, we generally use minimizing cache misses as a performance objective since it is a good proxy for makespan. We might be tempted to use HBM misses as the performance objective here. Surprisingly, it turns out that minimizing HBM misses correlates poorly with makespan for HBM for two reasons (formally proved in Section 3.7). First, unlike caches, HBM has the same access latency as DRAM. Second, multiple processors are accessing the HBM and potentially contending for the channel between HBM and DRAM and the amount of contention can have an impact on the makespan, not just the sheer number of HBM misses. Therefore, we focus directly on minimizing makespan. We establish the following:

- *Minimizing page faults does not correlate well with minimizing makespan.* An algorithm that optimizes the total number of HBM misses may have bad makespan (the time the last thread completes)— up to a $\Theta(p)$-factor worse than optimal. This is not true for the standard cache-replacement problem where the total number of cache misses is the surrogate objective for makespan [127, 70, 44].
- *Minimizing makespan is strongly NP-hard.*
- *Sharing the HBM-to-DRAM channel fairly does not work.* We consider how to design block-replacement policies for the HBM coupled with the First-Come-First-Serve (FCFS) algorithm for determining the order of accesses from HBM to DRAM. We show that even though LRU is a very good block-replacement policy, if we use FCFS in the HBM-to-DRAM channel, LRU performs poorly. In particular, with any constant amount of resource augmentation the makespan of using FCFS with LRU is $\Omega(p)$ away from the optimal policy in the worst case. This negative result establishes that more sophisticated management of the channel between HBM and DRAM is central to the designing a good algorithm for the problem. The seemingly fair FCFS policy is bad.

Our main positive results are simple online and offline algorithms for automatically managing HBM. What is interesting about HBM management is how starkly it differs from traditional caching policies, which are well understood in a serial setting but are challenging in multicore settings [103, 80].

- *Priority-based mechanism for managing the HBM-to-DRAM channel.* We give a priority-based policy for managing the channel between HBM and DRAM. We impose a pecking order on the cores, so that a high-priority core never has a request to DRAM blocked by a request from a lower-priority core. Our algorithms for HBM management are built around this priority-based mechanism.
- $O(1)$*-competitive algorithm for HBM management.* As a first step towards our online algorithm, we give an offline approximation algorithm for the makespan objective. We build upon the offline algorithm to obtain a simple online algorithm (with a more complicated analysis) that is $O(1)$-competitive, even without resource augmentation. The online algorithm is non-intuitive —it often preferentially allocates the HBM-to-DRAM channel to a single core while depriving other cores — but guarantees nearly optimal makespan. (The algorithm can treat cores fairly over time by periodically changing the pecking order among the cores.)
- *Approximation algorithms for objective functions measuring progress.* In addition to makespan, we consider another objective, total completion time, to measure progress towards completion. We show that this problem reduces to a particular resource-constrained scheduling problem. We then leverage techniques from scheduling theory to give a $O(1)$-approximation algorithm (offline) based on linear-program rounding.

3.1.2 Related Work

HBM-tuning and cache mode. Intel's Knights Landing (KNL) processor [92] features an implementation of HBM. Several recent papers have documented runtime improvements of 3-4x using this HBM in "cache mode" compared to using DRAM alone, when problem instances fit entirely in the HBM. For example, Li et al. studied eight kernels from scientific computing [98] on KNL and found 3-4x speedups on some instances of sparse matrix-vector multiplication, Cholesky decomposition, and dense matrix-matrix multiplication. They observed more modest 1-2x speedups for sparse matrix transpose and sparse triangular solve. Byun, et al. corroborate the KNL speedup for dense matrix-matrix multiplication [46]. Slota and Rajamanickam [128] observed 2-5x speedups in graph algorithms on instances far larger than HBM. Butcher et al. [45] also studied problems that are too large to fit into HBM. They optimized sorting on KNL and concluded that GNU parallel sort run in cache mode is not nearly as fast as a custom sorting algorithm (based upon concurrent calls to GNU serial sort), also run in cache mode [45].

This result is in keeping with the predictions made by Bender et al. [28, 26, 29]. Before KNL existed, they gave HBM-optimized sorting algorithms and obtained simulation results that predicted good speedups for these algorithms. However, this does not settle the question of automatic management of HBM. KNL's arbitration of HBM misses is handled by the DRAM controller. Although the actual protocol is proprietary, it is likely a solution based on [118]. Such arbitration is commonly called "first-ready first-come-first-served (FR-FCFS)." As the name implies, this is a variant of FCFS. We show in Section 3.5 that FCFS is not a good arbitration policy for HBM misses, and we conjecture that a future cache mode informed by this chapter may perform significantly better.

Multi-thread/multi-core paging. There exists a rich literature on multi-threaded and multi-core paging models. Feuerstein and Strejilevich de Loma [67] consider a paging model where

there are multiple threads but only a single core. They optimize the number of cache misses. Loma [60] and Seiden [123] give randomized algorithms for the same setting.

Hassidim [80] considers a paging model where there are multiple threads and multiple cores. He minimizes the makespan. López-Ortiz and Salinger [103] consider a similar model but minimize cache misses. They give lower bounds and an offline algorithm with a runtime exponential in the size of cache. Katti and Ramachandran [89] give a competitive algorithm for multi-core paging assuming that the interleaving of the request sequences of the cores are fixed. These classic paging results differ from our HBM model because in these prior results access times of near and far levels are different but there are still p channels between the cores and shared cache and between cache and DRAM. Therefore, these prior parallel caching results do not carry over to our setting, and vice versa.

When multiple threads access the same shared cache, the fraction of cache dedicated to any given thread can vary [21, 34, 25, 8]. Peserico [113] and Bender et al. [34, 32] formulated models for page replacement in a fluctuating cache, with the latter model serving as an algorithmic foundation for cache-adaptive analysis [34, 32, 100].

There are several analysis frameworks based upon assuming an underlying optimal paging algorithm. These include the seminal ideal cache-model of Frigo et al. [73, 74] and Prokop [74], which was based on Sleator and Tarjan's [127]'s classic paging results; cache-adaptive analysis [34, 32, 100]; and parallel caching models based on work stealing [52]. We can view this chapter as proposing an alternative setting for HBM, where the programmer can assume optimal paging for HBM, and then let the system make all decisions.

Finally, we note that there are many sequential and parallel models of the memory hierarchy [30, 57, 15, 16, 7, 6, 5, 37, 49, 86]. These include models where there are private caches associated with each core [19]. In contrast, our HBM model explicitly does not need to consider a private cache. Whatever happens in the private caches of each individual core is independent of the optimization problem in this chapter. As mentioned in [27], the performance characteristics of HBM set it apart from most other memory-hierarchy models, so that in some ways, HBM and DRAM are like siblings on the same level of the hierarchy, and in other ways they are stacked. This helps explain why the present optimization problem is so surprisingly different from prior work, and also why prior work provides little insight into how to deal with HBM.

3.2 HBM Model

Our model of HBM comprises a multi-core machine with p parallel channels connecting p cores to the HBM. The HBM has size M, and the DRAM (main memory) has no space limitation.

The increased bandwidth of HBM comes from multiple channels between it and the cores. There is only one channel between HBM and DRAM. Data is transferred along any of these channels in blocks of size B; see Figure 3.1. HBM can hold $k = M/B$ blocks. There can be up to p parallel block transfers from the HBM to the cores, but only one block at a time is transferred between DRAM and HBM. The roughly comparable access costs are captured by setting all block-transfer cost to 1. That is, it takes one time step to transfer a block from HBM to a core or from DRAM to HBM.

Unlike the Ideal Cache model [73, 74], our HBM model has two resources to manage: the HBM itself and the ***far channel*** between HBM and DRAM. The HBM is managed by a block-

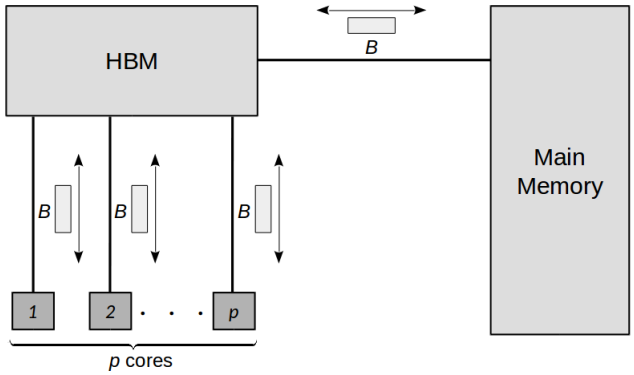


Figure 3.1: The HBM model with $\boldsymbol{p}$ cores and two levels of memory.

replacement policy, and the DRAM channel is managed by a ***far-channel arbitration policy.*** For example, we might consider LRU block replacement and FCFS far-channel arbitration.

Parallel program execution in HBM model. Each core runs its own stream of instructions, which for the purposes of the model, is a sequence of block requests. Thus, we denote $R^i = r_0^i, r_1^i, r_2^i, \ldots$ as the sequence of the blocks requested by core p_i on its dedicated channel to the HBM. We omit the core number when it is understood. We analyze the *disjoint* case in which the cores access disjoint sets of blocks, that is, for $i \neq j$ and $\forall q, s\ r_q^i \neq r_s^j$, as in prior work on parallel caching [80, 103]. This case emphasizes the computational issues that arise when the p programs compete for their own share of the HBM. Furthermore, the most common case when executing multithreaded programs is that the threads are disjoint or nearly disjoint [45, 27].

We say that a request r_j^i is ***served at time step t***, if the previous request to be served was r_{j-1}^i and the requested block r_j^i is transferred to core p_i at time step t. Request r_j^i is served as follows. If r_j^i is in HBM, then core p_i receives that block exactly one tick after the request. Otherwise, the block must be retrieved from DRAM via the far channel which takes at least one additional tick and may take many more depending on the request streams of other cores and the far-channel arbitration policy. Note that HBM hits have nonzero cost in our HBM model (since HBM latency is no better then DRAM latency) and are thus infinitely more expensive than the zero-cost cache hits of traditional caching models.

Primary objective: minimizing makespan. Given an HBM of size M and p disjoint request sequences of p cores, the objective is to find a contention-resolution policy for the HBM-DRAM channel and a block-replacement policy for the HBM so that the makespan is minimized.

3.3 Technical Overview

In the rest of the work, we propose methods to automatically manage HBM and we analyze them with our new HBM model. Our contributions, per section, are as follows:

- In this section, we explain how to navigate the algorithmic issues that distinguish HBM management from traditional cache management and we informally justify the makespan metric for HBM. We add a formal justification in Section 3.7.

- We give online algorithms for managing HBM asymptotically optimally with respect to the makespan objective (Section 3.4).
- We analyze the natural strategies for managing the two resources of the HBM model (variants of which are canonical results of computer architecture cited more than 1000 times [118]) and prove that they are not asymptotically good at managing HBM (Section 3.5).
- We prove the strong NP-hardness of the makespan minimization problem for HBM (Section 3.6)

Metrics for HBM management. HBM management must comprise policies for (1) dividing HBM's storage capacity among cores or DRAM regions, (2) evicting blocks from HBM, and (3) deciding which DRAM block requests to satisfy first, which we call "far-channel arbitration." In this work, we do not constrain (1). However, we note that KNL's cache mode employs direct-mapped caching (each block in DRAM has a fixed and unique destination in HBM), constraining both (1) and (2). Crucially, we will show that effective far-channel arbitration is the key to reducing the running time of a program in the HBM model.

We know that in single-core and multi-core paging models, minimizing the number of cache misses leads to a good approximation of the running time of a program [127, 70, 44]. The interesting difference in our HBM model is that the number of HBM misses no longer gives a good approximation to the running time. In particular, in Section 3.7, we show that there exist request sequences from p cores such that any policy that minimizes the number of HBM misses has a running time that is a factor of $\Theta(p)$ larger than the optimal running time. Moreover, unlike in traditional caching, resource augmentation in the form of larger HBM sizes is not necessary.

Hence, we turn directly to the makespan metric. We show that the problem of minimizing the makespan in HBM model is strongly NP-hard. The limited bandwidth between HBM and DRAM plays a pivotal role in the hardness proof.

So how should we deal with contending requests for the HBM-DRAM channel?

Natural far-channel arbitration policies do not work in the HBM model. One intuitive (but doomed) far-channel arbitration policy is to queue DRAM requests in First-Come-First-Serve (FCFS) order. Fairness would seem to dictate some sort of FCFS queue, and a canonical variant of this from [118] has been influential with vendors of DRAM controllers. However, we prove that FCFS is *not* a good far-channel arbitration policy for HBM. Even if we have a good eviction policy, such as LRU, a far-channel arbitration policy of FCFS queueing is provably non-optimal. In particular we show that even with d memory augmentation and s far-channel bandwidth augmentation, there exist p request sequences in which the makespan of FCFS with LRU is a $\Theta(\frac{p}{ds})$-factor away from that of the optimal policy.

A better strategy is to assign priorities to the cores. That is, there is a pecking order among the cores so that a high-priority core always beats a lower-priority core when one of its requests needs access to the HBM-DRAM channel. In Section 3.4 we analyze an online HBM-management strategy where the eviction policy is LRU but the far-channel arbitration policy is based on the pecking order. We prove that this simple scheme, which we call PRIORITY, is constant competitive with the optimal policy for minimizing makespan (without needing resource augmentation of either the HBM size or the far-channel bandwidth).

The theoretical wedge that this work drives between FCFS and priority-based HBM-DRAM channel arbitration is an important contribution of this work. Given that it is natural to have FCFS buffers queue up requests for the HBM-DRAM channel (and that FCFS variants are widely implemented in today's hardware), we believe that this negative queuing result could be quite

useful to hardware designers. A priority-based scheme is straightforward to implement in real hardware and leads to provably good algorithms for HBM management under our assumption of disjoint reference streams. Furthermore, we note that priority-based schemes are not inherently unfair. Our analysis still works if we change the priorities periodically over time.

Analysis of online competitive algorithm for the makespan. While the priority-based mechanism seems algorithmically simple, its analysis is more complicated. Thus, we explain the online analysis through an intermediary, an offline algorithm with a more complicated mechanism but a somewhat simpler analysis. We show that this algorithm, which we call k-PACKING, is an $O(1)$ approximation algorithm to makespan.

The offline algorithm k-PACKING divides the execution into *phases* of $\Theta(k)$ steps, where k is the size of the HBM. In each phase, each core makes an all-or-nothing decision about whether to execute its thread: either the thread makes $\Theta(k)$ progress or it does not run. If a thread runs, it grabs all of the resources from HBM that it needs by allocating space in the HBM and a channel bandwidth equal to the number of blocks that it accesses in that phase. Thus, in a phase, some threads make essentially full progress and others make none. k-PACKING performs a maximal packing of the threads into the phase. k-PACKING is a "very" offline policy because it requires $\Theta(k)$ look-ahead for each core, and k is very large.

The online algorithm PRIORITY does not have any lookahead, in contrast to k-PACKING. Hence, some cores may progress $\Theta(k)$ steps in a $\Theta(k)$-length phase, while others do not. If a core does not progress a full $\Theta(k)$ steps, we say that it *wastes* HBM capacity and bandwidth. We prove that the priority scheme guarantees at most a constant factor of these resources are wasted. This delivers the same performance guarantees as k-PACKING.

Approximation algorithm for offline total completion time. In addition to makespan, we also consider a different metric, namely the sum of completion times of the p cores. This metric encourages all of the threads to complete quickly, rather than just the last one. For this problem, we give an offline $O(1)$-approximation algorithm by reducing the HBM total completion time problem to a resource constrained scheduling problem. In particular, we divide each request sequence into chunks of size $\Theta(k)$. We denote a chunk as a task that takes $\Theta(k)$ time and an amount of resources equal to the number of blocks the chunk requests. We show that this simple reduction costs at most a constant factor more than the optimal total completion time. We then leverage techniques from scheduling theory to give an $O(1)$-approximation based on linear-program rounding. It remains an intriguing open problem whether there can exist an online algorithm that is constant competitive for the sum-of-completion-times objective.

3.4 $O(1)$-Competitive Online Algorithm for HBM Block Management

In this section we present an $O(1)$-competitive online algorithm for the makespan-minimization problem. We first give an offline $O(1)$-approximation algorithm. Then we show how to transform the offline strategy into an online strategy while retaining constant competitiveness.

One of the exciting aspects of our makespan-minimization problem is that proving constant competitiveness does not require resource augmentation. This result stands in stark contrast to most online caching problems, where resource augmentation is necessary to achieve good

competitive ratios. Nonetheless, the optimization problem is delicate. In Section 3.5 we show that some seemingly natural HBM caching policies achieve a competitive ratio as large as $\Theta(p)$.

In the rest of this section we prove the following theorem:

Theorem 9. *There exists an $O(1)$-competitive online algorithm for the makespan-minimization problem (without resource augmentation).*

3.4.1 Constant-approximation offline algorithm

This section gives an offline $O(1)$-approximation algorithm for the makespan-minimization problem, which we call the k-PACKING algorithm. We show the following:

Lemma 3. *There exists an offline constant-approximation algorithm for the makespan-minimization problem (without resource augmentation).*

We divide the request sequence $R^i = r_1, r_2, r_3, \ldots$ for each thread i into ***chunks***, where each chunk C_{ij} (except possibly the last) contains exactly $k/4$ requests[3]. Specifically,

$$R^i = \overbrace{r_1, r_2, \cdots, r_{k/4}}^{C_{i1}}, \overbrace{r_{1+k/4}, r_{2+k/4}, \cdots, r_{2k/4}}^{C_{i2}} \cdots$$

Executing a chunk C_{ij} means servicing each request in C_{ij}. A chunk C_{ij} is ***ready to run*** as soon as $C_{i,j-1}$ is executed. We associate a request with the block of memory needed to service the request.

Algorithm k-PACKING**.** The k-PACKING algorithm proceeds in ***phases***. In each Phase ϕ, k-PACKING executes at most one chunk from each thread. No chunks are partially executed. Let $\mathcal{C}$ be a set of chunks. Define the ***working set*** of $\mathcal{C}$, denoted $\mathcal{B}(\mathcal{C})$, to be the set of blocks requested in set $\mathcal{C}$. It is the union of the set of blocks over all the chunks in $\mathcal{C}$.

In Phase ϕ, k-PACKING executes a set $\mathcal{C}_\phi$ of ready-to-run chunks such that

1. $|\mathcal{B}(\mathcal{C}_\phi)| \leq k$,
2. Each thread executes either zero or one chunk in Phase ϕ, and
3. $\mathcal{C}_\phi$ is maximal. That is, no additional chunk can be added while satisfying Constraints 1 and 2.

Although chunks can be chosen greedily within a phase, k-PACKING itself is not greedy. That is, it may be possible for a thread to make forward progress in a phase without hindering any other thread—but k-PACKING does not execute any thread unless it can complete an entire chunk for that thread in the phase.

Because a phase ϕ is defined by the chunks $\mathcal{C}_\phi$ that it runs, we will overload notation, letting $\mathcal{B}(\phi)$ denote the set of blocks served in Phase ϕ. For generic input $\mathcal{I}$, let k-PACKING($\mathcal{I}$) denote running k-PACKING on instance $\mathcal{I}$.

Definition 8. *Each Phase ϕ in k-PACKING($\mathcal{I}$) has one of the following types.*

Contested: $3k/4 \leq |\mathcal{B}(\phi)| \leq k$

[3]Our proofs assume that k is a multiple of 4, fairly common for memory size, but we can adjust the proofs for general k.

Uncontested: $|\mathcal{B}(\phi)| < 3k/4$.

Lemma 4. *If $|\mathcal{B}(\phi)| < 3k/4$, then all unfinished threads execute a chunk in Phase ϕ.*

Proof. We prove this by contradiction. Let ϕ be a phase such that $|\mathcal{B}(\phi)| < 3k/4$. Assume that there is an unfinished thread p_i that does not execute a chunk in ϕ. However, a chunk of p_i has at most $k/4$ blocks. So we can add the ready-to-run chunk of p_i to Phase ϕ without violating Constraints 1 or 2 for phases. Therefore phase ϕ is not maximal, a violation of the third constraint. □

Lemma 5. *Suppose that k-PACKING($\mathcal{I}$) has X_1 contested phases and X_2 uncontested phases. Then the makespan of k-PACKING($\mathcal{I}$) is at most $\frac{5k}{4}X_1 + kX_2$.*

Proof. In every contested phase, at most k blocks are transferred from DRAM to HBM, requiring at most k time steps. Once the blocks are in HBM, they are served to the cores in at most $k/4$ time steps. So a contested phase finishes in $5k/4$ time units. Similarly, in every uncontested phase, at most $3k/4$ blocks are fetched from DRAM to HBM using at most $3k/4$ time steps. Once all the blocks are in the HBM, they are served to the cores in at most $k/4$ more time steps. So a uncontested phase finishes in k time units. □

For a set of phases Φ for an algorithm A we define $\eta_A(\Phi) \equiv \sum_{\phi\in\Phi} |\mathcal{B}(\phi)|$. When the set of distinct blocks in any phase fits in HBM (i.e. total at most k), $\eta_A(\Phi)$ gives an upper bound on the total time to bring blocks in from DRAM over all phases in Φ. This is the cost of running the algorithm normally, but artificially emptying HBM at phase boundaries. The system may need to bring in a block once for each phase it participates in. It can be served to its core from HBM multiple times within a single phase. For any instance k-PACKING($\mathcal{I}$), let Φ_1 be the set of contested phases.

Observation 1. *Suppose k-PACKING($\mathcal{I}$) has X_1 contested phases. Then $\eta_{k\text{-PACKING}}(\Phi_1) \geq (3k/4)X_1$.*

Analysis of OPT. Let OPT denote the optimal algorithm for the makespan-minimization problem. As with k-PACKING, we divide the execution OPT($\mathcal{I}$) on instance $\mathcal{I}$ into ***phases***. Each phase has a fixed length of exactly $k/4$ time steps (except possibly the last phase). Thus, Phase 1 contains the requests serviced in the first $k/4$ time steps, Phase 2 contains the requests serviced in the next $k/4$ time steps, and so on.

Observation 2. *Suppose OPT($\mathcal{I}$) runs in Y phases. Then its makespan is at most $(k/4)Y$ and at least $(k/4)(Y-1)+1$.*

We now compare the number of phases in OPT($\mathcal{I}$) versus k-PACKING($\mathcal{I}$).

Lemma 6. *Suppose k-PACKING($\mathcal{I}$) has X_2 uncontested phases and OPT($\mathcal{I}$) has Y phases. Then $Y \geq X_2$.*

Proof. From Lemma 4, in a uncontested phase, all the unfinished threads execute a chunk, which contains $k/4$ block requests. Let p_i be a thread that executes during the last uncontested phase. This implies that in all X_2 phases, p_i executes a chunk. As OPT can execute at most $k/4$ requests of p_i in each phase (length of each phase in OPT is $k/4$), OPT needs at least X_2 phases to serve thread p_i. □

Let Φ_{OPT} be the set of phases for the optimal algorithm for some instance $\mathcal{I}$. OPT has only one kind of phase. Since the algorithm is clear from context, use the shorthand $\eta(\Phi_{OPT})$ instead of $\eta_{OPT}(\Phi_{OPT})$.

Lemma 7. $\eta_{k\text{-PACKING}}(\Phi_1) \leq 2\eta(\Phi_{OPT})$.

Proof. Consider an arbitrary phase ϕ of OPT. Let B be a block that is requested by thread p at least once during phase ϕ. Let the first and last requests of block B in phase ϕ be the fth and jth reference to B respectively in thread p's request sequence. All $j - f + 1$ references to Block B in phase ϕ together contribute exactly 1 to $\eta(\Phi_{OPT})$. Let r'_f and r'_j be the references in thread p's request stream corresponding to the fth and jth reference to block B respectively. Because each phase of OPT has $k/4$ time steps, there are at most $k/4$ requests between r'_f and r'_j in thread p. Let ϕ_f be the phase in k-PACKING that contains reference r'_f and let ϕ_g be the next phase of k-PACKING that contains a reference to Block B. Both phase ϕ_f and phase ϕ_g execute $k/4$ requests for thread p (unless phase ϕ_g is the last phase for thread p.). Thus reference r'_j, is either in phase ϕ_f or phase ϕ_g. Thus all references to block B in phase ϕ of OPT are in at most two phases (of any type) in k-PACKING. They contribute at most 2 to $\eta_{k\text{-PACKING}}(\Phi_1)$. Summing over all phases in Φ_{OPT} proves the lemma. □

Lemma 8. *Suppose that* OPT *has* Y *phases. Then* $\eta(\Phi_{OPT}) \leq (5k/4)Y$.

Proof. Because a Phase ϕ of OPT has length $k/4$ (except a truncated last phase), at most $k/4$ blocks can be transferred from DRAM to HBM. In addition, there are at most k distinct blocks already present in HBM at the start of Phase ϕ. These can be accessed in parallel by the threads. Thus, altogether, in the phase, at most $5k/4$ distinct blocks can be accessed. Summing over all Y phases proves the lemma. □

Lemma 9. *Suppose that* k-PACKING *has* X_1 *contested phases and* OPT *has* Y *phases. Then* $X_1 \leq (10/3)Y$.

Proof. From Lemma 7, we know $\eta_{k\text{-PACKING}}(\Phi_1) \leq 2\eta(\Phi_{OPT})$. From Observation 1, $(3k/4)X_1 \leq \eta_{k\text{-PACKING}}(\Phi_1)$ and from Lemma 8, $\eta(\Phi_{OPT}) \leq (5k/4)Y$. Combining these, we get $X_1 \leq (10/3)Y$. □

PROOF OF LEMMA 3: Suppose that k-PACKING has X_1 contested and X_2 uncontested phases. Let $T(\mathcal{A})$ denote the makespan of an algorithm $\mathcal{A}$. Then from Lemma 5,

$$T(k\text{-PACKING}) \leq \frac{5k}{4}X_1 + kX_2.$$

Suppose OPT has Y phases. Then from Lemma 9, $X_1 \leq (10/3)Y$ and from Lemma 6, $X_2 \leq Y$. Combining these, we get the following.

$$T(k\text{-PACKING}) \leq \frac{5k}{4}\frac{10}{3}Y + kY.$$

However, from Observation 2, $T(\text{OPT}) \geq (k/4)(Y-1)$. Thus,

$$\begin{aligned} T(k\text{-PACKING}) &\leq \frac{5k}{4}\frac{10}{3}(Y-1) + \frac{50k}{12} + 4\frac{k}{4}(X_2 - 1) + k \\ &\leq \frac{50}{3}T(\text{OPT}) + 4T(\text{OPT}) + \frac{62k}{12}. \end{aligned}$$

Hence, the lemma follows as $T(k\text{-PACKING}) = O(T(\text{OPT}))$.

□

3.4.2 Online algorithm

In this section we introduce an online algorithm PRIORITY, which automatically guarantees that its execution on an instance $\mathcal{I}$, PRIORITY($\mathcal{I}$), has some of the structural properties that k-PACKING($\mathcal{I}$) does. Unlike k-PACKING, PRIORITY does not need to know the HBM size k or any future requests from the request sequences.

Specifically, the k-PACKING algorithm guarantees that in a phase (1) there are $\Theta(k)$ steps, (2) either the working set size is $\Theta(k)$ or every unfinished thread makes $\Theta(k)$ progress, (3) any thread that makes progress (without finishing) completes $\Theta(k)$ requests.

What makes the HBM model algorithmically interesting is the bandwidth bottleneck between HBM and DRAM. Given this bottleneck, the algorithmic concern/challenge is how to break ties when multiple block requests compete for the limited bandwidth. In the offline setting, we managed the tie-breaking issue by using size-$\Theta(k)$ chunks, but PRIORITY does not know k.

FCFS (First-Come-First-Serve) is a naturally fair policy for managing DRAM accesses. As we show in the next section, FCFS works poorly, at least when paired with an LRU page-replacement policy —see Section 3.5.

A better idea, at least when using LRU page replacement, is to assign priorities to threads, so that a high priority thread can never be blocked by a low-priority thread. This leads to constant competitiveness. Specifically, we prioritize block requests based on *which thread* made the request, with the highest-priority thread granted access to the DRAM. The specific priority order does not matter. With this far-channel arbitration policy, PRIORITY naturally does what we explicitly designed k-PACKING to do. Moreover, unlike most caching problems, resource augmentation is not necessary for constant competitiveness.

Let $R^i = r^i_1, r^i_2, r^i_3, \ldots$ be thread p_i's request sequence. Suppose that at time step t of some algorithm execution, thread p_i has served all requests through r^i_{j-1}. Let $u_i(t) = r^i_j$ denote thread p_i's first unserviced request at time t. We partition the threads into two sets, $P(t)$ and $\overline{P(t)}$ as follows. If r^i_j is in the HBM at the start of time step t, then $p_i \in P(t)$, and otherwise $p_i \in \overline{P(t)}$. At time 0, the HBM is empty, and hence for all i, $p_i \in \overline{P(0)}$.

Algorithm PRIORITY**.** We assign a fixed priority for each thread. Without loss of generality, say that thread p_i has priority i, where priority 1 is the highest.

In each time step t and for each thread p_i:

1. If $p_i \in P(t)$, then block $u_i(t)$ is transferred to p_i's core.
2. Otherwise, $p_i \in \overline{P(t)}$.
   (a) If p_i is the highest priority thread among the threads in $\overline{P(t)}$, then $u_i(t)$ is transferred from DRAM to HBM. If the HBM is full, then the least-recently-used block among all the cores (breaking ties arbitrarily) in HBM is replaced.
   (b) Otherwise, p_i ***stalls*** (i.e., waits, since only one block can be fetched from DRAM to HBM in each step).

It takes one time step to transfer a block from HBM to a core or from DRAM to HBM. Thus, if

$p_i \in P(t)$ (conditional in Step 1 holds), then after Step 1 in PRIORITY, $u_i(t+1) = r^i_{j+1}$. Otherwise, after PRIORITY executes Step 2, $u_i(t+1) = r^i_j$.

Analysis of PRIORITY. For the analysis (only), we divide PRIORITY's execution on instance $\mathcal{I}$, denoted PRIORITY($\mathcal{I}$), into phases of length k. Phase ϕ (for $\phi \geq 1$) begins at the start of time step $(\phi-1)k+1$ and finishes at the end of step ϕk. We say thread p_i is ***productive in Phase ϕ*** if and only if p_i serves at least $k/4$ requests in Phase ϕ or finishes the thread's execution. Otherwise, p_i is ***unproductive in Phase ϕ.***

Definition 9. *Let $\mathcal{B}(\phi)$ denote the set of distinct blocks that* PRIORITY($\mathcal{I}$) *serves in Phase ϕ. There are two types of phases:*

> **Contested:** $k/2 \leq |\mathcal{B}(\phi)| \leq k$ *or*
> **Uncontested:** $|\mathcal{B}(\phi)| < k/2$.

The following lemma shows uncontested phases are productive.

Lemma 10. *For any uncontested phase in* PRIORITY($\mathcal{I}$)*, all unfinished threads serve at least $k/4$ requests.*

Proof. Since there are fewer than $k/2$ distinct blocks accessed in Phase ϕ, then there are at most $k/2$ steps when any block is brought in from DRAM. Thus, since a phase has k time steps, there are at least $k/2$ steps when no thread needs a block that is not in HBM. During each of these steps, any active thread serves a page request from its sequence or finishes. □

The next two lemmas show that for every Phase ϕ, PRIORITY($\mathcal{I}$) satisfies the following two conditions:

- Each phase in PRIORITY($\mathcal{I}$) has at least one productive thread.
- In every contested phase, the productive threads alone serve at least $k/2$ distinct blocks.

Lemma 11. *Every phase of* PRIORITY($\mathcal{I}$) *has at least one productive thread.*

Proof. Let thread p_i have the highest priority among the threads that run in Phase ϕ. If p_i finishes its execution in Phase ϕ, then by definition, it is a productive thread. Otherwise, in each time step, a block of p_i is transferred from DRAM to HBM or from HBM to p_i's core. Being the highest priority thread, p_i never stalls. Since the phase has length k, thread p_i services at least $k/2$ block requests or finishes, making it a productive thread. □

Lemma 12. *For every contested phase ϕ, the productive threads access $g(\phi)$ distinct blocks, where $|\mathcal{B}(\phi)| \geq g(\phi) \geq k/2$.*

Proof. For ease of presentation, rename the threads that have not completed executing, so that p_1 is the highest priority thread, p_2 is the next highest priority thread, and so on.

We say that a thread p_i is ***active in step t*** if it is not stalled during the step. Thus, a block for p_i is transferred either from DRAM to HBM or from HBM to p_i's core. Say that thread p_i accesses h_i distinct blocks in Phase ϕ.

Thread p_1 is never stalled. Thus, it is active for k steps unless it finishes executing before the phase ends.

Thread p_2 can be stalled for at most h_1 time steps. This is because p_1 grabs the DRAM-to-HBM channel at most h_1 times during the phase. (Thread p_1 might grab the channel *fewer* than h_1 times, since the requested blocks might already be in HBM from a previous phase.) Thus, p_2 is active for at least $k - h_1$ steps unless it finishes executing before the phase ends.

Similarly, p_3 can be stalled for at most $h_1 + h_2$ time steps. This is because p_3 is only stalled when either p_1 or p_2 grabs the DRAM-to-HBM channel. In general, p_i can be stalled for at most $\sum_{j=1}^{i-1} h_j$ steps, and thus is active for $k - \sum_{j=1}^{i-1} h_j$ steps unless it finishes before the phase ends.

Let ℓ be the lowest priority thread in Phase ϕ such that $\sum_{j=1}^{\ell} h_j \geq k/2$.

We show that for all $i = 1 \ldots \ell$, thread p_i is productive. For $i = 1 \ldots \ell$, thread p_i is active for at least $k - \sum_{j=1}^{\ell-1} h_j \geq k/2$ steps unless it finishes earlier. If a thread finishes earlier, then by definition, it is productive. Otherwise, if a thread is active for $k/2$ steps, then it must serve at least $k/4$ block requests in its sequence, since it takes two steps to bring a block from DRAM to a thread's core. Hence, all ℓ threads are productive threads.

Thus, together all the productive threads access $\sum_{j=1}^{\ell} h_j \geq k/2$ distinct blocks, establishing the lemma. □

For a phase ϕ, let $\mathcal{B}^*(\phi)$ denote the set of distinct blocks requested by all the productive threads in phase ϕ. Let Φ_1 denote the set of contested phases. Similar to our previous notation, let $\eta^*_{\text{PRIORITY}}(\Phi_1) \equiv \sum_{\phi \in \Phi_1} |\mathcal{B}^*(\phi)|$.

Corollary 1. *Let* PRIORITY *have Z_1 contested phases. Then $\eta^*_{\text{PRIORITY}}(\Phi_1) \geq (k/2)Z_1$.*

Analysis of OPT**.** We divide OPT's execution on instance $\mathcal{I}$ into ***phases***. Each phase has a fixed length of exactly $k/4$ time steps (except possibly the last phase).

We now compare the number of phases in OPT versus PRIORITY. Our proof of the following lemma is similar to that of Lemma 7. However, since PRIORITY can not pack and execute chunks the way the offline algorithm k-PACKING does in a phase, we base the proof on the productive threads in a fixed-length phase. Later we show that considering only productive threads is enough to establish the constant-competitiveness of PRIORITY.

Lemma 13. $\eta^*_{\text{PRIORITY}}(\Phi_1) \leq 2\eta(\Phi_{OPT})$.

Proof. Consider an arbitrary phase ϕ of OPT. Let B be a block that is requested by thread p at least once during phase ϕ. Let the first and last requests of block B in phase ϕ be the fth and jth reference to B respectively in thread p's request sequence. All $j - f + 1$ references to Block B in phase ϕ together contribute exactly 1 to $\eta(\Phi_{OPT})$. Let r'_f and r'_j be the references in thread p's request stream corresponding to the fth and jth reference to block B respectively. Because each phase of OPT has $k/4$ time steps, there are at most $k/4$ requests between r'_f and r'_j in thread p. Let ϕ_f be the phase in PRIORITY that serves reference r'_f. Let ϕ^*_f be phase ϕ_f if thread p is productive in ϕ_f. Otherwise, let ϕ^*_f be the first phase after ϕ_f in PRIORITY where thread p is productive and it serves a reference to block B in $[r'_f, \ldots, r'_j]$. Let ϕ_g be the next phase of PRIORITY after phase ϕ^*_f that serves a reference to block B where thread p is productive. Both phase ϕ^*_f and phase ϕ_g execute $k/4$ requests for thread p (unless phase ϕ_g is the last phase for thread p.). Thus all the fth through jth request to block B are served by PRIORITY in phase ϕ^*_f or in phase ϕ_g, or in phases where thread p is not productive. Only phases ϕ^*_f and ϕ_g might contribute counts to $\eta^*_{\text{PRIORITY}}(\Phi_1)$, and together they contribute at most 2 to $\eta^*_{\text{PRIORITY}}(\Phi_1)$. Summing over all phases in Φ_{OPT} proves the lemma. □

Lemma 14. *Suppose* PRIORITY *has Z_1 contested phases and* OPT *has Y phases. Then $Z_1 \leq 5Y$.*

Proof. From Lemma 13, $\eta^*_{\text{PRIORITY}}(\Phi_1) \leq 2\eta(\Phi_{OPT})$. Also, from Corollary 1, $\eta^*_{\text{PRIORITY}}(\Phi_1) \geq (k/2)Z_1$ and from Lemma 8, $\eta(\Phi_{OPT}) \leq (5k/4)Y$. Combining these, we get $Z_1 \leq 5Y$. □

Lemma 15. *Suppose* PRIORITY *has Z_2 uncontested phases and* OPT *has Y phases. Then $Z_2 \leq Y$.*

Proof. The proof is essentially the same as that for Lemma 6. □

PROOF OF THEOREM 9: Suppose that PRIORITY has Z_1 contested and Z_2 uncontested phases. Then PRIORITY's makespan $T(\text{PRIORITY})$ satisfies the following:

$$T(\text{PRIORITY}) \leq k(Z_1 + Z_2).$$

Suppose OPT has Y phases. Using the results from Lemma 14 and Lemma 15, we get the following:

$$T(\text{PRIORITY}) \leq 6kY.$$

However, from Observation 2, $T(\text{OPT}) \geq (k/4)(Y-1)$. The theorem follows, since

$$T(\text{PRIORITY}) \leq 24T(\text{OPT}) + 6k = O(T(\text{OPT})).$$

□

Why do we use LRU for block replacement? Many arguments in proofs above assume that all references to a block within the same phase cost at most one access to DRAM. This assumes that once a block is brought in during a phase, it will stay in HBM for the rest of the phase. Thus future accesses to that block in the same phase are HBM hits. LRU is one way to ensure that new blocks coming in do not knock out very-recently-fetched blocks. Our phases have at most k distinct blocks. Thus the blocks brought in during the phase can all fit into HBM without evicting each other when using LRU.

3.5 FCFS with LRU is not a Good Policy in the HBM Model

In this section we consider a very natural contention-resolution policy FCFS for the DRAM-HBM channel with a widely used natural block replacement policy LRU. We show that the contention-resolution policy FCFS with the block replacement policy LRU (let us call it FCFS+LRU) works very poorly in the HBM model. We prove the following theorem.

Theorem 10. *There exists request sequences such that even with d memory augmentation and s bandwidth augmentation, the makespan of FCFS+LRU is $\Theta(\frac{p}{ds})$-factor away from that of the optimal policy.*

We give FCFS+LRU d-memory augmentation, that is, FCFS+LRU has an HBM of size k, whereas OPT has an HBM of size k/d. We assume that d divides k. Given the memory augmentation d, we set $k = 2pd$ where p is the number of cores. We could have chosen any larger value of k. Larger values of k capture reasonable HBM sizes and we would obtain the same lower bound. The request sequences would need to be updated accordingly.

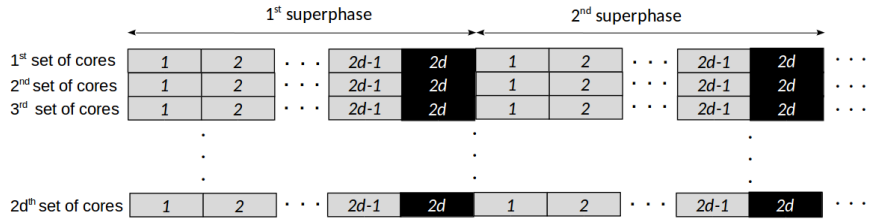


Figure 3.2: The execution of FCFS+LRU. Boxes with grey color represent light phases and boxes with black color represents heavy phases.

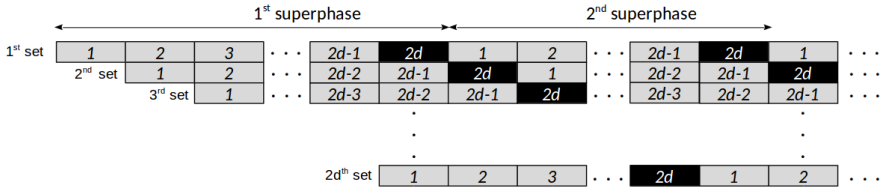


Figure 3.3: The execution of OPT. The sequences are shifted to align such that at most one set of cores run their heavy phases simultaneously.

Proof.

$$R^i = \Big(\overbrace{x^i_1, x^i_1, \ldots, x^i_1}^{\ell+i}, \ldots, \overbrace{x^i_{p-1}, x^i_{p-1}, \ldots, x^i_{p-1}}^{\ell+i}, \underbrace{\overbrace{x^i_1, x^i_2, \ldots, x^i_{k/p+1}, x^i_1, x^i_2, \ldots, x^i_{k/p+1}, \ldots}^{\ell+i}}_{\text{1 heavy phase}} \Big)^{\lambda}.$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{2d-1 \text{ light phases}}$$

The request sequence of core p_i is divided into phases of length $\ell + i$. There are two types of phases: light phases and heavy phases. In a light phase a single block is requested for $\ell + i$ times. In a heavy phase $k/p + 1$ blocks are requested in round-robin fashion until length ℓ and then the last requested block is repeated for i times. The additive term i in the length of a phase ensures synchronization among the cores in the execution of FCFS+LRU. If all p cores start executing their corresponding light phases at the same time, then all finish at the same time. Similarly, if all p threads start executing their corresponding heavy phases, then all finish at the same time. There are $2d - 1$ light phases followed by a heavy phase. Together these $2d$ phases are called a superphase. Request sequence R^i is the concatenation of such a super phase for λ times.

We divide the cores into $2d$ sets each containing $\frac{p}{2d}$ cores. We assume that $2d$ divides p. Let $\mathcal{P}_1 = \{p_j | 1 \leq j \leq \frac{p}{2d}\}$ denote the first set of $\frac{p}{2d}$ cores, and $\mathcal{P}_i = \{P_j | i\frac{p}{2d} \leq j \leq (i+1)\frac{p}{2d}\}$ denote the i-th set of $\frac{p}{2d}$ cores.

OPT has enough space in its HBM to hold a block of a light phase from each core in $(2d - 1)$ sets and $k/p + 1$ blocks of a heavy phase from each core in one set. Recall that each set has $p/2d$ cores.

$$\left(\frac{k}{p} + 1\right)\frac{p}{2d} + (2d - 1)\frac{p}{2d} = \frac{k}{2d} + p.$$
$$= k/d \text{ putting } k = 2pd.$$

The execution of FCFS+LRU. As FCFS+LRU cannot see the future, it starts executing all the cores simultaneously. All p cores start and finish their corresponding light phases at the same

time. Similarly, all the p cores start and finish their corresponding heavy phases at the same time. See Figure 3.2. As the total number of blocks of p heavy phases is larger than than size of HBM, every request in the heavy phase incurs an HBM miss. Hence, all the heavy phases run serially, thus taking at least $p\ell$ time steps. As there are λ superphases and each superphase has one heavy phase, FCFS+LRU takes at least $T(\text{FCFS+LRU}) = p\ell\lambda$ time steps to execute the whole program.

The execution of OPT. The optimal policy OPT aligns the request sequences such that no two cores run their heavy phases simultaneously. OPT does so by shifting the starting time of cores. It starts executing cores in set $\mathcal{P}_2$ when cores in set $\mathcal{P}_1$ start their 2-nd phase. Similarly, OPT starts executing cores in set $\mathcal{P}_3$ when cores in set $\mathcal{P}_2$ start their 2-nd phase. Note that cores in set $\mathcal{P}_1$ start their 3-rd phase at the same time. In general, cores in set $\mathcal{P}_{i+1}$ start when cores in set $\mathcal{P}_i$ start their 2-nd phase for $1 \leq i \leq 2d-1$. After this initial alignment, only $p/2d$ cores from one set run their heavy phase and all other cores run their corresponding light phases. As the HBM of OPT has enough space to hold all the blocks of a heavy phases from $p/2d$ cores each and a light phase from each of the rest of the cores, all the cores can progress simultaneously in every phase. See Figure 3.3.

OPT runs heavy phases from at most one set of $p/2d$ cores and rest of the cores runs light phases. Together, k/d blocks are fetched in k/d time steps and once all the blocks are in HBM, all p cores can run their phases in parallel, taking $l+p$ time at most. Hence, OPT finishes a phase in $(k/d + \ell + p) = \ell + 3p$ time steps as $k = 2pd$. Recall that a superphase has $2d$ phases and there are λ such superphases in each request sequence. As OPT shifts the starting time of the request sequences, OPT effectively runs $\lambda + 1$ superphases. OPT finishes a superphase in $(\ell + 3p)2d$ time steps. Choosing an appropriate value of ℓ with respect to p, we get $(\ell + 3p)2d \leq 3\ell d$. The whole program finishes in $T(\text{OPT}) = 3\ell d(\lambda + 1) \leq 4\ell d\lambda$ time steps.

Competitive ratio of FCFS+LRU. Applying the makespan of FCFS+LRU and OPT, we get the following competitive ratio of FCFS+LRU.

$$\frac{T(\text{FCFS+LRU})}{T(\text{OPT})} \geq \frac{p\ell\lambda}{4\ell d\lambda} = \frac{p}{4d}$$

If FCFS+LRU gets s bandwidth augmentation, then its running time is reduced by at most a factor of s as between HBM and DRAM s blocks can be transferred in one time step. Hence, the competitive ratio of FCFS+LRU becomes $\Theta\left(\frac{p}{sd}\right)$ and the theorem is proved.

□

3.6 NP-hardness of the Makespan-minimization Problem

In this section we show that the offline makespan-minimization problem is strongly NP-hard. Our proof is based on a polynomial-time reduction from the strongly NP-hard problem 3-partition.

3-partition. Given a set $A = \{a_1, a_2, \cdots, a_{3n}\}$ of $3n$ integers such that $\sum_{i=1}^{3n} a_i = nB$ and $B/4 < a_i < B/2$ for each $1 \leq i \leq 3n$, can $I = \{1, 2, \cdots, 3n\}$ be partitioned into disjoint sets $I_1, I_2, \cdots, I_n$, such that $\sum_{i \in I_j} a_i = B$ for each $1 \leq j \leq n$?

Reduction. Given an instance of the 3-partition problem with $3n$ integers $\{a_1, a_2, \cdots, a_{3n}\}$ and target sum B for each subset, we create an instance of the makespan-minimization problem as

follows. For each integer a_i, we create a request sequence $R^i = \left(r^i_1, r^i_2, \ldots, r^i_{a_i}\right)^{\lfloor 4B/a_i \rfloor}$, that is R^i is formed by repeating $\left(r^i_1, r^i_2, \ldots, r^i_{a_i}\right)$ for $\lfloor 4B/a_i \rfloor$ times. Recall that r^i_j denotes the j-th request in core p_i's request sequence. Therefore, all the $3n$ request sequences together have nB distinct blocks. We also create two auxiliary request sequences T_1 and T_2 as follows.

Sequence T_1 has length $(3nB + n + 1)$ where the first $3nB + n$ requests are all distinct, but then the last request is a repeat of the penultimate request.

$$T_1 = b_1, b_2, b_3, \ldots, b_{3nB+n-1}, b_{3nB+n}, b_{3nB+n}.$$

Sequence T_2 is a concatenation of n rounds. Each round consists of two consecutive phases. In the first phase of a round, T_2 requests the same block for $(2B + 1)$ times. In second phase, it requests $2B$ distinct blocks. Thus, each round is of length $(4B+1)$ and requests $(2B+1)$ distinct blocks. Let $x_i = (2B + 1)i$ where $0 \leq i < n$. Then,

$$T_2 = \underbrace{\overbrace{d_1, d_1, \ldots, d_1}^{2B+1}, \overbrace{d_2, d_3, \ldots d_{2B+1}}^{2B}}_{\text{Round } 1}, \ldots,$$
$$\underbrace{\overbrace{d_{x_i+1}, d_{x_i+1}, \ldots, d_{x_i+1}}^{2B+1}, \overbrace{d_{x_i+2}, d_{x_i+3}, \ldots, d_{x_i+2B+1}}^{2B}}_{\text{Round } i}, \ldots$$

There are $(3n + 2)$ cores and the HBM size is $(B + 2)$. Sequence T_1 and T_2 have $(3nB + n)$ and $(2nB + n)$ distinct block requests respectively. Sequence $R^1, R^2, \ldots, R^{3n}$ together have nB distinct block requests. The total number of distinct blocks is $(6nB + 2n)$.

Theorem 11. *An instance of 3-partition has a solution if and only if the derived makespan-minimization problem has a makespan of* $(6nB + 2n + 1)$.

Before proving Theorem 11, we first show some properties of any schedule of the derived makespan-minimization problem instance that has a makespan of $6nB + 2n + 1$.

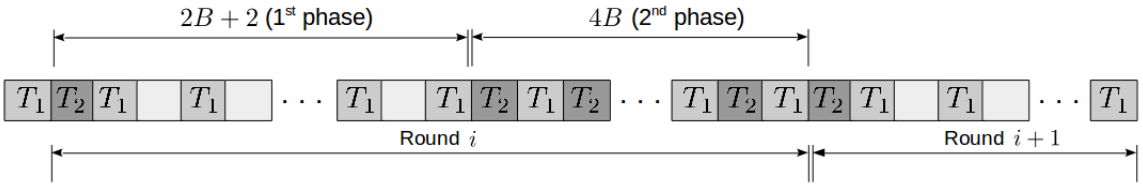


Figure 3.4: Each box represents a time step. Boxes with labels $\boldsymbol{T_1}$ and $\boldsymbol{T_2}$ denote that cores $\boldsymbol{T_1}$ and $\boldsymbol{T_2}$ fetch a block from DRAM respectively. A box without any label denotes that one of the $\boldsymbol{3n}$ cores fetches a block from DRAM. The first phase has $\boldsymbol{B}$ boxes without labels. Each box in the second phase is labelled by either $\boldsymbol{T_1}$ or $\boldsymbol{T_2}$.

Lemma 16. *Suppose that there is a schedule $\mathcal{S}$ with makespan $6nB + 2n + 1$. Then $\mathcal{S}$ fetches a block from DRAM to HBM in every time step except the last one.*

Proof. There are a total of $(6nB + 2n)$ distinct blocks. Hence, there must be at least $(6nB + 2n)$ HBM misses by any cache-replacement policy. If the target makespan is $(6nB + 2n + 1)$, the channel between the HBM and DRAM must always be busy except for the last time step (used for transferring the last block from HBM to a core). □

Observation 3. *Suppose that there is a schedule $\mathcal{S}$ of the derived makespan-minimization problem instance with makespan $(6nB + 2n + 1)$. Then $\mathcal{S}$ cannot evict a block from HBM unless the block is not requested in the future.*

Lemma 17. *The schedules of the auxiliary cores T_1 and T_2 are fixed in every solution of the derived makespan-minimization problem instance that has a makespan of $(6nB + 2n + 1)$.*

Proof. The target makespan is $(6nB + 2n + 1)$ and the last time step is used to serve a block from HBM. Hence, the first $(6nB + 2n)$ time steps can be used to fetch blocks from DRAM to HBM. Since T_1 requests $(3nB + n)$ distinct blocks, at every alternative time slot, T_1 must use the HBM-DRAM channel. In particular, T_1 must fetch a block in the first time step. Otherwise, it can not finish within the target makespan while fetching $(3nB + n)$ distinct blocks and serving the last distinct block twice. Hence, the schedule of T_1 is fixed: it fetches every odd timestep.

Similarly, the schedule of T_2 is fixed. In the first time step, thread T_1 brings in a block from DRAM. We show that in each of the remaining $(6nB + 2n)$ time steps, thread T_2 either brings in a block from DRAM or serves a block from HBM to its core. Recall that T_2 has n rounds. The first phase of each round takes $(2B + 2)$ steps because in the first step it brings a block from DRAM and serves the same block for $(2B + 1)$ steps. In the second phase, T_2 fetches $2B$ distinct blocks in every alternating step of $4B$ time steps. Hence, each round takes $6B+2$ steps. All n rounds are finished in $6nB+2n$ steps. Since T_2 starts executing at the second time step, it finishes execution in time $6nB + 2n + 1$ (Figure 3.4). □

Let $\mathcal{S}$ be a schedule that has makespan $(6nB + 2n + 1)$. The schedule of T_1 and T_2 are fixed. Except for the first time step, core T_2 is either fetching a block from DRAM or serving a block from HBM to the core. We say that when T_2 finishes round i, schedule $\mathcal{S}$ finishes round i. When T_2 finishes the first phase of round i, schedule $\mathcal{S}$ finishes the first phase of round i and similarly for phase 2 of round i.

Observation 4. *The auxiliary cores T_1 and T_2 always occupy one block each in HBM.*

Observation 5. *Let $\mathcal{S}$ be a schedule that has makespan $(6nB + 2n + 1)$. Then the first phase of each round of $\mathcal{S}$ has B time steps when the DRAM-HBM channel is used by neither T_1 nor T_2.*

Lemma 18. *Let $\mathcal{S}$ be a schedule that has makespan $(6nB + 2n + 1)$. Then no core besides T_1 and T_2 runs in two rounds of $\mathcal{S}$.*

Proof. We prove this by contradiction. Let core p run in rounds i and j. Core p can fetch blocks only during the first phase of round i. Since it continues to round j, it could not fetch all its blocks in the first phase. Otherwise, it could have finished its round-robin phase in the second phase of round i. That means some of the blocks that are fetched in round i, will be used in round j in the round-robin phase. From Observation 3, we know that once a block is fetched, it can not be kicked out unless it is not requested later. This implies that some blocks in HBM must be held for core p in round j.

There are B free time slots in round j and the HBM has B blocks left for cores other than T_1 and T_2. If some block is already occupied by some core at round j, then at least one time slot cannot bring in a block during the first phase of round j. This contradicts Lemma 16. □

Lemma 19. *Let $\mathcal{S}$ be a schedule that has makespan $(6nB+2n+1)$. Then exactly three cores besides T_1 and T_2 can run in a round of $\mathcal{S}$.*

Proof. No core besides T_1 and T_2 runs in two rounds. The core associated with integer a_i accesses a_i distinct blocks, so during the round where that core runs, it must read in a_i blocks during the B time slots when neither thread T_1 nor thread T_2 are accessing the DRAM. Since $B/4 < a_i < B/2$ for all a_i, at most three cores can share a round. Since there are $3n$ cores associated with integers a_i and only n blocks, then at least three blocks must run in any round. □

Proof of Theorem 11. Suppose that there is a solution to the 3-partition instance. The solution is a partition of $3n$ integers into n disjoint sets such that the sum of the integers in each set is B. We create a schedule of the derived makespan-minimization problem instance using the solution of the 3-partition instance. We schedule T_1 and T_2 as shown in Figure 3.4. There are n rounds in the schedule. The first phase in each round has B time-steps when neither T_1 nor T_2 use the HBM-DRAM channel. If the first set in the solution of 3-partition has integers a_i, a_j and a_k, we schedule core p_i, p_j and p_k in the first round. Recall that p_i has a_i distinct blocks. These three cores fetch a total of B blocks in the first phase of the first round and they execute the remaining round-robin accesses to these blocks by the end of the second phase of the round. The second phase has length $4B$ and the length of the request sequence of each core representing a_i is at most $4B$. Hence, p_i, p_j and p_k finish their execution in the second phase. Similarly, for each set in the 3-partition solution, we schedule the cores accordingly. As the n rounds finish in time $6nB + 2n + 1$, we have the target makespan.

Now suppose that there is a makespan of $(6nB + 2n + 1)$. Then from lemma 19, exactly three cores can run in a round. This gives a mapping from $3n$ cores to n rounds. As three cores are running in a round and total free slots (also total number of distinct blocks in these three cores) are B, this gives a 3-partition solution. Hence, *minimize-makespan* is strongly NP-hard. □

3.7 Performance Metric in HBM Model

In this section we show that minimizing a traditional scheduling performance metric like makespan is better for the HBM model than minimizing the number of HBM misses. In particular we show there exist request sequences where any policy that minimizes the number of HBM misses can have arbitrarily bad running time. We prove the following theorem.

Theorem 12. *There exist p request sequences such that any policy that minimizes the number of HBM misses when serving the sequences has a makespan that is a $\Theta(p)$ factor larger than the optimal makespan.*

Proof. There are p cores and the HBM has size k. The request sequence R^i for core p_i has length $n + 2k$ and uses k distinct blocks. The first n requests are np/k round-robin repetitions of k/p blocks. Then there are two round-robin repetitions of all k blocks. We call the last two length-k subsequences ***large passes***.

$$R^i = \overbrace{\underbrace{x_1^i, x_2^i, \ldots x_{k/p}^i}_{k/p}, \ldots \underbrace{x_1^i, x_2^i, \ldots x_{k/p}^i}_{k/p}}^{n}, \overbrace{\underbrace{x_1^i, x_2^i, \ldots x_k^i}_{k}, \underbrace{x_1^i, x_2^i, \ldots x_k^i}_{k}}^{2k}.$$

Because each core requests exactly k blocks in total, which exactly fills the size-k HBM, the fewest possible HBM misses is kp. This is achievable, for example, by running the request sequence for each core serially. Thus any policy that minimizes the HBM misses cannot evict a block once it is fetched from DRAM to HBM until its last reference is executed. Otherwise, there are at least $kp + 1$ HBM misses, which is not optimal.

The makespan of a single thread (i.e. when $p = 1$) is $n + 3k$. Serving the first k/p requests requires two time steps per element: one to bring the block in from DRAM and another to serve it from HBM to the core. The next n requests are HBM hits, so require one step each. These are the last $n - k/p$ requests in the first part of the sequence and the first k/p requests in the first large pass. Serving the remaining $k - k/p$ blocks in the first large pass requires two ticks each, and the final large pass requires k time steps. In all that is $2k/p + n + 2(k - k/p) + k = n + 3k$.

The last time each block is accessed by its core is during the last large pass. For a minimum-miss execution, if the threads execute in order, cores p_i and p_{i+1} can overlap for at most k time steps. Core p_{i+1} can bring in its first block when x_1^i is accessed for the last time at the start of the last large pass for R^i. In general the execution of R^i can overlap at most k with both R^{i-1} and R^{i+1}. So each sequence overlaps for $2k$ timeslots, except the first and last, which overlap for only k time slots. Let MinimumMisses represent any policy that minimizes the number of HBM misses. Then we have.

$$T(\text{MinimizeMisses}) \geq p(n + k) + 2k.$$

There is a feasible policy that runs faster for sufficiently large n. It brings the first k/p blocks for each thread into the HBM. Because threads can interleave, this requires time at most k. Then it can execute all p threads in parallel for their first n block requests with no HBM misses. The last two rounds for each core are almost serialized. Each core requires $2k$ time to bring in its k blocks, interleaved with reading the blocks the first time. There is one more time step to reread the first block before the next core can start bringing in its blocks. Thus each core controls the DRAM channel for $2k + 1$ time steps, and there are k steps at the end for the last sequence to finish. This is a loose analysis, since threads can start executing the first round robins as soon as their blocks are in and the first core can start bringing in the rest of its first large pass as other threads are finishing their first round robins. Still we have an upper bound on the time to execute this strategy:

$$T(\text{OPT}) \leq 2k + n + p(2k + 1).$$

Setting $n = p(2k + 1)$ suffices to show

$$T(\text{MinimizeMisses}) \geq \frac{p}{2} T(\text{OPT}).$$

Hence, the makespan of MinimizeMisses is a $\Theta(p)$-factor larger than the optimal makespan. □

3.7.1 How does uneven bandwidth affect makespan?

In this subsection, we show that how difference in bandwidth between HBM and DRAM makes the problem more interesting. In particular, we show that if the bandwidth of HBM and DRAM are the same, then by partitioning the HBM evenly among the cores, any reasonable policy achieves a makespan within a constant factor of optimal. However, when the bandwidth of DRAM and HBM are not the same, then there exists request sequences such that when the HBM is partitioned evenly, no block replacement policy can achieve a makespan within a constant factor of optimal.

Lemma 20. *Suppose that the bandwidths of HBM and DRAM are the same. Then any block replacement policy can achieve a 2-competitive makespan by partitioning the HBM evenly among the cores.*

Proof. We prove a much stronger claim that allocates a single block per core in the HBM and still have a 2-competitive makespan.

As the bandwidth of near and far memory is the same, we can model it as follows. From each core p_i $(1 \leq i \leq p)$ there is a channel to the near memory and from the near memory there are p channels to the far memory. Let R^i denote the request sequence of core P_i and ℓ_i denote the length of R^i, that is $\ell_i = |R^i|$. Then the OPT's makespan T_{OPT} would be at least $\max_{1\leq i\leq p}(\ell_i)$.

$$T_{\text{OPT}} \geq \max_{1\leq i\leq p}(\ell_i)$$

Now, let the strategy allocate one block for each core p_i in the near memory. This is always possible because the size of the near memory is at least p blocks. Now for each block request $r_j^{(i)}$ from core P_i, the cache replacement policy fetch the block from far memory and evict current block $r_{j-1}^{(i)}$ if present in the HBM. Hence, in one time step, the block is fetched from far memory and in the next time step, it is transferred to p_i. Thus, R^i will be executed by time $2\ell_i$. This gives a $2-$competitive makespan. As other cache replacement policies cannot be worse than this (it gets a HBM miss at every request), they are also at least 2-competitive. □

Lemma 21. *Suppose that the bandwidths of HBM and DRAM are not the same. Then there exists request sequences where by partitioning the HBM evenly among the cores, no block replacement policy, even in the offline setting, can achieve a makespan that is within a constant factor of optimal.*

Proof. Let the request sequence of core p_i be as follows.

$$R^i = (x_1^i, x_2^i)^{n/2}.$$

Let the size of HBM is p and the HBM is evenly shared by p cores, that is, every core has a single block to hold in HBM. While fetching a block for core p_i from DRAM, if the alloted place for p_i in HBM is full, a block from the alloted place is evicted to make room for the new block. As core p_i requests two blocks in round-robin fashion and p_i can hold a single block in HBM at any time, every request incurs a HBM miss irrespective to the block replacement policy. Hence, total HBM misses for p cores is np for any arbitrary block replacement policy.

However, instead of dividing the HBM evenly, a block replacement policy executes the first $p/2$ cores in parallel and allocates 2 blocks per core. As each core p_i has only two distinct blocks, $p/2$ cores will finish their execution in time $(p + n)$. This is because there are p HBM misses for

p distinct blocks and once they fit in HBM they do not incur any HBM-misses, thus making the makespan at most $(p + n)$. Similarly, after the first $p/2$ cores finish execution, the last $p/2$ cores also finish execution in another $(p + n)$ time steps. This makes the total makespan of $2(p + n)$.

This shows that when the HBM is partitioned evenly among the cores, no block-replacement policy can achieve a makespan that is constant factor from optimal. □

3.8 Minimizing Total Completion Time Offline

In this section, we consider the problem of minimizing total (average) completion time offline. To do so, we first show a reduction to a resource constrained scheduling problem. Then, we show that the new scheduling problem admits a $O(1)$ approximation via a linear program rounding. The goal of this section is to show the following theorem.

Theorem 13. *There is a polynomial time deterministic algorithm that computes a $O(1)$-approximate solution for the HBM problem for the total completion time objective.*

This theorem will follow from the following. We create a resource constrained scheduling problem and show in Lemma 22 that the total completion time of this new problem is bounded by at most a $O(1)$ of that of the best solution for the HBM problem. Then in Lemma 23, we show how any algorithm for the resource constrained scheduling problem can be converted back to the HBM problem while increasing the completion time by an $O(1)$ factor. Thus, we only need to show a $O(1)$-approximation for the new problem for the theorem to follow. This is shown in Theorem 14.

3.8.1 Reduction to a resource constrained scheduling problem

We reduce the problem into a resource constrained scheduling problem. Recall that in Section 3.4, each sequence $R^i = r_1, r_2, r_3, \ldots$ for each core p_i is divided into ***chunks***, where each chunk C_j (except possibly the last) contains exactly $k/4$ requests. We perform the same chunking of the requests.

$$R^i = \overbrace{r_1, r_2, \cdots, r_{k/4}}^{C_1}, \overbrace{r_{1+k/4}, r_{2+k/4}, \cdots, r_{2k/4}}^{C_2} \cdots$$

In the reduction, request sequences will always be completed in chunks of length $k/4$.

The reduction. We reduce a request sequence R^i to a job J_i as follows. If sequence R^i has w_i chunks, then we say that job J_i consists of w_i tasks. If the jth chunk of the sequence serves $r_{i,j}$ distinct blocks then the corresponding j-th task for job J_i has a resource requirement of $r_{i,j}$. The j-th task can only be processed after all $1, 2, \ldots j-1$ prior tasks for job J_i have completed. Each task takes $k/4$ time steps. A job is completed when all of its tasks are completed.

Reduced problem definition. Suppose that there is a set of p jobs that can be scheduled using k resources. At each point in time, a job can be allocated any number of resources, so long as at most k are assigned to all of the jobs. Job J_i consists of w_i tasks. The jth task for job J_i has a resource requirement $r_{i,j}$ and can only be processed after all $1, 2, \ldots j-1$ prior tasks for job J_i

have completed. All of the tasks are assumed to take unit time to finish. A job is completed when all of its tasks are completed. The goal is to minimize the total completion time of the jobs. We call this problem the **resource constrained scheduling problem (RCSP)**.

Subsequently we show that the optimal solution to this reduced resource constrained scheduling problem is within a constant factor of the optimal solution of the original problem in the HBM model.

Lemma 22. *Given any request sequence in the HBM problem, the total completion time for the RCSP is within a constant factor of the optimal total completion time of the original p request sequences in the HBM model.*

Proof. Let OPT be the optimal solution that minimizes the total completion time of the scheduling instance. Let S_i be all the jobs that complete between times $[2^i, 2^{i+1})$ in OPT. Consider using the makespan algorihtm k-PACKING to schedule *only* the jobs in S_i. This will result is a schedule of makespan at most $c{\cdot}2^{i+1}$ where c is a constant. Consider taking these schedules and concatenating them so S_1 is first then S_2 and so on. Notice that S_i completes at latest $c2^{i+2}$ after concatenating. Let O also denote the optimal solution's objective function and let A denote the total completion time of this schedule. We see that,

$$A \leq \sum_i c|S_i|2^{i+2} = 4c\sum_i |S_i|2^i = 2cO$$

Finally, consider any instance scheduled. The chunks scheduled by k-PACKING correspond to tasks in the RCSP instance. That is k-PACKING always schedules threads in request sequences of length $k/4$. When such a sequence is scheduled, schedule the corresponding task in the RCSP at the same time. The definition of the problem and k-PACKING ensures that at any time at most k resources are scheduled. Thus, we can use this schedule to give a solution to the RCSP problem with the same total completion time. Thus, the total completion time of the optimal solution for the RCSP instance is at most $2c = O(1)$ greater than the HBM problem.

□

Next we show that any schedule for RCSP can be converted to a schedule for the HBM problem with the same total completion time.

Lemma 23. *Consider any HBM instance and the corresponding RCSP instance. Given any schedule for RCSP, one can construct in polynomial time a schedule for the HBM instance where the objective only increases by a factor 5.*

Proof. Each task in the RCSP instance has the same 'unit' length of $k/4$. Consider a set of tasks X_t scheduled during one time step in RCSP instance. Each of these tasks corresponds to a chunk of requests of length $k/4$. By definition of the resource requirement, all of the tasks in X_t request at most k unique pages. We can complete all of these tasks in the HBM instance in $k + k/4$ time steps. k to load the HBM and $k/4$ to service all requests. Thus, each unit interval in RCSP can be completed in $k + k/4$ time in HBM instance. □

3.8.2 Solving the Scheduling Problem

In this section, we focus on the following problem we call the resource constrained scheduling problem. There is a set of n jobs that can be scheduled using k resources. At each point in time, a job can be allocated any number of resources, so long as at most k are assigned to all of the jobs. Each job i consists of w_i tasks. The jth task for job i has a resource requirement $r_{i,j}$ and can only be processed after all $1, 2, \ldots j-1$ prior tasks for job i have completed. All of the tasks are assumed to be unit time. A job is completed when all of its tasks are completed. The goal is to minimize the total completion time of the jobs. We show the following theorem.

Theorem 14. *There is a polynomial time 20 approximation algorithm for the resource constrained scheduling problem.*

We begin by showing that by loosing a factor 4 in the approximation ratio, we may assume that the algorithm is given $2k$ resources. In particular, we show that if we have a schedule using $2k$ resources then we can construct a schedule using k resources with at most a factor 4 larger objective.

Lemma 24. *Consider any algorithm A that has a total completion time of T using $2k$ resources. In polynomial time given A, there is an algorithm B that uses k resources with total completion time $4T$. This assumes $r_{i,j} \leq k$ for all i and j.*

Proof. Fix any time step t in the schedule A. We show that we can complete the work done during this time step in A in 4 steps in the schedule B. This ensures we can replicate the schedule of A, one time step at a time.

Let $a_1, a_2, \ldots a_\ell$ be the tasks assigned to time step t in A and let $r_1, r_2, \ldots r_\ell$ be their corresponding resource requirements. Assume that $r_1 \geq r_2 \geq \ldots r_\ell$. Let S_1, S_2, S_3 and S_4 be sets corresponding to the four time steps for t in B. From $i = 1$ to ℓ assign task a_i to any S_j such that $r_i + \sum_{i' \in S_j} r_{i'} \leq k$.

Now we just need to show that $r_i + \sum_{i' \in S_j} r_{i'} \leq k$ for some $j \in [4]$ when assigning any task. Say this is not the case when assigning a task i for the sake of contradiction. Then we know that $4r_i + \sum_{i' \in S_1 \cup S_2 \cup S_3 \cup S_4} r_{i'} > 3k$. Further, we know that $r_i \leq k$ by definition of the problem and therefore each set contains at least one item. The sets each include a task requiring greater resources than i by the ordering of how tasks are assigned. If $r_i \leq \frac{k}{2}$ then each set contains items of aggregate size greater than $\frac{k}{2}$ since i cannot fit in any set. However then the total resource requirement of the already assigned tasks exceeds $2k$, a contradiction to the schedule A. Otherwise, $r_i \geq \frac{k}{2}$. In this case there are at least five items of size at least $\frac{k}{2}$, again a contradiction. □

Leveraging the prior lemma, we give an algorithm. The algorithm uses $2k$ resources. The algorithm leverages rounding a linear program (LP). Let $x_{i,j,t}$ denote whether the jth task for job i is processed at time t. The linear programming formulation is the following. Notice that the LP using k resources and is a lower bound on the optimal solution.

$$\min \sum_{i=1}^{n} \sum_{t} t \cdot x_{i,w_i,t} \tag{3.1}$$

$$\text{s.t.} \quad \sum_{t} x_{i,j,t} = 1 \quad \forall i \in [n], j \in [w_i] \tag{3.2}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{w_i} r_i x_{i,j,t} \leq k \quad \forall t \tag{3.3}$$

$$x_{i,j,t} \leq \sum_{t' \leq t} x_{i,j-1,t'} \quad \forall i, j, t$$

$$0 \leq x_{i,j,t} \leq 1 \quad \forall i, j, t$$

The objective states that a job pays t for the completion time when its last task is processed. The first set of constraints ensures all tasks are scheduled. The second ensures that at most k resources are used at any point in time. The third ensures that the jth task for job i is completed after task $(j-1)$ for job i is completed. The last constraint ensures all tasks are scheduled.

Consider solving the LP to get a solution x. Let C_i be the first time that $\sum_{t' \leq t} x_{i,w_i,t} \geq \frac{1}{2}$. Re-index the jobs such that $C_1 \leq C_2 \leq \ldots \leq C_n$. The jobs are scheduled in this priority order. Every task of job i is scheduled before we consider scheduling job $i+1$. Depending on how resources are packed, it could be the case that job $i+1$ is scheduled at times before i completes.

Consider the ith job. For the jth task of the job, schedule this is the first available time step with $r_{i,j}$ resources available after task $j-1$ for job i is completed . A time slot is available when the task is assigned if at most $2k$ resources are used. We remark that all tasks for i are scheduled before job $i+1$ is considered.

We now show that this algorithm schedules the tasks by time $2C_i + w_i$.

Lemma 25. *Every job i is completed by time $2C_i + w_i$ in the algorithm.*

Proof. Fix any job i. Consider the last task a_{i,w_i} for job i. Say that this task is completed at some time t. There are two reasons this task is not completed before time t. For any time step $t' \leq t$ either (1) a task for i is being scheduled or (2) this time step does not have enough free resources to schedule the available task for job i at that time. Notice that if (2) occurs, then at least k resources are being used. This is because each task for i has size at most k, there are $2k$ resources available, and a task for i is available and not scheduled. We know that there are at most w_i time steps where (1) occurs. There cannot be more than $2C_i$ time steps when (2) because of the following. At any time t that is a (2) timestep, the LP must be using k resources to schedule tasks for some job $i' < i$. Every task for such a job i' is processed fractionally to at least a $\frac{1}{2}$ amount by time $C_{i'} \leq C_i$. Given that the LP can process at most kt work up to any time t, there must be at most $2C_i$ such timesteps. Thus, job i is completed at time $2C_i + w_i$. □

We can leverage the prior lemma to prove the theorem.

Proof of Theorem 14. Let O^* be the objective of the optimal solution using k resources. Lemma 25 ensures that the algorithm completes a job by time $2C_i + w_i$. The LP objective is at least $\frac{1}{2}\sum_i C_i$ since each job pays at least $\frac{1}{2}C_i$ in the objective. We also know that $\sum_i w_i$ is a lower bound on the

optimal solution. Thus the algorithm has an objective value of $5O^*$. Lemma 24 can transform the algorithm's solution into a valid solution using k resources at a cost of increasing the objective by a factor 4. This gives a total approximation ratio of 20.

□

Chapter 4

Avoiding Races with Extra Memory

A determinacy race occurs if two or more logically parallel instructions access the same memory location and at least one of them tries to modify its content. Races are often undesirable as they can lead to nondeterministic and incorrect program behavior. A data race is a special case of a determinacy race which can be eliminated by associating a mutual-exclusion lock with the memory location in question or allowing atomic accesses to it. However, such solutions can reduce parallelism by serializing all accesses to that location. For associative and commutative updates to a memory cell, one can instead use a reducer, which allows parallel race-free updates at the expense of using some extra space. More extra space usually leads to more parallel updates, which in turn contributes to potentially lowering the overall execution time of the program.

We start by asking the following question. Given a fixed budget of extra space for mitigating the cost of races in a parallel program, which memory locations should be assigned reducers and how should the space be distributed among those reducers in order to minimize the overall running time? We argue that under reasonable conditions the races of a program can be captured by a directed acyclic graph (DAG), with nodes representing memory cells and arcs representing read-write dependencies between cells. We then formulate our original question as an optimization problem on this DAG. We concentrate on a variation of this problem where space reuse among reducers is allowed by routing every unit of extra space along a (possibly different) source to sink path of the DAG and using it in the construction of multiple (possibly zero) reducers along the path. We consider two different ways of constructing a reducer and the corresponding duration functions (i.e., reduction time as a function of space budget).

We generalize our race-avoiding space-time tradeoff problem to a discrete resource-time tradeoff problem with general non-increasing duration functions and resource reuse over paths of the given DAG.

For general DAGs, we show that even if the entire DAG is available to us offline the problem is strongly NP-hard under all three duration functions, and we give approximation algorithms for solving the corresponding optimization problems. We also prove hardness of approximation for the general resource-time tradeoff problem and give a pseudo-polynomial time algorithm for series-parallel DAGs.

This work is published in the proceedings of SPAA 2019 [57].

4.1 Introduction

A determinacy race (or a general race) [108, 66] occurs if two or more logically parallel instructions access the same memory location and at least one of them modifies its content. Races are often undesirable as they can lead to nondeterministic and incorrect program behavior. A data race is a special case of a determinacy race which can be eliminated by associating a mutual-exclusion lock with the memory location in question or allowing only atomic accesses to it. Such a solution, however, makes all accesses to that location serial and thus destroys all parallelism. Figure 4.1 shows an example.

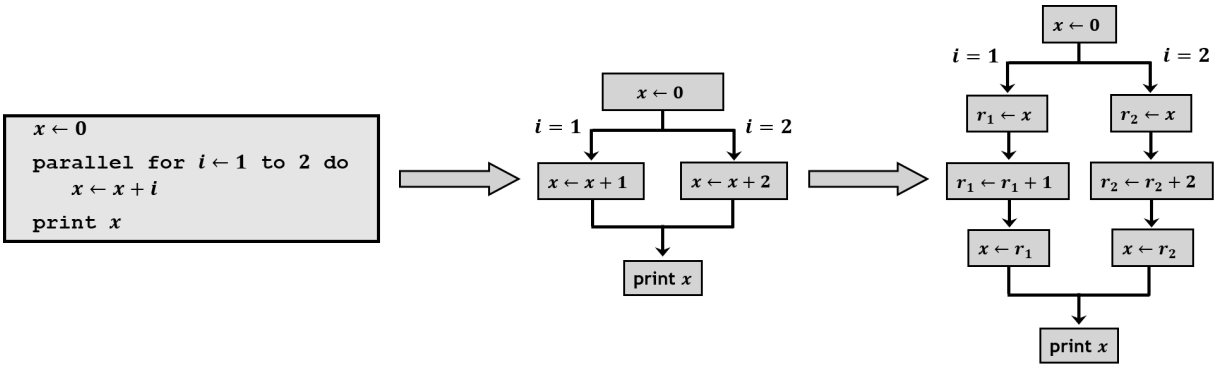


Figure 4.1: This figure shows a race on global variable x caused by two parallel threads trying to increment x, where r_1 and r_2 are local registers. The value printed by the 'print' statement depends on how the two threads are scheduled. Unless the two threads are executed sequentially, the print statement will print an incorrect result (either 1 or 2 depending on which thread updated x last).

One can use a reducer [72, 41, 117] to eliminate data races on a shared variable without destroying parallelism, provided the update operation is associative and commutative. Figure 4.2 shows the construction of a simple recursive binary reducer. For any integer $h > 0$ such a reducer is a full binary tree of height h and size $2^{h+1} - 1$ with the shared variable at the root. Each nonroot node is associated with a unit of extra space initialized to zero. All updates to the shared variable are equally distributed among the leaves of the tree. Each node has a lock and a waiting queue to avoid races by serializing the updates it receives, but updates to different nodes can be applied in parallel. As soon as a node undergoes its last update, it updates its parent using its final value. In fact, such a reducer can be constructed using only 2^h units of extra space because if a node completes before its sibling it can become its own parent (with ties broken arbitrarily) and the sibling then updates the new parent. Assume that the time needed to apply an update significantly dominates the execution time of every other operation the reducer performs and each update takes one unit of time to apply. Then a reducer of height h can correctly apply n parallel updates on a shared variable in $\lceil \frac{n}{2^h} \rceil + h + 1$ time provided at least 2^h processors are available. Hence, for large n, the speedup achieved by a reducer (w.r.t. serially and directly updating the shared variable) is almost linear in the amount of extra space used.

To see how extra space can speed up real parallel programs consider the iterative matrix multiplication code Parallel-MM shown in Figure 4.3 which multiplies two $n \times n$ matrices $X[1..n][1..n]$ and $Y[1..n][1..n]$ and puts the results in another $n \times n$ matrix $Z[1..n][1..n]$; that is, it sets $Z[i][j] = \sum_{1 \le k \le n} X[i][k] \times Y[k][j]$ for $1 \le i, j \le n$. Since every $Z[i][j]$ value can be

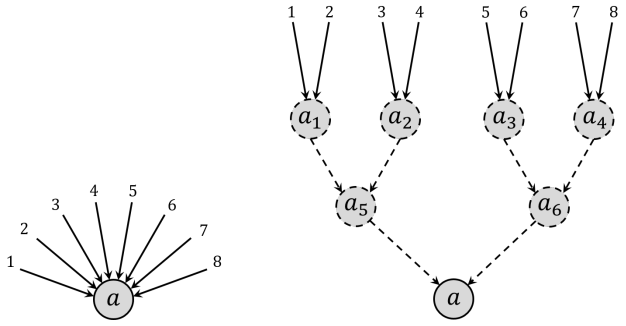


Figure 4.2: **[Left]** A memory location a with eight updates using an associative and commutative operator. **[Right]** The same location a with a recursive binary reducer of height two on top of it.

```
PARALLEL-MM(Z, X, Y, n)
1. parallel for i ← 1 to n do
2.     parallel for j ← 1 to n do
3.         Z[i][j] ← 0
4.         for k ← 1 to n do
5.             Z[i][j] ← Z[i][j] + X[i][k] × Y[k][j]
```

Figure 4.3: Parallel code that multiplies two $n \times n$ matrices $X[1..n][1..n]$ and $Y[1..n][1..n]$, and puts the result in $Z[1..n][1..n]$.

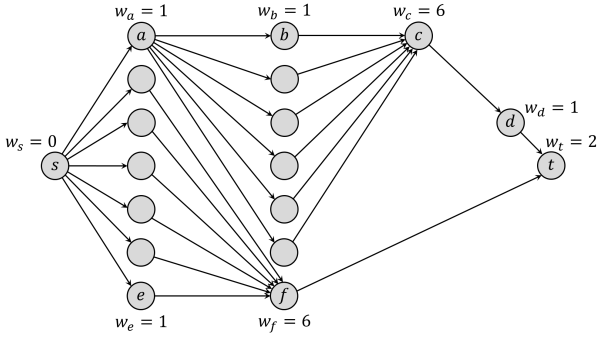


Figure 4.4: A DAG in which each node's work value is set to its in-degree. The makespan of this DAG is 11, and path $s \to a \to b \to c \to d \to t$ achieves it.

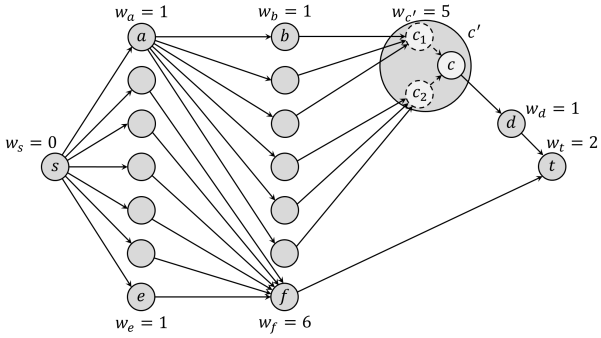


Figure 4.5: Node c from the DAG in Figure 4.4 has been replaced with a supernode c' in this figure which is nothing but node c with a reducer of height 1 on top. The makespan of this reduced DAG is 10, and path $s \to a \to b \to c_1 \to c \to d \to t$ achieves it.

computed independently of others, all iterations of the loops in Lines 1 and 2 can be executed in parallel without compromising correctness of the computation. However, the same is not true for the loop in Line 4 because if parallelized, for fixed values of i and j, all iterations of that loop will update the same memory location $Z[i][j]$ giving rise to data races and thus producing potentially incorrect results. Use of a mutual-exclusion lock or atomic updates for each $Z[i][j]$ will ensure correctness but in that case even with an unbounded number of processors, the code will take $\Theta(n)$ time to multiply the two $n \times n$ matrices. Now if we put a reducer of height h (integer $h \in [1, \log_2 n]$) at the top of each $Z[i][j]$ the time to fully update each $Z[i][j]$ and thus the overall running time of the code will drop to $\Theta\left(\frac{n}{2^h} + h\right)$ at the cost of using $n^2 \times 2^h$ units of extra space. Observe that when $h = 1$, the running time of the code almost halves using $2n^2$ units of extra space, and when $h = \lfloor \log_2 n \rfloor$, the running time drops to $\Theta(\log n)$ using $\Theta(n^3)$ extra space.

In order to analyze a program P with data races, we capture those races in a directed acyclic graph (DAG) $D(P)$, assuming that there are no cyclic read-write dependencies among the memory locations accessed by P. Figure 4.4 shows an example. We restrict P to the set of pro-

grams that perform $\mathcal{O}(1)$ other operations between two successive writes to the memory, e.g., PARALLEL-MM in Figure 4.3. We assume that an update operation is significantly more expensive than any other single operation performed by P and hence the costs of those operations can be safely ignored. Each node x of $D(P)$ represents a memory location, and a directed edge from node x to node y means that y is updated using the value stored at x. The in-degree $d_x^{(in)}$ of node x gives the number of times x is updated. With x we also associate a *work* value w_x and set $w_x = d_x^{(in)}$. Assuming that each update operation requires unit time to execute and each node has a lock and a wait queue to serialize the updates, the w_x value represents the time spent updating x (excluding all idle times). The w_x value also represents an upper bound on the time elapsed between the trigger time of any incoming edge of x and the time the edge completes updating x. We assume that updates along all outgoing edges of x trigger as soon as all incoming edges complete updating x. One can then make the following observation.

Observation 6. *The running time of P with an unbounded number of processors is upper bounded by the makespan of $D(P)$[1].*

Then one natural question to ask is the following.

Question 4.1.1. *Given a fixed budget of units of extra space to mitigate the cost of data races in P, which memory locations should be assigned reducers and how should the space be distributed among those reducers in order to minimize the makespan of $D(P)$?*

Figure 4.5 shows how to minimize the makespan of the DAG in Figure 4.4 using two units of extra space.

The question above ignores the possibility that space can be reused among reducers in $D(P)$. Indeed, after node x reaches its final value (i.e., updated $w_x = d_x^{(in)}$ times) it can release all (if any) space it used for its reducer which can then be reused by some other node y. A global memory manager can be used by the nodes to allocate/deallocate space for reducers. The following modified version of Question 4.1.1 now allows space reuse.

Question 4.1.2. *Repeat Question 4.1.1 but allow for space reuse among nodes of $D(P)$ by putting all extra space under the control of a global memory manager that each node calls to allocate space for its reducer right before its first update and to deallocate that space right after its last update.*

The problem with a single global memory manager is that it can easily become a performance bottleneck for highly parallel programs. Though better memory allocators have been developed for multi-core or multi-threaded systems [12, 35, 3, 121, 2], we can instead use an approach often used by recursive fork-join programs which avoids repeated calls to an external memory manager altogether along with the overhead of repeated memory allocations/deallocations. A single large segment of memory is allocated before the initial recursive call is made and a pointer to that segment is passed to the recursive call. Each recursive call splits and distributes its segment among its child recursive calls and reclaims the space when the children complete execution. So, we will assume that all the given extra space initially reside at the source node (i.e., node with

[1]To see why this is true start from the sink node and move backward toward the source by always moving to that predecessor y of the current node x that performed the last update on x and noting that after edge (y, x) was triggered it did not have to wait for more than $d_x^{(in)}$ time units to complete applying y's update to x.

in-degree zero). Then they flow along the edges toward the sink node (i.e., node with outdegree zero) possibly splitting along outgoing edges and merging at the tip of incoming edges as they flow. Each unit of space reaching node x moves out of x along some outgoing edge as soon as x becomes fully updated and those edges trigger. Every unit of space may participate in the construction of multiple reducers (possibly zero) along the path it takes.

Question 4.1.3. *Repeat Question 4.1.1 but now allow for space reuse among nodes of $D(P)$ by flowing each unit of space along a source to sink path and using it in the construction of zero or more reducers along that path.*

While several existing results [59, 64, 125, 85] can be extended to answer Questions 4.1.1 and 4.1.2, to the best of our knowledge, Question 4.1.3 had not been raised before. In this chapter we investigate answers to Question 4.1.3 by extending it to a more general resource-time tradeoff question posed on a DAG in which nodes represent jobs (not necessarily of updating memory locations), resources (not necessarily space) flow along source to sink paths, and an general duration function (i.e., time needed to complete a job as a function of the amount of resources used) is specified for each node. We consider the following three duration functions: general non-increasing function for the general resource-time question, and recursive binary reduction and multiway (k-way) splitting for the space-time case.

For general DAGs, we show that even if the entire DAG is available to us offline the problem is strongly NP-hard under all three duration functions, and we give approximation algorithms for solving the corresponding optimization problems. We also prove hardness of approximation for the general resource-time tradeoff problem and give a pseudo-polynomial time algorithm for series-parallel DAGs. Our main results are summarized in Table 4.1.

Duration function	Hardness	Hardness of Approximation
General non-increasing	strongly NP-hard	• makespan < 2 OPT with resources fixed • resource $< \frac{3}{2}$ OPT with makespan fixed
Recursive binary	strongly NP-hard	–
Multiway splitting	strongly NP-hard	–

Duration function	Approximation Results
General non-increasing	$\left(\frac{1}{\alpha}, \frac{1}{1-\alpha}\right)$ bi-criteria (resource, makespan), $0 < \alpha < 1$
Recursive binary	• makespan ≤ 4 OPT with resources fixed • $\left(\frac{4}{3}, \frac{14}{5}\right)$ bi-criteria (resource, makespan)
Multiway splitting	makespan ≤ 5 OPT with resources fixed

Table 4.1: Our main results on resource-time tradeoff problems in which resources are routed along source to sink paths (i.e., related to Question 4.1.3 and its generalization).

Related Work

While several prior works either directly or indirectly address Questions 4.1.1 (nonreusable resources) and 4.1.2 (globally reusable resources), to the best of our knowledge, Question 4.1.3 (reusable along flow paths) has not been considered before.

The well-known time-cost tradeoff problem (TCTP) is closely related to our nonreusable resources question. In TCTP, some activities are expediated at additional cost so that the makespan can be shortened. Deadline and budget problems are two TCTP variants with different objectives. While the deadline problem seeks to minimize the total cost to satisfy a given deadline, the budget problem aims to minimize the project duration to meet the given budget constraint [13]. Most researchers consider the tradeoff functions to be either linear continuous or discrete giving rise to linear TCTP and discrete TCTP, respectively.

Linear TCTP was formulated by Kelley and Walker in 1959 [91]. They assumed affine linear and decreasing tradeoff functions. In 1961, linear TCTP was solved in polynomial time using network flow approaches independently by Fulkerson [76] and Kelley [90]. Phillips and Dessouky [115] later improved that result.

In 1997, De et al. [59] proved that discrete TCTP is NP-hard. For this problem, Skutella [125] proposed the first approximation algorithm under budget constraints which achieves an approximation ratio of $\mathcal{O}(\log r)$, where r is the ratio of the maximum duration of any activity to the minimum one. Discrete TCTP can also be used to approximate the TCTP with general time-cost tradeoff functions, see, e.g., Panagiotakopoulos [111] and Robinson [119]. For details on discrete TCTP see De et al. [58].

Our problem with globally reusable resources (Question 4.1.2) is very similar to the problem of scheduling precedence-constrained malleable tasks [137]. In 1978, Lenstra and Rinnooy Kan [95] showed that no polynomial time algorithm exists with approximation ratio less than $\frac{4}{3}$ unless P = NP. About 20 years later, Du and Leung [64] showed that the problem is strongly NP-hard even for two units of resources. In 2002, under the monotonous penalty assumptions of Blayo et al [36], Lepère et al. [96] first proposed the idea of two-step algorithms – computing an allocation first, and then scheduling tasks, and used this idea [97] to design a algorithm that achieve an approximation ratio of ≈ 5.236. In the first phase, they approximate an allocation using Skutella's algorithm [125]. Similarly, based on Skutella's approximation algorithm, Jansen and Zhang [85] devised a two-phase approximation algorithm with the best-known ratio of ≈ 4.730598 and showed that the ratio is tight when the problem size is large. For more details on the problems of scheduling malleable tasks with precedence constraints, please check Dutot et al. [65].

There are memory allocators based on global memory manager for multi-core or multi-threaded systems such as scalloc [12], Hoard [35], llalloc [3], Streamflow [121], and TCMalloc [2]. They use thread-local space for memory allocation and a global manager for memory deallocation/reuse. For the global manager, they use concurrent data structures. However, these data structures can not completely avoid the need for synchronization [12, 81, 124] without compromising correctness.

4.2 Preliminaries, Problem Formulation

In general, the option to use reducers to trade off between extra space and the time to complete race-free writing operations leads to a *discrete resource-time tradeoff problem*, where, here, the valuable "resource" is the space that is added, in order to reduce the time necessary for the write operations. By investing in additional space, we can reduce the time it takes to do conflict-free write operations.

We formalize the discrete resource-time tradeoff problem. Consider a DAG, $D = (V, E)$,

whose vertices V correspond to jobs, and whose edges represent precedence relations among jobs. Without loss of generality, we assume that the DAG has a single source and a single sink vertex. The duration of a job depends on how much resource it receives. For each job $v \in V$, there is a non-increasing duration function $t_v(r)$ that denotes the time required to complete job v using r units of resources. We call $\langle r, t_v(r)\rangle$ a *resource-time tuple* associated with job (vertex) v. We consider three classes of duration functions – general non-increasing step functions, k-way splitting functions, and recursive binary splitting functions.

General non-increasing step function. Let l_v be the number of resource-time tuples associated with job v. The i-th resource-time tuple is $\langle r_{v,i}, t_v(r_{v,i})\rangle$ where $1 \leq i \leq l_v$. Then, the duration function $t_v(r)$ is a step function with l_v steps described as follows:

$$t_v(r) = \begin{cases} t_v(r_{v,i}), & \text{if } r_{v,i} \leq r < r_{v,i+1}, 1 \leq i < l_v, \\ t_v(r_{v,l_v}). & \text{if } r_{v,l_v} \leq r, \end{cases} \tag{4.1}$$

where $r_{v,1} = 0, r_{v,j} < r_{v,j+1}$ and $t_v(r_{v,j}) \geq t_v(r_{v,j+1})$ for $1 \leq j < l_v$.

***k*-way splitting.** A *k-way split reducer* utilizes k units of extra space, $S_v = \{s_1, s_2, .., s_k\}$, associated with a vertex v, with $2 \leq k \leq d_v^{(in)}$, such that the write operations associated with incoming edges at v are distributed among the vertices S_v, which then have edges linking each s_i to v. The duration function that results from k-way split reducers is given by

$$t_v(r) = \begin{cases} t_v(0), & \text{if } k \in \{0, 1\} \\ \lceil t_v(0)/k \rceil + k, & \text{if } 2 \leq k \leq \lfloor \sqrt{t_v(0)} \rfloor \\ t_v(\lfloor \sqrt{t_v(0)} \rfloor). & \text{if } \lfloor \sqrt{t_v(0)} \rfloor < k. \end{cases} \tag{4.2}$$

Recursive binary splitting. The duration function that results from a recursive binary split reducer is given by a step function, as follows. The resource-time tuples are defined for $r = 0$ and 2^i where $0 \leq i \leq k$ and $k = \lfloor \log_2 t_v(0) - \log_2 \log_2 e \rfloor$. The duration function $t_v(2^k) = \lceil t_v(0)/2^k \rceil + k + 1$ is minimized when $k = \lfloor \log_2 t_v(0) - \log_2 \log_2 e \rfloor$ (by differentiating $t_v(2^k)$ w.r.t. k).

$$t_v(r) = \begin{cases} t_v(0), & \text{if } r = 0, 1 \\ \lceil t_v(0)/2^i \rceil + i + 1, & \text{if } r = 2^i, 2 \leq i \leq k \\ t_v(2^i), & \text{if } 2^i \leq r < 2^{i+1}, 2 \leq i \leq k \\ t_v(2^k), & \text{if } i > k \end{cases} \tag{4.3}$$

When utilizing a reducer, extra space serves as the limited resource and the time taken for race-free writing at a vertex v is the duration of the job corresponding to v. Both the k-way splitting duration function and the recursive binary splitting duration function are special cases of general non-increasing function.

We consider jobs whose duration functions are of the types described above, and we distinguish between two optimization problems, depending on the objective function:

Minimum-Makespan Problem. Given a resource budget of B, assign the resources to the vertices V such that the makespan of the project is minimized. Resources can be reused over a path.

Minimum-Resource Problem. Given a makespan target of T, minimize the amount of resources to achieve target makespan. Resources can be reused over a path.

Finally, we remark that instead of jobs corresponding to vertices of the DAG, we can transform the DAG D into another DAG D' in which jobs correspond to edges of D', and the precedence relations among jobs are enforced by introducing dummy edges, as follows: For each node v in D, we introduce an edge $e_v = (a_v, b_v)$ in D' (which then has the corresponding duration function, specified, e.g., by resource-time tuples). For each edge (u, v) of D, we introduce a dummy edge, $e = (b_u, a_v)$ in D', from the endpoint b_u of edge $e_u = (a_u, b_u)$ to the origin a_v of edge $e_v = (a_v, b_v)$, with resource-time function $t_e(r) = 0$ for all valid resource levels r.

4.3 Approximation Algorithms

4.3.1 Bi-criteria Approximation for Non-increasing Duration Functions

We use linear programming in our approximation algorithms. First, we relax the discrete duration function to a linear one. We transform the DAG so that a relaxed linear non-increasing duration function can be used. The transformation happens in two steps.

Activity on arc reduction. We reduce the input DAG D into an equivalent DAG $D^{'}$ with activities on arcs instead of nodes. This is a simple transformation described earlier in Section 4.2.

Activity with two tuples. Following [125], we create a DAG $D^{''}$ from $D^{'}$ such that all activities in $D^{''}$ are still on arcs and each such activity has at most 2 resource-time tuples as shown in Figure 4.6(b). Let j be a job with $l_j \geq 2$ resource-time tuples $\langle r_{j,i}, t_j(r_{j,i})\rangle, 1 \leq i \leq l_j$ with $0 = r_{j,1} < r_{j,2} < \cdots < r_{j,l_j}$ and $t_j(r_{j,1}) \geq t_j(r_{j,2}) \geq \cdots \geq t_j(r_{j,l_j})$ (following Equation 4.1). Let edge (u, v) of $D^{'}$ represent job j. We add l_j parallel chains, each consisting of two edges in $D^{''}$ (Figure 4.6). For $1 \leq i \leq l_j$, we create a chain of two edges (u, u_i) and (u_i, v). We create a job j_i for arc (u, u_i) and associate two resource-time tuples with it. For $1 \leq i < l_j$, job j_i can be finished either using 0 resource in $t_j(r_{j,i})$ units of time or using $(r_{j,i+1} - r_{j,i})$ units of resource in 0 unit of time. The logic is that job j's duration can be reduced from $t_j(r_{j,i})$ to $t_j(r_{j,i+1})$ provided the resource difference $(r_{j,i+1} - r_{j,i})$ is allocated to j_i. Thus the duration function is $t_{j_i}(0) = t_j(r_{j,i})$ and $t_{j_i}(r_{j,i+1} - r_{j,i}) = 0$. Job j_{l_j}'s (bottom most edge in the l_j parallel edges for job j) duration cannot be further improved from $t_j(r_{j,l_j})$ units of time by using extra resources. The resource-time tuple at edge (u_i, v) is $\langle 0, 0\rangle$ where $1 \leq i \leq l_j$.

There is a canonical mapping of resource usages and durations for jobs j_i to that of job j. Let x_i be the units of resource used for job j_i, then for job j, $\sum_{i=1}^{l_j} x_i$ units of resource are used. The time taken to finish job j is $\max\{t_{j_i}(x_i) | 1 \leq i \leq l_j\}$. Without loss of generality, if we use 0 unit of resource for job j_i if $t_{j,i}(0) \leq \max\{t_{j,1}(x_1), t_{j,2}(x_2), \cdots, t_{j,i-1}(x_{i-1})\}$, then this mapping is bijective. Thus we get the following lemma.

Lemma 26. *Any approximation algorithm $\mathcal{A}$ on DAG $D^{''}$ (activity on edge and each edge has at most two resource-time tuples) with an approximation ratio α implies an approximation algorithm with the same approximation ratio α on general DAG D (activity on vertex and each job can have more than two resource-time tuples).*

From now on, we will only consider DAGs whose edges represent jobs, with each edge having at most two resource-time tuples.

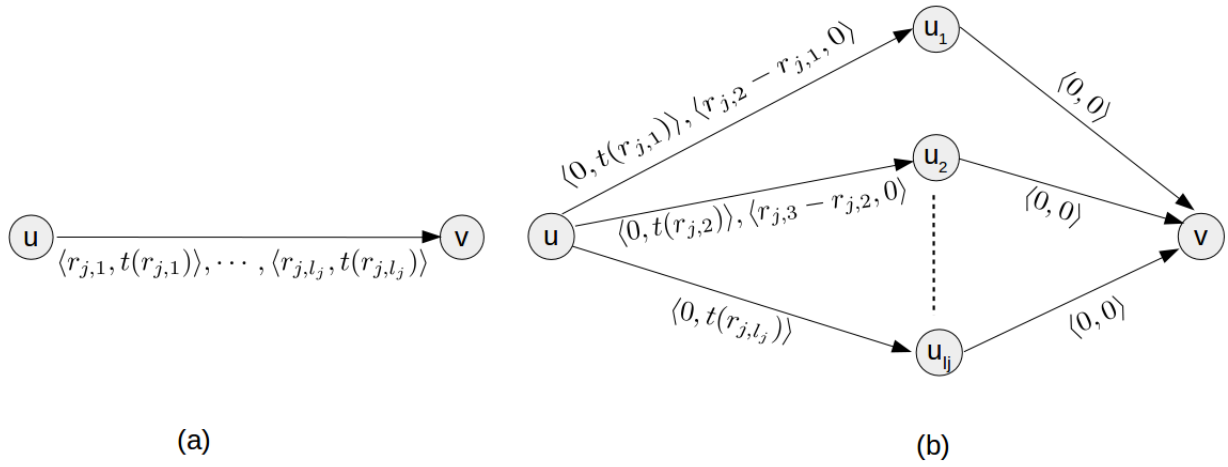


Figure 4.6: Transforming $(\boldsymbol{a})$ a DAG with $l_j \geq 2$ resource-time tuples on each arc into $(\boldsymbol{b})$ one with at most two resource-time tuples on each arc (Section 4.3.1)

.

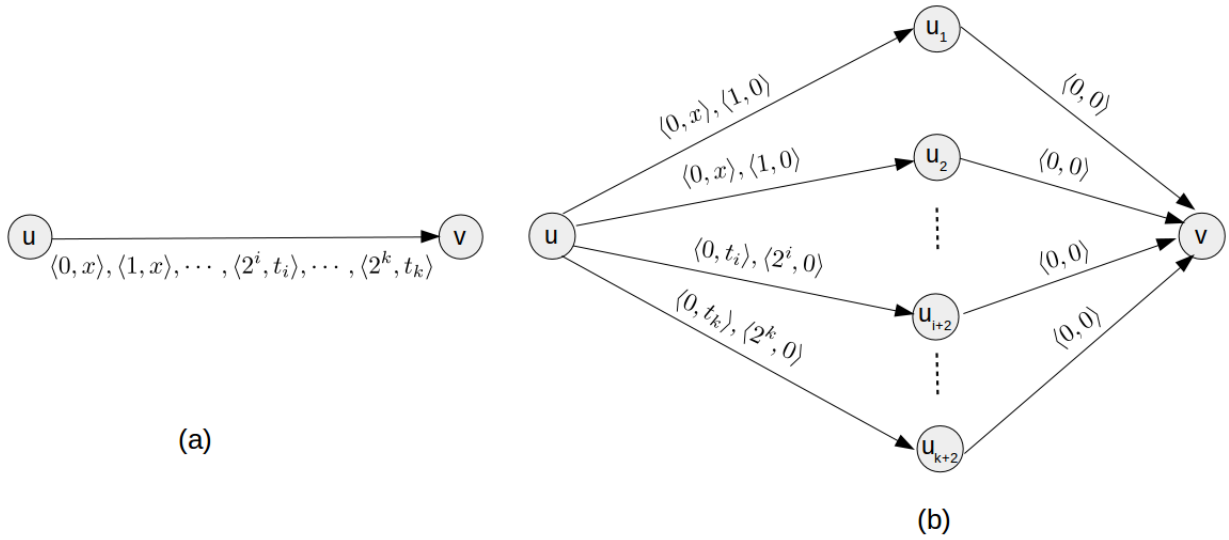


Figure 4.7: Transforming $(\boldsymbol{a})$ a DAG with $(k+1)$ resource-time tuples on each arc based on the recursive binary splitting function into $(\boldsymbol{b})$ one with at most two resource-time tuples on each arc (Section 4.3.3)

.

Linear relaxation. In $D^{''}$, any edge (u, v) can have either two resource-time tuples $\{\langle 0, t_{(u,v)}(0)\rangle, \langle r_{(u,v)}, 0\rangle\}$ or a single resource-time tuple $\{\langle 0, t_{(u,v)}(0)\rangle\}$. With linear relaxation, $r \in [0, r_{(u,v)}]$ units of resource can be used to reduce the completion time of the job corresponding to edge (u, v) that has two resource-time tuples. The corresponding duration function $t_{(u,v)}(r)$ is as follows:

$$t_{(u,v)}(r) = \frac{t_{(u,v)}(0)}{r_{(u,v)}} r \text{ for } r \in [0, r_{(u,v)}] \tag{4.4}$$

The linear duration function $t_{(u,v)}(r)$ for the job (u, v) with single resource-time tuple is as follows:

$$t_{(u,v)}(r) = t_{(u,v)}(0) \text{ for all } r \geq 0 \tag{4.5}$$

Linear programming formulation. Since we are allowed to reuse resources over a path we can model the problem as a network flow problem where resources are allowed to flow from the source to the sink in the DAG $D^{''}$. Let E be the set of edges in $D^{''}$. Let $f_{(u,v)}$ denote the amount of resources that flow through the edge (u, v). Using linear relaxation on edge (u, v), the time taken to finish the activity is $t_{(u,v)}(f_{(u,v)})$. Let the vertices in $D^{''}$ denote events. From now onwards, we use a vertex and its corresponding event synonymously. Let $E_v = \{(x, v)\}$ be the set of edges that are incident on vertex v. Event v occurs if and only if all the jobs corresponding to the edges in set E_v are finished. Let T_v denote the time when event v occurs. Let s and t denote the source vertex and the sink vertex, respectively. For source vertex s, we assume $T_s = 0$. All variables are non-negative.

Constraints:

$$f_{(u,v)} \leq r_{(u,v)} \ , \ \forall (u, v) \text{ with two resource-time tuples.} \tag{4.6}$$

$$T_u + t_{u,v}(f_{(u,v)}) \leq T_v \ , \ \forall (u, v) \in E \tag{4.7}$$

$$\sum_w f_{(v,w)} + \sum_u f_{(u,v)} = 0 \ , \ \forall v \notin \{s, t\} \tag{4.8}$$

$$\sum_k f_{(s,k)} \leq B \tag{4.9}$$

Objective function:

$$\min T_t \tag{4.10}$$

Inequality 4.6 upper bounds the resource flow variable $f_{(u,v)}$ for edges with two tuples. This ensures that these variables remain in the range $[0, r_{(u,v)}]$ and the duration function is linear in this range. Note that there is no such upper bound on the edges with single resource-time tuple (except the trivial total resource budget B upper bound). This allows the flow of more resources over an edge that can be used later on a path. Equation 4.8 is a flow conservation constraint for all the vertices $v \notin \{s, t\}$. Inequality 4.9 constrains the flow of resources from source s to be upper bounded by the resource budget.

Solving the LP and rounding. We first solve the LP described above. This might give solution as fractional flow f_e^* and duration $t_e(f_e^*)$ at edge $e = (u, v)$. Let the resource-time tuples at edge e be $\{\langle 0, t_e(0)\rangle, \langle r_e, 0\rangle\}$. The range of feasible duration of activity e is $[0, t_e(0)]$. We divide this range into two parts $[0, \alpha t_e(0)), [\alpha t_e(0), t_e(0)]$ where $0 < \alpha < 1$. If $t_e(f_e^*) \in [0, \alpha t_e(0))$ we round it down to 0, otherwise, we round it up to $t_e(0)$. Observe that in the first case, the resource requirement at e can be increased by at most a factor of $1/(1-\alpha)$. In the second case, the completion time can be increased at most by a factor of $1/\alpha$. Let $f_e^{'}$ denote the rounded integer resource requirement at edge e.

Computing min-flow. After rounding the LP solution, we get an integral resource requirement $f_e^{'} \in \{0, r_e\}$ for every edge e. We now compute a min-flow through this DAG where $f_e^{'}$ serves as the lower bound on the flow through (or resource requirement at) edge e.

Constraints:

$$f_{(u,v)} \geq f_{(u,v)}^{'}\ ,\ \forall (u,v) \in E \tag{4.11}$$

$$\sum_w f_{(v,w)} + \sum_u f_{(u,v)} = 0\ ,\ \forall v \notin \{s,t\} \tag{4.12}$$

Objective function:

$$\min \sum_k f_{(s,k)} \tag{4.13}$$

Let, f and f^* be the optimal solutions of LP 4.11–4.13 and LP 4.6–4.10, respectively.

Lemma 27. *$f^*/(1-\alpha)$ is a feasible solution of min-flow LP 4.11–4.13.*

Proof. Let f_e^* be the optimal solution of LP 4.6–4.10. We know that $f_e^{'} \leq f_e^*/(1-\alpha)$. Hence, $f^*/(1-\alpha)$ is a feasible solution of that LP as it meets the resource requirement $f_e^{'}$ at every edge e. □

Lemma 28. *f is an integral flow and $f \leq f^*/(1-\alpha)$, where $0 < \alpha < 1$.*

Proof. The minflow problem has integral optimality. If f is the optimal solution then it is an integral flow. From lemma 27 we know that $f^*/(1-\alpha)$ is a feasible solution of LP 4.11–4.13. Since f is optimal and $f^*/(1-\alpha)$ is a feasible flow, we have, $f \leq f^*/(1-\alpha)$. □

Bi-criteria approximation. We now summarize our bi-criteria approximation result for general non-increasing duration functions:

Theorem 15. *For any $\alpha \in (0,1)$, there is a $(1/\alpha, 1/(1-\alpha))$ bi-criteria approximation algorithm for the discrete resource-time tradeoff problem with an general non-increasing duration function which allows resource reuse over paths.*

Proof. First, we know from lemma 28 that f is an integral flow and $f \leq f^*/(1-\alpha)$, where $0 < \alpha < 1$.

Second, we claim that the makespan of the DAG used in the minflow LP 4.11–4.13 is at most a factor of $1/\alpha$ away from that of the LP 4.6–4.10 solution. Let us consider any $s-t$ path MP. The makespan is at least the sum of completion times of the edges in MP. Now, after rounding the LP 4.6–4.10 solution, the completion time of an edge may increase at most by a factor of α. Hence, the sum of duration of edges along any path is increased at most by a factor of α, thus the makespan will be increased by at most a factor of α. □

4.3.2 Single-criteria Approximation for k-Way and Recursive Binary Splitting

First, observe the prior section gives us a bi-criterian approximation for both k-way and recursive binary splitting. Setting $\alpha = 1/2$ in Theorem 15, we obtain a $(2, 2)$ bi-criteria approximation. Now, after LP rounding, say a job j uses $\overline{r_j}$ units of resource and takes $\overline{t_j}$ units of time. Then the optimal solution uses $r_j^* \geq \overline{r_j}/2$ units of resource and takes $t_j^* \geq \overline{t_j}/2$ units of time for job j. Recall that job j consists of l_j parallel jobs j_i where $1 \leq i \leq l_j$. Hence, $\overline{r_j}$ is the sum of the resource (after rounding) used by l_j parallel jobs and $\overline{t_j}$ is the maximum time (after rounding) taken by l_j parallel jobs.

Approximation algorithm for *k*-way splitting. To obtain a single-criteria approximation, in the case of k-way splitting, we use at most r_j^* units of resource for job j. If $\overline{r_j} > r_j^*$, we reduce $\overline{r_j}$ to k (a nonnegative integer) units of resource such that $k \leq r_j^*$. Using k units of resource, job j takes $t_j(k)$ units of time to complete.

Lemma 29. $\lceil d/k \rceil + k \leq 2.5\overline{t_j}$ *for* $\overline{r_j} > 3$ *where* $d = t_j(0)$ *and* $k = \lfloor \overline{r_j}/2 \rfloor$.

Proof. Since $k = \lfloor \overline{r_j}/2 \rfloor \geq \overline{r_j}/2.5$ for $\overline{r_j} > 3$, we have $\lceil d/k \rceil \leq d/k + 1 \leq 2.5d/\overline{r_j} + 1 \leq 2.5\lceil d/\overline{r_j} \rceil + 1$. Also since $k = \lfloor \overline{r_j}/2 \rfloor \leq \overline{r_j} + 1$ and $2.5\overline{r_j} \geq \overline{r_j} + 2$ for $\overline{r_j} > 3$, we have $\lceil d/k \rceil + k \leq 2.5\lceil d/\overline{r_j} \rceil + 1 + \overline{r_j} + 1 \leq 2.5\left(\lceil d/\overline{r_j} \rceil + \overline{r_j}\right)$. Hence, $t_j(k) \leq 2.5\overline{t_j}$. □

Lemma 30. *If* $\overline{r_j} > 3$ *then* $t_j(k) \leq 5t_j^*$.

Proof. We know $t_j(k) = \lceil d/k \rceil + k$ as $k \geq 4$. Also in lemma 29, we prove $t_j(k) \leq 2.5\overline{t_j}$. However, we show that $\overline{t_j} \leq 2t_j^*$. Hence, combining these two results we get $t_j(k) \leq 5t_j^*$. □

Lemma 31. *If* $t_j^* = d/4$ *then* $r_j^* \geq 2$.

Proof. Recall that in D'', job j is represented as l_j parallel jobs j_i where $1 \leq i \leq l_j$. The resource-time tuples of jobs j_1 and j_2 are $\{\langle 0, d\rangle, \langle 2, 0\rangle\}$ and $\{\langle 0, \lceil d/2 \rceil + 2\rangle, \langle 1, 0\rangle\}$, respectively. To attain $d/4$ duration, j_1 requires at least $3/2$ units of resource and job j_2 requires $1/2$ unit of resource (applying linear relaxation). Hence, $r_j^* \geq (3/2 + 1/2) = 2$ units of resource to achieve $t_j^* = d/4$. □

Lemma 32. *If* $\overline{r_j} \leq 3$ *then* $t_j(k) \leq 4t_j^*$.

Proof. If $\overline{r_j} \leq 3$ and $r_j^* < 2$, then we round down $\overline{r_j}$ to $k = 0$. So, from Lemma 31 it follows that after rounding down to 0 unit of resource, job j takes $d \leq 4t^*$ units of time.

If $\overline{r_j} \leq 3$ and $r_j^* \geq 2$, then we round $\overline{r_j}$ to $k = 2$. It is true that $t_j(2) \leq 2t_j(3)$ because $(\lceil d/2 \rceil + 2) \leq 2(\lceil d/3 \rceil + 3)$. Also, $t_j(3) \leq t_j(\overline{r_j}) \leq 2t_j^*$. Combining this two results we get $t_j(2) \leq 4t^*$. □

So, now we have the following result.

Theorem 16. *There is a* 5*-approximation algorithm for the minimum-makespan problem with* k*-way splitting duration function.*

Proof. Combining Lemmas 32 and 30 we get $t_j(k) \leq 5t_j^*$ for all valid $\overline{r_j}$. This proves that the makespan is at most 5 times the optimal solution. We now calculate the total amount of resource required to flow from the source of $D^{'}$. We compute a min-flow in $D^{'}$ where k is the resource requirement for job j. Note that we are now working on $D^{'}$ that does not have l_j parallel chains for job j. Let f be the min flow from the source of $D^{'}$ such that all the resource requirements are met. The flow f^* from the LP solution before rounding is also a valid flow for the resource requirement k for job j as $k \leq r_j^*$. We know that min-flow gives an optimal integral solution. Hence, $f \leq f^*$. □

Approximation algorithm for recursive binary splitting. We have the following result.

Theorem 17. *There is a 4-approximation algorithm for the minimum-makespan problem with recursive binary splitting function.*

Proof. As in the case of k-way splitter, to get a single-criteria approximation, we use no more than r_j^* units of resource for job j. If $\overline{r_j} > r_j^*$, we reduce $\overline{r_j}$ to $\overline{r_j}/2$. We know that $t_j(\overline{r_j}/2) \leq 2t_j(\overline{r_j})$ from the properties of the recursive binary splitting function. Thus, $t_j(\overline{r_j}/2) \leq 2t_j(\overline{r_j}) \leq 4t_j(r_j^*) = 4t_j^*$. □

4.3.3 Improved Bi-criteria Approximation for Recursive Binary Splitting Functions

Putting $\alpha = 3/4$ in Theorem 15 we obtain a $(4/3, 4)$ bi-criteria approximation algorithm for general non-increasing duration functions. Hence, if we use $4/3$ times more resources than OPT (i.e., the optimal solution), we are guaranteed to get a makespan within factor of 4 of OPT. In this section we show that the bound can be improved to $(4/3, 14/5)$ for recursive binary splitting functions.

For a node with in-degree x, the resource-time tuples based on the recursive binary splitting function are as follows: $\{\langle 0, x\rangle, \langle 1, x\rangle, \langle 2, t_1\rangle, ..., \langle 2^i, t_i\rangle, \langle 2^{i+1}, t_{i+1}\rangle ..., \langle 2^k, t_k\rangle\}$ where $t_j = \lceil x/2^j \rceil + j + 1$ for $j \geq 2$ and $k = \lfloor \log_2 x - \log_2 \log_2 e \rfloor$ is the largest value of j for which t_j decreases with the increase of j. See Figure 4.7.

After solving LP 4.6–4.10 from Section 4.3.1, we sum up the (possibly fractional) resources allocated to all the l_j parallel edges corresponding to job j. Let r be that sum. Let t be the maximum among the time values given by the LP solution for the l_j parallel edges. Thus, the LP takes t units of time for job j.

We round r to an integer $\overline{r}$ based on the following criteria.

$$\overline{r} = \begin{cases} 0, & \text{if } r < 1 \\ 2^i & \text{if } 2^i \leq r < (2^i + 2^{i+1})/2, 0 \leq i \leq k \\ 2^{i+1}, & \text{if } (2^i + 2^{i+1})/2 \leq r < 2^{i+1}, 0 \leq i \leq k \end{cases}$$

We want to find a constant ρ, such that if $t = t_i/\rho$, then the LP must use at least $(2^i + 2^{i+1})/2 = 3(2^{i-1})$ units of resources. We compute r as follows. In Figure 4.7(b), each of the top two edges (u, u_1) and (u, u_2) requires $(1 - (1/x)t)$ units of resource to finish in time t. Each edge (u, u_{j+2})

for $1 \leq j \leq i+1$ requires $\left(2^j - (2^j/t_j)t\right)$ units of resource to finish in time t. Summing over all these edges, we get the expression of r

$$r = 2\left(1 - \frac{1}{x}t\right) + \sum_{j=1}^{i+1}\left(2^j - \frac{2^j}{t_j}t\right) = 8 \cdot (2^{i-1}) - \frac{t_i}{\rho}\left(2/x + \sum_{j=1}^{i+1}\frac{2^j}{t_j}\right)$$

Since we want to have $r \geq 3(2^{i-1})$, we want to find the smallest value of ρ such that

$$\frac{t_i}{\rho}\left(2/x + \sum_{j=1}^{i+1}\frac{2^j}{t_j}\right) \leq 5 \cdot (2^{i-1}) \Rightarrow \quad \rho \geq 1/5\left(\frac{t_i}{2^{i-2}x} + \sum_{j=1}^{i+1}\frac{t_i}{2^{i-j-1}t_j}\right).$$

Now,

$$\begin{aligned}
&\frac{t_i}{2^{i-2}x} + \sum_{j=1}^{i+1}\frac{t_i}{2^{i-j-1}t_j} = \frac{\lceil\frac{x}{2^i}\rceil + i + 1}{x(2^{i-2})} + \sum_{j=1}^{i+1}\frac{\lceil\frac{x}{2^i}\rceil + i + 1}{(\lceil\frac{x}{2^j}\rceil + j + 1)2^{i-j-1}} \\
&< \frac{\frac{x}{2^i} + i + 2}{x(2^{i-2})} + \sum_{j=1}^{i+1}\frac{\frac{x}{2^i} + i + 2}{(\frac{x}{2^j} + j + 1)2^{i-j-1}} \\
&= \frac{1}{2^i}\frac{1}{2^{i-2}} + \frac{i+2}{x(2^{i-2})} + \sum_{j=1}^{i+1}\frac{\frac{1}{2^{i-j}}(\frac{x}{2^j} + j + 1) + i + 2 - \frac{j}{2^{i-j}} - \frac{1}{2^{i-j}}}{(\frac{x}{2^j} + j + 1)2^{i-j-1}} \\
&\leq \left(\frac{i+2}{x}\frac{1}{2^{i-2}} + \sum_{j=1}^{i+1}\frac{i+2}{(\frac{x}{2^j} + j + 1)2^{i-j-1}}\right) + \left(\frac{1}{2^i}\frac{1}{2^{i-2}} + \sum_{j=1}^{i+1}\frac{1}{2^{i-j}}\frac{1}{2^{i-j-1}}\right) \\
&= \left(\frac{i+2}{x}\frac{1}{2^{i-2}}\right) + \left(\sum_{j=1}^{i+1}\frac{i+2}{(\frac{x}{2^j} + j + 1)2^{i-j-1}}\right) + \left(\frac{32}{3} + \frac{1}{3}\frac{1}{4^{i-1}}\right)
\end{aligned}$$

Let, $A = \frac{i+2}{x}\frac{1}{2^{i-2}}$, $B = \sum_{j=1}^{i+1}\frac{i+2}{(\frac{x}{2^j}+j+1)2^{i-j-1}}$ and $C = 32/3 + \frac{1}{3}\frac{1}{4^{i-1}}$.
Note that $i + 2 = (i + 1) + 1 \leq (\log_2 x - \log_2\log_2 e) + 1$, since $i + 1 \leq k$. Hence,

$$A \leq \frac{(\log_2 x - \log_2\log_2 e) + 1}{x}\frac{1}{2^{i-2}} \leq \frac{2}{e}\frac{1}{2^{i-2}}.$$

Now, $x/2^j + j + 1 \geq (\log_2 x - \log_2\log_2 e + \frac{1}{\ln 2})$ and hence,

$$\begin{aligned}
B &\leq \sum_{j=1}^{i+1}\frac{(\log_2 x - \log_2\log_2 e) + 1}{(\log_2 x - \log_2\log_2 e + \frac{1}{\ln 2} + 1)}\frac{1}{2^{i-j-1}} \\
&< \sum_{j=1}^{i+1}\frac{1}{2^{i-j-1}} = 2 - \frac{1}{2^{i-2}}.
\end{aligned}$$

Thus, $A + B + C < \frac{2}{e}\frac{1}{2^{i-2}} + 2 - \frac{1}{2^{i-2}} + 32/3 + \frac{1}{3}\frac{1}{4^{i-1}} \leq 14$.
Therefore, $(t_i/x)\frac{1}{2^{i-2}} + \sum_{j=1}^{i+1}\frac{t_i}{t_j}\frac{1}{2^{i-j-1}} < 14$.

So, by setting $\rho = 14/5$, we get $\rho > 1/5 \left((t_i/x)\frac{1}{2^{i-2}} + \sum_{j=1}^{i+1} \frac{t_i}{t_j}\frac{1}{2^{i-j-1}} \right)$.
Summarizing, we get the following lemmas from the computation above.

Lemma 33. *To achieve a duration of $t = t_i/(14/5)$ for any job j, the LP solution uses at least $3(2^{i-1})$ units of resources for $0 \leq i \leq k$.*

Lemma 33 implies the following.

Lemma 34. *If the LP uses $2^i \leq r < 3(2^{i-1})$ units of resources and we round r down to $\overline{r} = 2^i$ where $0 \leq i \leq k$, then $t_i \leq (14/5)t$ where t is the duration from the LP solution.*

Lemma 35. *With $r < 1$ units of resource, the LP cannot achieve a duration of $t < x/2$ for job j.*

Proof. The first edge has resource-time tuples $\{\langle 0, x\rangle, \langle 1, 0\rangle\}$. To achieve a duration of $x/2$, the LP has to use $1/2$ unit of resource on the first edge. The second edge also has the same resource-time tuples $\{\langle 0, x\rangle, \langle 1, 0\rangle\}$, and it also takes $1/2$ unit of resource. Thus, the first two edges alone need 1 unit of resource to achieve a duration of $x/2$ for all l_j parallel edges of job j. □

Lemma 35 implies the following.

Lemma 36. *If the LP uses $r < 1$ unit of resource and we round r down to 0, then $t_i \leq 2t$, where t is the duration from the LP solution.*

Lemma 37. *If r rounded to $\overline{r}$ then $\overline{r} \leq (4/3)r$*

Proof. When we use $\overline{r} = 2^{i+1}$ units of resource after rounding, the LP uses at least $3(2^{i-1}) \leq r \leq 2^{i+1}$ units. Thus, $\overline{r} \leq (4/3)r$. □

From Lemma 34 and Lemma 37, we get the following theorem.

Theorem 18. *There is a $(4/3, 14/5)$ bi-criteria approximation algorithm for the discrete resource-time tradeoff problem with resource reuse along paths when the recursive binary duration function is used.*

4.3.4 Exact Algorithm for Series-Parallel Graphs

We consider now the special case in which the underlying DAG D is a series-parallel graph. A series-parallel graph G can be transformed into (and represented as) a rooted binary tree T_G in polynomial time by decomposing it into its atomic parts according to its series and parallel compositions (see, e.g., [107]). In T_G, the leaves correspond to the vertices of G. Internal nodes of T_G are labeled as "s" or "p" based on series or parallel composition. We associate each internal node v of T_G with the series-parallel graph G_v, induced by the leaves of the subtree rooted at v.

Let $T(v, \lambda)$ denote the makespan of G_v using $0 \leq \lambda \leq B$ units of resources where B is the resource budget. We want to solve for $T(s, B)$, where s is the root of T_G. This can be done using dynamic programming, solving for the leaves first, and then progressing upward to the root of T_G. We compute $T(v, \lambda)$ as follows which assumes that node v corresponds to job j if it is a leaf, otherwise it has two children v_1 and v_2.

If v is a leaf, $T(v, \lambda) = t_j(\lambda)$. If v is an internal node with left child v_1 and right child v_2, then we compute $T(v, \lambda)$ as follows. If v is an internal node with label "s", then $T(v, \lambda) = T(v_1, \lambda) +$

$T(v_2, \lambda)$. If v is an internal node with label "p", then $T(v, \lambda) = min_{0 \leq i \leq \lambda}[max(T(v_1, i), T(v_2, \lambda - i))]$.

There are $\mathcal{O}(m)$ nodes in T_G if G has m edges. For each node v we compute $T(v, \lambda)$ for $0 \leq \lambda \leq B$. Computing $T(v, \lambda)$ for any particular value of λ takes $\mathcal{O}(\lambda)$ time, since, if the node is a "p" node, then for $0 \leq i \leq \lambda$ we need to look up values $T(v_1, i)$. Thus, for any internal node v, it takes $\sum_{\lambda=0}^{B} \mathcal{O}(\lambda) = \mathcal{O}(B^2)$ time. As there are $\mathcal{O}(m)$ nodes in T_G, the (pseudo-polynomial) time complexity of the algorithm is $\mathcal{O}(mB^2)$.

4.4 NP-Hardness

In this section we give a variety of NP-hardness and inapproximability results related to the discrete time-resource tradeoff problem in the offline setting (i.e., when the entire DAG is available offline). All problems consider the version where there is resource reuse over paths, but they vary the cost-function, graph structure, and minimization goal. Section 4.4.1 gives several reductions from 1-in-3SAT. Theorem 19 gives a base reduction for the problem with general non-increasing duration function which will provide the ideas and structure for later more complex proofs. Theorems 20 and 21 adapt this proof to give constant factor inapproximability for the minimum-resource and minimum-makespan problems. Section 4.4.2 adapts the NP-hardness proof to apply when the cost function is restricted to be the recursive binary splitting and the k-way splitting.

Section 4.4.3 considers the problem in bounded treewidth graphs. We show weak NP-hardness by a reduction from Partition.

4.4.1 Reuse Over a Path with General Non-increasing Duration Function

Theorem 19. *It is (strongly) NP-hard to decide if there exists a solution to the (offline) discrete resource-time tradeoff problem, with resource reuse over paths and a non-increasing duration function, satisfying a resource bound B and a makespan bound T.*

Our proof is based on a polynomial-time reduction from the strongly NP-hard problem 1-in-3SAT [120]: Given n variables ($V_i, 1 \leq i \leq n$) and m clauses ($C_j, 1 \leq j \leq m$),with each clause a disjunction of three literals, is there a truth assignment to the variables such that each clause has exactly one true literal?

Variable gadget. The gadget for variable V consists of nodes $V^{(1)}$, $V^{(2)}$, $V^{(3)}$, $V^{(4)}$, $V^{(5)}$, and $V^{(6)}$ as shown in Figure 4.8(a). We show in the hardness proof that a variable gadget will get exactly one unit of extra resource, otherwise the makespan will be greater than the target makespan of 1. Sending one unit of resource to node $V^{(2)}$ (Figure 4.8(a)) corresponds to setting the variable V to TRUE and sending the unit of resource to $V^{(3)}$ corresponds to setting V to FALSE. The remaining vertices ensure the extra resource is used in the variable and not transferred into one of the clauses.

Clause gadget. The gadget corresponding to clause C has 10 vertices $C^{(i)}$ ($1 \leq i \leq 10$) as shown in Figure 4.8(b). Arcs $(C^{(1)}, C^{(2)})$, $(C^{(2)}, C^{(4)})$, $(C^{(1)}, C^{(3)})$ and $(C^{(3)}, C^{(4)})$ have resource-time pairs as $\{\langle 0, 1\rangle, \langle 1, 0\rangle\}$. If clause C has three literals V_i, V_j and V_k, then vertex $C^{(5)}$ is connected to the vertices $V_i^{(3)}$, $V_j^{(3)}$ and $V_k^{(2)}$. These vertices correspond to $\neg V_i$, $\neg V_j$ and V_k respectively. Vertex

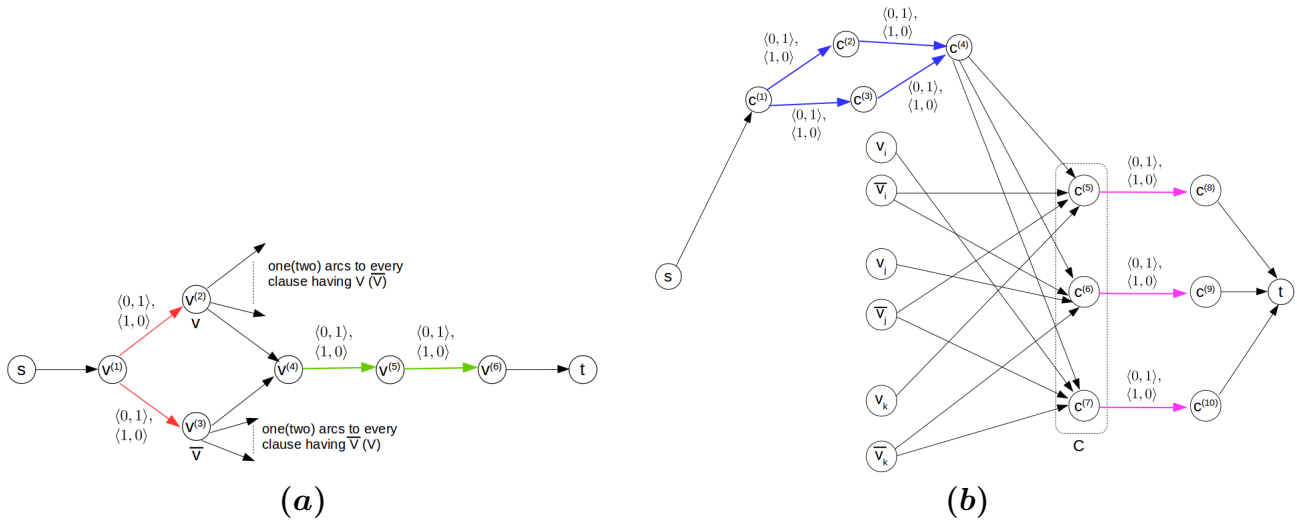


Figure 4.8: (a) Gadget for variable V, and (b) gadget for clause $C = (V_i \vee V_j \vee V_k)$ (Section 4.4.1).

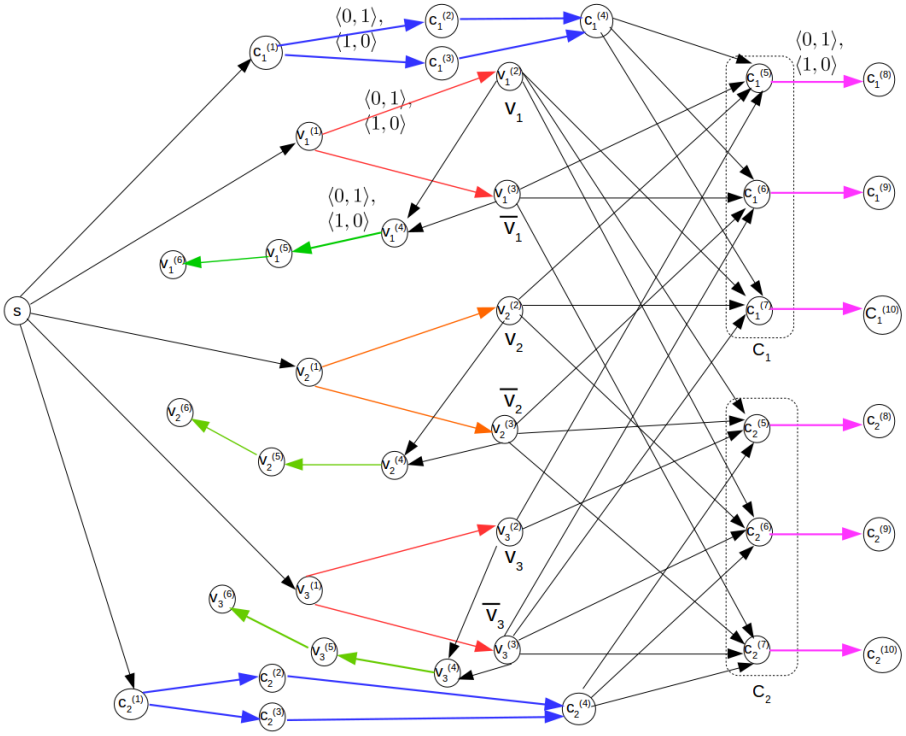


Figure 4.9: The complete construction for $(V_1 \vee \neg V_2 \vee V_3) \wedge (\neg V_1 \vee V_2 \vee V_3)$ is satisfiable with the truth assignment: $V_1 =$ TRUE, $V_2 =$ TRUE, $V_3 =$ FALSE (Section 4.4.1).

$C^{(6)}$ is connected to $V_i^{(3)}, V_j^{(2)}$ and $V_k^{(3)}$. These vertices correspond to $\neg V_i, V_j$ and $\neg V_k$. Vertex

$C^{(7)}$ is connected to $V_i^{(2)}, V_j^{(3)}$ and $V_k^{(3)}$. These vertices correspond to $V_i, \neg V_j$ and $\neg V_k$. Arcs $(C^{(5)}, C^{(8)})$, $(C^{(6)}, C^{(9)})$, and $(C^{(7)}, C^{(10)})$ have resource-time pairs as $\{\langle 0, 1\rangle, \langle 1, 0\rangle\}$. The part of the clause gadget consisting of $C^{(1)}, C^{(2)}, C^{(3)}$ and $C^{(4)}$ demand at least two units of memory be allocated there and then these units of resource go to satisfy two of $C^{(5)}, C^{(6)}$ and $C^{(7)}$. There is still one of these lines that has no allocated resource so it's cost is 1. Thus the corresponding variable must have had it's path length reduced (by setting it true).

Figure 4.9 shows the complete construction of $(V_1 \vee \neg V_2 \vee V_3) \wedge (\neg V_1 \vee V_2 \vee V_3)$ as an example.

Lemma 38. *There exists a solution to the input instance of 1-in-3SAT iff there exists a valid flow of resources through the DAG achieving a makespan of* 1 *under a resource bound of* $B = n + 2m$.

Proof. **Forward direction.** We prove that if there is a solution to the 1-in-3SAT instance with n variables and m clauses, then the reduced DAG has a solution of makespan 1 with $(n+2m)$ units of resource. If a variable V's truth assignment is TRUE, then we allow one unit of resource to flow through vertex $V^{(2)}$ along the path $\langle S, V^{(1)}, V^{(2)}, V^{(4)}, V^{(5)}, V^{(6)}, T\rangle$, otherwise we allow one unit of resource to flow through vertex $V^{(3)}$ along the path $\langle S, V^{(1)}, V^{(3)}, V^{(4)}, V^{(5)}, V^{(6)}, T\rangle$. For every clause C, we allow one unit of resource to flow through the path $\langle S, C^{(1)}, C^{(2)}, C^{(4)}\rangle$ and another unit of resource through the path $\langle S, C^{(1)}, C^{(3)}, C^{(4)}\rangle$. Thus, 2 units of resource can be flowed from vertex C^4. In a valid assignment of 1-in-3SAT, for each clause C, exactly 2 vertices of $C^{(5)}, C^{(6)}$ and $C^{(7)}$ will have the earliest start time of 1 and the other one will have 0 (Table 4.2).

Also, if only one literal is true in a clause, exactly two vertices among $C^{(5)}, C^{(6)}$ and $C^{(7)}$ need one unit of extra resource each to meet the makespan requirement (from Table 4.2). We are allowed to flow 2 units of resource from vertex $C^{(4)}$. Thus the project makespan is 1 using $(n + 2m)$ units of resource.

Backward direction. Now, we prove that if there exists a solution of makespan 1 using $(n+2m)$ units of resource in the reduced DAG, then there also exists a solution to the 1-in-3SAT instance. To achieve a makespan of 1, every variable gadget needs 1 unit of resource and each clause gadget needs 2 units of resource, otherwise the makespan would be greater than 1. Also, any resource that is used in a variable gadget cannot be used further in any other variable or clause gadget because the resource can be reused over a path only. Similarly, any resource that is used in any clause gadget, cannot be reused in any other gadget. Only one vertex that is either $V^{(2)}$ or $V^{(3)}$, will have the earliest start time 0. Both cannot be 0, as there is only 1 unit of resource per variable gadget. Both cannot be 1 as in a clause C where the literal V or $\neg V$ is present, each of $C^{(5)}, C^{(6)}$ and $C^{(7)}$ would have earliest starting time of 1. This requires use of 3 units of resource in the clause gadget C to achieve a makespan of 1. However, each clause gadget can have exactly 2 units of resource. Thus, for every variable, it has to be a valid assignment (V is set to either TRUE or FALSE). From Table 4.2, if a clause has exactly one TRUE literal, then the clause gadget requires 2 units of resource to achieve a makespan of 1. Otherwise, the clause gadget would have a makespan of 2 with the same amount of resource or would require more resource to achieve the target makespan of 1. Thus, each clause has exactly one TRUE literal. This satisfies the 1-in-3SAT instance. □

We also prove hardness of approximation, both for the minimum-makespan problem and for the minimum-resource problem. We begin with the minimum-makespan problem.

V_i	V_j	V_k	$C^{(5)}$	$C^{(6)}$	$C^{(7)}$
True	True	True	$max(1,1,0)=1$	$max(1,0,1)=1$	$max(0,1,1)=1$
False	True	True	$max(0,1,0)=1$	$max(0,0,1)=1$	$max(1,1,1)=1$
True	False	True	$max(1,0,0)=1$	$max(1,1,1)=1$	$max(0,0,1)=1$
True	True	False	$max(1,1,1)=1$	$max(1,0,0)=1$	$max(0,1,0)=1$
False	False	True	$max(0,0,0)=0$	$max(0,1,1)=1$	$max(1,0,1)=1$
False	True	False	$max(0,1,1)=1$	$max(0,0,0)=0$	$max(1,1,0)=1$
True	False	False	$max(1,0,1)=1$	$max(1,1,0)=1$	$max(0,0,0)=0$
False	False	False	$max(0,0,1)=1$	$max(0,1,0)=1$	$max(1,0,0)=1$

Table 4.2: Makespan at vertices $C^{(5)}$, $C^{(6)}$ and $C^{(7)}$ for different truth value assignments to V_i, V_j and V_k in Figure 4.8(b).

Theorem 20. *The minimum-makespan discrete resource-time tradeoff problem that allows resources to be reused only over paths cannot have a polynomial-time approximation algorithm with approximation factor less than* 2 *unless* $P = NP$.

Proof. We prove the theorem by contradiction. Let's assume that there is a polynomial time approximation algorithm with factor less than 2. Given a formula with n variables and m clauses, we construct the reduced DAG as described in the proof of Lemma 38. If the formula is a valid 1-in-3SAT instance, then OPT (i.e., the optimal solution) has a makespan of 1 using $(n + 2m)$ units of resource in the reduced DAG. The approximation algorithm will return a schedule with makespan less than 2 using $(n + 2m)$ units of resource. If the formula is not a valid 1-in-3SAT instance, then OPT's makespan is greater than or equal to 2. So, the approximation algorithm will have a schedule with makespan greater than or equal to 2. Thus, using a polynomial time algorithm one can solve a strongly NP-hard problem. This is a contradiction. Hence, there exists no polynomial time approximation algorithm for resource-time-reuse-path problem with factor less than 2 unless $P = NP$. □

Now, we turn attention to the minimum-resource problem:

Theorem 21. *The minimum-resource discrete resource-time tradeoff problem that allows resources to be reused only over paths cannot have a polynomial-time approximation algorithm with approximation factor less than* $3/2$ *unless* $P = NP$.

Proof. (Sketch) The proof uses a reduction from 1-in-3SAT; the construction is similar to that in the proof of Theorem 19, but has several key differences that make it considerably more intricate.

First, for each variable x_i we have a gadget similar to before (Figure 4.8(a)), with the option to send one unit of resource on one of two two-edge paths via a vertex, with the choice of which path indicating whether the variable is set to true or to false. Unlike the previous construction, we chain the variable gadgets together into a path of gadgets, from a source s to a sink t. Refer to Figure 4.10. A single unit of resource will be moved along the path, using one of each pair of two-edge paths, according to the truth assignments of the variables. A single directed edge, with options $\langle 1, 0\rangle$ and $\langle 0, M\rangle$, links variable x_i gadget to variable x_{i+1} gadget. Node s is connected to the variable x_1 gadget with an edge with $\langle 0, 0\rangle$. A property of this construction is that the entry node of the x_i gadget is reached by the unit of resource at exactly time $i - 1$, and the exit node of this gadget is reached at time exactly i. At time n the one unit of resource that traverses the

path of variable gadgets emerges at time n. Finally, there is also an edge directly from s to t with options $\langle 1, n\rangle$ and $\langle 0, M\rangle$. In total, two units of resource will be moved through this part of the DAG: one will follow a path through the variable gadgets, according to the truth assignments of the variables, and the other will go directly along the edge (s, t). Both units of resource will arrive at t at time n.

The clause gadget consists of three vertices, each representing a literal. Each clause has an entry vertex and an exit vertex, and they are chained into a path of gadgets, with clauses ordered in a specific way, as described below. Refer to Figure 4.11. The exit vertex of one clause has an edge connecting it to the next clause in the order; these edges have specially chosen duration values in order to serve as "buffers", as described below. The variable portion of the DAG feeds into the path of clause gadgets, with the 2 units of resource that arrive at t at time n moving along an edge that feeds into the first of the sequence of clause gadgets. Each of the three vertices of a clause gadget corresponds to a literal; each has an input edge coming from one of the two vertices of the variable gadget corresponding to the literal, according to whether the variable appears positively or negatively in the clause. These incoming edges have durations that are carefully chosen, so that the timing is as follows: For a clause with variables x_i, x_j, and x_k, the two units of resource (which came through the variable portion of the DAG before entering the path of clause gadgets) will arrive at the entry to the clause at exactly time $n + i + j + k$. The incoming edges from variables to the clause literals have durations chosen just so that the precedence constraints are satisfied "just in time", for the two units of resource to pass through the clause gadget literals that are *not* true (using edges with duration 0, based on the resource of 1), while the one true literal vertex (who was reached within the clause gadget via an edge of duration 1, instead of 0, since there was no resource associated with it) is reached 1 unit of time sooner (from the variable gadget), to compensate. The net result is that both units of resource emerge out of a clause at time $n + 1 + i + j + k$, ready to pass into the buffer and the next clause gadget. The buffers are selected carefully.

Then, we claim that we can achieve makespan A using just the 2 units of resource if and only if the variables are assigned to satisfy the 1-in-3SAT. If the variables are assigned in a way that does not yield all clauses to be true, then we will need at least 3 units of resource to achieve the target makespan. Thus, it is NP-hard to distinguish between needing 2 units and needing 3 units of resource. This implies that it is NP-hard to achieve an approximation ratio better than factor 3/2. □

4.4.2 Reuse Over a Path with Recursive Binary Splitting and k-Way Splitting

We have seen a (strong) NP-hardness proof (Theorem 19) for the discrete resource-time tradeoff problem with general non-increasing duration functions. In this subsection we strengthen this result by showing that the problem remains hard even when the duration functions arise from recursive binary split reducers and k-way split reducers. The proof uses the same general technique as in Section 4.4.1, but requires more complex gadgets to deal with the restricted duration functions.

Composite node. A composite node v of order k is a gadget of $(k + 2)$ nodes as shown in Figure 4.12. A composite node can have only one incoming edge and only one outgoing edge.

Without using any extra resource, a composite node of order k takes $(k+2)$ units of time to finish its activities. This is because there is one write operation on vertex v_1, one write operation on vertex v_i $(2 \leq i \leq k+1)$ and k write operations on vertex v_{k+2}. Using 2 units of resource with the k-way splitting function, all activities can be completed in $(2 + k/2 + 2) = (k/2 + 4)$ time. Similarly using 2 units of resource with recursive binary splitting function, all activities will be completed in $(2 + k/2 + \log 2 + 1) = (k/2 + 4)$ time. Thus using 2 units of resource, composite node v takes $(k/2 + 4)$ units of time using either function.

Variable gadget. The gadget for variable V consists of 3 composite nodes and other nodes as shown in Figure 4.13. Composite nodes $V^{(2)}$ and $V^{(2)}$ are of order $2x$. Composite node $V^{(4)}$ is of order $8x$. There is a chain of $4x$ nodes from $V^{(2)}$ to $V^{(5)}$ inclusive. Similarly there is a chain of $4x$ nodes from $V^{(3)}$ to $V^{(6)}$ inclusive. We will see that unless a variable gadget gets exactly 2 units of resource, its makespan will be greater than $(7x + 2y + 12)$ which we will use as the target makespan later in our hardness proof. The values of x and y will be described shortly. Sending 2 units of resource to node $V^{(2)}$ (Figure 4.13) corresponds to setting the variable V to TRUE and sending 2 units of resources to $V^{(3)}$ corresponds to setting V to FALSE. We will see that sending one unit of resource to $V^{(2)}$ and one unit of resource to $V^{(3)}$ will make the makespan greater than the target makespan.

Clause gadget. The gadget corresponding to clause C has 13 vertices $C^{(i)}$ $(1 \leq i \leq 13)$ as shown in Figure 4.14. Vertices $C^{(2)}$ and $C^{(3)}$ are composite nodes each of order $8x$. If clause C has three literals V_i, V_j and V_k, then vertex $C^{(5)}$ is connected to the vertices $V_i^{(6)}, V_j^{(6)}$ and $V_k^{(5)}$. These vertices correspond to $\neg V_i, \neg V_j$ and V_k respectively. Vertex $C^{(6)}$ is connected to $V_i^{(6)}, V_j^{(5)}$ and $V_k^{(6)}$. These vertices correspond to $\neg V_i, V_j$ and $\neg V_k$. Vertex $C^{(7)}$ is connected to $V_i^{(5)}, V_j^{(6)}$ and $V_k^{(6)}$. These vertices correspond to $V_i, \neg V_j$ and $\neg V_k$. There are 3 composite nodes $C^{(8)}, C^{(9)}$ and $C^{(10)}$ each of order $2x$. There is a chain of $7x + 11$ vertices from s to each vertex in $\{C^{(11)}, C^{(12)}, C^{(13)}\}$. We define the "earliest finish time" of a node v as the time when all the write operations at v are finished.

In a valid assignment of 1-in-3SAT, we show that for each clause C, exactly 2 vertices of $C^{(5)}, C^{(6)}$ and $C^{(7)}$ will have earliest finish time of $(6x + 5)$ and the other one will have earliest finish time of $(5x + 8)$. (Table 4.3)

Value of x. There is only one vertex $(V^{(7)})$ with out-degree zero in every variable gadget V. Also, in every clause gadget C, there are three vertices $C^{(11)}, C^{(12)}$ and $C^{(13)}$, each with zero out-degree. So, if we connect all such vertices to the sink vertex t, then in-degree at t will be $(n + 3m)$. Let k be the smallest power of 2 such that $k \geq (n + 3m)$. We perform a recursive binary splitting at vertex t. Let y be the height of the binary splitting at t where $y = \log k$. To make $8x > (7x + 2y + 12)$, we define $x = max\big((2y + 13), 8\big)$. Hence, the path from any vertex from $\{V^{(7)}, C^{(11)}, C^{(12)}, C^{(13)}\}$ to sink t will take time $2y$.

Truth value assignment. Setting variable V to TRUE implies sending 2 units of resource through composite vertex $V^{(2)}$. The corresponding earliest finish time at vertex $V^{(5)}$ is $1 + (x + 4) + 4x = 5x + 5$ and at vertex $V^{(6)}$ is $1 + (2x + 2) + 4x = 6x + 3$. Similarly, setting variable V to FALSE implies sending 2 units of resource through vertex $V^{(3)}$. The corresponding earliest finish time at vertex $V^{(5)}$ is $1 + (2x + 2) + 4x = 6x + 3$ and at vertex $V^{(6)}$ is $1 + (x + 4) + 4x = 5x + 5$.

Lemma 39. *There exists a solution to the input instance of 1-in-3SAT iff there exists a valid flow of resource through the reduced DAG achieving a makespan of at most* $7x + 2y + 12$ *using at most*

$2n + 4m$ *units of resource.*

Proof. **Forward direction.** We now prove that if there is a solution to the 1-in-3SAT instance with n variables and m clauses, then the reduced DAG has a makespan of $7x + 2y + 12$ with $2n + 4m$ units of resource.

If a variable V is set to TRUE, then we allow 2 units of resource to flow through vertex $V^{(2)}$ along the path $\langle S, V^{(1)}, V^{(2)}, V^{(4)}\rangle$, otherwise, we allow 2 units of resource to flow through vertex V^3 along the path $\langle S, V^{(1)}, V^{(3)}, V^{(4)}\rangle$. Assigning TRUE to variable V implies that the earliest finish times at vertex $V^{(5)}$ and $V^{(6)}$ are $5x + 5$ and $6x + 3$, respectively. Also, the earliest finish time at vertex $V^{(7)}$ is $1 + (2 + 2x) + 1 + 2 + (4x + 4) + x + 2 = 7x + 12$. In Figure 4.14, there are 3 writers from variable gadgets that write on each of the nodes in $\left\{C^{(5)}, C^{(6)}, C^{(7)}\right\}$. If there are multiple writers ready to write to the same vertex at the same time, we serialize the write operations. For example, if $V_i = TRUE, C_j = FALSE$ and $V_k = FALSE$, then the writer from variable gadget V_i is ready to write at time $5x + 5$. The writers from V_j and V_k are ready to write at time $6x + 3$. Hence, all three write operations can be completed at time $\max\{5x + 6, 6x + 4, 6x + 5\} = 6x + 5$. From Table 4.3, it is evident that in clause C, if only one literal is TRUE and the other two are FALSE, then among $C^{(5)}, C^{(6)}$ and $C^{(7)}$ only one vertex has an earliest finish time of $5x + 8$ and the other two have $6x + 5$. The vertex with starting time $5x + 8$, can finish the activity corresponding to composite node (one of $C^{(8)}, C^{(9)}$ and $C^{(10)}$) of order $2x$, in another $2x + 2$ units of time without using any resource. Hence, it will finish at time $5x + 8 + 2x + 2 = 7x + 10$. Each of the other two vertices with earliest finish time of $6x + 5$ takes 2 units of resource flowing from vertex $C^{(4)}$ and finishes the composite node's activity at time $(6x + 5) + (x + 4) = 7x + 9$. There is a chain of $7x + 11$ nodes from the source vertex to each of the vertices in $\left\{C^{(11)}, C^{(12)}, C^{(13)}\right\}$. Thus, the earliest finish time at each of those three vertices is $7x + 12$. Together, with $2y$ units of time to sink vertex t, the total makespan is $7x + 2y + 12$.
Backward direction. To achieve a makespan of $7x + 2y + 12$, every variable gadget requires 2 units of resource and each clause gadget requires 4, otherwise the makespan will be $8x$ which is larger than $7x + 2y + 12$ because $x > 2y + 12$. Also, any resource used in a variable gadget cannot be used further in any other variable or clause gadget because the resource can be reused over a path only. Similarly, any resource used in any clause gadget cannot be reused in any other gadget. Only one vertex that is either $V^{(5)}$ or $V^{(6)}$, will have the earliest finish time of $5x + 5$. Both cannot be $5x + 5$, as there is only 2 units of resource per variable gadget. Both cannot be $6x + 3$ as in a clause C where the literal V or $\neg V$ is present, there is an edge from either $V^{(5)}$ or $V^{(6)}$ to each of $C^{(5)}, C^{(6)}$ and $C^{(7)}$. This requires clause gadget C to get 6 units of resource to achieve a makespan $\leq 7x + 2y + 12$. But each clause gadget can have exactly 4 units of resource. Thus, for every variable V, for it to be a valid assignment, V is set to either TRUE or FALSE. From Table 4.3, if a clause has exactly one TRUE literal, then one of the vertices from $C^{(5)}, C^{(6)}$ and $C^{(7)}$ has the earliest finish time of $5x + 8$ and the other two have $6x + 5$. This requires to have 4 units of resource to achieve the earliest finish time $\leq 7x + 10$ at each of the vertices from $\left\{C^{(8)}, C^{(9)}, C^{(10)}\right\}$. This can be achieved by assigning 2 units of resource to those two composite nodes (from $C^{(8)}, C^{(9)}$ and $C^{(10)}$) that start executing at time $6x + 5$. The composite node that can start at time $5x + 8$ does not use any extra resource. If the clause does not have exactly one TRUE literal, then the clause gadget would require 6 units of resource to achieve the target makespan. However, we just argued that each clause gadget can have exactly 4 units of resource. Thus, each clause has exactly one TRUE literal and the 1-in-3SAT instance is also satisfied. □

V_i	V_j	V_k	$C^{(5)}$	$C^{(6)}$	$C^{(7)}$
T	T	T	$max(a, a+1, b) = a+1$	$max(a, b, a+1) = a+1$	$max(b, a, a+1) = a+1$
F	T	T	$max(b, a, b+1) = a$	$max(b, b+1, a) = a$	$max(a, a+1, a+2) = a+2$
T	F	T	$max(a, b, b+1) = a$	$max(a, a+1, a+2) = a+2$	$max(b, b+1, a) = a$
T	T	F	$max(a, a+1, a+2) = a+2$	$max(a, b, b+1) = a$	$max(b, a, b+1) = a$
F	F	T	$max(b, b+1, b+2) = b+2$	$max(b, a, a+1) = a+1$	$max(a, b, a+1) = a+1$
F	T	F	$max(b, a, a+1) = a+1$	$max(b, b+1, b+2) = b+2$	$max(a, a+1, b) = a+1$
T	F	F	$max(a, b, a+1) = a+1$	$max(a, a+1, b) = a+1$	$max(b, b+1, b+2) = b+2$
F	F	F	$max(b, b+1, a) = a$	$max(b, a, b+1) = a$	$max(a, b, b+1) = a$

Table 4.3: Earliest start time at vertices $C^{(5)}, C^{(6)}$ and $C^{(7)}$ for different assignment of truth values of variable V_i, V_j and V_k in Figure 4.14, where $a = (6x + 4)$ and $b = (5x + 6)$.

4.4.3 Underlying Bounded Treewidth Graph

Let $G(D)$ be the undirected graph obtained by ignoring the directedness of the edges of a given DAG D. In the case that $G(D)$ is a graph of bounded treewidth,[2] we show that the offline minimum-makespan and minimum-resource problems on D are (weakly) NP-hard. (Note that Theorem 19 proving the strong NP-hardness of the problems does not assume that the underlying undirected graph is of bounded treewidth.)

Theorem 22. *It is weakly NP-hard to decide if there exists a solution to the (offline) discrete resource-time tradeoff problem, with resource reuse over paths and a non-increasing duration function, satisfying a resource bound B and a makespan bound T, provided the undirected graph obtained by ignoring the directedness of the edges of the input DAG is of bounded treewidth.*

The proof of this theorem is based on a reduction from PARTITION[77]. The construction is shown in Figure 4.15. The input instance is a set $S = \{s_1, s_2, \ldots, s_n\}$ of n positive integers; let $B = \sum_{i=1}^{n} s_i$. The PARTITION problem asks if there is a partition of S into subsets S_1 and S_2 such that the sums of the values in the two subsets are the same (i.e., exactly $B/2$). In this construction we have a total of B resources to allocate in our program. The value M is chosen to be greater than $B/2$, the target makespan, ensuring that memory resources must be allocated to these nodes. This ensures that at least s_i units of resource pass through each $v_i^{(1)}$, constructing our numbers. From each $v_i^{(1)}$ there are two choices of nodes, $v_i^{(2)}$ and $v_i^{(3)}$, to pass the resources onto each of which will either utilize s_i resources or increase the makespan on that path by s_i. The pair also funnel the resources into a sink vertex $\overline{v_0}$ with a potential makespan cost of M which ensures that their resources cannot be passed along to nodes $v_j^{(2)}$ and $v_j^{(3)}$ to the right (i.e., $j > i$). Thus the top and bottom paths represent our two sets and for each v_i we must allocate s_i makespan to either the top or the bottom path. Thus a total makespan of $B/2$ can only be achieved iff there is a partition of the s_i's into two sets such that each set sums to $B/2$.

To see that the constructed graph has bounded treewidth, let $V_i = \{v_i^{(j)}\}$, where $1 \leq j \leq 7$. Vertices $v_i^{(7)}$ for $1 \leq i \leq n$ are connected to the sink vertex $\overline{v_0}$. Then G has a tree decomposition T with nodes S_i, $1 \leq i \leq n$, as shown in Figure 4.16, with S_i defined as follows: $S_1 = \{v_0, \overline{v_0}\} \cup V_1$; $S_i = \{v_0, \overline{v_0}\} \cup V_{i-1} \cup V_i$, for $2 \leq i \leq n$. We claim that T is a valid tree decomposition. It is

[2]Recall that a tree decomposition of a graph $G = (V, E)$ is a tree T with nodes $X_1, X_2, \ldots, X_n$, $X_i \subseteq V$, satisfying: (1) $\bigcup_i X_i = V$; (2) For edge $(u, v) \in E$ there exists a X_i with $u, v \in X_i$; (3) For any two nodes, X_i and X_j, in T, if node X_k is in the (unique) path between X_i and X_j in T, then $X_i \cap X_j \subseteq X_k$. The *width* of the tree decomposition is $\max_i |X_i| - 1$, and the *treewidth* of G is the minimum width over all tree decompositions of G.

evident that $\cup_{1 \leq i \leq n} S_i = V$. From the construction of S_j $(1 \leq j \leq n)$, it is clear that, for each edge (u, v) of the graph G, there exists a node S_j with $u, v \in S_j$. For any S_i and S_j, with $j > i+1$ and $1 \leq i \leq (n-2)$, we have $S_i \cap S_j = \{v_0, \overline{v_0}\}$, and, for any node S_k $(i < k < j)$, on the path between S_i and S_j, we have $v_0 \in S_k$ and $\overline{v_0} \in S_k$, so that $S_i \cap S_j \subseteq S_k$. Thus, T is a valid tree decomposition, and it has width 15 ($\max_i |S_i| - 1 = 15$), so the treewidth of G is at most 15.

4.5 Alternate hardness proof from numerical 3D matching

We give a polynomial-time reduction from the numerical 3-dimensional matching problem to the discrete resource-time tradeoff problem (with resource reuse over paths and a non-increasing duration function).

Numerical 3-dimensional matching problem: Given $A = \{a_1, a_2, \cdots a_n\}$, $B = \{b_1, b_2, \cdots b_n\}$, and $C = \{c_1, c_2, \cdots c_n\}$, partition $A \cup B \cup C$ into n triples $S_i \in A \times B \times C$ of equal sum $T = (\sum A + \sum B + \sum C)/n$.

Given an instance of the numerical 3D matching problem, we create a DAG D with source s and sink t as shown in Figure 4.18. For each $a_i \in A$, there is an edge (s, a_i) in D. The space-time tradeoff function at edge (s, a_i) is $\{\langle 0, \infty \rangle, \langle n, a_i \rangle\}$. Recall that, this means that with zero resource, it takes infinite time to finish the activity (s, a_i) and with n units of resource it finishes in time a_i. We create a gadget that has n incoming edges and n outgoing edges. We call the gadget a *bipartite matcher* (Figure 4.17) as it matches (a $1:1$ mapping) the incoming edges to the outgoing edges. We describe the bipartite matcher in the next paragraph. For each $b_i \in B$, there is an edge $(b_i, b_i^{'})$ in D. The tradeoff function at edge $(b_i, b_i^{'})$ is $\{\langle 0, \infty \rangle, \langle n, b_i \rangle\}$. We put all the n edges $(b_i, b_i^{'})$ to a bipartite matcher as its incoming edges. For each $c_i \in C$, there is an edge (c_i, t) in D. The tradeoff function at edge (c_i, t) is $\{\langle 0, \infty \rangle, \langle n, c_i \rangle\}$.

The bipartite matcher gadget. The gadget has n incoming edges at vertices $\{x_1, x_2, \cdots x_n\}$ and n outgoing edges from $\{z_1, z_2, \cdots z_n\}$. It maps the vertices from $\{x_1, x_2, \cdots x_n\}$ to those in $\{z_1, z_2, \cdots z_n\}$. The mapping is one to one. This works as follows. There are n units of incoming resource at each vertices x_i. Every outgoing edge (x_i, y_i^j) from x_i $(1 \leq j \leq n)$ has a tradeoff function $\{\langle 0, \infty \rangle, \langle 1, 0 \rangle\}$. Hence, each of the outgoing edges (x_i, y_i^j) from x_i gets one unit of resource. The tradeoff function at edge (y_i, z_i) is $\{\langle 0, \infty \rangle, \langle 1, 0 \rangle\}$ which forces y_i^j to send one unit of resource to y_i. The tradeoff function at edge $(y_i^j, z_j^{'})$ is $\{\langle 0, M \rangle, \langle 1, 0 \rangle\}$. Thus, if y_i^j sends one unit of resource to y_i, it cannot send any resource to $z_j^{'}$ forcing the activity $(y_i^j, z_j^{'})$ to take M units of time to finish. Here, $M > \max_{1 \leq i \leq n}(a_i) + \max_{1 \leq i \leq n}(b_i) + \max_{1 \leq i \leq n}(c_i)$. The tradeoff function at edge $(z_j^{'}, z_j)$ is $\{\langle 0, \infty \rangle, \langle n-1, 0 \rangle\}$. There are n incoming edges $(y_i^j, z_j^{'})$ to $z_j^{'}$. Out of these n incoming edges, $(n-1)$ edges flow $n-1$ units of resource to $z_j^{'}$ which are then used for the activity at $(z_j^{'}, z_j)$.

We now show the mapping through an example. Suppose x_1 is mapped to z_3. Then the corresponding flow is as follows: one unit of resource flows from y_1^3 to y_1. As the total incoming flow of resource at vertex y_1^3 is one, no resource flows from y_1^3 to $z_3^{'}$. However, one unit of resource flows from each y_i^3 except y_1^3 to $z_3^{'}$. The earliest start time (EST) along path $\langle x_1, y_1^3, z_3^{'} \rangle$ is $EST(x_1) + M$ while that along path $\langle x_i, y_i^3, z_3^{'} \rangle$ for $i \neq 1$ is $EST(x_i)$. This makes the earliest start time at $z_3^{'}$, $EST(z_3^{'}) = \max\{EST(x_1) + M, EST(x_i)\} = EST(x_1) + M$. This holds true because $M > \max_{1 \leq i \leq n}(a_i) + \max_{1 \leq i \leq n}(b_i) + \max_{1 \leq i \leq n}(c_i)$. Also, $n-1$ units of resource flow

to $z_3^{'}$ and they are used for the activity $(z_3^{'}, z_3)$ to finish in time 0. Observe that no y_1^i except y_1^3 can send resource to y_1. The gadget has a total resource-inflow of n^2. Each of $(z_i^{'}, z_i)$ requires $n-1$ units of resource that sums up to $n^2 - n$ units of resource. Each of (y_i, z_i) requires one unit of resource, that sum up to n units of resource. If two of y_1^i sends a unit of resource each to y_1, then the total resource left to be used by all $(z_i^{'}, z_i)$ is at most $n^2 - n - 1$. Thus at least one of $(z_i^{'}, z_i)$ won't get $n-1$ units of resource and will take infinite time. Hence, mapping x_i to y_j corresponds to flowing one unit of resource from y_i^j to y_i and vice-versa; this makes a one-to-one mapping from $\{x_1, x_2, \cdots x_n\}$ to $\{z_1, z_2, \cdots z_n\}$.

Lemma 40. *There exists a solution to a input instance of numerical 3D matching if and only if there exists a valid flow of resource in the DAG such that the makespan is $2M + T$ with resource bound $B = n^2$.*

Proof. If there is a solution in the input instance of numerical 3D matching, then there are n sets, each of type $\{a_i, b_j, c_k\}$ such that $a_i + b_j + c_k = T$. We use first bipartite matcher gadgets to map a_i to b_j and the second bipartite matcher to map $b_j^{'}$ to c_k. Each bipartite matcher contributes M in the makespan. $(s, a_i), (b_j, b_j^{'})$ and (c_k, t) adds T to the makespan. Thus the makespan is eaxctly $2M + T$.

If the reduced DAG admits a makespan of $2M + T$ using n^2 units of resource, then there is also a solution to the input instance of numerical 3D matching. From the construction of bipartite 3D matching, there is a one-to-one mapping from a_i to b_j and from $b_j^{'}$ to c_k. As the makespan is $2M + T$ and each bipartite matcher contributes M to the makespan, this gives a solution to numerical 3D matching. □

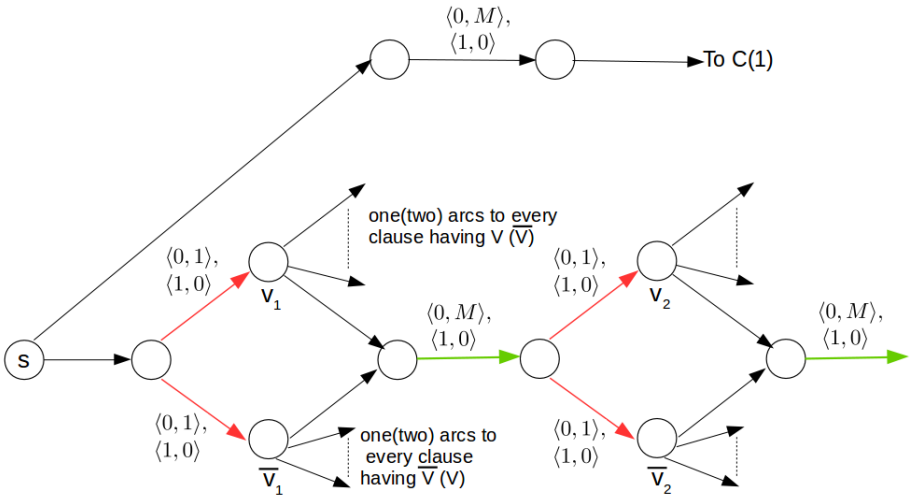


Figure 4.10: The variable gadgets chained together for the hardness of approximation of the minimum-resource problem (Theorem 21).

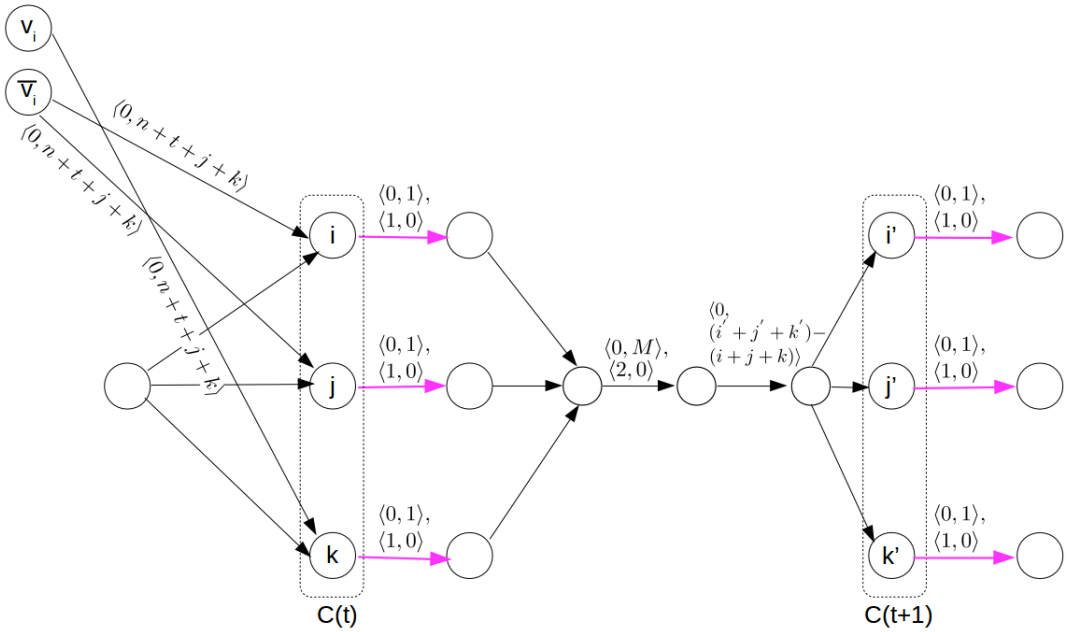


Figure 4.11: The clause gadgets chained together for the hardness of approximation of minimum-resource problem.

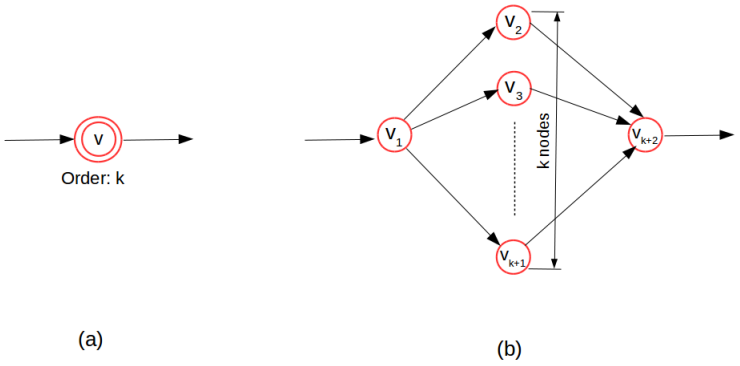


Figure 4.12: Composite node (Section 4.4.2).

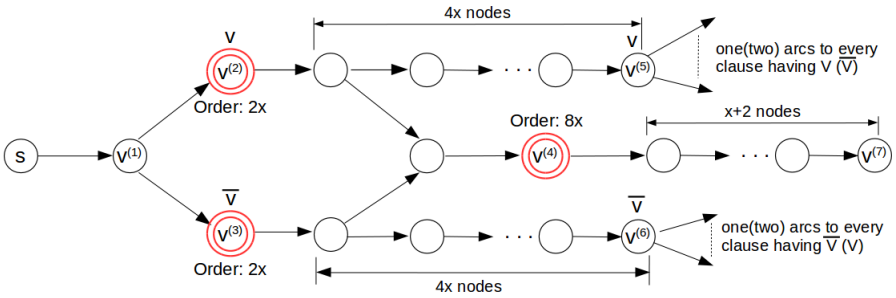


Figure 4.13: Gadget for variable V (Section 4.4.2).

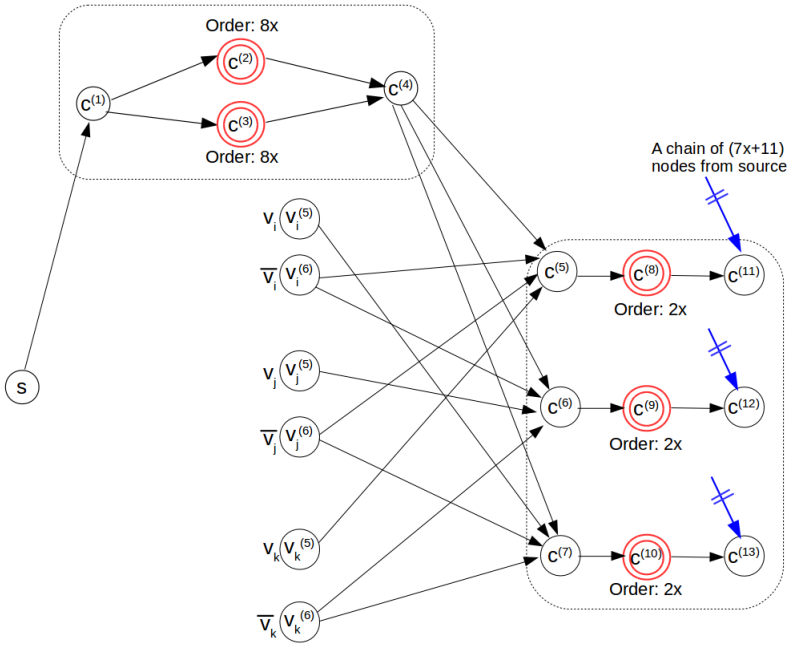


Figure 4.14: Gadget for clause $C = (V_i \vee V_j \vee V_k)$ (Section 4.4.2).

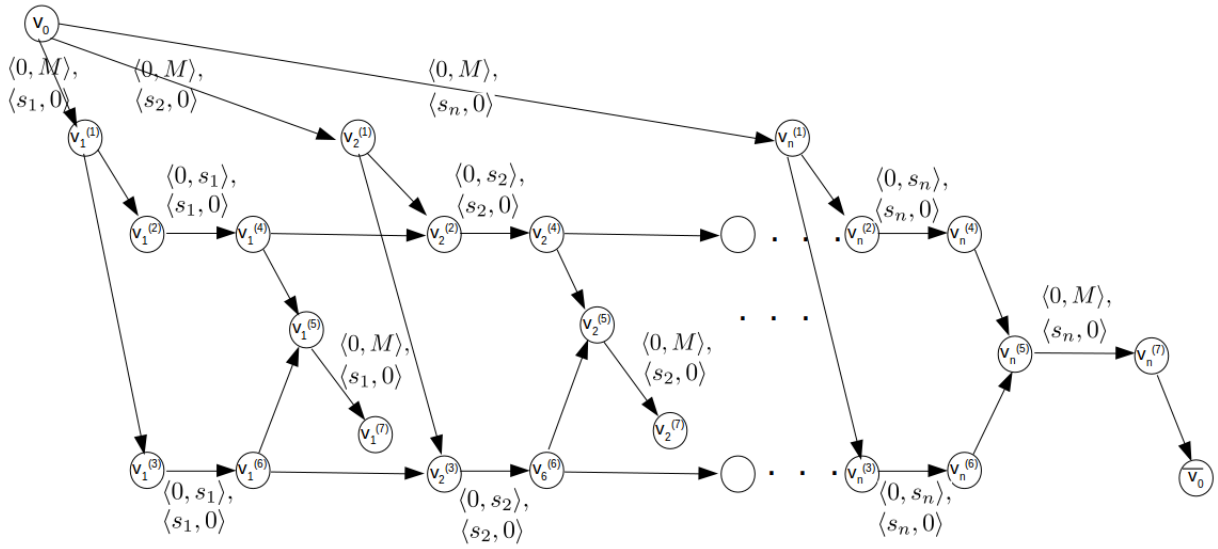


Figure 4.15: Construction for (weak) NP-hardness proof for graphs with bounded treewidth (Section 4.4.3).

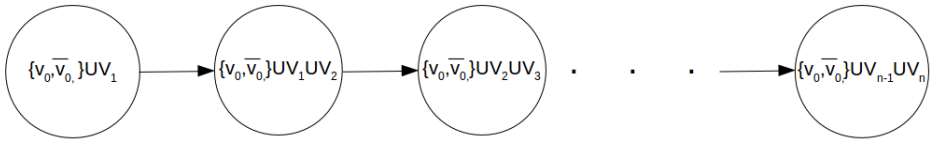


Figure 4.16: Tree decomposition of graph G (Section 4.4.3).

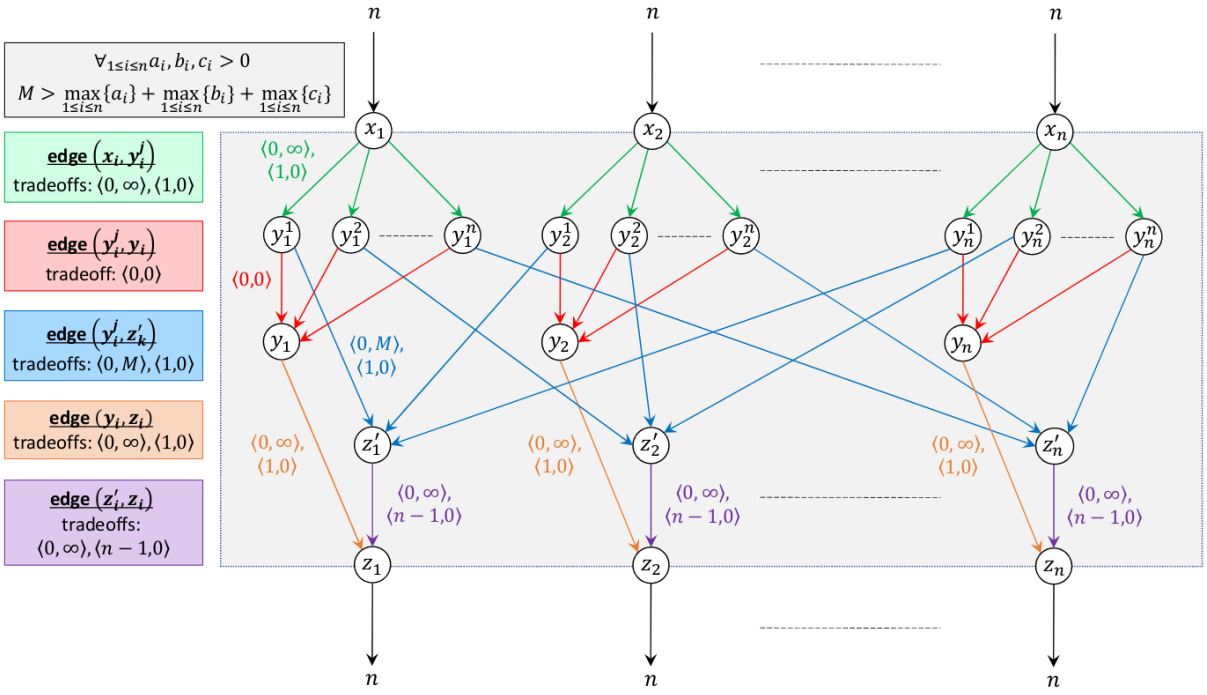


Figure 4.17: Bipartite matcher gadget (Section 4.5).

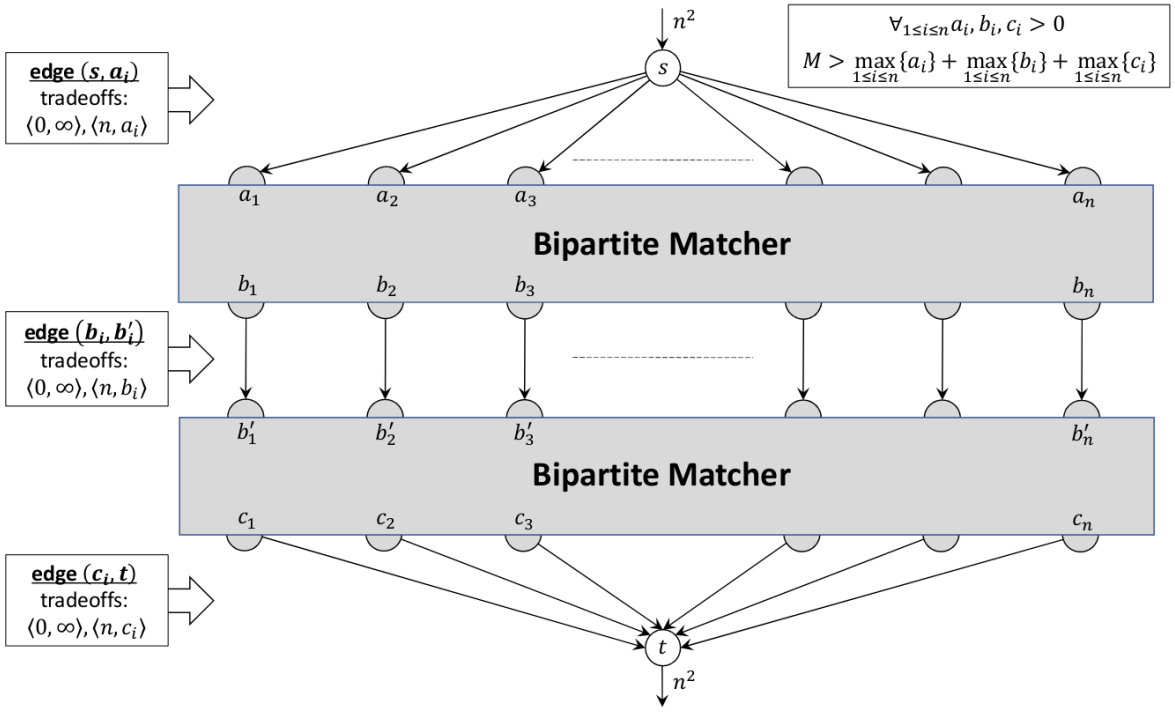


Figure 4.18: Reduced DAG from a numerical 3D matching instance (Section 4.5).

Chapter 5

Reducing Synchronization Cost with Extra Memory

The binary-forking model is a parallel computation model, formally defined by Blelloch et al., in which a thread can fork a concurrent child thread, recursively and asynchronously. The model incurs a cost of $\Theta(\log n)$ to spawn or synchronize n tasks or threads. The binary-forking model realistically captures the performance of parallel algorithms implemented using modern multithreaded programming languages on multicore shared-memory machines. In contrast, the widely studied theoretical PRAM model does not consider the cost of spawning and synchronizing threads, and as a result, algorithms achieving optimal performance bounds in the PRAM model may not be optimal in the binary-forking model. Often, algorithms need to be redesigned to achieve optimal performance bounds in the binary-forking model and the non-constant synchronization cost makes the task challenging.

We design efficient parallel algorithms in the binary-forking model without atomics for three fundamental problems: Strassen's (and Strassen-like) matrix multiplication (MM).

This work can also be found in Arxiv [11].

5.1 Introduction

We present efficient algorithms with optimal/near-optimal span[1] for several fundamental problems in the binary-forking model without locks and atomic instructions. The binary-forking model was introduced in Blelloch et al. [40] (see also [4, 24, 38, 39, 57]) to accurately capture the performance of algorithms designed for modern multi-core shared-memory machines. In this model, the computation starts with a single thread, and as the computation progresses, threads are created dynamically and asynchronously; the computation finishes when all threads end. A thread can spawn/fork a concurrent asynchronous child thread while it progresses simultaneously and such forking of threads can happen recursively; hence the model is called the binary-forking model. The model also includes a "join" operation to synchronize the threads. Though the model introduced in [40] allows the use of atomic instructions, we do not use them in this work.

[1] Span/depth is the running time of an algorithm with an unbounded number of processors.

The binary-forking model is closely related to the well-studied PRAM model [84]. The PRAM model is strictly more powerful than the binary-forking model; however, it does not correlate well with modern architectures. In the PRAM model, computation progresses in synchronous steps. Modern architectures employ new techniques such as use of multiple caches, processor pipelining, branch prediction, hyper-threading, and many more, which give rise to many asynchronous events such as cache misses, varying clock speed, interrupts, etc., thus demanding the development of a parallel computation model where computation can proceed asynchronously. Asynchronous thread creation in the binary-forking model makes it an ideal candidate for modeling parallel computation in modern architectures. As pointed out in [40], this is the model underlying many widely used parallel programming languages/environments such as Cilk [75], the Java fork-join framework [71], Intel TBB [136], and the Microsoft Task Parallel Library [133].

One can trivially reduce any algorithm designed for the PRAM model to an algorithm for the binary-forking model, incurring an $O(\log n)$-factor blow-up in the span while keeping the work[2] asymptotically the same as in the PRAM model. Spawning n threads takes $\Theta(1)$ time and $\Theta(n)$ work in the PRAM model—making the synchronization cost (span) constant. This synchronization can be simulated by using a binary tree of $\log n$ depth and $\Theta(n)$ nodes in the binary-forking model. Each internal node in the binary tree corresponds to a binary-forking operation, and the n leaves correspond to the n spawned threads.

A direct simulation of an optimal-span PRAM algorithm may not produce an algorithm with optimal span in the binary-forking model. For example, Cole's parallel merge sort [51] achieves optimal $\Theta(\log n)$ span and $\Theta(n \log n)$ work in the PRAM model. The binary-tree reduction increases the span to $\Theta\left(\log^2 n\right)$ while keeping the work asymptotically the same. On the other hand, by increasing work to $\Theta(n^2)$, it becomes trivial to get a $\Theta(\log n)$ span sorting algorithm — each item independently computes its rank in the final sorted list in $\Theta(\log n)$ time and $\Theta(n)$ work by comparing itself with all n elements. However, neither algorithm is optimal in the binary-forking model — the former has non-optimal span while the latter performs non-optimal work. Cole and Ramachandran [53] presented a deterministic sorting algorithm with $\mathcal{O}(\log n \log\log n)$ span and optimal $\Theta(n \log n)$ work in the binary-forking model. Recently, Ramachandran and Shi [116] gave a data-oblivious sorting algorithm in the binary-forking model with optimal work and $\mathcal{O}(\log n \log\log n)$ span. Very recently, Blelloch et al. [40] used atomic test-and-set operations to design a randomized sorting algorithm with $\Theta(\log n)$ span w.h.p.[3] in n and $\Theta(n \log n)$ work in expectation. Hence, finding an optimal (both in span and work) sorting algorithm without using atomic instructions remains an interesting and non-trivial open problem in the binary-forking model. We encounter the span blow-up problem when running other fundamental low-span PRAM algorithms, such as those for Strassen's matrix multiplication and Fast Fourier Transform (FFT), in the binary-forking model. Both algorithms have $\Theta(\log n)$ span in the PRAM model, which blow up by a factor of $\log n$ and $\log\log n$, respectively, in the binary-forking model.

Algorithms for the binary-forking model face a major challenge: how to avoid the blow-up in span (synchronization cost) without blowing up work? Surprisingly, it turns out that using extra space, we can tackle this challenge. By extra space, we mean the space allocated from RAM (from shared heap memory), not from processors private registers or stack memory.

[2]Work is the number of operations performed by a parallel algorithm on a serial computer.

[3]An event ξ occurs with high probability (w.h.p.) in n provided it occurs with probability at least $1 - \frac{c}{n^\alpha}$ for constants $\alpha \geq 1$ and $c > 0$.

An important specification of any parallel computation model is how the model resolves contentions when multiple processors try to read/write the same memory location. The two extremes of such contention rules are (1) exclusive: at most, one processor can read/write a memory location in each time step; (2) concurrent: any number of processors can read/write a memory location in each time step. Gibbons, Matias, and Ramachandran [78] proposed a more realistic contention rule, called concurrent-read-queued-write (CRQW), that reflects well modern parallel machines. In this rule, concurrent reads to a memory location take unit time, and concurrent writes to a memory location take time equal to the total number of concurrent writes to the location. As exclusive writes are too strict and concurrent writes are too relaxed (ignores high-contention cost), most of the modern parallel machines' contention properties reflect well the queued write rule instead of the exclusive or concurrent rules. In this work, we consider CRQW as the contention rule for the binary-forking model.

Our Contributions. We present an optimal $O(\log n)$ span algorithm for Strassen's Matrix Multiplication (MM) with only a $\Theta(\log\log n)$-factor blow-up in work as well as a near-optimal $O(\log n \log\log n)$ span algorithm with no asymptotic blow-up in work.

A remarkable feature of our algorithms is that we avoid the use of atomic instructions except possibly inside the join operations which are implemented by the runtime system.

Major Techniques. The extra $\log n$ factor in the span of the standard parallelization of Strassen's MM algorithm in the binary-forking model arises from the fact that it spends $\Theta\left(\log \frac{n}{2^i}\right)$ time (synchronization cost) computing intermediate results at recursion level i for each $i \in [0, \log_2 n]$ which requires only $\mathcal{O}(1)$ time in the PRAM model. We observe that none of those intermediate matrices need to be explicitly computed or stored to compute the final output. Indeed, each cell in the final output matrix can be computed directly in $\mathcal{O}(\log n)$ time from the two original input matrices of the algorithm. This ***single-point computation method*** can be used to compute all the cells in the output matrix simultaneously in $\Theta(\log n)$ span. However, this approach blows up the work performed by the algorithm by up to a $\Theta(n^2)$ factor because the approach does not reuse intermediate results. We avoid this work blow-up by computing and temporarily storing the intermediate results at $\Theta(\log\log n)$ carefully 'chosen levels' of recursion, which eliminates the need for implicitly recomputing the intermediate results over and over again. So, all single-point computations proceed in stages where a stage includes all levels of recursion between two consecutive 'chosen levels,' and synchronizations happen only at stage boundaries with all threads executing asynchronously within every stage. We show that this *stage-based approach* reduces the work blow-up from $\Theta(n^2)$ factor to only $\Theta(\log\log n)$ factor while achieving the optimal $\Theta(\log_2 n)$ span.

The technique described above works for all Strassen-like algorithms, including Victor Pan's $\mathcal{O}(n^{2.795})$ work algorithm [110]. We remark that while we use additional techniques specific to the problems to achieve better work and span bounds, the main contribution is devising the general technique to enable limited work-sharing among the single-point computations using extra space.

Binary-Forking Model. Binary-forking model captures the current multi-core shared-memory systems. Many parallel algorithms are based on binary-forking model [4, 24, 38, 39, 57]. Com-

putations in the binary-forking model can be viewed as a series-parallel DAG where each node represents a thread's instruction. The root of the tree is the first instruction of the starting thread. Each node has at most two children. If node u denotes the i-th instruction of thread t and u has only one child v, then v denotes the $(i+1)$-th instruction of thread t. If node u has two children v and w, then v represents the $(i+1)$-th instruction of thread t and w represents the first instruction of the new forked thread t'. The binary-forking model includes "join" instructions to join the forking threads. They are modeled as a node with two incoming edges. The work of the computation is the number of nodes in the series-parallel DAG and the span of the computation is the length of the longest path in the DAG assuming unbounded resources such as processors and space.

Performance Metrics of a Parallel Program. We use the *work-span* model [54] to analyze the performance of parallel programs executed on shared-memory multicore machines. The *work* of a multithreaded program, denoted by $T_1(n)$, where n is the input parameter, is defined as the total number of CPU operations it performs when executed on a single processor. The *span* $T_\infty(n)$ of a program which is also known as its *critical-path length* or *depth*, is the maximum number of operations performed on any single processor when the program is run on an unbounded number of processors. The *parallel running time* $T_p(n)$ of a program when run on p processors under a greedy scheduler is given by $T_p(n) = \mathcal{O}\left(T_1(n)/p + T_\infty(n)\right)$. The *parallelism*, computed by the ratio of $T_1(n)$ and $T_\infty(n)$, is the average amount of work performed by the program in each step of its critical path.

5.2 Strassen's Matrix Multiplication

Suppose $w = \log_2 7$. Strassen's matrix multiplication (MM) algorithm [131] performs $\mathcal{O}\left(n^w\right)$ work (i.e., multiplications and additions), unlike the classic MM algorithm that performs $\mathcal{O}\left(n^3\right)$ work. A straightforward parallelization of Strassen's MM leads to $\Theta\left(\log^2 n\right)$ span. Our goal is to design a parallel Strassen's MM in the binary-forking model without using locks and atomic instructions to achieve an optimal span of $\mathcal{O}\left(\log n\right)$ without affecting the work bound of $\Theta\left(n^w\right)$.

In this chapter, we present parallel Strassen MM algorithms (i) having optimal $\mathcal{O}\left(\log n\right)$ span and $\mathcal{O}\left(n^w \log\log n\right)$ work, i.e., work very close to that of the standard Strassen's MM; and (ii) having $\mathcal{O}\left(n^w\right)$ work and $\mathcal{O}\left(\log n \log\log\log n\right)$ span, i.e., very close to optimal span.

The core ideas and techniques used in our algorithms are as follows. We first perform *single-point computation*, i.e., computation of a single cell of the output matrix independently from that of other cells/entries. This implies that all cells of the output matrix are computed independently in $\mathcal{O}\left(\log n\right)$ span. However, as there is no work-sharing across multiple threads, the total work blows up to $\mathcal{O}\left(n^{w+2}\right)$. We enable partial work-sharing across threads by saving intermediate computations at carefully selected levels of recursion. By splitting the recursion tree into *stages* and allowing work-sharing across stages, we are able to reduce the work to very close to $\mathcal{O}\left(n^w\right)$. Hence, by using single-point computations in stages, we are able to obtain good work and span bounds. We use this algorithm to design other parallel Strassen's MM algorithms with different advantages.

5.2.1 k-way Strassen's MM.

The k-way Strassen's MM [131, 54], for $k \in [1, 7]$, executes the child nodes in exactly $\lceil 7/k \rceil$ parallel steps without executing more than k child nodes at a time.

Lemma 41 ([131, 54]). *The k-way Strassen's MM has a complexity of $\mathcal{O}(n^w)$ work, $\mathcal{O}\left(\log^2 n\right)$ span if $k = 7$, $\mathcal{O}\left(n^{\log_2 \lceil 7/k \rceil}\right)$ span if $k \neq 7$, $\mathcal{O}(n^2 \log n)$ space if $k = 4$, and $\mathcal{O}\left(n^{\max(2, \log_2 k)}\right)$ space if $k \neq 4$.*

Proof. The work, span, and extra space recurrences for the k-way Strassen's MM are as follows. If $n = 1$, then $T_1(n) = \mathcal{O}(1)$ and $T_\infty(n) = \mathcal{O}(1)$. If $n > 1$, then

$$T_1(n) = 7T_1(n/2) + \mathcal{O}\left(n^2\right), \qquad T_\infty(n) = \lceil 7/k \rceil\, T_\infty(n/2) + \mathcal{O}(\log n),$$
$$S_\infty(n) = kS_\infty(n/2) + \mathcal{O}\left(n^2\right).$$

Solving these recurrences, we have the lemma. □

The work of the k-way Strassen's MM for any value of k is $\mathcal{O}\left(n^{\log_2 7}\right)$. The k-way algorithm gives a trade-off between span and space. When $k = 1$, we get the standard Strassen's algorithm [131]. When $k = 7$, we get the standard parallel Strassen's MM [54] that spawns all the child nodes in parallel achieving $\mathcal{O}(\log^2 n)$ span and occupying $\mathcal{O}\left(n^{\log_2 7}\right)$ space.

5.2.2 Strassen-S MM.

In this section, we present a parallel Strassen's MM algorithm, as shown in Figure 5.2, that achieves the optimal span of $\mathcal{O}(\log n)$ with only a $\mathcal{O}(\log \log n)$ factor increase in the work compared with the classical sequential Strassen's MM algorithm. In this algorithm, we multiply two matrices U and V and store the matrix product in X. We first construct the required data structures as shown in Figure 5.3. We then compute the input matrices (U and V) at all nodes in the recursion tree in parallel in $\mathcal{O}(\log n)$ span. Finally, we compute the output matrix (X) at all nodes in the recursion tree in $\mathcal{O}(\log n)$ span.

[Step 1. Compute the Input Matrices.] Consider the standard 7-way parallel Strassen's MM. The height of the recursion tree is $\mathcal{O}(\log n)$ and in each level, the total cost of forking and synchronizing threads to compute the input matrices is $\mathcal{O}(\log n)$. Hence, the total span for computing input matrices at all nodes in the recursion tree is $\mathcal{O}\left(\log^2 n\right)$. We can reduce the span to $\mathcal{O}(\log n)$ using single-point computation.

A cell of an input matrix (U or V) at a node of the recursion depends on at most two cells of the corresponding input matrix at its parent node. This implies that each cell in an input matrix at a leaf node depends on at most $2^{\log n} = n$ cells in the corresponding input matrix at the root node. If we were to compute all input cells of all input matrices at all nodes, the total work would explode to $\mathcal{O}(n^w \times n) = \mathcal{O}(n^{w+1})$. To keep the work very close to $\mathcal{O}(n^w)$, we split the entire recursion tree into stages. We then use single-point computation of input cells in stages.

For this algorithm, we have $\mathcal{O}(\log \log n)$ stages so that the work performed in each stage is $\mathcal{O}(n^w)$. Using single-point computation in each stage, we are able to achieve the desired optimal span of $\mathcal{O}(\log n)$ limiting the total work to $\mathcal{O}(n^w \log \log n)$.

```
STRASSEN-S(X, U, V)                                                    ▷ X ← U × V
1. (Ū, V̄, X̄, U quad, V quad, X branch) ← CONSTRUCT-DATA-STRUCTURES(U, V)
2. COMPUTE-INPUT-MATRICES(Ū, 0, 0, U quad); COMPUTE-INPUT-MATRICES(V̄, 0, 0, V quad)
3. COMPUTE-OUTPUT-MATRICES(X̄, 0, 0, X branch)
4. X ← X̄[0][0]
```

```
COMPUTE-INPUT-MATRICES(Z, stage_id, root_id, quad)
1. height ← #levels in the stage; #leaves ← 7^height
2. parallel for node ← 0 to #leaves −1 do
3.     leaf_id ← (root_id −1) × #leaves + node
4.     parallel for i ← 0 to n − 1 do
5.         parallel for j ← 0 to n − 1 do
6.             Z[stage_id][leaf_id][i, j] ← COMPUTE-INPUT-CELL(Z, stage_id, leaf_id, i, j, n, height, quad)
7.     if not last stage then COMPUTE-INPUT-MATRICES(Z, stage_id + 1, leaf_id, quad)
```

```
COMPUTE-INPUT-CELL(Z, stage_id, node_id, i, j, n, height, quad)
1. if height = 0 then return Z[stage_id][node_id][i, j]
2. parent_id ← (node_id / 7), branch_id ← node_id mod 7
3. parallel for k ← 0 to 1 do
4.     t[k] ← 0; coeff ← quad[branch_id][k].coeff
5.     if coeff ≠ 0 then
6.         new_i ← n× quad[branch_id][k].shift_i + i; new_j ← n× quad[branch_id][k].shift_j + j
7.         t[k] ← coeff × COMPUTE-INPUT-CELL(Z, stage_id, parent_id, new_i, new_j, 2n, height −1, quad)
8. return t[0] + t[1]
```

```
COMPUTE-OUTPUT-MATRICES(Z, stage_id, root_id, branch)
1. height ← #levels in the stage; #leaves ← 7^height
2. parallel for node ← 0 to #leaves −1 do
3.     leaf_id ← (root_id −1) × #leaves + node
4.     if last stage then Z[stage_id][leaf_id][0,0] ← U[stage_id][leaf_id][0,0] × V[stage_id][leaf_id][0,0]
5.     else COMPUTE-OUTPUT-MATRICES(Z, stage_id + 1, leaf_id, branch)
6. parallel for i ← 0 to n − 1 do
7.     parallel for j ← 0 to n − 1 do
8.         COMPUTE-OUTPUT-CELL(Z, stage_id, root_id, i, j, n, height, branch)
```

Figure 5.1: The STRASSEN-S MM algorithm (part 1).

In this step, we compute the input matrices of all nodes in the recursion tree. The step consists of $h + 1$ sequential stages: $0, 1, \ldots, h$, as shown in Figure 5.4, such that the height of stage i is

COMPUTE-OUTPUT-CELL(Z, stage_id, node_id, i, j, n, height, branch)

1. **if** height $= 0$ **then return** Z[stage_id + 1][node_id][i, j]
2. shift_i $\leftarrow$ [$i > n/2$]; shift_j $\leftarrow$ [$j > n/2$] ▷ [ ] is the Iversion bracket
3. quad_id $\leftarrow$ 2 shift_i+ shift_j; new_i $\leftarrow$ $i - (n/2)$ shift_i; new_j $\leftarrow$ $j - (n/2)$ shift_j
4. **parallel for** $k \leftarrow 0$ **to** 6 **do**
5. $t[k] \leftarrow 0$; coeff $\leftarrow$ branch[quad_id][k]
6. **if** coeff $\neq 0$ **then**
7. child_id $\leftarrow$ (node_id -1) $\times 7 + k$
8. $t[k] \leftarrow$ coeff $\times$ COMPUTE-OUTPUT-CELL(Z, stage_id, child_id, new_i, new_j, $n/2$, height -1, branch)
9. **return** $\sum_{k=0}^{6} t[k]$

Figure 5.2: The STRASSEN-S MM algorithm (part 2).

branch_id	$k=0$ ⟨shift_i, shift_j, coeff⟩	$k=1$ ⟨shift_i, shift_j, coeff⟩
0	$\langle 0,0,1\rangle$	$\langle 1,1,1\rangle$
1	$\langle 1,0,1\rangle$	$\langle 1,1,1\rangle$
2	$\langle 0,0,1\rangle$	$\langle -,-,0\rangle$
3	$\langle 1,1,1\rangle$	$\langle -,-,0\rangle$
4	$\langle 0,0,1\rangle$	$\langle 0,1,1\rangle$
5	$\langle 1,0,1\rangle$	$\langle 0,0,-1\rangle$
6	$\langle 0,1,1\rangle$	$\langle 1,1,-1\rangle$

branch_id	$k=0$ ⟨shift_i, shift_j, coeff⟩	$k=1$ ⟨shift_i, shift_j, coeff⟩
0	$\langle 0,0,1\rangle$	$\langle 1,1,1\rangle$
1	$\langle 0,0,1\rangle$	$\langle -,-,0\rangle$
2	$\langle 0,1,1\rangle$	$\langle 1,1,-1\rangle$
3	$\langle 1,0,1\rangle$	$\langle 0,0,-1\rangle$
4	$\langle 1,1,1\rangle$	$\langle -,-,0\rangle$
5	$\langle 0,0,1\rangle$	$\langle 0,1,1\rangle$
6	$\langle 1,0,1\rangle$	$\langle 1,1,1\rangle$

quad_id	k						
	0	1	2	3	4	5	6
0	1	0	0	1	−1	0	1
1	0	0	1	0	1	0	0
2	0	1	0	1	0	0	0
3	1	−1	1	0	0	1	0

Figure 5.3: Data structures required for the STRASSEN-S MM algorithm. Left: $Uquad$ and $Vquad$. Right: $Xbranch$.

fixed at $c_i \log n$, where h and c_i are given below:

$$c_i = \begin{cases} 0 & \text{if } i = -1, \\ 1 - \alpha^{i+1} & \text{if } i \in [0, h-1], \text{ such that } w = \log_2 7, \alpha = \dfrac{1}{w-1}, \text{ and } h = \log_{w-1} \dfrac{\log n}{\log\log\log n}. \\ 1 & \text{if } i = h. \end{cases} \tag{5.1}$$

Please refer to Figure 5.2 (the COMPUTE-INPUT-MATRICES algorithm) for computing the input matrices (U and V) for all the leaf nodes in all stages. We start from stage 0. For any given stage, denoted by stage id, we can easily compute the topmost level, called the root level and the bottommost level, called the leaf level. It is also easy to list out all indices of the leaf nodes in a given stage. So, for all leaf nodes, for all cells in the input matrix in a particular leaf node, we invoke the function COMPUTE-INPUT-CELL. This function computes the value of a specific cell in the input matrix of a leaf node.

The working of the COMPUTE-INPUT-CELL algorithm is as shown in Figure 5.5 (left). The figure shows the way in which a highlighted cell in the U matrix at a leaf node with id 05 (in base-7 system, for simplicity) is computed. As the last digit of the index is 5, it means that the leaf node is the 5th child of its parent. From the logic of the Strassen's MM algorithm, we know that the U

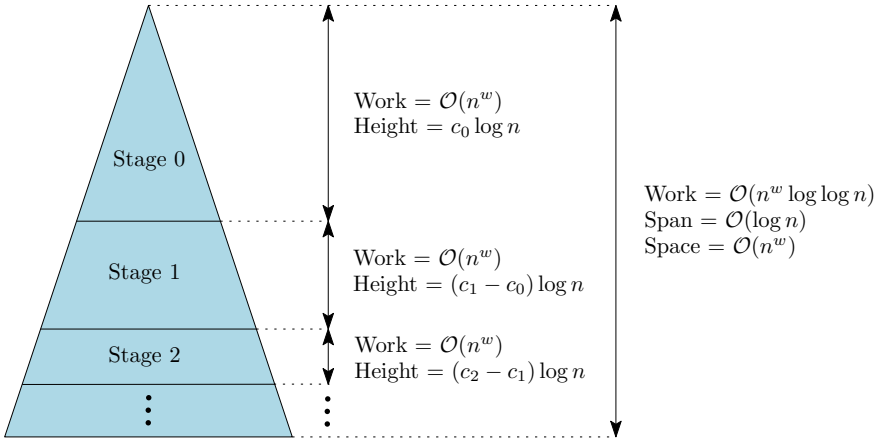
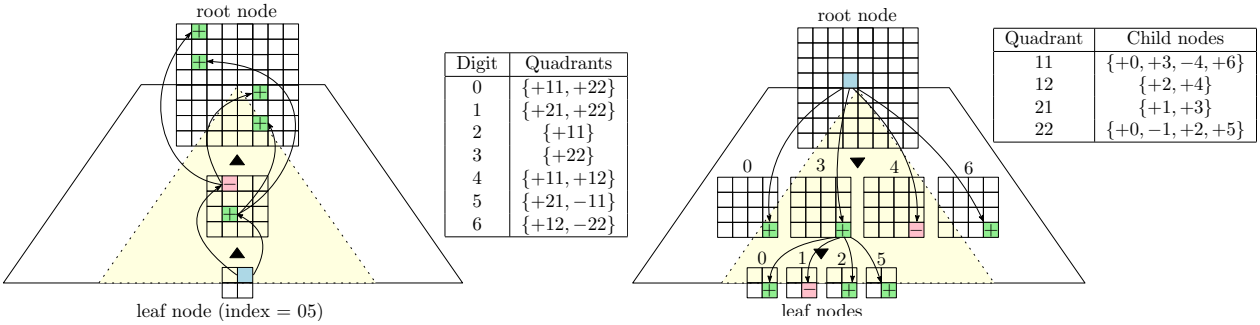


Figure 5.4: Stages in the STRASSEN-S MM algorithm for computing the input matrices.



Digit	Quadrants
0	$\{+11, +22\}$
1	$\{+21, +22\}$
2	$\{+11\}$
3	$\{+22\}$
4	$\{+11, +12\}$
5	$\{+21, -11\}$
6	$\{+12, -22\}$

Quadrant	Child nodes
11	$\{+0, +3, -4, +6\}$
12	$\{+2, +4\}$
21	$\{+1, +3\}$
22	$\{+0, -1, +2, +5\}$

Figure 5.5: Left: Single-point computation of a cell in the input matrix U at a leaf node in a stage. Right: Single-point computation of a cell in the output matrix X at a root node in a stage. (If there is an arrow from cell a to cell b, it means that cell a depends on cell b.)

matrix in the 5th child node of a parent node is computed by subtracting the first quadrant (-11) from the third quadrant $(+21)$ of the U matrix of the parent node. Hence, we can exactly know the two cells in the U matrix of the parent node on which the highlighted cell in the U matrix of the leaf node depends. Also, we can compute the highlighted cell in $\mathcal{O}(1)$ time using the two cells of the parent node. The first digit of the index of the lead node is 0. This means that the parent node of the leaf node is the 0th child of its parent (i.e., the leaf node's grandparent). From the logic of the Strassen's MM algorithm, we know that the U matrix in the 0th child node of a parent node is computed by adding the first quadrant $(+11)$ to the fourth quadrant $(+22)$ of the U matrix of the parent node. Using this approach, we can trace the path from the leaf node to its ancestor at the root level. So, each cell in the leaf node depends on 2 cells in its parent node which in turn depends on 4 cells in its parent node and so on until we reach a node at the root level. In this way, we can spawn multiple threads that recursively compute each cell at the leaf node using cells from its ancestor at the root level of the stage. The span for computing each cell is simply the height of the stage i.e, the number of levels in that stage.

Once all the cells in a leaf node with id 05 are computed, the algorithm recursively and asynchronously invokes COMPUTE-INPUT-MATRICES for the next stage with this leaf node as the new root. The base case of the COMPUTE-INPUT-MATRICES algorithm is when the algorithm reaches the last stage at which we compute the cells at the leaf nodes using the exact same idea.

Lemma 42. *COMPUTE-INPUT-MATRICES has a complexity of $\mathcal{O}(n^w \log\log n)$ work, $\mathcal{O}(\log n)$ span, and $\mathcal{O}(n^w)$ space.*

Proof. [Work.] We compute the input matrices in $h+1$ stages $S_0, S_1, \ldots, S_h$, where $h = \log_{w-1}(\log n / \log\log\log n)$. Suppose W_i define the work done at stage S_i. We first come up with a generic formula for W_i. We use a direct proof to show that $W_i = \mathcal{O}(n^w)$, which implies that the total work is $\sum_{i=0}^{h} W_i = \mathcal{O}(n^w \log\log n)$.

We compute W_i for $i \in [0, h-1]$. #Nodes at the leaf level of stage S_i is $7^{c_i \log n} = n^{wc_i}$. The #cells in a matrix at the leaf level is $(n/2^{c_i \log n})^2 = n^{2(1-c_i)}$. Each cell in a matrix at the leaf level depends on $\mathcal{O}(2^{(c_i - c_{i-1})\log n}) = \mathcal{O}(n^{c_i - c_{i-1}})$ cells in a matrix at the root level of the stage. Hence, $W_i = \mathcal{O}\left(n^{wc_i} n^{2(1-c_i)} n^{c_i - c_{i-1}}\right) = \mathcal{O}\left(n^{c_i(w-1) - c_{i-1} + 2}\right)$. To show that $W_i = \mathcal{O}(n^w)$ for all $i \in [0, h-1]$, it is enough to prove that $c_i(w-1) - c_{i-1} + 2 = w$. We substitute the values of c_i and c_{i-1} from equation 5.1 to get: $c_i(w-1) - c_{i-1} + 2 = (1 - (1/(w-1))^{i+1})(w-1) - (1 - (1/(w-1))^i) + 2 = w$.

We now compute W_h. The height of the first h stages is $c_{h-1} \log n$. So, the height of the last stage S_h is $\log n - c_{h-1} \log n$. Substituting the value of c_{h-1} from equation 5.1 and simplifying, we get the height of stage S_h as $\log\log\log n$. There are n^w nodes at S_h. The size of a matrix at a leaf node is 1×1. Each cell depends on $2^{\log\log\log n} = \log\log n$ cells in a matrix at the root level of stage S_h. Hence, work done at the last stage is $W_h = \mathcal{O}(n^w \log\log n)$.

Combining the work of the first h stages and the last stage, we get $T_1(n) = \sum_{i=0}^{h-1} W_i + W_h = \mathcal{O}(n^w \log\log n)$.

[Span.] Let $T_\infty(m, i)$ denote the span of the COMPUTE-INPUT-MATRICES algorithm starting from stage i where a matrix at the root level is of size $m \times m$. We give a recursive formula to compute $T_\infty(m, i)$. Then, the total span for the algorithm is $T_\infty(n, 0)$.

Consider the COMPUTE-INPUT-MATRICES algorithm. Let $\Delta c_i = c_i - c_{i-1}$. #Nodes at the leaf level of stage S_i is $7^{\Delta c_i \log n}$. Launching these nodes in parallel (line 2) incurs a span of $\mathcal{O}(\Delta c_i w \log n)$. A matrix at the leaf level will be of size $m/(2^{\Delta c_i \log n}) \times m/(2^{\Delta c_i \log n})$. Spawning COMPUTE-INPUT-CELL function for all cells (lines 4, 5) incur a span of $\mathcal{O}(2\log m - 2\Delta c_i \log n)$. Executing the COMPUTE-INPUT-CELL algorithm incurs $\mathcal{O}(\Delta c_i \log n)$ span. Adding all these spans give us $\mathcal{O}(\Delta c_i \log n + \log m)$.

The span of stage S_i recursively depends upon the span of stage S_{i+1} as the matrices at the leaf level of stage S_{i+1} are constructed from the leaf level matrices of stage S_i. Hence, $T_\infty(m, i)$ can be recursively defined using the previous analysis as: $T_\infty(m, i) = \mathcal{O}(\log\log\log n)$ if $i = h$ and $T_\infty(m, i) = \mathcal{O}(\Delta c_i \log n + \log m) + T_\infty(m^{1-\Delta c_i}, i+1)$ if $i < h$. Substituting the values of c_i from equation 5.1, we get $\Delta c_i = 1 - \alpha^{i+1} - (1 - \alpha^i) = \alpha^i(1-\alpha) = \mathcal{O}(\alpha^i)$. We know that m starts with n and decreases by a factor of $n^{\Delta c_i}$ for every stage. Hence, $m = n^{1-c_{i-1}} = n^{\alpha^i}$, which implies that $\log m = \alpha^i \log n$.

By unrolling the recursion and using the fact that α^i is a geometric series and $\alpha < 1$, we compute the total span as $T_\infty(n, 0) = \sum_{i=0}^{h-1} \alpha^i \log n + T_\infty(n, h) = \mathcal{O}(\log n)$.

[Space.] The total space is dominated by the space used by the data structures. There are n^w matrices at the leaf level for each of the input matrices U and V. Each such matrix is of size 1×1. Hence, space usage is $\mathcal{O}(n^w)$. □

[Step 2. Compute the Output Matrices.] The logic used to compute the output matrices is very similar to that used to compute the input matrices. A cell of the output matrix (X) at a node of the

recursion depends on at most four cells of the corresponding output matrices at its child nodes. This implies that each cell in the output matrix at the root node depends on at most $4^{\log n} = n^2$ cells in the corresponding output matrices at the leaf nodes. If we were to compute all output cells of all output matrices at all nodes, the total work would explode to $\mathcal{O}(n^w \times n^2) = \mathcal{O}(n^{w+2})$. To keep the work very close to $\mathcal{O}(n^w)$, we split the entire recursion tree into stages. We then use single-point computation of output cells in stages.

In this step, we compute the output matrix of all nodes in the recursion tree. The phase consists of $h+1$ sequential stages: $0, 1, \ldots, h$, as shown in Figure 5.4 (replace c_i's with d_i's), such that the height of stage i is fixed at $d_i \log n$, where h and d_i are given below:

$$d_i = \begin{cases} 0 & \text{if } i = -1, \\ 1 - \beta^{i+1} & \text{if } i \in [0, h-1], \\ 1 & \text{if } i = h. \end{cases} \text{ such that } w = \log_2 7, \beta = \frac{4-w}{2}, h = \log_{\frac{2}{4-w}} \frac{2 \log n}{\log \log \log n} - 1. \tag{5.2}$$

In step 1, we computed the input matrices in the top-down fashion. In contrast, in this step, we construct the output matrices at different recursion levels in a bottom-up fashion. In other words, we compute the last stage S_h first, then stage S_{h-1}, and so on until stage S_0. At stage S_0, the final output matrix X will be of size $n \times n$.

Please refer to Figure (the Compute-Output-Matrices algorithm) for computing the output matrix for all leaf nodes in all stages. We first descend the tree until we reach the last stage S_h. We know that all cells in the leaf nodes of this stage (or the recursion tree) already store the input U and V matrices using which we can compute the output matrices at that level. Using these output matrices at the leaf level of S_h, we compute the output matrices at the root level of S_h (or the leaf level of S_{h-1}). Using these matrices at the leaf level of S_{h-1}, we compute the output matrices at the root level of S_{h-1}. This process continues until we reach the root level of S_0 (or the root node of the entire recursion tree), which is the desired matrix product.

The way an output matrix at the root level is computed from the output matrices at the leaf level of stage S_i is as follows. For all cells in the output matrix at the root level, we invoke the function Compute-Output-Cell. This function computes the final value at that cell.

The working of the Compute-Output-Cell algorithm is shown in Figure 5.5 (right). The figure shows the way in which a highlighted cell in the output matrix at the root level is computed. The highlighted cell belongs to the first quadrant (11) of the output matrix. From the logic of the Strassen's MM algorithm, we know that the first quadrant of the output matrix of a node is computed by adding the output matrices of the 0th, 3rd, and 6th child nodes and subtracting that of the 4th child node. We can compute the highlighted cell from four cells in the next level in $\mathcal{O}(1)$ time. Now, consider the output cell in the 3rd child node of the root node. This cell belongs to the third quadrant (22) of that matrix. Again, from the logic of the Strassen's MM algorithm, the fourth quadrant of the matrix is computed by adding the output matrices of the 0th, 2nd, and 5th child nodes and subtracting that of the 1st child node. We continue the process until we reach the leaf level of that stage.

Once cells in the output matrices at the root level of a stage S_i are computed, the algorithm will proceed to computing the cells in the output matrices at the root level of stage S_{i-1} recursively until we reach the root node of the entire recursion tree.

Lemma 43. *COMPUTE-OUTPUT-MATRICES has a complexity of $\mathcal{O}(n^w \log\log n)$ work, $\mathcal{O}(\log n)$ span, and $\mathcal{O}(n^w)$ space.*

Proof. [Work.] We compute the output matrix in $h+1$ stages $S_0, S_1, \ldots, S_h$, where $h = \log_{w-1}(\log n / \log\log\log n)$. Suppose W_i defines the work done at stage S_i. We first come up with a generic formula for W_i. We use a direct proof to show that $W_i = \mathcal{O}(n^w)$, which implies that the total work is $\sum_{i=0}^{h} W_i = \mathcal{O}(n^w \log\log n)$.

We compute W_i for $i \in [0, h-1]$. Each output matrix at the root level of stage S_i is constructed from the output matrices at the leaf level of the stage. All cells in all output matrices at the root level of stage S_i are computed in parallel. #Nodes at the root level of stage S_i is $7^{d_{i-1}\log n} = n^{wd_{i-1}}$. Each such matrix has $(n^{1-d_{i-1}})^2 = n^{2(1-d_{i-1})}$ cells. Each cell at a recursion level ℓ depends on at most 4 output cells in recursion level $\ell+1$. Hence, each cell in a matrix at the root level of stage S_i depends on $\mathcal{O}(4^{(d_i-d_{i-1})\log n}) = \mathcal{O}\left(n^{2(d_i-d_{i-1})}\right)$ cells in a matrix at the leaf level of the stage. Hence, $W_i = \mathcal{O}\left(n^{wd_{i-1}} n^{2(1-d_{i-1})} n^{2(d_i-d_{i-1})}\right) = \mathcal{O}\left(n^{2d_i+(w-4)d_{i-1}+2}\right)$.

To show that $W_i = \mathcal{O}(n^w)$ for all $i \in [0, h-1]$, it is enough to prove that $2d_i + (w-4)d_{i-1} + 2 = w$. We substitute the values of d_i and d_{i-1} from equation 5.2 and simplify to get: $2d_i + (w-4)d_{i-1} + 2 = w$.

We now compute W_h. We see that $W_h = \mathcal{O}(n^{2d_h+(w-4)d_{h-1}+2})$ using the equation aforementioned. We substitute the values of d_h, d_{h-1}, h, and β from equation 5.2 and simplify to obtain $W_h = \mathcal{O}(n^w \cdot n^{(4-w)(1-d_{h-1})}) = \mathcal{O}(n^w \cdot n^{2\beta^{h+1}}) = \mathcal{O}(n^w \log\log n)$.

Combining the work of the first h stages and the last stage, we get $T_1(n) = \sum_{i=0}^{h-1} W_i + W_h = \mathcal{O}(n^w \log\log n)$.

[Span.] Let $T_\infty(m, i)$ denote the span of the COMPUTE-OUTPUT-MATRICES algorithm starting from stage i where a matrix at the leaf level is of size $m \times m$. We give a recursive formula to compute $T_\infty(m, i)$. Then, the total span for the algorithm is $T_\infty(n, 0)$.

Consider the COMPUTE-OUTPUT-MATRICES algorithm. Let $\Delta d_i = d_{i+1} - d_i$. #Nodes at the leaf level of stage S_i is $7^{\Delta d_i \log n}$. Launching these nodes in parallel (line 2) incurs a span of $\mathcal{O}(\Delta d_i w \log n)$. A matrix at the leaf level will be of size $m/(2^{\Delta d_i \log n}) \times m/(2^{\Delta d_i \log n})$. Spawning COMPUTE-OUTPUT-CELL function for all cells (lines 6, 7) incur a span of $\mathcal{O}(2\log m - 2\Delta d_i \log n)$. Executing the COMPUTE-OUTPUT-CELL algorithm incurs $\mathcal{O}(\Delta d_i \log n)$ span. Adding all these spans give us $\mathcal{O}(\Delta d_i \log n + \log m)$.

The span of stage S_i recursively depends upon the span of stage S_{i+1} as the output matrices at the leaf level of stage S_{i+1} are constructed from the leaf level matrices of stage S_i. Hence, $T_\infty(m, i)$ can be recursively defined using the previous analysis as: $T_\infty(m, i) = \mathcal{O}(\log\log\log n)$ if $i = h$ and $T_\infty(m, i) = \mathcal{O}(\Delta d_i \log n + \log m) + T_\infty(m^{1-\Delta d_i}, i+1)$ if $i < h$. Substituting the values of d_i from equation 5.2, we get $\Delta d_i = 1 - \beta^{i+2} - (1 - \beta^{i+1}) = \beta^{i+1}(1-\beta) = \mathcal{O}(\beta^i)$. We know that m starts with n and decreases by a factor of $n^{\Delta d_i}$ for every stage. Hence, $m = n^{1-d_{i-1}} = n^{\beta^i}$, which implies that $\log m = \beta^i \log n$.

By unrolling the recursion and using the fact that β^i is a geometric series and $\beta < 1$, we compute the total span as $T_\infty(n, 0) = \sum_{i=0}^{h-1} \beta^i \log n + T_\infty(n, h) = \mathcal{O}(\log n)$.

[Space.] Using a similar analysis as given in Lemma 43, space usage is $\mathcal{O}(n^w)$.

□

Theorem 23. *The STRASSEN-S MM algorithm has a complexity of $\mathcal{O}(n^w \log\log n)$ work, $\mathcal{O}(\log n)$ span, and $\mathcal{O}(n^w)$ space.*

Proof. The theorem follows from lemmas 42 and 43. □

5.2.3 Strassen-S-Adaptive MM.

We design a parallel Strassen's MM algorithm Strassen-S-Adaptive with space-span trade-off, which for any given s amount of space in the range $[n^2, n^w]$, achieves the optimal span for that space and performing work very close to $\mathcal{O}(n^w)$. Suppose we are given the input matrices U and V. We need to compute the output matrix X using space $s \in [n^2, n^w]$. Then, the algorithm works as follows. Observe that there are $\log n$ levels in the recursion tree of the Strassen's MM algorithm. We split the entire recursion tree, at level t, into two parts: the top part and the bottom part. The threshold level t depends on the value s. We execute the classical sequential Strassen's MM algorithm in the top portion of the recursion tree and the Strassen-S algorithm in the bottom portion of the tree.

Theorem 24. *The Strassen-S-Adaptive MM algorithm has a complexity of $\mathcal{O}(n^w \log\log n)$ work and $\mathcal{O}((n^w/s)\log n)$ span, given $\Theta(s)$ amount of space.*

Proof. Let $T_1(n,s), T_\infty(n,s)$, and $S_\infty(n,s)$ denote work, span, and space of Strassen-S-Tunable. Let $T_1(n), T_\infty(n)$, and $S_\infty(n)$ denote work, span, and space of Strassen-S. Note that Strassen-S does not have a space parameter.

We run the sequential Strassen's MM algorithm for the first t levels of the recursion tree. At level t, the size of the matrices is $n/2^t \times n/2^t$ and the number of matrices is 7^t. We have

$$
\begin{aligned}
T_1(n,s) &= 7T_1(n/2,s) + \mathcal{O}\left(n^2\right) = \cdots = 7^t T_1(n/2^t) + \mathcal{O}\left(7^t n^2\right) \\
T_\infty(n,s) &= 7T_\infty(n/2,s) + \mathcal{O}(\log n) = \cdots = 7^t T_\infty(n/2^t) + \mathcal{O}\left(7^t \log n\right) \\
S_\infty(n,s) &= S_\infty(n/2,s) + \mathcal{O}\left(n^2\right) = \cdots = S_\infty(n/2^t) + \mathcal{O}\left(n^2\right) = \mathcal{O}\left((n/2^t)^w + n^2\right)
\end{aligned}
$$

Equating the total space usage with s, we get $s = \Theta\left((n/2^t)^w + n^2\right)$. We simplify this expression to get the two expressions $n/2^t = \Theta\left((s-n^2)^{(1/w)}\right)$ and $7^t = \Theta\left(n^w/(s-n^2)\right)$. Substituting the two expressions in the span and work equations, we have

$$
\begin{aligned}
T_\infty(n/2^t) &= \mathcal{O}\left(\log^2(n/2^t)\right) = \mathcal{O}\left(\log^2 s\right) = \mathcal{O}(\log n) \\
T_\infty(n,s) &= 7^t T_\infty(n/2^t) + \mathcal{O}\left(7^t \log n\right) = \mathcal{O}\left((n^w/s)\log n\right) \\
T_1(n/2^t) &= (n/2^t)^w \log\log(n/2^t) \\
T_1(n,s) &= 7^t \cdot T_1(n/2^t) + \mathcal{O}\left(7^t n^2\right) = \mathcal{O}\left(n^w \log\log n\right)
\end{aligned}
$$

□

Strassen-W-Adaptive MM We can show that by using the sequential 1-way Strassen's MM until recursion level t (depends on s units of space) and then switching to the 7-way Strassen's MM instead of Strassen-S, we can achieve work bound the same as that of the classical Strassen's MM, but the span increases by an extra $\mathcal{O}(\log n)$ factor.

Theorem 25. *The Strassen-W-Adaptive MM algorithm has a complexity of $\mathcal{O}(n^w)$ work and $\mathcal{O}\left((n^w/s)\log^2 n\right)$ span, given $\Theta(s)$ amount of space.*

Proof. The proof is similar to that of Theorem 24 except that $T_\infty(n/2^t) = \mathcal{O}\left(\log^2 n\right)$ and $T_1(n/2^t) = (n/2^t)^w$. □

Corollary 2. *With $s = \Theta(n^2)$ units of space, (i) Strassen-S-Adaptive has a complexity of $\mathcal{O}(n^w \log\log n)$ work and $\mathcal{O}(n^{w-2}\log n)$ span and (ii) Strassen-W-Adaptive has a complexity of $\mathcal{O}(n^w)$ work and $\mathcal{O}\left(n^{w-2}\log^2 n\right)$ span.*

Strassen-W MM We ask the following question. If the work is bounded by $\mathcal{O}(n^w)$, what is the best span achievable by a parallel Strassen's MM algorithm? It turns out that with $\mathcal{O}(n^w)$ work bound one can achieve $\mathcal{O}(\log n \log\log\log n)$ span.

We split the entire recursion tree, at level $\log(n/(\log\log n)^{1/(w-2)})$, into two parts: the top and the bottom parts. We execute the Strassen-S in the top portion and the quadratic space Strassen-W-Adaptive algorithm in the bottom portion.

Theorem 26. *The Strassen-W MM algorithm has a complexity of $\mathcal{O}(n^w)$ work, $\mathcal{O}(\log n \cdot \log\log\log n)$ span, and $\mathcal{O}(n^w/\log\log n)$ space.*

Proof. There are m matrices of size $(n/m)\times(n/m)$ at the switching level t, where $m = n/(\log\log n)^{1/(w-2)}$. At each node at the threshold level, we add two matrices with two for loops. Adding two matrices has $\mathcal{O}(\log(n/m))$ span and $\mathcal{O}((n/m)^2)$ work. From Theorem 23, the span and work for Strassen-S are $\mathcal{O}(\log m \cdot \log(n/m))$ and $\mathcal{O}((m^w \log\log m)\cdot(n/m)^2)$ respectively. The span and work for Strassen-W-Tunable in m^w leaves are $\mathcal{O}\left((n/m)^{(w-2)}\log^2(n/m)\right)$ and $\mathcal{O}(m^w(n/m)^w)$ respectively. Combining the span from Strassen-S and Strassen-W-Tunable, we get the expression for span for the hybrid algorithm as follows.

When $m = n/(\log\log n)^{1/(w-2)}$, then $n/m = (\log\log n)^{1/(w-2)}$. We compute span as

$$\begin{aligned} T_\infty(n) &= \mathcal{O}\left(\log m \cdot \log(n/m) + (n/m)^{w-2}\log^2(n/m)\right) \\ &= \mathcal{O}(\log n \cdot \log\log\log n)\,. \end{aligned}$$

Combining work from both Strassen-S and Strassen-W-Tunable, we get the total work as:

$$\begin{aligned} T_1(n) &= \mathcal{O}\left((m^w \log\log m)(n/m)^2 + m^w(n/m)^w\right) \\ &= \mathcal{O}\left((n^w/(\log\log n)^{\frac{w}{w-2}})(\log\log n)^{\frac{2}{w-2}} + n^w\right) = \mathcal{O}(n^w)\,. \end{aligned}$$

We use $s = \Theta(n^w/\log\log n)$ space for the whole algorithm.

$$S_\infty(n) = \mathcal{O}\left(m^w(n/m)^2\right) = \mathcal{O}\left((n^w/(\log\log n)^{\frac{w}{w-2}})(\log\log n)^{\frac{2}{w-2}}\right) = \mathcal{O}(n^w/\log\log n)\,.$$

□

Strassen-like MM Algorithms. Let recursive algorithm ALG multiply two input matrices U and V of size $n\times n$ and produce output matrix X, that is $X = U\cdot V$. ALG divides U into $m\times m$ blocks each of size $(n/m)\times(n/m)$. Similarly, ALG divides the other input matrix V and output matrix X. Suppose that algorithm ALG has R recursive calls in each level of recursion. Then ALG creates R temporary matrices each for both input matrices and the output matrix. In particular,

the computation of ALG in each level of recursion is as follows. For each $r = 0, 1, \ldots, R-1$, Here $A^{(r)}, B^{(r)}$ and $C^{(r)}$ represent temporary matrices.

$$
\begin{aligned}
&A^{(r)} \longleftarrow 0; B_{(r)} \longleftarrow 0 \\
&A^{(r)} \longleftarrow A^{(r)} + \alpha_{i,k}^{(r)} U_{i,k} \text{ for } i, k = 0, 1, \cdots, m-1 \\
&B^{(r)} \longleftarrow B^{(r)} + \beta_{k,j}^{(r)} V_{k,j} \text{ for } k, j = 0, 1, \ldots, m-1 \\
&C^{(r)} \longleftarrow A^{(r)} \cdot B^{(r)} \\
&X_{i,j} \longleftarrow X_{i,j} + \gamma_{i,j}^{(r)} C^{(r)} \text{ for } i, j = 0, 1, \ldots, m-1.
\end{aligned}
$$

We call ALG a Strassen-like MM algorithm [105]. Sequential algorithm ALG does $\mathcal{O}(n^w)$ work where $w = \log_m R$. In Strassen, $m = 2$ and $R = 7$.

It is important to observe that the approaches used by Strassen-S, Strassen-W, Strassen-S-Adaptive, and Strassen-W-Adaptive MM algorithms apply to all Strassen-like MM algorithms.

5.3 Lower Bounds.

We give the following lower bound of any parallel version of Strassen's MM algorithm using s units of space in the binary-forking model.

Theorem 27. *Let A be a parallel version of Strassen's MM algorithm which uses s units of extra space. Then A's span is $\Omega(\max(n^w/s, \log n))$ in the binary-forking model.*

Proof. We consider the binary-forking model without atomics [40]. In this model, every binary operation (addition, subtraction, multiplication and division) is associated with a memory location. Specifically, the output of such binary operation needs to be written to a memory location. Concurrent reads from a memory location are allowed, while concurrent writes are not.

Let A be a matrix multiplication serial algorithm with $\Theta(n^w)$ work where $w \geq 2$. Let $T_\infty(n, w, s)$ denote the span of any parallel algorithm B that parallelize the serial algorithm A using s units of space. We do not make any restriction on the number of processors used. We remark that only heap space is counted in s (all our previous algorithms also allocate space from heap memory).

We get the first lower bound as follows. When we use s units of memory locations, then from the pigeonhole principle, there must exist a memory location that is subjected to $\Theta(n^w/s)$ write operations. As concurrent writes to a memory location are not allowed, the following lower bound holds: $T_\infty(n, w, s) = \Omega(n^w/s)$.

We get the second lower bound from a more general PRAM CREW model. Thus, it holds for a more restricted binary-forking model. When we have an unbounded #memory locations and unbounded #processors, the following lower bound [84] holds for the computation of $x_1 + x_2 + \cdots + x_n$ (array-sum) where every $s_i \in \{0, 1\}$: $T_\infty(n, w, \infty) = \Omega(\log n)$.

As matrix multiplication is at least as hard as array-sum, it must have $\Omega(\log n)$ span. Combining both the lower bounds, we get the following: $T_\infty(n, w, s) = \Omega(\max(n^w/s, \log n))$. □

Chapter 6

Conclusion

In this dissertation we tackle three vital challenges that arise in shared-memory multiprocessor systems: (1) efficiently managing the memory shared by multiple processors, (2) avoiding race conditions with minimal loss in parallelism, and (3) synchronizing processors efficiently to minimize span. The contribution of this dissertation is thus three-fold. (1) To handle the first challenge, we develop an algorithmic foundation for automated management of shared-memory in multilevel-memory systems; (2) to handle the second challenge, we give provably good algorithms that schedule memory in a parallel computation to avoid race conditions and achieve optimal or near-optimal span (parallel running time); (3) to handle the third challenge, we present techniques that take into account synchronization costs among the processors and yet achieve optimal span at the cost of slightly increasing work.

We summarize the contribution of each chapter as follows.

6.1 Algorithmic Foundation of Parallel Paging

The fundamental result of this work is that a seemingly unrelated problem called green paging, with a memory capacity between k and k/p, is essentially equivalent to parallel paging, with memory capacity k and p processors. Moreover, we obtain tight bounds for both. It is interesting to note that, informally, an optimal parallel solution must also be "green"; this validates the folklore principle in "practical parallelism" that extra parallelism should never be purchased at a(n excessive) cost in work-efficiency.

A crucial difference in parallel paging from classic paging, or even from pure page replacement with variable memory capacity, is that any online parallel paging algorithm is at best $\Theta(\log p)$ competitive, even with $O(1)$ resource augmentation: i.e., informally, online memory allocation is much harder than online page replacement, at least in the presence of significant parallelism or significant capacity flexibility. Crucially, the source of this $\log p$ factor appears to be the lack of knowledge about future locality of the computation. In practice, future locality can often be gauged, whether by profiling or simply because it does not change too often during the course of a computation. It would be interesting to incorporate this knowledge into our model, showing if and to what extent it can rid us of the $\log p$ factor. Also, we strongly believe that the $\log p$ factor can in fact subsume the logarithmic factor in ε when translating an efficient green paging strategy into a parallel paging algorithm with a low completion time for the all but the εp slowest

processors (and that eliminating one will eliminate the other); this is also an obvious avenue for future work.

We would also remark that to simplify our analysis we have often used constant factors that are larger that necessary. Reducing constants, both theoretically and experimentally, through better analysis and more sophisticated algorithms, would be crucial to the practicality of our (or in fact any) general scheme for both green and parallel paging.

6.2 How to Manage High-Bandwidth Memory Automatically

There are a number of relevant theoretical and practical questions we leave for future work. For instance, we would like to understand the practical impact of this work. Motivated by the discussion of the work by Butcher et al. [45] in Section 3.1.2, suppose that an HBM manufacturer like Intel had known about prioritized far-channel arbitration when they were creating the relevant system software. Might a differently-designed cache mode have given (for example) GNU::parallel sort a speedup of 1.5x over what it gets now? From the algorithm design perspective, we can ask the following questions: (1) Is there a solution to the makespan problem when the request sequences are not disjoint? (2) Relatedly, we currently consider the p cores to be running their own independent sequential jobs; what if the cores were running one (or more) parallel job(s)? (3) What kind of far-channel arbitration policy works with other block replacement policies, for instance, direct-mapped cache? (4) What if we made our model more general such that the far channel bandwidth was asymptotically larger than 1, but still asymptotically smaller than p?

6.3 Avoiding Races with Extra Memory

We introduce the discrete resource-time tradeoff problem with resource reuse, in which each unit of resource is routed along a source to sink path and is possibly used and reused to expedite activities encountered along that path. We assume that a general duration function (i.e., time needed to complete an activity as a function of the amount of resources used) is associated with each activity. We consider two different objective functions: (1) optimize makespan given a limited resource budget and (2) optimize resource requirements given a target makespan.

Our original motivation was to mitigate the cost of data races in shared-memory parallel programs. We achieve this by using extra space to reduce the time it takes to perform conflict-free write operations to shared memory locations. We consider three types of duration functions: general non-increasing functions, recursive binary reduction functions, and multiway (k-way) splitting functions.

We present the first hardness and approximation hardness results as well as the first approximation algorithms for our problems. We show that the makespan optimization problem is strongly NP-hard under all three types of duration functions. When the duration function is general non-increasing we also show that it is strongly NP-hard to achieve an approximation ratio less than 2 for the makespan optimization problem and less than $\frac{3}{2}$ for the resource optimization problem. We give a $\left(\frac{1}{\alpha}, \frac{1}{1-\alpha}\right)$ bi-criteria (resource, makespan) approximation algorithm for that same duration function, where $0 < \alpha < 1$. We present an improved $\left(\frac{4}{3}, \frac{14}{5}\right)$ bi-criteria approxima-

tion algorithm for the recursive binary reduction function. We also give 4- and 5-approximation algorithms for the makespan optimization problem under recursive binary reduction functions and multiway (k-way) splitting functions, respectively.

6.4 Reducing Synchronization Cost with Extra Memory

We presented several fundamental low-span algorithms in the binary-forking model without using locks and atomic instructions. Our parallel algorithms perform work (almost) the same as that of the serial algorithms from which they are derived. All our results improve known results in the binary-forking model with and without atomics.

We introduced the technique of *single-point computation in stages* through Strassen's MM to carefully set a balance between high work-sharing and high span of the given algorithm and low work-sharing and low span of the single-point computation variant to obtain parallel algorithms with optimal/near-optimal span without work blow-up. This technique can be used to design efficient parallel algorithms for other problems too.

An interesting open question that remains is whether there exists a parallel Strassen's MM algorithm that performs $O(n^w)$ work and achieves $O(\log n)$ span simultaneously.

Bibliography

[1] High-performance on-package memory, January 2015. `http://www.micron.com/products/hybrid-memory-cube/high-performance-on-package-memory`.

[2] Fast, multi-threaded malloc() and nifty performance analysis tools. http://code.google.com/p/gperftools/, Google gperftools.

[3] Lockless memory allocator. http://locklessinc.com/, llalloc.

[4] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *ACM symposium on Parallel algorithms and architectures*, pages 1–12, 2000.

[5] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 305–314, May 1987.

[6] A. Aggarwal, A. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, pages 3–28, March 1990.

[7] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[8] K. Agrawal, M. Bender, R. Das, W. Kuszmaul, E. Peserico, and M. Scquizzato. Brief announcement: Green paging and parallel paging. In *Proc. 32st ACM on Symposium on Parallelism in Algorithms and Architectures*, 2020.

[9] K. Agrawal, M. A. Bender, R. Das, W. Kuszmaul, E. Peserico, and M. Scquizzato. Brief announcement: Green paging and parallel paging. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 493–495, 2020.

[10] K. Agrawal, M. A. Bender, R. Das, W. Kuszmaul, E. Peserico, and M. Scquizzato. Tight bounds for parallel paging and green paging. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3022–3041. SIAM, 2021.

[11] Z. Ahmad, R. Chowdhury, R. Das, P. Ganapathi, A. Gregory, and M. M. Javanmard. Low-depth parallel algorithms for the binary-forking model without atomics. *arXiv preprint arXiv:2008.13292*, 2020.

[12] M. Aigner, C. M. Kirsch, M. Lippautz, and A. Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *ACM SIGPLAN Notices*, volume 50, pages 451–469. ACM, 2015.

[13] C. Akkan, A. Drexl, and A. Kimms. Network decomposition-based benchmark results for the discrete time–cost tradeoff problem. *European Journal of Operational Research*, 165(2):339–358, 2005.

[14] S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *SODA*, volume 99, pages 31–40. Citeseer, 1999.

[15] M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. In *Proc. 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 580–589, 1996.

[16] M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. *Algorithmica*, 32(2):277–301, February 2002.

[17] S. Angelopoulos, R. Dorrigiv, and A. López-Ortiz. On the separation and equivalence of paging strategies. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 229–237. Society for Industrial and Applied Mathematics, 2007.

[18] S. Angelopoulos and P. Schweitzer. Paging and list update under bijective analysis. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 1136–1145. SIAM, 2009.

[19] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 197–206, 2008.

[20] N. Bansal, N. Buchbinder, and J. Naor. A primal-dual randomized algorithm for weighted paging. *Journal of the ACM (JACM)*, 59(4):1–24, 2012.

[21] R. Barve and J. S. Vitter. External memory algorithms with dynamically changing memory allocations. Technical report, Duke University, 1998.

[22] R. D. Barve, E. F. Grove, and J. S. Vitter. Application-controlled paging for a shared cache. *SIAM Journal on Computing*, 29(4):1290–1303, 2000.

[23] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[24] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 145–156, 2016.

[25] M. Bender, R. Chowdhury, R. Das, R. Johnson, W. Kuszmaul, A. Lincoln, Q. Liu, J. Lynch, and H. Xu. Closing the gap between cache-oblivious and cache-adaptive analysis. In *Proc. 32st ACM on Symposium on Parallelism in Algorithms and Architectures*, 2020.

[26] M. A. Bender, J. Berry, S. D. Hammond, K. S. Hemmert, S. McCauley, B. Moore, B. Moseley, C. A. Phillips, D. Resnick, and A. Rodrigues. Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation. In *Proc. 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Hyderabad, INDIA, May 2015.

[27] M. A. Bender, J. W. Berry, S. D. Hammond, K. S. Hemmert, S. McCauley, B. Moore, B. Moseley, C. A. Phillips, D. Resnick, and A. Rodrigues. Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation. *Journal of Parallel and Distributed Computing*, 102:213–228, 2017.

[28] M. A. Bender, J. W. Berry, S. D. Hammond, K. S. Hemmert, S. McCauley, B. Moore, B. Moseley, C. A. Phillips, D. S. Resnick, and A. Rodrigues. Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation. *Journal of Parallel and Distributed Computing*, 102:213–228, 2017.

[29] M. A. Bender, J. W. Berry, S. D. Hammond, B. Moore, B. Moseley, and C. A. Phillips. k-means clustering on two-level memory systems. In B. Jacob, editor, *Proc. 2015 International Symposium on Memory Systems, (MEMSYS)*, pages 197–205, Washington DC, USA, October 2015.

[30] M. A. Bender, A. Conway, M. Farach-Colton, W. Jannen, Y. Jiao, R. Johnson, E. Knorr, S. McAllister, N. Mukherjee, P. Pandey, D. E. Porter, J. Yuan, and Y. Zhan. Small refinements to the dam can have big consequences for data-structure design. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 265–274, Phoenix, AZ, June 2019.

[31] M. A. Bender, E. D. Demaine, R. Ebrahimi, J. T. Fineman, R. Johnson, A. Lincoln, J. Lynch, and S. McCauley. Cache-adaptive analysis. In *Proc. 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–144, 2016.

[32] M. A. Bender, E. D. Demaine, R. Ebrahimi, J. T. Fineman, R. Johnson, A. Lincoln, J. Lynch, and S. McCauley. Cache-adaptive analysis. In *Proc. 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–144, July 2016.

[33] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 958–971, 2014.

[34] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proc. 25th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 958–971, Portland, OR, USA, January 2014.

[35] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 117–128. ACM, 2000.

[36] E. Blayo, L. Debreu, G. Mounie, and D. Trystram. Dynamic load balancing for ocean circulation model with adaptive meshing. In *European Conference on Parallel Processing*, pages 303–312. Springer, 1999.

[37] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510. Society for Industrial and Applied Mathematics, 2008.

[38] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA*, volume 8, pages 501–510. Citeseer, 2008.

[39] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *ACM symposium on Parallelism in algorithms and architectures*, pages 355–366, 2011.

[40] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 89–102, 2020.

[41] O. A. R. Board. OpenMP: A proposed industry standard API for shared memory programming. *White Paper*, 1997. url: http://www.openmp.org/specs/mp-documents/paper/paper.ps.

[42] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[43] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995.

[44] A. Borodin, P. Raghavan, S. Irani, and B. Schieber. Competitive paging with locality of reference. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 249–259. Citeseer, 1991.

[45] N. Butcher, S. L. Olivier, J. Berry, S. D. Hammond, and P. M. Kogge. Optimizing for knl usage modes when data doesn't fit in mcdram. In *Proceedings of the 47th International Conference on Parallel Processing*, page 37. ACM, 2018.

[46] C. Byun, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, et al. Benchmarking data analysis and machine learning applications on the intel knl many-core processor. *arXiv preprint arXiv:1707.03515*, 2017.

[47] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *Proceedings of the USENIX Summer 1994 Technical Conference (USTC)*, pages 171–182, 1994.

[48] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 402–412, 2007.

[49] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, et al. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115. ACM, 2007.

[50] M. Chrobak. SIGACT news online algorithms column 17. *SIGACT News*, 41(4):114–121, 2010.

[51] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[52] R. Cole and V. Ramachandran. Bounding cache miss costs of multithreaded computations under general schedulers. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 351–362, 2017.

[53] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. *ACM Transactions on Parallel Computing*, 3(4):1–31, 2017.

[54] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[55] R. Das, K. Agrawal, M. Bender, J. Berry, B. Moseley, and C. Phillips. How to manage high-bandwidth memory automatically. In *Proc. 32st ACM on Symposium on Parallelism in Algorithms and Architectures*, 2020.

[56] R. Das, K. Agrawal, M. A. Bender, J. Berry, B. Moseley, and C. A. Phillips. How to manage high-bandwidth memory automatically. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 187–199, 2020.

[57] R. Das, S.-Y. Tsai, S. Duppala, J. Lynch, E. M. Arkin, R. Chowdhury, J. S. Mitchell, and S. Skiena. Data races and the discrete resource-time tradeoff problem with resource reuse over paths. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 359–368, 2019.

[58] P. De, E. J. Dunne, J. B. Ghosh, and C. E. Wells. The discrete time-cost tradeoff problem revisited. *European Journal of Operational Research*, 81(2):225–238, 1995.

[59] P. De, E. J. Dunne, J. B. Ghosh, and C. E. Wells. Complexity of the discrete time-cost tradeoff problem for project networks. *Operations research*, 45(2):302–306, 1997.

[60] A. S. de Loma. New results on fair multi-threaded paging. *Electronic Journal of SADIO*, 1(1):21–36, 1998.

[61] D. W. Doerfler. Trinity: Next-generation supercomputer for the asc program. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2014.

[62] R. Dorrigiv, M. R. Ehmsen, and A. López-Ortiz. Parameterized analysis of paging and list update algorithms. *Algorithmica*, 71(2):330–353, 2015.

[63] R. Dorrigiv, A. López-Ortiz, and J. I. Munro. On the relative dominance of paging algorithms. *Theoretical Computer Science*, 410(38-40):3694–3701, 2009.

[64] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.

[65] P.-F. Dutot, G. Mounié, and D. Trystram. Scheduling parallel tasks: Approximation algorithms, 2004.

[66] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[67] E. Feuerstein and A. S. de Loma. On-line multi-threaded paging. *Algorithmica*, 32(1):36–60, 2002.

[68] E. Feuerstein and A. Strejilevich de Loma. On-line multi-threaded paging. *Algorithmica*, 32(1):36–60, 2002.

[69] 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)*, pages 626–634, 1995.

[70] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[71] `http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html`.

[72] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the twenty-first annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90. ACM, 2009.

[73] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual ACM Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.

[74] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, Jan. 2012.

[75] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[76] D. R. Fulkerson. A network flow computation for project cost curves. *Management science*, 7(2):167–178, 1961.

[77] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[78] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous pram model. In *European Conference on Parallel Processing*, pages 277–292. Springer, 1996.

[79] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi. Elastic caching. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 143–156, 2019.

[80] A. Hassidim. Cache replacement policies for multicore processors. In A. C. Yao, editor, *Proc. Innovations in Computer Science (ICS)*, pages 501–509, 2010.

[81] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *ACM SIGPLAN Notices*, volume 48, pages 317–328. ACM, 2013.

[82] S. Irani. Competitive analysis of paging. In *Developments from a June 1996 Seminar on Online Algorithms: The State of the Art*, pages 52–73, 1998.

[83] S. Irani, A. R. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. *SIAM Journal on Computing*, 25(3):477–497, 1996.

[84] J. JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley, 1997.

[85] K. Jansen and H. Zhang. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Transactions on Algorithms (TALG)*, 2(3):416–434, 2006.

[86] M. M. Javanmard, P. Ganapathi, R. Das, Z. Ahmad, S. Tschudi, and R. Chowdhury. Toward efficient architecture-independent algorithms for dynamic programs. In *International Conference on High Performance Computing*, pages 143–164. Springer, 2019.

[87] S. Kamali and H. Xu. Beyond worst-case analysis of multicore caching strategies. In *Symposium on Algorithmic Principles of Computer Systems (APOCS21)*. SIAM, 2021.

[88] A. K. Katti and V. Ramachandran. Competitive cache replacement strategies for shared cache environments. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 215–226, 2012.

[89] A. K. Katti and V. Ramachandran. Competitive cache replacement strategies for shared cache environments. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 215–226. IEEE, 2012.

[90] J. E. Kelley Jr. Critical-path planning and scheduling: Mathematical basis. *Operations research*, 9(3):296–320, 1961.

[91] J. E. Kelley Jr and M. R. Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173. ACM, 1959.

[92] `http://www.hpcwire.com/2014/06/24/`.

[93] P. Kogge and J. Shalf. Exascale computing trends: Adjusting to the" new normal"'for computer architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.

[94] R. Kumar, M. Purohit, Z. Svitkina, and E. Vee. Interleaved caching with access graphs. In *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1846–1858, 2020.

[95] J. K. Lenstra and A. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

[96] R. Lepère, G. Mounié, and D. Trystram. An approximation algorithm for scheduling trees of malleable tasks. *European Journal of Operational Research*, 142(2):242–249, 2002.

[97] R. Lepere, D. Trystram, and G. J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. *International Journal of Foundations of Computer Science*, 13(04):613–627, 2002.

[98] A. Li, W. Liu, M. R. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song. Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 26, 2017.

[99] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008.

[100] A. Lincoln, Q. C. Liu, J. Lynch, and H. Xu. Cache-adaptive exploration: Experimental results and scan-hiding for adaptivity. In *Proc. 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 213–222, 2018.

[101] A. López-Ortiz and A. Salinger. Minimizing cache usage in paging. In *Proceedings of the 10th Workshop on Approximation and Online Algorithms (WAOA)*, pages 145–158, 2012.

[102] A. López-Ortiz and A. Salinger. Paging for multi-core shared caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science conference (ITCS)*, pages 113–127, 2012.

[103] A. López-Ortiz and A. Salinger. Paging for multi-core shared caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 113–127. ACM, 2012.

[104] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[105] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the bsp model. *Algorithmica*, 24(3-4):287–297, 1999.

[106] I. Menache and M. Singh. Online caching with convex costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 46–54, 2015.

[107] R. H. Möhring. Computationally tractable classes of ordered sets. In *Algorithms and order*, pages 105–193. Springer, 1989.

[108] R. H. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[109] `http://nnsa.energy.gov/mediaroom/pressreleases/trinity`.

[110] V. Y. Pan. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *Symposium on Foundations of Computer Science*, pages 166–176, 1978.

[111] D. Panagiotakopoulos. A cpm time-cost computational algorithm for arbitrary activity cost functions. *INFOR: Information Systems and Operational Research*, 15(2):183–195, 1977.

[112] K. Panagiotou and A. Souza. On adequate performance measures for paging. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 487–496, 2006.

[113] E. Peserico. Paging with dynamic memory capacity. *CoRR*, abs/1304.6007, 2013.

[114] E. Peserico. Paging with dynamic memory capacity. In *Proceedings of the 36th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 56:1–18, 2019.

[115] S. Phillips Jr and M. I. Dessouky. Solving the project time/cost tradeoff problem using the minimal cut concept. *Management Science*, 24(4):393–400, 1977.

[116] V. Ramachandran and E. Shi. Data oblivious algorithms for multicores. *arXiv preprint arXiv:2008.00332*, 2020.

[117] J. Reinders. *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[118] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM, 2000.

[119] D. R. Robinson. A dynamic programming solution to cost-time tradeoff for cpm. *Management Science*, 22(2):158–166, 1975.

[120] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 216–226. ACM, 1978.

[121] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, pages 84–94. ACM, 2006.

[122] M. Scquizzato. *Paging on Complex Architectures*. PhD thesis, University of Padova, 2013.

[123] S. S. Seiden. Randomized online multi-threaded paging. *Nordic Journal of Computing*, 6(2):148–161, 1999.

[124] N. Shavit. Data structures in the multicore age. *Communications of the ACM*, 54(3):76–84, 2011.

[125] M. Skutella. Approximation algorithms for the discrete time-cost tradeoff problem. *Mathematics of Operations Research*, 23(4):909–929, 1998.

[126] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[127] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, Feb. 1985.

[128] G. M. Slota and S. Rajamanickam. Experimental design of work chunking for graph algorithms on high bandwidth memory architectures. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 875–884. IEEE, 2018.

[129] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.

[130] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41:1054–1068, 1992.

[131] V. Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[132] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[133] `https://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx`.

[134] T. N. Theis and H.-S. P. Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[135] D. Thiébaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41:665–676, 1992.

[136] `https://www.threadingbuildingblocks.org`.

[137] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the fourth annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332. ACM, 1992.

[138] J. Wells, B. Bland, J. Nichols, J. Hack, F. Foertter, G. Hagen, T. Maier, M. Ashfaq, B. Messer, and S. Parete-Koon. Announcing supercomputer summit. Technical report, ORNL (Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States)), 2016.

[139] R. S. Williams. What's next?[the end of moore's law]. *Computing in Science & Engineering*, 19(2):7–13, 2017.

[140] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. *ACM SIGARCH Computer Architecture News*, 37(3):174–183, 2009.

[141] N. E. Young. Online paging and caching. 2008.