Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems

Pramod Ganapathi

A **Dissertation** presented to the faculty of **Stony Brook University**
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the field of **Computer Science**
specializing in **Algorithms**

Advisor: **Rezaul Chowdhury**

Dec 2016

Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems

Pramod Ganapathi

Dissertation Committee:

Rezaul Chowdhury, Stony Brook University

Michael A. Bender, Stony Brook University

Yanhong A. Liu, Stony Brook University

Barbara Chapman, Stony Brook University

Kunal Agrawal, Washington University

Dedicated to mathematics, puzzles, algorithms, & philosophy.

Acknowledgments

Many lives are directly or indirectly responsible for this 3.5 year work and for my amazing Ph.D. life. In this small section, I would like to express my sincere and heartful gratitude to all of them.

First and foremost, I thank this enigmatic cosmos for everything. I thank my biological parents – (mother) Varada and (father) Ganapathi Hegde for their love, care, and affection and for their sacrifice in bringing me up; and all my wonderful relatives, especially (older brother) Prashanth Hegde, (paternal uncle) Hegde Jadugar, (cousin) Sudha Hegde, and (cousin) Shobha Hegde, for always showering love upon me. Most importantly, I thank the lady I loved, for introducing me to the world of beauty, love, dreams, imagination, and addiction. I am infinitely indebted to that Goddess for the eternal happiness and unconditional love she has given to me. I also thank my closest friend Darshan S. Palasamudram for countless hours of brain-drilling philosophical discussions and ground-breaking analyses on hundreds of topics.

From the bottom of my heart, I would like to thank my research guru, my algorithms teacher, my great advisor, and the superhero of my Ph.D. life – Rezaul Chowdhury, for introducing me to the world of organized research and rigorous algorithmic thinking. He stressed importance on novelty, technicality, mathematical rigorousness, presentation, and academic integrity. He taught me that a world of problems is in fact a world of opportunities because with each problem there comes an opportunity to solve the problem. My problem-solving ability has increased tremendously under his tutelage.

Rezaul is the reason for my transformational thinking from "The given problem is impossible to solve" to "There must be a very simple solution to the given problem and I will find that solution in 3 days". In the initial years of my Ph.D., I viewed problems as lions and myself as a deer whose only goal was to run away from the lions and survive. In the final years of my Ph.D., I saw myself as a powerful lion and problems as deers and my only aim was to attack and nail the deers down one-by-one. In this way, Rezaul rewired my belief system.

I am eternally grateful to Rezaul for having given me a diamond opportunity to design AUTOGEN (an algorithm to discover other algorithms), which is the greatest computer algorithm I have ever (co-)designed, which now forms the core of my Ph.D. dissertation. Rezaul gave me a beautiful opportunity to lead and advise 8 master's and 2 undergrad students for their research projects. He gave me a lot of freedom to choose the problems I like and take as much time as necessary to solve them. I thank Rezaul in a non-terminating recursive loop for everything he has taught me.

I thank my algorithms teacher Michael A. Bender for exponentially increasing my love

My lovely American family: Susan, Carlo, and their son John Colatosti deserve a special thanks for treating me as a member of their own family and for their continuous love and affection. My mom Susan exposed me to the American culture, took me to several wonderful places in Long Island, and taught me the ways of appreciating the beauty of nature, experiencing life, and being grateful for the blessings in life. The long bike rides, breakfasts at Port Jefferson, church visits, frequent dinners, walks on the shore of West Meadow beach, and camping at Smith Point beach, all these experiences with Susan now seek existence only in my memories. I thank my another American family: Pamela and Frank Murphy for their love and affection. Pamela and I shared a lot of common interests such as good conversations, photography, nature, arts, and so on. I thank her for being so nice and always caring. Thanks to my close roommates: Ayon Chakraborty, Chirag Mandot, and Franklin George in celebrating life with chit-chats, tasty food, discussions, arguments, drink parties, pranks, dances, and beach trips. Thanks to Pavithra K. Srinath and Pallavi Ghosh for their support and encouragement; and several other friends for all memorable fun moments. I am also extremely thankful to the West Meadow beach, my most favorite location near Stony Brook, where I have spent an uncountable number of romantic evenings watching the sunset and contemplating on the deeper meanings of life, love, and the world.

Finally, I wish to thank everything in this aesthetically beautiful and enigmatic cosmos that is directly or indirectly responsible for this work and my mind-blowingly-awesome life.

Pramod Ganapathi
Stony Brook, USA
December 2016

Abstract

This dissertation is an excursion into computer-discovered algorithms and computer-aided algorithm design. In our vision of the not-so-distant future of computing, machines will take the responsibility of designing and implementing most, if not all, machine-specific highly efficient nontrivial algorithms with provable correctness and performance guarantees. Indeed, as the complexity of computing hardware grows and their basic architectures keep changing rapidly, manually redesigning and reimplementing key algorithms for high performance on every new architecture is quickly becoming an impossible task. Automation is needed.

We design algorithms that can design other algorithms. Our focus is on auto-generating algorithms for solving dynamic programming (DP) recurrences efficiently on state-of-the-art parallel computing hardware. While DP recurrences can be solved very easily using nested or even tiled nested loops, for high performance on modern processors/coprocessors with cache hierarchies, portability across machines, and automatic adaptivity to runtime fluctuations in the availability of shared resources (e.g., cache space) highly nontrivial recursive divide-and-conquer algorithms are needed. But, these recursive divide-and-conquer algorithms are difficult to design and notoriously hard to implement for high performance. Furthermore, new DP recurrences are being encountered by scientists every day for solving brand new problems in diverse application areas ranging from economics to computational biology. But, how does an economist or a biologist without any formal training in computer science design an efficient algorithm for evaluating his/her DP recurrence on a computer? Well, we can help!

We present AUTOGEN – an algorithm that given any blackbox implementation of a DP recurrence (e.g., inefficient naive serial loops) from a wide class of DP problems, can automatically discover a fast recursive divide-and-conquer algorithm for solving that problem on a shared-memory multicore machine. We mathematically formalize AUTOGEN, prove its correctness, and provide theoretical performance guarantees for the auto-discovered algorithms. These auto-generated algorithms are shown to be efficient (e.g., highly parallel with highly optimizable kernels, and cache-, energy-, and bandwidth-efficient), portable (i.e., cache- and processor-oblivious), and robust (i.e., cache- and processor-adaptive).

We present AUTOGEN-WAVE – a framework for computer-assisted discovery of fast divide-and-conquer wavefront versions of the algorithms already generated by AUTOGEN. These recursive wavefront algorithms retain all advantages of the AUTOGEN-discovered algorithms on which they are based, but have better and near-optimal parallelism due to the wavefront order of execution. We also show how to schedule these algorithms with provable performance guarantees on multicore machines.

We present VITERBI – an efficient cache-oblivious parallel algorithm to solve the Viterbi recurrence, as a first step toward extending AUTOGEN to handle DP recurrences with ir-

regular data-dependent dependencies. Our algorithm beats the current fastest Viterbi algorithm in both theory and practice.

We present Autogen-Fractile – a framework for computer-aided design of high-performing and easily portable recursively tiled/blocked algorithms for a wide class of DP problems having fractal-type dependencies. These recursively tiled algorithms have excellent cache locality and excellent parallel running time on multi-/many-core machines with deep memory hierarchy.

We present Autogen-Tradeoff – a framework that can be used to design efficient and portable not-in-place algorithms to asymptotically increase the parallelism of some of the Autogen-discovered algorithms without affecting cache-efficiency.

This dissertation takes the first major step on which to build computing systems for automatically designing efficient and portable nontrivial algorithms. We hope that more work follows in this science of algorithm discovery.

Contents

Chapter 1

Introduction & Background

"All life is problem-solving", says Karl Popper, a famous philosopher. Considering problem-solving itself as a problem, can we solve the problem of problem-solving? In the world of computer science, problem-solving often means coming up with algorithms to solve computer science problems. Solving the problem of problem-solving means coming up with an algorithm that can automatically come up with various other algorithms for solving a multitude of problems. In this dissertation, we answer affirmatively that we can solve the problem of problem-solving for a class of problems by designing algorithms / frameworks that can automatically / semi-automatically discover other efficient and portable nontrivial algorithms to solve a wide class of dynamic programming problems.

1.1 Motivation and vision

Programming is hard. Programming typically involves understanding the given problem deeply, identifying / coming up with data structures to structure and organize the data, finding provably correct algorithms for solving the problem that access data from the data structures efficiently, and coding the algorithms and data structures to take the problem specification as input and output the desired solutions. *Performance* is usually the most important measure of goodness of correct programs. In computer science, almost all paths lead not to Rome but to develop high-performing programs.

A majority of today's algorithms are sequential. Each step of execution of a serial algorithm consists of an instruction. The speed of the sequential computers has improved exponentially for several years through increase in the number of transistors on an integrated circuit. The improvement is now coming at very high costs. As a result, manufacturers are building *parallel computers* with multiple processing elements which are cost-effective and also improve the total computing speed. To solve problems on parallel computers efficiently, we need to design parallel algorithms and each step of execution in such algorithms consists of multiple instructions.

Real-life applications [Chen et al., 2014] [Kim et al., 2014] such as databases (of Google, Facebook, Microsoft, Twitter, Netflix, IBM, Amazon, etc), cryptography, computer graphics, computer vision, weather forecasting, crystallography, simulation of galaxy formation and planetary movements, gene analysis, material science, circuit design, defense, geology, molecular science, condensed matter, etc involve gigantic volume of computations which when executed sequentially might take months to years. *Parallelism* is one of the key

factors in improving performance. Hence, designing parallel algorithms typically leads to high-performing programs.

Modern computers have a tree-like hierarchy of caches having different sizes and access times. A typical algorithm has to access data and when the data does not fit in a particular cache it moves back-and-forth between different levels of caches. Reducing the total number of such data transfers improves the overall performance. *Cache locality* is yet another key factor to improve performance. Hence, designing cache-efficient algorithms (i.e., algorithms with good cache locality) might lead to more efficient programs.

Most of today's cache-efficient parallel algorithms are cache-aware and/or processor-aware. This means the algorithms need to know the machine parameters such as cache size, block size, number of levels of caches, number of processors etc. There are several types of machine architectures and it is not feasible to write cache-efficient parallel programs for every single one of them. *Resource-obliviousness* (e.g: processor- and cache-obliviousness) is a key factor to achieve portability. Hence, designing resource-oblivious algorithms leads to portable programs that execute with reasonable performance on various machine architectures with a little or no change in the code.

Designing algorithms that are efficient (i.e., parallel and cache-efficient) and portable (i.e., processor- and cache-oblivious) for a shared-memory multicore parallel machine is complicated. It requires expertise in different domains such as algorithms, data structures, parallel computing, compilers, machine architecture, and so on. Manually designing fast and portable algorithms is not scalable because the number of problems to solve is very large and solving each problem requires expertise in different domains. *Automatic discovery* of the desired algorithms is the solution. Hence, we need to design algorithms / frameworks so that the ordinary programmers and computational scientists can use these systems to automatically / semi-automatically discover fast and portable algorithms in their respective fields.

In this dissertation, we design algorithms / frameworks to automatically / semi-automatically discover fast, portable, and robust algorithms to solve a wide class of dynamic programming problems. In the subsequent sections of this chapter, we discuss the fundamentals necessary to understand the entire dissertation. At the end of this chapter, we briefly describe the scope of the dissertation and our contributions.

1.2   Cache-efficient algorithms

In this section, we define several terms related to cache-efficient algorithms and discuss the models of computation used for designing cache-efficient algorithms.

We all know from physics that no physical signal (thoughts are not included) can travel faster than light. Due to this reason, for a given memory access time or *cache latency*, there is a physical limit on how large a cache (or memory) can be. Therefore a cache cannot be fast, compact, and have a large memory size, simultaneously. This problem is called the *memory wall problem* [Sanders, 2003].

The simplest and the most common way to solve the memory wall problem is to use a hierarchy of caches having different access times or latencies. Modern computers often have a tree-like hierarchy of private and shared caches. Caches at different levels have different sizes and latencies. Reducing the number of memory transfers in various levels of caches reduces the overall runtime of an algorithm.

An algorithm that performs asymptotically fewer number of memory transfers (w.r.t another algorithm solving the same problem) is called a *cache-efficient algorithm*. An algorithm is called *cache-optimal* if no other algorithm (to solve the same problem) can incur asymptotically fewer number of memory transfers than the former. In order to make good use of cache, an algorithm must have the following two features:

* ✶ [*Spatial cache locality*.] Whenever a cache block is brought into the cache, it contains as much useful data as possible.
* ✶ [*Temporal cache locality*.] Whenever a cache block is brought into the cache, as much useful work as possible is performed on this data before removing the block from the cache.

Throughout literature, computer scientists have developed various mathematical models to analyze the number of memory transfers between consecutive levels of cache. Here, we summarize a few important models.

1.2.1   I/O model

I/O model [Aggarwal et al., 1988] is also known as external-memory model, disk access machine (DAM) model, cache-aware model, and cache-conscious model. The model consists of a single CPU and a 2-level memory hierarchy: internal-memory and external-memory. The internal-memory can be viewed as a cache and the external-memory can be viewed as memory. The aim of the model is to mathematically model a computer's memory and compute theoretically the number of memory transfers between the two caches.

The internal-memory is made up of M/B blocks each of size B bytes. The size of a block is called the *block size* B and the size of the internal-memory is called the *cache size* M. The external-memory is of infinite size. We can consider RAM to be the internal-memory and hard disk to be the external-memory. The transfer of data between internal-memory and external-memory happens in units of blocks, each of size B.

If the data block required by an algorithm is present in the cache, then the event is called a *cache hit*. On the other hand, if the data block is not found in the cache, then the event is called a *cache miss* (or page fault). If there is a cache miss, then the data block that is referenced must be brought from the from the external-memory. The cache complexity (or I/O complexity) is measured in terms of the number of cache misses incurred. The cache misses affects the number of block or I/O transfers. Our aim in algorithm design in this model is to develop algorithms that minimizes the number of cache misses thereby decreasing the running time of the algorithms.

The limitations of the algorithm designed in this model are:

* ✶ [*Non-adaptibility for all memory levels*.] The algorithms in the I/O model are tailored to optimize cache complexity for a single level of cache.
* ✶ [*Non-portability*.] The algorithms in this model know the cache parameters. Hence, the same algorithms cannot run on all machine architectures without change. In many applications, if it is not possible to know the cache parameters such as number of cache levels, cache sizes, and block sizes, which makes the algorithms non-portable.
* ✶ [*Cache non-adaptivity*.] The algorithms assume that the available space in cache is fixed and not changing dynamically over time. In real life scenarios, the available cache space will be changing dynamically because multiple processes will be running simultaneously and sharing cache.

1.2.2 Cache-oblivious model

This model, also called the *ideal-cache model* [Frigo et al., 1999], is an extension of the I/O model, in which the algorithms do not depend on the cache parameters such as the number of levels of cache, cache sizes, and block sizes. Though the algorithms are analyzed for two adjacent levels of a memory hierarchy, they apply to any two adjacent levels of the memory hierarchy. Though the analysis makes use of cache parameters, the algorithms do not. Hence, they are portable and cache-adaptive.

The assumptions of the model are:

(1) [*Optimal page replacement policy*.] An optimal page replacement algorithm chooses data block that will be accessed furthest in the future. But, such an algorithm is difficult to implement because it is difficult to know beforehand which page is going to be used and when, unless all memory patterns of the software being run is known beforehand. The *least recently used (LRU)* and *first in first out (FIFO)* algorithms increase the cache complexity by at most a constant factor [Sleator and Tarjan, 1985] and hence are commonly used in practice.

(2) [*Automatic page replacement*.] When a cache miss occurs, the requested data block is automatically transferred from the external-memory to the internal-memory by the hardware and/or operating system.

(3) [*Full associativity*.] When a data block is transfered from the external-memory to internal-memory, it can be placed anywhere in internal-memory. There are three types of cache associativity: (*i*) *direct mapped*, where a cache block can go to one spot in the cache. It is easy to find a cache block but it does not have flexibility; (*ii*) *n-way set associative*, where the cache is made up of sets each of size n. Often, n is 2 or 4. The larger the value of n, fewer the number of sets, and fewer number of bits are needed to encode them. The larger the value of n, fewer the number of cache conflicts and lower the miss rates but the hardware costs increase; and (*iii*) *fully-associative*, where a cache block can go to any spot of the cache. Though there is flexibility, it is extremely difficult to implement such a strategy. In practice, a tradeoff is made to reduce the cache conflicts and miss rates and also not increasing hardware costs. Full-associativity can be implemented in software.

1.2.3 Parallel cache-oblivious model

The *parallel cache-oblivious (PCO) model* [Blelloch et al., 2011] is a modification to the cache-oblivious model. The PCO model is a modification of the cache-oblivious model. The model allows for arbitrary imbalances among tasks and works for a very general memory hierarchy called *parallel memory hierarchy (PMH) model*.

A computation is modeled as a series of tasks, strands, and parallel blocks. A parallel block consists of several tasks and a strand is a single thread of execution unit. The cache complexity measure defined in the paper accounts for all cache misses at a particular level in the memory hierarchy. The cache complexity does not account for shared data blocks among parallel threads. Hence, for a shared cache, the cache complexity in this model is p times that in the cache-oblivious model. Still, for several algorithms the cache complexity measure defined in the paper matches asymptotically with the serial cache complexity of the cache-oblivious model.

1.3 Cache-oblivious algorithms

In 1988, Aggarwal and Vitter introduced the I/O model in a seminal paper [Aggarwal et al., 1988] titled "The Input/Output Complexity of Sorting and Related Problems". The algorithms defined in the I/O model are called I/O-efficient algorithms or cache-aware algorithms. The performance of these algorithms are not measured by how many computations they perform but instead by how many I/O misses (or page faults) they incur. These algorithms must know the size of the available cache and the block size. The I/O-efficient algorithms typically execute fast but are not portable.

The algorithms designed in the cache-oblivious model are called cache-oblivious algorithms. These algorithms in their implementations do not make use of cache-parameters such as cache size M, block size B, number of levels in the cache hierarchy, etc. These algorithms do not have any information about the cache hierarchy of the machine it runs on. Due to this reason, such algorithms are *portable* and can run on any shared-memory parallel machine ranging from tiny smart phones to the compute nodes of gigantic supercomputers.

All classic iterative algorithms in literature we have seen in text books are cache-oblivious as they do not make use of cache parameters. The fact that they are cache-oblivious (and hence portable) does not compel us to use them in all our applications. We care for two key factors in order of priority: *performance* and *portability*. If the algorithms are too slow, then probably we do not care for portability. We will go for non-portable fast algorithms than portable slow algorithms. Then why should we care for cache-oblivious algorithms?

In 1999, Frigo et al. published a landmark paper called "Cache-Oblivious Algorithms" [Frigo et al., 1999] in which they showed that it is possible to develop *efficient cache-oblivious* algorithms that are simultaneously theoretically fast and portable. These algorithms exploit temporal locality and are cache-oblivious because they are often based on the *recursive divide-and-conquer* algorithm design technique.

Cache-oblivious algorithms and/or data structures [Demaine, 2002, Kumar, 2003, Brodal, 2004] have been developed to solve a variety of problems. A few cache-oblivious algorithms and/or data structures are designed for rectangular matrix multiplication, rectangular matrix transpose, fast Fourier transform (FFT), funnelsort (cache-oblivious merge sort), and distribution sort [Frigo et al., 1999, Frigo et al., 2012], Jacobi multipass filter [Prokop, 1999], B-trees [Bender et al., 2000, Bender et al., 2005a], searching, partial-persistence, and planar point location [Bender et al., 2002a], searching [Brodal et al., 2002], funnel heap (priority queue) [Brodal and Fagerberg, 2002], priority queue, list ranking, tree algorithms, directed breadth first search (BFS) and depth first search (DFS), undirected BFS, and minimal spanning forest [Arge et al., 2002a], computational geometry problems [Brodal and Fagerberg, 2002], dynamic dictionary [Bender et al., 2002b], orthogonal range searching [Agarwal et al., 2003], FFT [Frigo and Johnson, 2005], stencil computations [Frigo and Strumpen, 2005], mesh layout for visualization [Yoon et al., 2005], parallel B-trees [Bender et al., 2005b], R-trees [Arge et al., 2005], longest common subsequence [Chowdhury and Ramachandran, 2006], string B trees [Bender et al., 2006a], parenthesis problem [Chowdhury and Ramachandran, 2008], databases [He and Luo, 2008], parallel matrix multiplication, parallel 1-D stencil computation, and other problems [Frigo and Strumpen, 2009], Floyd-Warshall's all-pairs shortest path [Chowdhury and Ramachandran, 2010], 3-D convex hull and 2-D segment intersection [Chan and

Chen, 2010], range minimum query [Hasan et al., 2010], multicore-oblivious algorithms for matrix transposition, prefix sum, FFT, sorting, Gaussian elimination paradigm, and list ranking [Chowdhury et al., 2013].

Experimental evaluation of cache-oblivious algorithms can be found in [Rahman et al., 2001, Olsen and Skov, 2002, Ladner et al., 2002, Kumar, 2003, Yotov et al., 2007, Chowdhury, 2007, Frigo and Strumpen, 2007, Brodal et al., 2008, Tithi et al., 2015, Tang et al., 2015, Chowdhury et al., 2016b, Chowdhury et al., 2016a].

Throughout this dissertation, we will be interested in algorithms that are simultaneously fast and portable i.e., cache-efficient, cache-oblivious, and parallel.

1.4 Parallel algorithms

Real-life applications (see the papers in [Bischof et al., 2008]) such as databases (of Google, Facebook, Twitter, Netflix, etc), cryptography, computer graphics, computer vision, weather forecasting, crystallography, simulation of galaxy formation and planetary movements, gene analysis, material science, circuit design, defense, geology, molecular science, condensed matter, etc will have gigantic number of computations which when executed serially might take months to years. Developing fast parallel algorithms to such applications will be of tremendous value, saves energy and time. Hence, designing parallel algorithms [Karp and Ramachandran, 1988] is of utmost significance.

A machine is called a *parallel computer* if it performs multiple operations at a time. Parallel machines are important to reduce the execution time to complete a task. Now-a-days most machines including smartphones, laptops, desktops, workstations, and supercomputers run parallel computations. Smartphones, laptops, desktops, and compute nodes of supercomputers use the shared-memory multicore model as the machine model. Due to the importance of shared-memory machine algorithms, the entire dissertation is based on the shared-memory multicore model.

1.4.1 Dynamic multithreading model

In the dynamic multithreading model [Graham, 1969, Brent, 1974, Eager et al., 1989, Cormen et al., 2009], the programmer simply specifies the logical parallelism in a computation without worrying about load balancing the computations. A scheduler that is part of the concurrency platform will do the necessary scheduling / load balancing different tasks on to different processors.

The major advantages of the model are:

✵ [*Simple keywords*.] The model defines majorly three keywords: *spawn* for nested parallelism (when functions can run in parallel), *parallel* for parallel loops, and *sync* for synchronization of threads.

✵ [*Simple performance metrics*.] The model gives easy ways to quantify parallelism using *work* and *span*.

✵ [*Well-suited for divide-and-conquer*.] The model is well-suited for divide-and-conquer algorithms to extract parallelism at every level of the recursion.

✵ [*Works well in practice*.] The model works well in practice and hence is used in several dynamic multithreading systems such as Cilk [Blumofe et al., 1996b, Frigo et al., 1998], Cilk++ [Leiserson, 2010], Intel Cilk Plus [Plus, 2016], and OpenMP [Chapman

et al., 2008].

The most common theoretical model to model a multithreaded program execution is using directed acyclic graphs (DAGs). In this *DAG model*, every node is a computation. A node x precedes a node y, denoted by $x \succ y$, if x must be executed before y can begin. If two nodes x and y are such that if neither $x \succ y$ nor $y \succ x$, then they are executed in parallel, denoted by $x \parallel y$. Figure 1.2(a), which has been adapted from [Blumofe, 1995], gives an example DAG representation for a multithreaded program execution. We characterize a DAG using work and span as described below, which is also called the *work-span model*.

Work. The work T_1 of a multithreaded program is defined as the total number of instructions executed on a machine. If we assume that an instruction takes a constant time to execute, then the work is the serial running time of the program. In a DAG, the work is equal to the total number of nodes, assuming every node executes a single instruction. For example, in Figure 1.2(a), the work is 20 as there are 20 nodes in the DAG.

The *parallel running time T_p*, for p processors can be lower-bounded using work and it is called the *work law*. It is straightforward to see that the parallel running time on a parallel machine with p processors must be at least T_1/p because at most p instructions can run in parallel at a time i.e., $T_p \geq T_1/p$.

Span. The span T_∞ of a multithreaded computation is defined as the maximum number of instructions executed by any processor when there are an infinite number of processors. It is the longest path from the root (the only node with indegree 0) to a leaf (node with outdegree 0) of the DAG. Span is also called *critical path length* or *depth*. For example, in Figure 1.2(a), the span is 10 as the longest path from root to leaf in the dag has 10 nodes.

The parallel running time T_p, for p processors can be lower-bounded using span and it is called the *span law*. It is straightforward to see that the running time on p processors cannot be smaller than the running time on an infinite number of processors i.e., $T_p \geq T_\infty$.

Parallelism. Parallelism is defined as the ratio of work and span i.e., T_1/T_∞. It represents the maximum speedup that can be achieved from an infinite number of processors. For our DAG in Figure 1.2(a), the parallelism is $T_1/T_\infty = 20/10 = 2$. This means we cannot achieve more than 2 speedup whatever might be the number of processors.

The *goodness* of a parallel algorithm is measured from its work and span. A parallel algorithm typically reduces the span, thereby increasing the parallelism. Several cache-oblivious algorithms mentioned in Section 1.3 can be parallelized. Sometimes the algorithms might have to be changed suitably to allow proper parallelization.

1.4.2   Parallel memory hierarchy model

In this memory model, a parallel computer consists of several processors connected through a shared memory, where any processor can access any location in the shared memory.

A very generic memory model to represent an example of shared-memory multicore model is the *parallel memory hierarchy (PMH)* model as shown in Figure 1.1. The parallel memory hierarchy (PMH) [Blelloch et al., 2011] model models the memory hierarchy of many real parallel systems. It consists of $h+1$ levels as shown in Figure. The leaves of the tree at level-0 are the p processors and the internal nodes at level-i $(i \in [1, h])$ are the ideal

caches. Each cache at a level l is represented by $\langle M_l, B_l, f_l \rangle$, which means the cache size is M_l, the cache has f_l children (aka branch factor or fanout), and the cache line size between the cache and any of its children is B_l.

Here are the assumptions from [Blelloch et al., 2011]: (i) $B_i \geq 1$, $M_i \geq B_i$ for $i \in [1, h]$; (ii) $M_h = \infty$, $M_0 = p_0 = 0$, $B_0 = 1$; (iii) $B_i \geq B_{i-1}$ for $i \in [1, h-1]$; and (iv) $M_i/B_i \geq p_i M_{i-1}/B_{i-1}$ for $i \in [2, h]$ and $M_1/B_1 \geq p_1$.
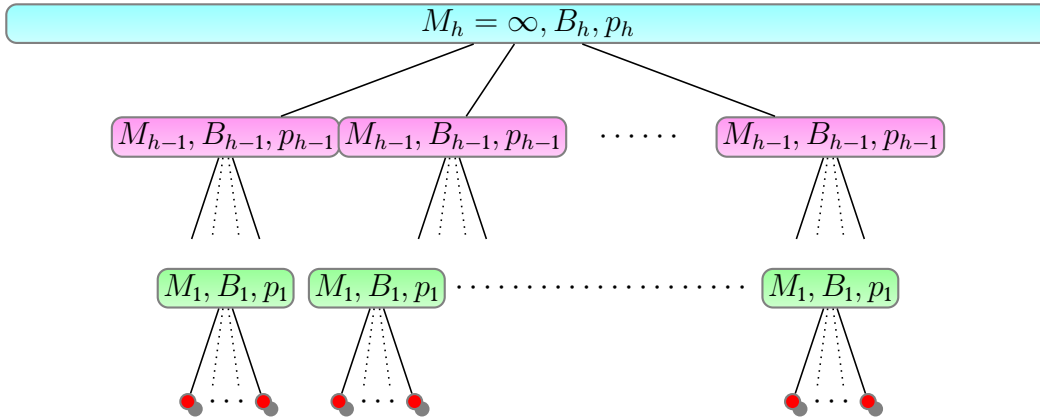
Figure 1.1: The parallel memory hierarchy model.

A cache can be shared by all processors that are rooted at that cache. Algorithms divide a task (or problem) into subtasks and each subtask is executed as a *thread* on a processor. When multiple threads are run to process multiple subtasks the problem will be solved efficiently and this process is termed *multithreading*. Also, algorithms developed in this shared-memory model are generally faster because the shared variable that is modified by a processor is immediately accessible to the other processors and they need not send / receive messages (which is a rather slow process).

1.4.3 Parallel algorithms

Similar to sequential algorithm design techniques, we also have techniques for designing parallel algorithms. Some of them are: divide-and-conquer; randomization for sampling [Rajasekaran and Sen, 1993], symmetry breaking [Luby, 1986], and load balancing; parallel pointer techniques such as pointer jumping [Wyllie, 1979], Euler tour [Tarjan and Vishkin, 1985], graph contraction [Miller and Reif, 1989, Miller and Reif, 1991]; ear decomposition [Maon et al., 1986, Miller and Ramachandran, 1992]; finding small graph separators [Reif, 1993]; hashing [Karlin and Upfal, 1988, Vishkin, 1984]; and iterative techniques [Bertsekas and Tsitsiklis, 1989].

Several parallel algorithms [Blelloch and Maggs, 2010] have been proposed, for example, prefix sums [Stone, 1975], list ranking [Reid-Miller, 1994, Anderson and Miller, 1990, Anderson and Miller, 1991], and graph algorithms [Reif, 1993, Jaja, 1992, Gibbons and Rytter, 1989].

1.5 Scheduling algorithms

The total running time of a multithreaded program on a parallel machine depends not only on minimizing work and span but also how well the computations are scheduled on different processors, which is done by an algorithm called scheduler.

A *scheduler* maps parallel tasks of a multithreaded program to different processors in a parallel machine. It load balances the work among different processing elements automatically without the intervention of the programmer. Several schedulers have been proposed, analyzed, and implemented in literature. Optimal algorithms scheduled using bad schedulers does not lead to high-performance and similarly slow algorithms scheduled using the best schedulers also do not lead to great performance. For the best performance, we need parallel algorithms with good cache efficiency and good parallelism and also scheduling algorithms (or schedulers) that exploit these characteristics. Please refer [Kwok and Ahmad, 1999] for a survey on schedulers.

It is important to note that a *job scheduler* is different from a *task scheduler*. A job scheduler schedulers different processes to a processor (or more processors). A job scheduler is part of an operating system. An operating system takes the responsibility of scheduling jobs. On the other hand, a task scheduler schedules different tasks or threads of work units to different processors or processing elements. Task / thread scheduling is part of parallel algorithms. A programming language runtime system takes the responsibility of scheduling tasks / threads.

1.5.1 Greedy scheduler

A *greedy scheduler* tries to perform as much work as possible at every step. Let p denote the number of processors in a parallel machine. At any execution step, if there are fewer than p tasks then all tasks are executed and the step is called an *incomplete step*. On the other hand, if there are greater than or equal to p tasks at any step, then any p of the tasks are executed and the step is called a *complete step*.

From [Graham, 1969, Brent, 1974], we get $T_p \leq T_1/p + T_\infty$. In [Eager et al., 1989] it was shown that any greedy scheduler will satisfy this bound. Also, the methodology of using work and span to analyze parallel algorithms was proposed. It is easy to show that $T_p \leq 2T_p^*$, where T_p^* is the parallel running time of optimal scheduling. Finding an optimal schedule is an NP-complete problem [Garey and Johnson, 2002]. The greedy scheduling for an example DAG when $p = 1$ and $p = 2$ are shown in Figures 1.2(b, c), respectively. Greedy schedulers perform very well in practice.

1.5.2 Parallel depth first scheduler

A *parallel depth first (PDF) scheduler* [Blelloch et al., 1995, Blelloch et al., 1999, Blelloch and Gibbons, 2004] is a priority scheduler that schedules most recent ready nodes. The priority to tasks is given to those tasks the sequential program would have executed earlier. The schedule is equivalent to the depth first search and hence the scheduler keeps scheduling the nodes depth-wise until there are no ready nodes. As the schedule happens in depth first order it is true that the cache performance of this scheduling approach is better than that of a randomized greedy schedule. However, the scheduling is centralized (and not distributed) and this puts burden on a single thread and affects performance.
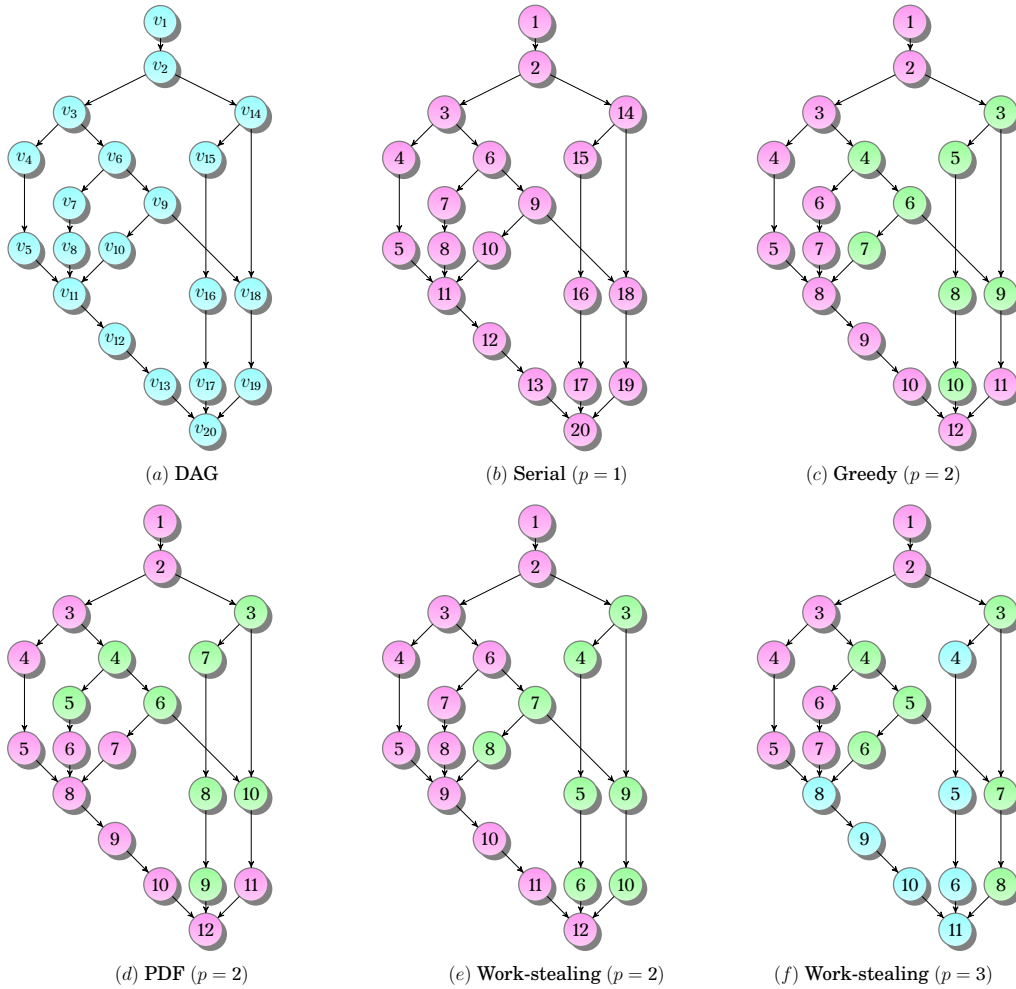
Figure 1.2: (a) A directed acyclic graph (DAG) representation for a multithreaded program execution. Scheduling of the DAG using: (b) greedy scheduler for $p = 1$. (c) greedy scheduler for $p = 2$. (d) PDF scheduler for $p = 2$. It is also called 2-DF schedule. (e) work-stealing scheduler for $p = 2$. (f) work-stealing scheduler for $p = 3$; In (b, c, d, e, f), the numbers represent the time steps; and red, green, and blue nodes represent that they are executed by processors 1, 2, and 3, respectively.

Using a PDF scheduler we obtain $T_p \leq T_1/p + T_\infty$ and a parallel space complexity of $S_p \leq S_1 + \mathcal{O}(pT_\infty)$, where S_1 is the space complexity when there is only 1 processor. The PDF scheduling for an example DAG when $p = 2$ is shown in Figure 1.2(d).

1.5.3 Work-stealing scheduler

A *work stealing scheduler* [Blumofe and Leiserson, 1998, Blumofe and Leiserson, 1999] is a distributed scheduler that distributes parallel tasks among the available processors to minimize the total execution time by allowing a processor to steal work from other randomly selected processor(s) when it runs out of work. Though the work stealing concept is old it was popularized by its implementation in the C-based Cilk multithreaded parallel programming runtime system [Blumofe et al., 1996b, Frigo et al., 1998].

In the scheduler, every processor maintains its own dequeue of tasks. Every processor

pops ready threads for execution and pushes spawned threads from its dequeue. Whenever a processors runs out of tasks, it steals from the top of the dequeue of a processor selected at random.

For any parallel algorithm that is run under the randomized work-stealing scheduler on a parallel machine with private caches, the expected parallel running time [Arora et al., 2001] and the parallel cache complexity [Acar et al., 2000] is given by $T_p = T_1/p + \mathcal{O}(T_\infty)$ and $Q_p = \mathcal{O}(Q_1 + p(M/B)T_1)$, respectively. Also, the parallel space complexity is bounded as $S_p \leq pS_1$, where S_1 is the serial space complexity. The work-stealing scheduling for an example DAG when $p = 2$ and $p = 3$ are shown in Figures 1.2(e, f) respectively. Work stealing schedulers perform very well in practice and hence are used in Cilk [Blumofe et al., 1996b, Frigo et al., 1998], Cilk++ [Leiserson, 2010], Intel Cilk Plus [Plus, 2016], and OpenMP [Chapman et al., 2008]. They are also used in Java, Microsoft Visual Studio, GCC, and several other runtime systems.

1.5.4 Space-bound scheduler

A *space-bounded scheduler* [Chowdhury et al., 2013, Blelloch et al., 2011, Simhadri et al., 2014] was designed to overcome the limitation of the work-stealing scheduler and achieve optimal parallel cache efficiency. The scheduler works for recursive divide-and-conquer parallel computation where a constant number of subtasks are generated by any task. In this scheduling method, the programmer sends a space bound (the upper bound on the amount of space that will be consumed by a task) of a task as a hint to the scheduler. When a task is anchored to a cache at some level, then the task will only be executed by the processors under the subtree rooted at that cache.

For an algorithm that is run under the space-bounded scheduler, $T_p = \Theta(T_1/p)$ and $Q_p = \mathcal{O}(Q_1)$. The scheduler cannot be used to schedule the computation DAG of Figure 1.2 (a) because the scheduler works only for trees of constant branch factor.

1.6 Algorithm design techniques

Algorithm design techniques are like problem-solving strategies that give generic methods and processes to crack algorithmic problems. There is an old proverb: "Give a man a fish and you feed him for a day; teach a man to fish and you feed him for a lifetime." Adapting the proverb to algorithm design, we can state: "Give a man an algorithm and you solve his current algorithmic problem; teach a man several algorithm design techniques and you solve most of his algorithmic problems."

There are several algorithm design strategies [Levitin, 2011] that can be used to design new algorithms for new problems such as brute force, divide-and-conquer, decrease-and-conquer, transform-and-conquer, greedy technique, dynamic programming, iterative improvement, backtracking, and branch-and-bound.

The algorithms presented in this dissertation use a combination of the two algorithm design techniques: divide-and-conquer and dynamic programming. In subsequent sections, we will briefly describe the important aspects of the two algorithm design techniques.

1.6.1   Divide-and-conquer

Divide-and-conquer is a powerful algorithm design strategy used to solve a problem by breaking it down into smaller and simpler subproblems. The general plan of a divide-and-algorithm is as follows:

1. [*Divide*.] The given problem is divided into smaller subproblems (typically the subproblems are independent and are of the same size).
2. [*Conquer*.] The subproblems are solved (typically using recursion)
3. [*Combine*.] If necessary, the solutions to the subproblems are combined to form the solution to the original problem.

Recursive divide-and-conquer algorithms have the following advantages:

⋆ [*Complexity analysis*.] They can be represented succinctly and can be analyzed for complexities using recurrence relations [Bentley, 1980].
⋆ [*Efficiency*.] They are usually efficient [Levitin, 2011] in the sense that they reduce the total number of computations. E.g.: Comparison-based sorting algorithms that use divide-and-conquer require $\mathcal{O}(n \log n)$ comparisons.
⋆ [*Easy parallelization*.] They can be parallelized easily [Mou and Hudak, 1988, Blelloch and Maggs, 2010] as the subproblems are typically independent, which can be run in parallel.
⋆ [*Cache-efficiency and cache-obliviousness*.] They often are (or can be made) cache-efficient and cache-oblivious [Frigo et al., 1999, Chatterjee et al., 2002, Frens and Wise, 1997].
⋆ [*Processor-obliviousness*.] They often are (or can be made) processor-oblivious [Frigo et al., 1999, Chowdhury and Ramachandran, 2008].
⋆ [*Energy efficiency*.] They can be energy efficient as they reduce the total number of computations or the number of cache misses [Tithi et al., 2015].

For example, some of the fastest sorting algorithms such as merge sort and quicksort are based on divide-and-conquer.

1.6.2   Dynamic programming (DP)

Dynamic programming (DP) is an algorithm design technique used to solve a problem by breaking it down into smaller and simpler subproblems. The method is used to solve problems that have the properties of overlapping subproblems and optimal substructure. DP is especially used to solve optimization problems. Surprisingly, many combinatorial problems that typically require an exponential time for the standard algorithms to solve, can be solved in polynomial time and space using DP. This is the reason some algorithmists compare DP to a magical wand that turns stones into gold.

DP is used in a variety of fields such as operations research, parsing of ambiguous languages [Giegerich et al., 2004], sports and games [Romer, 2002, Duckworth and Lewis, 1998, Smith, 2007], economics [Rust, 1996], finance [Robichek et al., 1971], and agriculture [Kennedy, 1981]. In computational biology, several significant problems such as protein-homology search, gene-structure prediction, motif search, analysis of repetitive genomic elements, RNA secondary-structure prediction, and interpretation of mass spectrometry data [Bafna and Edwards, 2003, Durbin et al., 1998, Gusfield, 1997, Waterman et al., 1995] make use of dynamic programming.

In general, dynamic programming can be described as filling a table efficiently. The

table to be filled is called DP table and it consists of cells in a grid format. Initially, only a few cells are filled. Every cell can be computed using the values of already computed cells using a recurrence relation. Typically, all the cells of the DP table has to be filled using the recurrence relation. Often, we will be interested in the final values of one or more cells of the DP table, we call *goal cells*.

Dynamic programming can be implemented in two ways:

* [*Top-down*.] In this approach (also called memoization), the cells that the goal cell(s) depend upon are computed and stored. This process happens recursively (from root to leaves in the recursion tree) and only those cells that are required are stored / cached. The approach might lead to more space usage due to recursion.

* [*Bottom-up*.] In this approach, all cells will be computed exactly once in a bottom-up fashion (from leaves to root in the recursion tree) finally reaching the goal cell(s). This approach is the most widely used approach to implement dynamic programming algorithms.

Dynamic programming algorithms have the following advantages:

* [*Efficiency*.] Several problems that require exponential time naively can be solved using DP in polynomial time.

* [*Suited for discrete optimization problems*.] DP is well-suited to solve discrete optimization (minimization or maximization) problems as it explores all possible ways and selects the one that optimizes a given condition.

Methods to solve discrete optimization problems are called mathematical programming (or mathematical optimization) methods [Lew and Mauch, 2006]. It includes methods such as linear programming, quadratic programming, iterative methods, dynamic programming etc. There are efficient algorithms such as simplex method to solve linear programming problems. However, there is no universal efficient algorithm to solve all dynamic programming problems largely due of its generality.

Differences between plain recursion and dynamic programming

Consider the Fibonacci example. We know that the Fibonacci number $F(i)$ is defined recursively as

$$F(i) = \begin{cases} i & \text{if } i = 0 \text{ or } 1, \\ F(i-1) + F(i-2) & \text{if } i > 1. \end{cases} \tag{1.1}$$

The recursion tree for this computation is given in Figure 1.3. The complexity of this algorithm (or the number of nodes in the recursion tree) is of the order $\Theta(\phi^n)$, where ϕ is the golden ratio. Using dynamic programming, we can write the recurrence as

$$F[i] = \begin{cases} i & \text{if } i = 0 \text{ or } 1, \\ F[i-1] + F[i-2] & \text{if } i > 1. \end{cases} \tag{1.2}$$

The recursion-DAG is shown in Figure 1.3. The complexity improves drastically to $\Theta(n)$. A summary table of the differences between simple recursion and dynamic programming techniques is given in Table 1.1.

In dynamic programming, finding the subproblems is the toughest part. In fact, depending on the identified subproblems, a problem can be solved in different ways using the same dynamic programming technique.
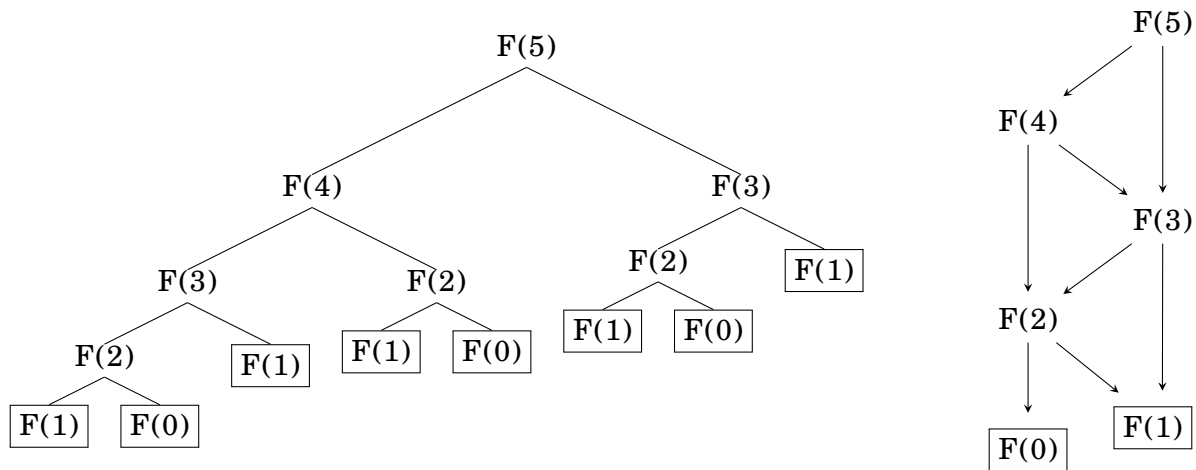
Figure 1.3: Computation of the Fibonacci number F(5) with base cases represented by rectangles. Left: Recursion tree when simple recursion is used. Right: Computation DAG when dynamic programming is used.

Feature	Recursion	Dynamic programming
Execution	Top-down	Bottom-up
Storing / caching	No	Yes
Unrolling	Recursion tree	Computation DAG
Complexity	Exponential (typically)	Polynomial
Subproblems	Overlapping / independent	Overlapping
Recurrence	Uses parentheses (to denote functions)	Uses square brackets (to denote table)

Table 1.1: Differences between recursion and dynamic programming.

1.6.3 Divide-and-conquer dynamic programming

We can combine divide-and-conquer and dynamic programming techniques. The divide-and-conquer algorithms to solve dynamic programming problems are superior to existing algorithms both in theory and in practice. Such algorithms have all the advantages of divide-and-conquer (with the exception of reducing the total number of computations). Often, the algorithms are:

　✶ [*Efficient*.] – cache-efficient, parallel, and energy efficient.
　✶ [*Portable*.] – cache-oblivious and processor-oblivious.
　✶ [*Robust*.] – cache-adaptive.

　Due to all the reasons mentioned above, divide-and-conquer technique is by far the most powerful way to implement dynamic programming algorithms. Almost all of our algorithms presented in this dissertation are based on divide-and-conquer except the algorithms that discover them.

1.7　Dynamic programming implementations

Dynamic programs are described through recurrence relations that specify how the cells of a DP table must be filled using already computed values for other cells. Dynamic programs are traditional implemented iteratively i.e., using series of loops. Standard iterative algorithms do not have temporal locality, do not achieve very high parallelism (typically, for problems having complicated non-local dependencies), and have poor bandwidth efficiency.

We can improve the performance of iterative algorithms using blocking / tiling / strip-mining. The tiled iterative algorithms are cache-efficient but they are cache-aware. Also, they do not achieve higher parallelism (for complicated non-local dependencies) and are not cache-adaptive. Table 1.2 summarizes the differences between the three algorithms. It is important to note that we can improve to parallelism of iterative and tiled iterative algorithms to near-optimal by using wavefront-order execution. However, the programs become horrendously complicated for complicated non-local DP dependencies.

In the entire dissertation, we will be interested only in recursive divide-and-conquer parallel DP algorithms.

Feature	Iterative algorithms	Tiled iterative algorithms	Recursive D&C algorithms
Cache-efficiency	Bad	Excellent	Excellent
Cache-obliviousness	Yes	No	Yes
Cache-adaptivity	Yes	No	Yes
Higher parallelism	No	No	Yes
Highly optimizable kernels	NA	No	Yes
Energy efficiency	Bad	Excellent	Excellent
Bandwidth efficiency	Bad	Excellent	Excellent

Table 1.2: Comparison of different features in iterative, tiled iterative, and recursive divide-and-conquer dynamic programming algorithms. NA means not applicable.

1.7.1 Iterative DP algorithms

These are the standard looping algorithms consisting of series of *for* and/or *while loops*, some conditional statements, and assignments. These loop-based codes are straightforward to implement, often have *good spatial cache locality*, and benefit from hardware prefetchers. But looping codes suffer in performance from *poor temporal cache locality*. Indeed, when the DP table is too large to fit into cache, scanning the entire table over and over again means that no significant portion of it can be retained in the cache for reuse.

Iterative DP implementations are also often *inflexible* in the sense that the loops and the data in the DP table cannot be suitably reordered in order to optimize for better spatial locality, parallelization, and/or vectorization. Such inflexibility arises because the codes often read from and write to the same DP table, and thus imposing strict read-write ordering of the cells.

1.7.2 Blocked / tiled iterative DP algorithms

The method of blocking / tiling is called *loop tiling* or *strip mining*. Tiled iterative DP algorithms are iterative algorithms with the DP table being blocked / tiled such that a tile exactly fits the cache. When a tile is brought to cache, as much work as possible is done before kicking the tile out of the cache. Hence, these algorithms have *excellent temporal locality*. However, the optimal tile size is a function of the cache size and hence these algorithms are *cache-aware* and *non-portable*.

1.7.3   Recursive divide-and-conquer DP algorithms

Our two most important priorities are: *performance* and *portability*. Iterative algorithms suffer in performance and tiled iterative algorithms are non-portable. These limitations are handled by the recursive divide-and-conquer algorithms as they lead to both high performance and portability.

A necessary condition for an algorithm for a problem to have temporal locality is that the work (total number of computations) done by the algorithm must be asymptotically larger than the total space usage.

Hence, if we have a cache-oblivious recursive divide-and-conquer DP algorithm for a problem, it does not necessarily mean that it is cache-efficient (through temporal locality). The minimum criteria the algorithm has to satisfy is that the average number of computations per unit of space must be $\omega(1)$ (asymptotically more than constant).

Recursive divide-and-conquer algorithms consist of one or more recursive functions. Because of their recursive nature such algorithms are known to have excellent (and often optimal) temporal locality. Efficient implementations of these algorithms use iterative kernels when the problem size becomes reasonably small. But unlike standard loop-based DP codes, the loops inside these iterative kernels can often be easily reordered, thus allowing for better spatial locality, vectorization, parallelization, and other optimizations.

The sizes of the iterative kernels are determined based on vectorization efficiency and overhead of recursion, and not on cache sizes, and thus the algorithms remain *cache-oblivious* and *portable*. Unlike tiled looping codes these algorithms are also *cache-adaptive* [Bender et al., 2014] — they passively self-adapt to fluctuations in available cache space when caches are shared with other concurrently running programs.

1.8   Automatic algorithm discovery

Automation is a revolutionary idea. Automation is the process of using automatic devices and computing machines to replace the routine human effort and labor. For example, online shopping systems such as amazon.com, mass manufacturing of products, ATM machines, ticket machines, food machines, automated testing, robotics, GPS navigation, vehicles, traffic lights, video surveillance, automated replies, self-driving cars, and thousands of other stuff. Automation has several advantages:

- ✮ [*Saves time*.] Automation replaces manual effort with machine effort. Hence, it saves a lot of time, energy, and money of humans. Reducing the time improves productivity.
- ✮ [*Increases accuracy*.] Machines perform the work they are designed or programmed to do. As they do not take decisions (like we do), they follow a human's command as it is and they do not deviate from what is supposed to be done. Hence, this machine process maintains the accuracy and does not give an opportunity for humans to make mistakes :).
- ✮ [*Deeper understanding of the process*.] Automating a human effort leads to a deeper understanding of the process itself and this understanding might be very useful.

In computer science, automation is majorly used to replace the human labor in the software development process. Automation is also used to some extent in problem solving. There are automated frameworks for domain specific languages (DSL), e.g.: Pochoir [Tang et al., 2011], automatically generate high performing implementations for certain problems

from known efficient algorithms. However, there is little work for automation of nontrivial design of algorithms [Kant, 1985] given just the problem specifications. This dissertation is the first major step in the science of discovery of nontrivial efficient cache-oblivious algorithms.

Designing algorithms is hard. Discovering a fast and portable algorithm requires expertise from different fields such as algorithmics, data structures, parallel algorithms, computer architecture, compiler design, operating system, and so on. A domain expert (human) who has a deeper understanding of the problem and who has a good experience in solving similar problems can design an efficient algorithm given sufficient time, energy, and resources. However, this method does not scale very well when the number of problems to be solved gets bigger and bigger. Also, the designed algorithms must be proved correct, analyzed for their complexities, and implemented. Hence, an intelligent way to address this issue is to build machines to solve problems automatically or in computer science terminology – design an algorithm that discovers other algorithms.

Having a system that can automatically generate highly efficient parallel implementations from easy-to-write and concise specifications of the problem is extremely useful. Computational scientists who specialize in subjects other than computer science such as biology, chemistry, physics, material science, and so on without a formal training in computer science find it difficult to develop highly efficient algorithms for the state-of-the-art supercomputers. A system that automates the complicated task of efficient algorithm discovery saves hundreds of thinking hours and brings the power of supercomputing closer to programmers without deep computer science (CS) background.

1.8.1 Program synthesis

Program synthesis [Steier and Anderson, 2012] is a field in computer science that is related to automatic programming or automatic algorithm design. The idea for program synthesis originated in the 1960s with the aim to automate programming using artificial intelligence. In recent decades, formal mathematical methods are used to specify, derive (using deductive methods), and verify (using formal verification methods such as theorem provers) algorithms.

There is a large literature using formal methods to synthesize programs [Klonatos et al., 2013, Armando et al., 1999]. However, all these methods use different types of logical calculus and they require a lot of user interaction such as specifying the functions or other parameters that the algorithm to be designed must use. This is the reason most of these methods are not fully automatic.

1.8.2 Polyhedral compilers

There are several systems to automatically parallelize programs [Bondhugula et al., 2008, Banerjee et al., 1993, Bae et al., 2013, Feautrier, 1996, Hendren and Nicolau, 1990, Campanoni et al., 2012]. Some systems are described for loops and other systems are described for trees and DAGs. The literature of parallelizers is vast. A majority of these parallelizers use loop transformations in the polyhedral model [Feautrier and Lengauer, 2011]. Some of these systems can also optimize the programs for cache locality and hence they generate cache-efficient parallel algorithms.

A few drawbacks of the polyhedral compilers or parallelizers are:

⋆ [*Cache-awareness*.] The generated algorithms depend on the cache parameters and hence are cache-aware and not cache-oblivious.

⋆ [*Loop programs on arrays*.] The application of the model is limited to loops that work on arrays.

In the polyhedron model, the point-sets form a polyhedra in $\mathbb{Z}^d$. The point-sets are represented as solutions to a system of affine inequalities $Ax \leq b$, where A is a constant matrix, x is a variable vector, and b is a constant vector. The solutions can be found using loop transformations and integer linear programming.

1.9   Dissertation outline

In this section, we give the statement of the dissertation; contributions of the dissertation to theoretical and practical computer science; and finally the organization of topics in the dissertation.

1.9.1   Dissertation statement

The main motivation for the work in the dissertation is to bring automated algorithm design to the masses. Developing algorithms(s) (resp. framework(s)) that can be used to automatically (resp. semi-automatically) discover algorithms can be very useful to computational scientists and programmers with less expertise in computer science. This dissertation gives evidence to support the statement:

> It is possible to develop algorithm(s) / framework(s) to automatically / semi-automatically discover algorithms that are simultaneously nontrivial, fast, portable, and robust, which can be used to solve to a wide class of dynamic programming problems.

1.9.2   Dissertation contributions

The major contributions of the dissertation are as follows:

⋆ [Autogen.] We design an algorithm called Autogen that for a wide class of dynamic programming problems automatically discovers nontrivial efficient (cache-efficient, parallel, and energy-efficient), portable (processor- and cache-oblivious), and robust (processor- and cache-adaptive) recursive divide-and-conquer algorithms given iterative descriptions of the DP recurrences. We prove the correctness of Autogen, analyze the cache-complexities of Autogen-discovered algorithms, and implement Autogen.

⋆ [Autogen-Wave.] We design a framework called Autogen-Wave that for a wide class of dynamic programming problems semi-automatically discovers non-trivial recursive divide-and-conquer wavefront algorithms from standard divide-and-conquer algorithms and DP recurrences. Autogen-Wave is the sequel of Autogen as it takes the Autogen-discovered algorithms as input and discovers algorithms that have near-optimal parallelism (retaining all the other advantages of the input algorithms).
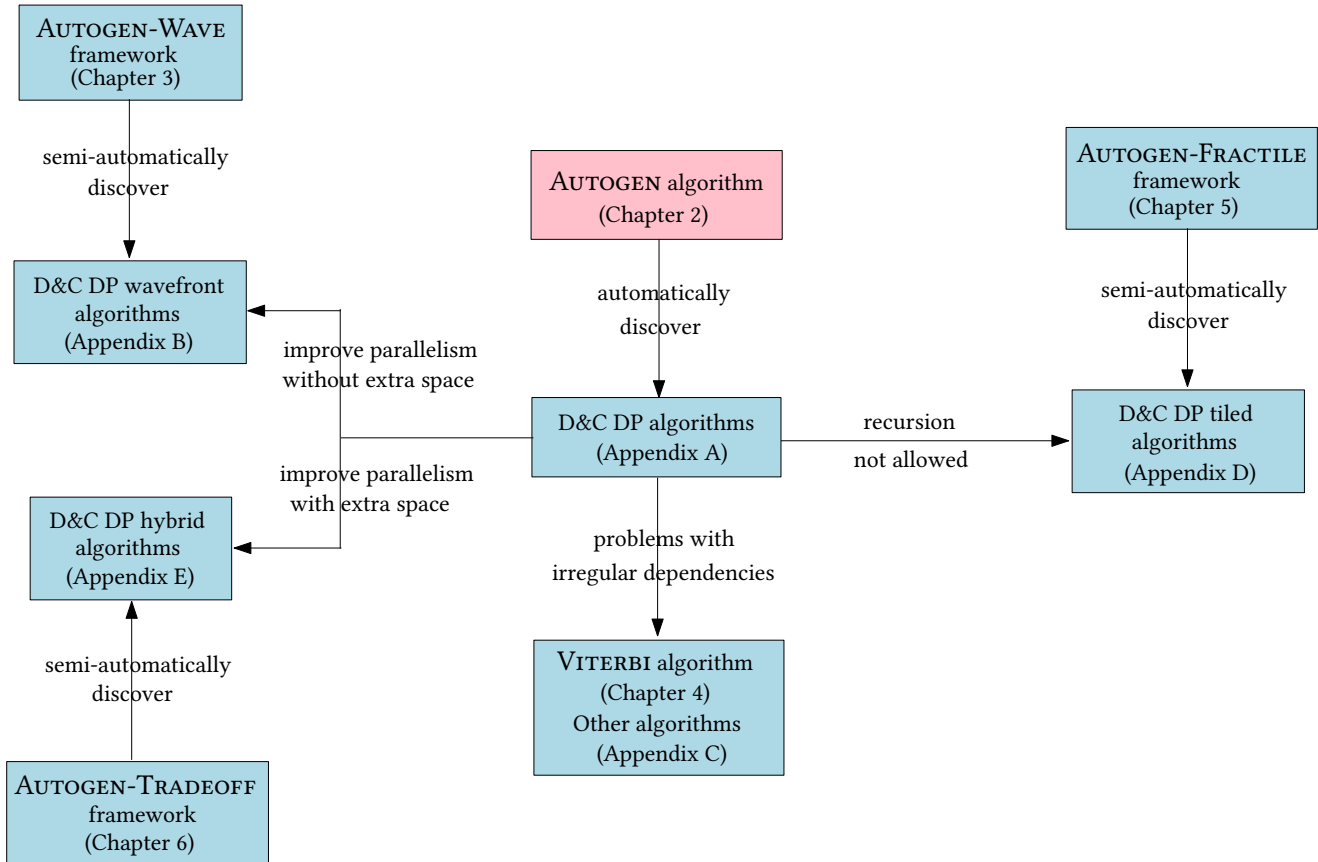
Figure 1.4: Dissertation map.

* [*VITERBI.*] AUTOGEN cannot be directly used to develop efficient algorithms for data-dependent / data-sensitive dynamic programs. The Viterbi recurrence is an extremely important example of a data-dependent dynamic program for which there is no efficient resource-oblivious algorithm. We design the first provably correct efficient resource-oblivious parallel Viterbi algorithm.
* [*AUTOGEN-FRACTILE.*] We design a framework called AUTOGEN-FRACTILE that can be used to semi-automatically discover efficient cache-aware tiled algorithms based on recursive divide-and-conquer, for a wide class of dynamic programming problems. We prove strong theoretical bounds on cache locality and parallelism. We design the fastest GPU algorithms for several dynamic programs using the framework.
* [*AUTOGEN-TRADEOFF.*] We design a framework called AUTOGEN-TRADEOFF that can be used to design algorithms to asymptotically increase the parallelism of some of the AUTOGEN-discovered algorithms without affecting cache-efficiency, but using extra space.

The relation between all the chapters in the dissertation and how they relate to the theme of the dissertation is shown in Figure 1.4. A list of my papers appear in Table 1.3. The work in the dissertation is a joint contribution. The experiments (included in the dissertation for completeness) are the work of Jesmin Jahan Tithi, Stephen Tschudi, Rathish Das, Mohammad Mahdi Javanmard, Yunpeng Xiao, and Isha Khanna.

No.	Paper	Conf / journal	In thesis?
Major contribution			
1	*AUTOGEN: Automatic Discovery of Cache-Oblivious Parallel Recursive Algorithms for Solving Dynamic Programs* with Rezaul Chowdhury, Jesmin Jahan Tithi, Charles Bachmeier, Bradley Kuszmaul, Charles E. Leiserson, Armando Solar-Lezama, & Yuan Tang	PPoPP 2016 (Invited to TOPC)	Chap. 2
2	*AUTOGEN: Automatic Discovery of Cache-Oblivious Parallel Recursive Algorithms for Solving Dynamic Programs* with Rezaul Chowdhury, Jesmin Jahan Tithi, Charles Bachmeier, Bradley Kuszmaul, Charles E. Leiserson, Armando Solar-Lezama, & Yuan Tang	Under prep.	Chap. 2
3	*Provably Efficient Scheduling of Cache-Oblivious Wavefront Algorithms* with Rezaul Chowdhury, Jesmin Jahan Tithi, & Yuan Tang	Under prep.	Chap. 3
4	*An Efficient Cache-Oblivious Parallel Viterbi Algorithm* with Rezaul Chowdhury, Vivek Pradhan, Jesmin Jahan Tithi, & Yunpeng Xiao	Euro-Par 2016	Chap. 4
5	*A Framework for Designing External-Memory RAM-Oblivious GPU Algorithms for Dynamic Programs* with Rezaul Chowdhury, Rathish Das, Mohammad Mahdi Javanmard, & Stephen Tschudi	Under review	Chap. 5
6	*Space-Parallelism Tradeoff for Cache-Oblivious Parallel Algorithms* with Rezaul Chowdhury, Mohammad Mahdi Javanmard, Isha Khanna, Premadurga Kolli, & Stephen Tschudi	Under prep.	Chap. 6
7	*Divide-and-Conquer Variants of Bubble, Selection, & Insertion Sorts* with Rezaul Chowdhury	Under prep.	Chap. A
8	*A Framework to Discover Combinatorial Algorithms* with Rama Badrinath & Abhiram Natarajan	Under prep.	✗
9	*Premtuatoins & PAttERns* with Rezaul Chowdhury	Under prep.	✗
10	*The Range 1 Query (R1Q) Problem* with Michael A. Bender, Rezaul Chowdhury, Samuel McCauley, & Yuan Tang	COCOON 2014 (Invited to TCS)	✗
11	*The Range 1 Query (R1Q) Problem* with Michael A. Bender, Rezaul Chowdhury, Samuel McCauley, & Yuan Tang	TCS 2016	✗
Minor contribution			
12	*Cache-Oblivious Wavefront: Improving Parallelism of Recursive Dynamic Programming Algorithms Without Losing Cache-Efficiency* with Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, & Rezaul Chowdhury	PPoPP 2015	✗
13	*The I/O Complexity of Computing Prime Tables* with Michael A. Bender, Rezaul Chowdhury, Alex Conway, Martin Farach-Colton, Rob Johnson, Samuel McCauley, Bertrand Simon, & Shikha Singh	LATIN 2016	Chap. A (part)
14	*High-Performance Energy-Efficient Recursive Dynamic Programming with Matrix-Multiplication-like Flexible Kernels* with Jesmin Jahan Tithi, Aakrati Talati, Sonal Aggarwal, & Rezaul Chowdhury	IPDPS 2015	Chap. A (part)

Table 1.3: My papers. In mathematics and theoretical computer science, we often follow the convention of listing authors in alphabetical order of last names.

1.9.3 Dissertation organization

The organization of the topics in the dissertation is as follows. We present our AUTOGEN algorithm, its proof of correctness, and cache complexity of AUTOGEN-discovered algorithms in Chapter 2. We present recursive divide-and-conquer algorithms to several dynamic programming problems in Appendix A. In Chapter 3, we present our AUTOGEN-WAVE framework that can be used to discover divide-and-conquer wavefront algorithms. We present the AUTOGEN-WAVE-discovered divide-and-conquer wavefront algorithms in Appendix B. In Chapter 4, we present a provably cache-efficient cache-oblivious parallel VITERBI algorithm. Efficient algorithms are presented for DP problems with irregular data dependencies in Appendix C. In Chapter 5, we present the AUTOGEN-FRACTILE framework that can be used

to discover highly efficient recursive divide-and-conquer-based tiled algorithms. In Appendix D, we present several divide-and-conquer tiled algorithms. Finally, we present our AUTOGEN-TRADEOFF framework in Chapter 6 and present hybrid algorithms in Appendix E.

Chapter 2

Automatic Discovery of Efficient Divide-&-Conquer DP Algorithms

We present AUTOGEN — an algorithm that for a wide class of dynamic programming (DP) problems automatically discovers highly efficient cache-oblivious parallel recursive divide-and-conquer algorithms from inefficient iterative descriptions of DP recurrences. AUTOGEN analyzes the set of DP table locations accessed by the iterative algorithm when run on a DP table of small size, and automatically identifies a recursive access pattern and a corresponding provably correct recursive algorithm for solving the DP recurrence. We use AUTOGEN to autodiscover efficient algorithms for several well-known problems. Our experimental results show that several autodiscovered algorithms significantly outperform parallel looping and tiled loop-based algorithms. Also these algorithms are less sensitive to fluctuations of memory and bandwidth compared with their looping counterparts, and their running times and energy profiles remain relatively more stable. To the best of our knowledge, AUTOGEN is the first algorithm that can automatically discover new nontrivial divide-and-conquer algorithms.

2.1   Introduction

AUTOGEN is an algorithm for automatic discovery of efficient recursive divide-and-conquer *dynamic programming* (DP) algorithms for multicore machines from naive iterative descriptions of the dynamic programs. DP [Bellman, 1957, Sniedovich, 2010, Cormen et al., 2009] is a widely used algorithm design technique that finds optimal solutions to a problem by combining optimal solutions to its overlapping subproblems, and explores an otherwise exponential sized search space in polynomial time by saving solutions to subproblems in a table and never recomputing them. DP is extensively used in computational biology [Bafna and Edwards, 2003, Durbin et al., 1998, Gusfield, 1997, Waterman et al., 1995], and in many other application areas including operations research, compilers [Lew and Mauch, 2006], sports [Romer, 2002, Duckworth and Lewis, 1998], games [Smith, 2007], economics [Rust, 1996], finance [Robichek et al., 1971] and agriculture [Kennedy, 1981].

Dynamic programs are described through recurrence relations that specify how the cells of a DP table must be filled using already computed values for other cells. Such recurrences are commonly implemented using simple algorithms that fill out DP tables iteratively. These loop-based codes are straightforward to implement, often have good *spatial cache*

locality[1], and benefit from hardware prefetchers. But looping codes suffer in performance from poor *temporal cache locality*[2]. Iterative DP implementations are also often *inflexible* in the sense that the loops and the data in the DP table cannot be suitably reordered in order to optimize for better spatial locality, parallelization, and/or vectorization. Such inflexibility arises because the codes often read from and write to the same DP table, and thus imposing strict read-write ordering of the cells.

Recursive divide-and-conquer DP algorithms (see Table 2.1) can often overcome many limitations of their iterative counterparts. Because of their recursive nature such algorithms are known to have excellent (and often optimal) temporal locality. Efficient implementations of these algorithms use iterative kernels when the problem size becomes reasonably small. But unlike in standard loop-based DP codes, the loops inside these iterative kernels can often be easily reordered, thus allowing for better spatial locality, vectorization, parallelization, and other optimizations. The sizes of the iterative kernels are determined based on vectorization efficiency and overhead of recursion, and not on cache sizes, and thus the algorithms remain *cache-oblivious*[3] [Frigo et al., 1999] and more *portable* than cache-aware tiled iterative codes. Unlike tiled looping codes these algorithms are also *cache-adaptive* [Bender et al., 2014] — they passively self-adapt to fluctuations in available cache space when caches are shared with other concurrently running programs.

For example, consider the dynamic program for solving the parenthesis problem [Galil and Park, 1994] in which we are given a sequence of characters $S = s_1 \cdots s_n$ and we are required to compute the minimum cost of parenthesizing S. Let $C[i, j]$ denote the minimum cost of parenthesizing $s_i \cdots s_j$. Then the DP table $C[0 : n, 0 : n]$ is filled up using the following recurrence:

$$C[i,j] = \begin{cases} \infty & \text{if } 0 \leq i = j \leq n, \\ v_j & \text{if } 0 \leq i = j - 1 < n, \\ \min_{i \leq k \leq j} \{(C[i,k] + C[k,j]) + w(i,k,j)\} & \text{if } 0 \leq i < j - 1 < n; \end{cases} \quad (2.1)$$

where the v_j's and function $w(\cdot, \cdot, \cdot)$ are given.

Figure 3.1 shows a serial looping code LOOP-PARENTHESIS implementing Recurrence 3.1. Though the code is really easy to understand and write, it suffers from poor cache performance. Observe that the innermost loop scans one row and one column of the same DP table C. Assuming that C is of size $n \times n$ and C is too large to fit into the cache, each iteration of the innermost loop may incur one or more cache misses leading to a total of $\Theta(n^3)$ cache misses in the ideal-cache model [Frigo et al., 1999]. Such extreme inefficiency in cache usage makes the code bandwidth-bound. Also this code does not have any parallelism as none of the three loops can be parallelized. The loops cannot also be reordered without making the code incorrect[4] which makes the code difficult to optimize.

Figure 3.1 shows the type of parallel looping code PAR-LOOP-PARENTHESIS one would write to solve Recurrence 3.1. We can analyze its parallel performance under the *work-span model* [Cormen et al., 2009] (chapter 27) which defines the *parallelism* of a code

---

[1]Spatial locality — whenever a cache block is brought into the cache, it contains as much useful data as possible.

[2]Temporal locality — whenever a cache block is brought into the cache, as much useful work as possible is performed on this data before removing the block from the cache.

[3]Cache-oblivious algorithms — algorithms that do not use the knowledge of cache parameters in the algorithm description.

[4]compare this with iterative matrix multiplication in which all 6 permutations of the three nested loops produce correct results

PAR-LOOP-PARENTHESIS(C,n)

1. **for** $t \leftarrow 2$ **to** $n-1$ **do**
2.   **parallel for** $i \leftarrow 1$ **to** $n-t$ **do**
3. $j \leftarrow t+i$
4.   **for** $k \leftarrow i+1$ **to** j **do**
5. $C[i,j] \leftarrow \min\left(\begin{array}{l} C[i,j], C[i,k]+ \\ C[k,j]+w(i,k,j) \end{array}\right)$

Parallelizer

LOOP-PARENTHESIS(C,n)

1. **for** $i \leftarrow n-1$ **to** 0 **do**
2.   **for** $j \leftarrow i+2$ **to** n **do**
3.   **for** $k \leftarrow i$ **to** j **do**
4. $C[i,j] \leftarrow \min\left(\begin{array}{l} C[i,j], C[i,k]+ \\ C[k,j]+w(i,k,j) \end{array}\right)$

AUTOGEN

$\mathcal{A}_{par}(\langle X,X,X\rangle)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}par}(\langle X,X,X\rangle)$
2. **else**
3.   **par:** $\mathcal{A}_{par}(\langle X_{11},X_{11},X_{11}\rangle)$, $\mathcal{A}_{par}(\langle X_{22},X_{22},X_{22}\rangle)$
4. $\mathcal{B}_{par}(\langle X_{12},X_{11},X_{22}\rangle)$

$\mathcal{B}_{par}(\langle X,U,V\rangle)$

1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}par}(\langle X,U,V\rangle)$
2. **else**
3. $\mathcal{B}_{par}(\langle X_{21},U_{22},V_{11}\rangle)$
4.   **par:** $\mathcal{C}_{par}(\langle X_{11},U_{12},X_{21}\rangle)$, $\mathcal{C}_{par}(\langle X_{22},X_{21},V_{12}\rangle)$
5.   **par:** $\mathcal{B}_{par}(\langle X_{11},U_{11},V_{11}\rangle)$, $\mathcal{B}_{par}(\langle X_{22},U_{22},V_{22}\rangle)$
6. $\mathcal{C}_{par}(\langle X_{12},U_{12},X_{22}\rangle)$
7. $\mathcal{C}_{par}(\langle X_{12},X_{11},V_{12}\rangle)$
8. $\mathcal{B}_{par}(\langle X_{12},U_{11},V_{22}\rangle)$

$\mathcal{C}_{par}(\langle X,U,V\rangle)$

1. **if** X is a small matrix **then** $\mathcal{C}_{loop\text{-}par}(\langle X,U,V\rangle)$
2. **else**
3.   **par:** $\mathcal{C}_{par}(\langle X_{11},U_{11},V_{11}\rangle)$, $\mathcal{C}_{par}(\langle X_{12},U_{11},V_{12}\rangle)$,
    $\mathcal{C}_{par}(\langle X_{21},U_{21},V_{11}\rangle)$, $\mathcal{C}_{par}(\langle X_{22},U_{21},V_{12}\rangle)$
4.   **par:** $\mathcal{C}_{par}(\langle X_{11},U_{12},V_{21}\rangle)$, $\mathcal{C}_{par}(\langle X_{12},U_{12},V_{22}\rangle)$,
    $\mathcal{C}_{par}(\langle X_{21},U_{22},V_{21}\rangle)$, $\mathcal{C}_{par}(\langle X_{22},U_{22},V_{22}\rangle)$

Figure 2.1: Left upper half: A parallel looping code that evaluates Recurrence 3.1. Left lower half: AUTOGEN takes the serial parenthesis algorithm as input and automatically discovers a recursive divide-and-conquer cache-oblivious parallel algorithm. Initial call to the divide-and-conquer algorithm is $\mathcal{A}_{par}(\langle C,C,C\rangle)$, where C is an $n \times n$ DP table and n is a power of 2. The iterative base-case kernel of a function $\mathcal{F}_{par}$ is $\mathcal{F}_{loop\text{-}par}$. Right: Pictorial representation of the recursive divide-and-conquer algorithm discovered by AUTOGEN. Data in the dark red blocks are updated using data from light blue blocks.

as T_1/T_∞, where T_p ($p \in [1,\infty)$) is the running time of the code on p processing cores (without scheduling overhead). Clearly, the parallelism of PAR-LOOP-PARENTHESIS is $\Theta(n^3)/\Theta(n^2) = \Theta(n)$. If the size M of the cache is known the code can be tiled to improve its cache performance to $\Theta\left(n^3/\left(B\sqrt{M}\right)\right)$, where B is the cache line size. However, such

rigid cache-aware tiling makes the code less portable, and may contribute to a significant loss of performance when other concurrently running programs start to use space in the shared cache.

Finally, Figure 3.1 shows the type of algorithm AUTOGEN would generate from the serial code. Though designing such a parallel recursive divide-and-conquer algorithm is not straightforward, it has many nice properties. First, the algorithm is cache-oblivious, and for any cache of size M and line size B it always incurs $\Theta\left(n^3/\left(B\sqrt{M}\right)\right)$ cache misses which can be shown to be optimal. Second, its parallelism is $\Theta\left(n^{3-\log_2 3}\right) = \omega\left(n^{1.41}\right)$ which is asymptotically greater than the $\Theta\left(n\right)$ parallelism achieved by the parallel looping code. Third, since the algorithm uses recursive blocking, it can passively self-adapt to a correct block size (within a small constant factor) as the available space in the shared cache changes during runtime. Fourth, it has been shown that function $\mathcal{C}_{loop\text{-}par}$ is highly optimizable like a matrix multiplication algorithm, and the total time spent inside $\mathcal{C}_{loop\text{-}par}$ asymptotically dominates the time spent inside $\mathcal{A}_{loop\text{-}par}$ and $\mathcal{B}_{loop\text{-}par}$ [Tithi et al., 2015]. Hence, reasonably high performance can be achieved simply by optimizing $\mathcal{C}_{loop\text{-}par}$.

We ran the recursive algorithm and the parallel looping algorithm from Figure 3.1 both with and without tiling on a multicore machine with dual-socket 8-core 2.7 GHz Intel Sandy Bridge processors ($2 \times 8 = 16$ cores in total), per-core 32 KB private L1 cache and 256 KB private L2 cache, and per-socket 20 MB shared L3 cache, and 32 GB RAM shared by all cores. All algorithms were implemented in C++, parallelized using Intel Cilk Plus extension, and compiled using Intel C++ Compiler v13.0. For a DP table of size 8000×8000, the recursive algorithm without any nontrivial hand-optimizations ran more than 15 times faster than the non-tiled looping code, and slightly faster than the tiled looping code when each program was running all alone on the machine. When we ran four instances of the same program (i.e., algorithm) on the same socket each using only 2 cores, the non-tiled looping code slowed down by almost a factor of 2 compared to a single instance running on 2 cores, the tiled looping code slowed down by a factor of 1.5, and the recursive code slowed down by a factor of only 1.15. While the non-tiled looping code suffered because of bandwidth saturation, the tiled looping code suffered because of its inability to adapt to cache sharing.

In this chapter, we present AUTOGEN — an algorithm that for a very wide class of DP problems can *automatically discover* efficient cache-oblivious parallel recursive divide-and-conquer algorithms from naive serial iterative descriptions of DP recurrences (see Figure 2.2). AUTOGEN works by analyzing the set of DP table locations accessed by the input serial algorithm when run on a DP table of suitably small size, and identifying a recursive fractal-like pattern in that set. For the class of DP problems handled by AUTOGEN the set of table locations accessed by the algorithm is independent of the data stored in the table. The class includes many well-known DP problems such as the parenthesis problem, pairwise sequence alignment and the gap problem as well as problems that are yet to be encountered. AUTOGEN effectively eliminates the need for human involvement in the design

Input	AUTOGEN	Output
serial iterative DP algorithm	analyze the trace of the input algorithm on a small DP table	parallel recursive DP algorithm: cache-efficient, cache-oblivious, cache-adaptive, energy-efficient
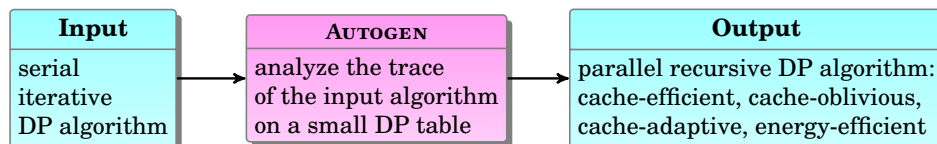
Figure 2.2: Input and output of AUTOGEN.

		$\mathcal{I}\text{-}\mathcal{DP}$		$\mathcal{R}\text{-}\mathcal{DP}$	
	Work	Serial cache	Span	Serial cache	Span
Problem	(T_1)	comp. (Q_1)	(T_∞)	comp. (Q_1)	(T_∞)
Parenthesis problem	$\Theta\left(n^3\right)$	$\Theta\left(n^3\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$
Floyd-Warshall's APSP 3-D	$\Theta\left(n^3\right)$	$\Theta\left(n^3/B\right)$	$\Theta\left(n\log n\right)$	$\Theta\left(n^3/B\right)$	$\mathcal{O}\left(n\log^2 n\right)$
Floyd-Warshall's APSP 2-D	$\Theta\left(n^3\right)$	$\Theta\left(n^3/B\right)$	$\Theta\left(n\log n\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n\log^2 n\right)$
LCS / Edit distance	$\Theta\left(n^2\right)$	$\Theta\left(n^2/B\right)$	$\Theta\left(n\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n^{\log 3}\right)$
Multi-instance Viterbi	$\Theta\left(n^3 t\right)$	$\Theta\left(n^3 t/B\right)$	$\Theta\left(nt\right)$	$\Theta\left(n^3 t/(B\sqrt{M})\right)$	$\Theta\left(nt\right)$
Gap problem	$\Theta\left(n^3\right)$	$\Theta\left(n^3\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$
Protein accordion folding	$\Theta\left(n^3\right)$	$\Theta\left(n^3/B\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n\log n\right)$
Spoken-word recognition	$\Theta\left(n^2\right)$	$\Theta\left(n^2/B\right)$	$\Theta\left(n\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n^{\log 3}\right)$
Function approximation	$\Theta\left(n^3\right)$	$\Theta\left(n^3/B\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$
Binomial coefficient	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n^2/B\right)$	$\Theta\left(n\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n^{\log 3}\right)$
Bitonic traveling salesman	$\Theta\left(n^2\right)$	$\Theta\left(n^2/B\right)$	$\Theta\left(n\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n\log n\right)$
Matrix multiplication	$\Theta\left(n^3\right)$	$\Theta\left(n^3/B\right)$	$\Theta\left(n\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n\right)$
Bubble sort	$\Theta\left(n^2\right)$	$\Theta\left(n^2/B\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n\right)$
Selection sort	$\Theta\left(n^2\right)$	$\Theta\left(n^2/B\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n\right)$
Insertion sort	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n^2/B\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n^{\log 3}/(BM^{\log 3 - 1})\right)$	$\mathcal{O}\left(n\right)$

Table 2.1: Work (T_1), serial cache complexity (Q_1), and span (T_∞) of $\mathcal{I}$-$\mathcal{DP}$ and $\mathcal{R}$-$\mathcal{DP}$ algorithms for several DP problems. Here, n = problem size, M = cache size, B = block size, and p = #cores. By T_p we denote running time on p processing cores. We assume that the DP table is too large to fit into the cache, and $M = \Omega\left(B^d\right)$ when $\Theta\left(n^d\right)$ is the size of the DP table. On p cores, the running time is $T_p = \mathcal{O}\left(T_1/p + T_\infty\right)$ and the parallel cache complexity is $Q_p = \mathcal{O}\left(Q_1 + p(M/B)T_\infty\right)$ with high probability when run under the randomized work-stealing scheduler on a parallel machine with private caches. The problems in the lower section are non-DP problems. For insertion sort, T_1 for $\mathcal{R}$-$\mathcal{DP}$ is $\mathcal{O}\left(n^{\log 3}\right)$.

of efficient cache-oblivious parallel algorithms for all present and future problems in that class.

Our contributions. Our major contributions are as follows:

(1) [*Algorithmic*.] We present AUTOGEN — an algorithm that for a wide class of DP problems automatically discovers highly efficient cache-oblivious parallel recursive divide-and-conquer algorithms from iterative descriptions of DP recurrences. AUTOGEN works by analyzing the DP table accesses (assumed to be independent of the data in the table) of an iterative algorithm on a table of small size, finding the dependencies among different orthants of the DP table recursively, and constructing a tree and directed acyclic graphs that represent a set of recursive functions corresponding to a parallel recursive divide-and-conquer algorithm. We prove the correctness of the algorithms generated by AUTOGEN.

(2) [*Experimental*.] We have implemented a prototype of AUTOGEN which we have used to autogenerate efficient cache-oblivious parallel recursive divide-and-conquer algorithms (pseudocodes) from naive serial iterative descriptions of several DP recurrences. We present experimental results showing that several autogenerated algorithms without any nontrivial hand-tuning significantly outperform parallel looping

codes in practice, and have more stable running times and energy profiles in a multi-programming environment compared to looping and tiling algorithms.

Related work. Systems for auto-generating fast iterative DP implementations (not algorithms) exist. The Bellman's GAP compiler [Giegerich and Sauthoff, 2011] converts declarative programs into optimized C++ code. A semi-automatic synthesizer [Pu et al., 2011] exists which uses contraint-solving to solve linear-time DP problems such as maximal substring matching, assembly-line optimization and the extended Euclid algorithm.

There are systems to automatically parallelize DP loops. EasyPDP [Tang et al., 2012] requires the user to select a directed acyclic graph (DAG) pattern for a DP problem from its DAG patterns library. New DAG patterns can be added to the library. EasyHPS [Du et al., 2013] uses the master-slave paradigm in which the master scheduler distributes computable sub-tasks among its slaves, which in turn distribute subsubtasks among slave threads. A pattern-based system exists [Liu and Schmidt, 2004] that uses generic programming techniques such as class templates to solve problems in bioinformatics. Parallelizing plugins [Reitzig, 2012] use diagonal frontier and row splitting to parallelize DP loops.

To the best of our knowledge, there has been no previous attempt to automate the process of discovering efficient cache-oblivious and cache-adaptive parallel recursive algorithms by analyzing the memory access patterns of naive serial iterative algorithms. The work that is most related to AUTOGEN, but completely different in many aspects is Pochoir [Tang et al., 2011]. Pochoir translates simple specifications of a stencil[5] into high-performing parallel code implementing an efficient cache-oblivious parallel algorithm. While Pochoir tailors the implementation of the same cache-oblivious algorithm (known as the trapezoidal decomposition algorithm) to different stencil computations, AUTOGEN discovers a (possibly) brand new efficient parallel cache-oblivious algorithm for every new DP problem it encounters.

Compiler technology for automatically converting iterative versions of matrix programs to serial recursive versions is described in [Ahmed and Pingali, 2000]. The approach relies on heavy machineries such as dependence analysis (based on integer programming) and polyhedral techniques. AUTOGEN, on the other hand, is a much simpler stand-alone algorithm that analyzes the data access pattern of a given naive (e.g., looping) serial DP code when run on a small example, and inductively generates a provably correct parallel recursive algorithm for solving the same DP.

2.2   The AUTOGEN algorithm

In this section, we present an algorithm called AUTOGEN that automatically converts an iterative algorithm to an efficient cache-oblivious parallel recursive divide-and-conquer algorithm for a wide variety of DP problems.

Definition 1 ($\mathcal{I}$-$\mathcal{DP}$ / $\mathcal{R}$-$\mathcal{DP}$ / *AUTOGEN*)**.** *Let $\mathcal{P}$ be a given DP problem specified through a DP recurrence. An iterative (or loop-based) algorithm for $\mathcal{P}$ is called $\mathcal{I}$-$\mathcal{DP}$. A cache-oblivious parallel recursive divide-and-conquer algorithm (if it exists) for $\mathcal{P}$ is called $\mathcal{R}$-$\mathcal{DP}$. The algorithm that automatically generates an $\mathcal{R}$-$\mathcal{DP}$ given any implementation for $\mathcal{P}$ is called AUTOGEN.*

---

[5]a stencil is a dynamic program in which the value of a spatial cell at any time step depends on its neighboring cells in a constant number of previous time steps

The input to AUTOGEN can be any implementation ($\mathcal{I}$-$\mathcal{DP}$, tiled $\mathcal{I}$-$\mathcal{DP}$, $\mathcal{R}$-$\mathcal{DP}$, or some other implementation) of the given DP recurrence. As the simplest implementation of a DP recurrence is an iterative algorithm or $\mathcal{I}$-$\mathcal{DP}$, in this chapter we assume $\mathcal{I}$-$\mathcal{DP}$s as inputs to AUTOGEN.

Algorithm. The four main steps of AUTOGEN are:

(1) [*Cell-set generation.*] A cell-set is generated from a sample run of the given $\mathcal{I}$-$\mathcal{DP}$ on a problem of small size. See Section 2.2.1.

(2) [*Algorithm-tree construction.*] An algorithm-tree is constructed from the cell-set in which each node represents a subset of the cell-set and follows certain rules. See Section 2.2.2.

(3) [*Algorithm-tree labeling.*] The nodes of the tree are labeled with function names and these labels represent a set of recursive divide-and-conquer functions in an $\mathcal{R}$-$\mathcal{DP}$ under Assumption 1. See Section 2.2.3.

(4) [*Algorithm-DAG Construction.*] For every unique function, we construct a directed acyclic graph (DAG) that shows both the order in which the functions are to be executed and the parallelism involved. See Section 2.2.4.

We make the following assumption for an $\mathcal{R}$-$\mathcal{DP}$.

Assumption 1 (*Number of functions*)**.** *The number of distinct recursive functions in an $\mathcal{R}$-$\mathcal{DP}$ is upper bounded by a constant.*

Example. AUTOGEN works for arbitrary d-D ($d \geq 1$) DP problems under the assumption that each dimension of the DP table is of the same length and is a power of 2. For simplicity of exposition, we explain AUTOGEN by applying it on an $\mathcal{I}$-$\mathcal{DP}$ for the parenthesis problem, which updates a 2-D DP table.

In the parenthesis problem [Galil and Park, 1994], we are given a sequence of characters $S = s_1 \cdots s_n$ and we would like to compute the minimum cost of parenthesizing S. This problem represents a class of problems such as optimal matrix chain multiplication, string parsing for context-free grammar (e.g., CYK algorithm), RNA secondary structure prediction, optimal natural join of database tables (e.g., Selinger algorithm), construction of optimal binary search trees, optimal polygon triangulation, maximum perimeter inscribed polygon, and offline job scheduling minimizing total flow time of jobs. Let $C[i, j]$ denote the minimum cost of parenthesizing $s_i \cdots s_j$. Then, the DP table $C[0 : n, 0 : n]$ is filled up using the following recurrence.

$$C[i,j] = \begin{cases} \infty & \text{if } 0 \leq i = j \leq n, \\ v_j & \text{if } 0 \leq i = j - 1 < n, \\ \min_{i \leq k \leq j} \left\{ (C[i,k] + C[k,j]) + w(i,k,j) \right\} & \text{if } 0 \leq i < j - 1 < n; \end{cases} \tag{2.2}$$

where v_j's are given, and function $w()$ can be computed without additional memory accesses.

Based on the recurrence relation above, we write the serial $\mathcal{I}$-$\mathcal{DP}$ for the parenthesis problem as illustrated in Figure 2.3. In the rest of the section, we show how to convert this serial $\mathcal{I}$-$\mathcal{DP}$ into an $\mathcal{R}$-$\mathcal{DP}$ using AUTOGEN.

```
LOOP-PARENTHESIS(C, n)
  1. for i ← n − 1 to 0 do
  2.    for j ← i + 2 to n do
  3.       for k ← i to j do
  4.          C[i, j] ← min (C[i, j], C[i, k] + C[k, j] + w(i, k, j))
```

```
PAR-LOOP-PARENTHESIS(C, n)
  1. for t ← 2 to n − 1 do
  2.    parallel for i ← 1 to n − t do
  3.       j ← t + i
  4.       for k ← i + 1 to j
  5.          C[i, j] ← min (C[i, j], C[i, k] + C[k, j] + w(i, k, j))
```

Figure 2.3: Serial and parallel $\mathcal{I}$-$\mathcal{DP}$ for the parenthesis problem (Recurrence 3.1). Initialization is not shown.

2.2.1   Cell-set generation

In this step, a cell-set is generated for a given DP problem. We define a few terms that will be used to define a cell-set.

A *cell* is a grid point in a DP table identified by its d-D coordinates, e.g., $(5, 7)$ represents a cell in a 2-D DP table. A cell is *fully updated* when the value of the cell is final and does not change in the future, otherwise it is *partially updated*. A cell x is *fully dependent* on a cell y when x is partially or fully updated with the fully updated value of y. A cell x is said to be *partially dependent* on a cell y when x is to be updated from the combined values of y and some other cells. We say that two cells x and y in a DP table are *adjacent* iff they differ only in one coordinate value and that difference is exactly 1. A *path* from cell x to cell y in a DP table is a sequence of cells starting from x and ending at y such that every two consecutive cells in the sequence are adjacent.

Definition 2 (*Region*). *For simplicity, a region is defined for 2-D. Let a 2-D DP table be represented as $C[0 : n − 1][0 : n − 1]$. Then, a region at a level $i \in [0, \log n]$ is defined as a $\frac{n}{2^i} \times \frac{n}{2^i}$ square block in C with its top-left corner at $(j \cdot 2^i, k \cdot 2^i)$, where $j, k \in [0, 2^i − 1]$.*

A d-D DP table C is called a level-0 region. The orthants of identical dimensions of the level-0 region are called level-1 regions. Generalizing, the orthants of level-i regions are called level-$(i+1)$ regions. In d-D, a region at a level i can be divided into 2^d regions at level $i + 1$, where $i \in [0, \log n)$.

Definition 3 (*Depends, dependencies*). *A cell x depends on another cell y, represented by $x \rightsquigarrow y$, if cell x is updated using information from cell y, e.g., $(5, 13) \rightsquigarrow (4, 12)$. A region X depends on another region Y, represented by $X \rightsquigarrow Y$, if $\exists$ cells $x \in X, y \in Y$ such that $x \rightsquigarrow y$.*

The set of all dependency relations $x \rightsquigarrow y$ of a cell x is called the dependencies of x. Similarly, the set of all dependency relations $X \rightsquigarrow Y$ of a region X is called the dependencies of X.

Definition 4 (*Update relation, cell-/region-tuple*). *We assume that each iteration of the innermost loop of the given $\mathcal{I}$-$\mathcal{DP}$ performs the following update:*

$$C[x] \leftarrow f(C^1[y_1], C^2[y_2], \ldots, C^s[y_s]) \; or \tag{2.3}$$

$$C[x] \leftarrow C[x] \oplus f(C^1[y_1], C^2[y_2], \ldots, C^s[y_s]), \tag{2.4}$$

where $s \geq 1$; x is a cell of table C; y_i is a cell of table C^i; $\oplus$ is an associative operator (such as min, max, $+$, $\times$); and f is an arbitrary function.

We call the tuple $\langle C[x], C^1[y_1], \ldots, C^s[y_s] \rangle$ a cell-tuple. Let $C[X], C^1[Y_1], \ldots, C^s[Y_s]$ be regions such that $x \in X$, and $y_i \in Y_i$. Then we call the tuple $\langle C[X], C^1[Y_1], \ldots, C^s[Y_s] \rangle$ a region-tuple. In simple words, a cell-tuple (resp. region-tuple) gives information of which cell (resp. region) is being written by reading from which cells (resp. regions). The size of a cell-/region-tuple is $1 + s$.

Definition 5 (*Cell-set*). *For any given $\mathcal{I}$-$\mathcal{DP}$, the set of all cell-tuples for all cells in its DP table is called a cell-set.*

Given an $\mathcal{I}$-$\mathcal{DP}$, we modify it such that instead of computing its DP table, it generates the cell-set for a problem of suitably small size. For a problem of suitably small size and instead of computing the DP table we simply generate the cell-set. For example, for the parenthesis problem, we choose $n = 64$ and generate the cell-set $\{\langle C(i, j), C(i, k), C(k, j) \rangle\}$, where C is the DP table, $0 \leq i < j - 1 < n$, and $i \leq k \leq j$. A method of choosing a good small problem size is explained in Section 2.3.1.

2.2.2 Algorithm-tree construction

In this step, we create an algorithm-tree using the cell-set generated in Section 2.2.1.

Definition 6 (*Algorithm-tree*). *Given an $\mathcal{I}$-$\mathcal{DP}$, a tree representing a hierarchy of recursive divide-and-conquer functions which is used to find a potential $\mathcal{R}$-$\mathcal{DP}$ is called an algorithm-tree.*

An algorithm-tree is a collection of nodes placed at different levels. The way we construct level-i nodes in an algorithm-tree is by analyzing the dependencies between level-i regions using the cell-set. Every node in the algorithm-tree represents a subset of the cell-set satisfying certain region-tuple dependencies. Suppose the algorithm writes into DP table C, and reads from tables $C^1, \ldots, C^s$ (they can be same as C). The algorithm-tree is constructed as follows.

At level 0, as per Definition 2, the only regions possible are the entire tables $C, C^1, \ldots, C^s$. We analyze the cell-tuples of the cell-set to identify the region-tuples at this level. As all the write cells belong to C and all the read cells belong to $C^1, \ldots, C^s$, the only possible region-tuple is $\langle C, C^1, \ldots, C^s \rangle$. We create a node for this region-tuple and it forms the root node of the algorithm-tree. It represents the entire cell-set. For example, for parenthesis problem, as all the write and read cells belong to the same DP table C, the root node will be $\{\langle C, C, C \rangle\}$.

The level-1 nodes are found by distributing the cell-tuples belonging to the root node among region-tuples of level 1. The level-1 regions are obtained by dividing the DP table C into four quadrants: C_{11} (top-left), C_{12} (top-right), C_{21} (bottom-left), and C_{22} (bottom-right). Similarly, each C^i for $i \in [1, s]$ is divided into four quadrants: $C^i_{11}, C^i_{12}, C^i_{21}$, and C^i_{22}. The cell-tuples of the cell-set are analyzed to find all possible nonempty region-tuples at level 1. For example, if a cell-tuple $\langle c, c_1, \ldots, c_s \rangle$ is found to have $c \in C_k$ and $c_i \in C^i_{k_i}$ for $i \in [1, s]$ and $k, k_i \in \{11, 12, 21, 22\}$, then we say that $\langle c, c_1, \ldots, c_s \rangle$ belongs to region-tuple $\langle C_k, C^1_{k_1}, \ldots, C^s_{k_s} \rangle$. Different problems will have different nonempty region-tuples depending on their cell dependencies. For the parenthesis problem, there are four nonempty level-1 region-tuples and they are $\langle C_{11}, C_{11}, C_{11} \rangle$, $\langle C_{22}, C_{22}, C_{22} \rangle$, $\langle C_{12}, C_{11}, C_{12} \rangle$, and $\langle C_{12}, C_{12}, C_{22} \rangle$.

(a) Before applying Rule 1. (b) After applying Rule 1.
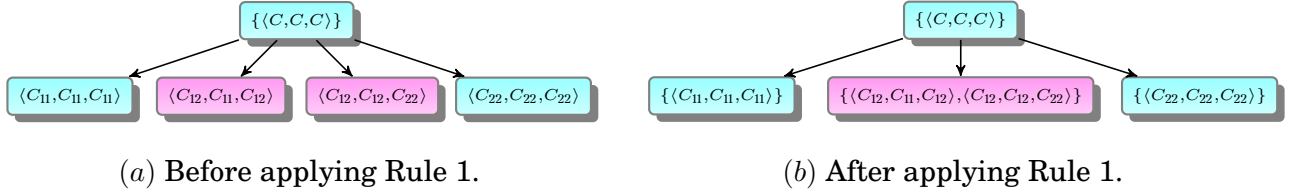
Figure 2.4: First two levels of the algorithm-tree for parenthesis problem before and after applying Rule 1.

Sometimes two or more region-tuples are combined into a node. The region-tuples that write to and read from the same region depend on each other for the complete update of the write region. Hence, two or more region-tuples are combined into a node if they read from and write to the same region. The following rule guarantees that such region-tuples are processed together to to satisfy the read-write constraints as described in Property 2 and avoid incorrect results.

> **Rule 1** (*Combine region-tuples into node*). *Two region-tuples at the same level of an algorithm-tree denoted by $\langle W, R_1, \ldots, R_s \rangle$ and $\langle W, R'_1, \ldots, R'_s \rangle$ are combined into a single node provided $\exists i, j \in [1, s]$ such that $R_i = W = R'_j$.*

For example, for the parenthesis problem, at level 1, the two region-tuples $\langle C_{12}, C_{11}, C_{12} \rangle$ and $\langle C_{12}, C_{12}, C_{22} \rangle$ are combined into a single node $\{\langle C_{12}, C_{11}, C_{12} \rangle, \langle C_{12}, C_{12}, C_{22} \rangle\}$ (see Figure 2.4(b)). The other two nodes are $\{\langle C_{11}, C_{11}, C_{11} \rangle\}$ and $\{\langle C_{22}, C_{22}, C_{22} \rangle\}$. The three nodes represent three mutually disjoint subsets of the cell-set and have different region-tuple dependencies. Once we find all level 1 nodes, we recursively follow the same strategy to find the nodes of levels ≥ 2 partitioning the subsets of the cell-set further depending on their region-tuple dependencies.

In summary, we follow the following four step process to find the child nodes of a node $\mathcal{F}$ present at level k:
  (1) Find the regions at level $k + 1$ that belong to the regions of $\mathcal{F}$.
  (2) Analyze the cell-tuples present in $\mathcal{F}$ and use the regions found from the previous step to find the region-tuples at level $k + 1$.
  (3) Combine region-tuples into nodes following Rule 1.
  (4) Split the cell-tuples of $\mathcal{F}$ into different nodes as per their region-tuple dependencies.

2.2.3 Algorithm-tree labeling

In this step, the nodes of the algorithm-tree are labeled with function names. Two nodes are given the same function name when the following two are the same:
  (1) Output fingerprint, to denote the same kind of child nodes (subregion-tuples).
  (2) Input fingerprint, to denote the same kind of node (region-tuples).
  The definitions and the rule follow.

Definition 7 (*Output fingerprint*). *The output fingerprint of a node is the set of all output fingerprints of its region-tuples. The output fingerprint of a region-tuple is defined as the set of all its subregion-tuples present in the child nodes. A subregion-tuple of a region-tuple $\langle W, R_1, \ldots, R_s \rangle$ is defined as a tuple $\langle w, r_1, \ldots, r_s \rangle$ where $w, r_i \in \{11, 12, 21, 22\}$ such that $\langle W_w, R_{r_1}, \ldots, R_{r_s} \rangle$ is a region-tuple, where $\forall i \in [1, s]$.*

Definition 8 (*Input fingerprint*)**.** *The input fingerprint of a node is the set of all input fingerprints of its region-tuples. The input fingerprint of a region-tuple $\langle X_1, \ldots, X_{1+s} \rangle$ is a tuple $\langle p_1, \ldots, p_{1+s} \rangle$, where $\forall i \in [1, 1+s]$, p_i is the smallest index $j \in [1, i]$ such that $X_j = X_i$.*

In practice, for most algorithms we have seen, it is enough if we simply consider the output fingerprints alone for giving the same function name. Having input fingerprint increases the number of functions but it makes the rule stronger.

> **Rule 2** (*Same function name*)**.** *Two nodes in an algorithm tree are given the same function name provided the two nodes have the same output and input fingerprints.*

For example, in the parenthesis problem, nodes $\{\langle C_{12}, C_{11}, C_{12} \rangle, \langle C_{12}, C_{12}, C_{22} \rangle\}$ and $\{\langle C_{1221}, C_{1122}, C_{1221} \rangle, \langle C_{1221}, C_{1221}, C_{2211} \rangle\}$ satisfy Rule 2, and so get the same function name.

We assume that an algorithm-tree satisfies the one-way function generation property as defined below. We will make use of the property in later sections.

> **Property 1** (*One-way function generation*)**.** *In an algorithm-tree, if a node labeled $\mathcal{F}$ is an ancestor of a node labeled $\mathcal{G}$, then there exists no node labeled $\mathcal{G}$ that is an ancestor of a node labeled $\mathcal{F}$. We call this property the one-way function generation property.*

The height of an algorithm-tree is determined by the threshold level as defined below.

Definition 9 (*Threshold level*)**.** *In an algorithm-tree, at least one new function is invoked at every level starting from level 0 till a certain level l, beyond which no new functions are invoked. We call l the threshold level and it is upper bounded by a constant as per Assumption 1.*

We stop the process of labeling at a level l where no new functions are invoked. This means, all functions present in levels $[1, l]$ call themselves.

The two steps: algorithm-tree construction and algorithm-tree labeling can be combined into a single step, in which case, it is sufficient if we build the algorithm-tree and label it till level l. In this paper, we have separated them into two different steps for pedagogical reasons.

Figure 2.5(a) illustrates a small part of the algorithm-tree for the parenthesis problem with three functions $\mathcal{A}, \mathcal{B},$ and $\mathcal{C}$. We see that $\mathcal{A}$ calls $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{B}$ calls $\mathcal{B}$ and $\mathcal{C}$ and $\mathcal{C}$
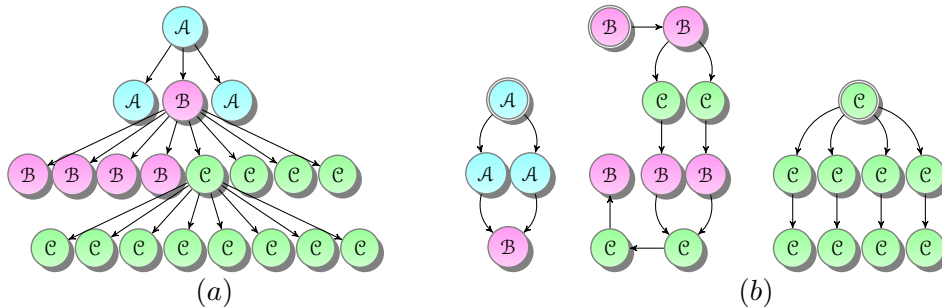
Figure 2.5: (a) A small part of the algorithm-tree for the parenthesis problem. Due to space constraints, only three nodes are expanded. (b) DAGs for the three functions $\mathcal{A}, \mathcal{B},$ and $\mathcal{C}$ in the parenthesis problem showing the order of execution of functions.

calls $\mathcal{C}$ only. The root node is given the function name $\mathcal{A}$. The function $\mathcal{A}$ at level 1 calls $\{\mathcal{A}, \mathcal{A}, \mathcal{B}\}$. At level 2, a new function $\mathcal{B}$ is generated. After expansion, $\mathcal{B}$ at level 2 calls $\{\mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}\}$. A new function $\mathcal{C}$ is generated at level 3. On expanding, the function $\mathcal{C}$ calls itself $\{\mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}\}$. As no new functions are generated at level 4, we stop. The three recursive recursive functions have different functionalities due to the relationship between the read and write regions.

In the parenthesis problem, the smallest level at which no new functions are called is $l = 4$. The three functions generated in the levels $[1, 3]$ are represented in the form

$$\mathcal{F} \mapsto \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_{m_{\mathcal{F}}}\} \hspace{3cm} [l = k],$$

where the function (or node) $\mathcal{F}$, which is called for the first time at level k, in turn calls the functions $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_{m_{\mathcal{F}}}$.

$$\mathcal{A} \mapsto \{\mathcal{A}, \mathcal{A}, \mathcal{B}\} \hspace{3cm} [l = 1]$$
$$\mathcal{B} \mapsto \{\mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}\} \hspace{2cm} [l = 2]$$
$$\mathcal{C} \mapsto \{\mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}\} \hspace{2cm} [l = 3]$$

2.2.4 Algorithm-DAG construction

In this step, we construct a directed acyclic graph (DAG) for every function. An algorithm-tree does not give information on (a) the sequence in which a function calls other functions, and (b) the parallelism involved in executing the functions. The DAGs address these two issues using the rules that follow.

We define a few terms before listing the rules.

Definition 10 ($W()$, $R()$). *Given a function $\mathcal{F}$, we define $\boldsymbol{W}(\mathcal{F})$ and $\boldsymbol{R}(\mathcal{F})$ as the write region and the set of read regions of the region-tuples in $\mathcal{F}$, respectively. For a region-tuple $T = \langle W, R_1, \ldots, R_s \rangle$, we define $\boldsymbol{W}(T) = W$ and $\boldsymbol{R}(T) = \{R_1, \ldots, R_s\}$.*

Definition 11 (*Flexibility*). *A region-tuple T is called flexible provided $\boldsymbol{W}(T) \notin \boldsymbol{R}(T)$, i.e., the region-tuple does not write to a region it reads from. A function is called flexible if all of its region-tuples are flexible.*

Definition 12 (*Function ordering*). *If a function $\mathcal{F}$ calls two functions $\mathcal{F}_1$ and $\mathcal{F}_2$, then the ordering between $\mathcal{F}_1$ and $\mathcal{F}_2$ can be any of the following three: (a) $\mathcal{F}_1 \to \mathcal{F}_2$ i.e., $\mathcal{F}_1$ is called before $\mathcal{F}_2$, (b) $\mathcal{F}_1 \leftrightarrow \mathcal{F}_2$ i.e., either $\mathcal{F}_1 \to \mathcal{F}_2$ or $\mathcal{F}_2 \to \mathcal{F}_1$, and (c) $\mathcal{F}_1 \| \mathcal{F}_2$ i.e., $\mathcal{F}_1$ can be run in parallel with $\mathcal{F}_2$.*

If a function $\mathcal{F}$ calls two functions $\mathcal{F}_1$ and $\mathcal{F}_2$, then the order in which $\mathcal{F}_1$ and $\mathcal{F}_2$ are executed is determined by the following rules.

> **Rule 3** (*Different write regions*). *If $\boldsymbol{W}(\mathcal{F}_1) \neq \boldsymbol{W}(\mathcal{F}_2)$ and $\boldsymbol{W}(\mathcal{F}_1) \in \boldsymbol{R}(\mathcal{F}_2)$, then $\mathcal{F}_1 \to \mathcal{F}_2$.*

> **Rule 4** (*Same write region*). *If $\boldsymbol{W}(\mathcal{F}_1) = \boldsymbol{W}(\mathcal{F}_2)$, $\mathcal{F}_1$ is flexible but $\mathcal{F}_2$ is not, then $\mathcal{F}_1 \to \mathcal{F}_2$.*

> **Rule 5** (*Same write region*)**.** *If* $\mathbf{W}(\mathcal{F}_1) = \mathbf{W}(\mathcal{F}_2)$ *and both* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *are flexible, then* $\mathcal{F}_1 \leftrightarrow \mathcal{F}_2$.

> **Rule 6** (*Different write regions*)**.** *If* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *satisfy none of the rules 3, 4 and 5, then* $\mathcal{F}_1 || \mathcal{F}_2$.

Based on the rules above, we construct a DAG for every function. Let a node in a DAG be called a *dnode*. We modify the constructed DAGs by deleting unwanted edges from them as per the following rules. The set of all modified DAGs for all functions represents an $\mathcal{R}$-$\mathcal{DP}$ for the given $\mathcal{I}$-$\mathcal{DP}$.

> **Rule 7** (*DAG*)**.** *In the algorithm-tree, if a function* $\mathcal{F}$ *calls two functions* $\mathcal{F}_1$ *and* $\mathcal{F}_2$, *then in the DAG of* $\mathcal{F}$, *we create dnodes* d_1 *and* d_2 *corresponding to* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *(if they are not already present), respectively. If* $\mathcal{F}_1 \rightarrow \mathcal{F}_2$, *then we add a directed edge from* d_1 *to* d_2. *On the other hand, if* $\mathcal{F}_1 \leftrightarrow \mathcal{F}_2$, *then we add a directed edge either from* d_1 *to* d_2 *or from* d_2 *to* d_1, *but not both.*

> **Rule 8** (*Algorithm-DAG*)**.** *Let* d_1, d_2 *and* d_3 *be three dnodes in a DAG. If there are directed edges from* d_1 *to* d_2 *and from* d_2 *to* d_3, *then mark the directed edge from* d_1 *to* d_3. *Mark all such edges in the DAG until no more edges can be marked. Finally, delete all marked edges from the DAG.*

The redundant functions can be removed using extra space using the following rule. After the reduction, the functions can be made to satisfy Property 1.

> **Rule 9** (*Function reduction*)**.** *Let the size of the DP table be* n^d. *Any function* $\mathcal{F}$ *in the autogenerated* $\mathcal{R}$-$\mathcal{DP}$ *can be removed using an additional space of* $\mathcal{O}\left(n^{d-1}\right)$, *if the cells in* $\mathbf{W}(\mathcal{F})$ *depends on* $\mathcal{O}\left(n^{d-1}\right)$ *cells of* $\mathbf{R}(\mathcal{F})$.

Thus, starting from a simple $\mathcal{I}$-$\mathcal{DP}$, we can automatically generate an $\mathcal{R}$-$\mathcal{DP}$ using Autogen. As an example, for the parenthesis problem, an $\mathcal{R}$-$\mathcal{DP}$ is given in Figure 2.6 and illustrated in Figure 2.5 and 2.7. The algorithm includes three functions $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, each of which is represented by a DAG. Function $\mathcal{B}$ contains two region-tuples $\langle X, U, X \rangle$ and $\langle X, X, V \rangle$, but because of space constraints we write them as $\langle X, U, V \rangle$.

2.3 Correctness of Autogen

In this section, we first describe a method of choosing a good small problem size for the sample run, then define a class of DP problems called Fractal-dp on which Autogen works, and finally provide a proof of correctness for Autogen.

2.3.1 Threshold problem size

Given a DP problem, there exists a minimum value of n (assumed to be a power of 2) for which one can build an algorithm-tree with the number of levels more than the threshold level for that problem. Suppose the $\mathcal{R}$-$\mathcal{DP}$ algorithm of the problem includes at least m

$\mathcal{A}_{par}(\langle X, X, X \rangle)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}par}(\langle X, X, X \rangle)$
2. **else**
3.   **parallel:** $\mathcal{A}_{par}(\langle X_{11}, X_{11}, X_{11} \rangle)$, $\mathcal{A}_{par}(\langle X_{22}, X_{22}, X_{22} \rangle)$
4. $\mathcal{B}_{par}(\langle X_{12}, X_{11}, X_{22} \rangle)$

$\mathcal{B}_{par}(\langle X, U, V \rangle)$

1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}par}(\langle X, U, V \rangle)$
2. **else**
3. $\mathcal{B}_{par}(\langle X_{21}, U_{22}, V_{11} \rangle)$
4.   **parallel:** $\mathcal{C}_{par}(\langle X_{11}, U_{12}, V_{21} \rangle)$, $\mathcal{C}_{par}(\langle X_{22}, X_{21}, V_{12} \rangle)$
5.   **parallel:** $\mathcal{B}_{par}(\langle X_{11}, U_{11}, V_{11} \rangle)$, $\mathcal{B}_{par}(\langle X_{22}, X_{22}, V_{22} \rangle)$
6. $\mathcal{C}_{par}(\langle X_{12}, U_{12}, X_{22} \rangle)$
7. $\mathcal{C}_{par}(\langle X_{12}, X_{11}, V_{12} \rangle)$
8. $\mathcal{B}_{par}(\langle X_{12}, U_{11}, V_{22} \rangle)$

$\mathcal{C}_{par}(\langle X, U, V \rangle)$

1. **if** X is a small matrix **then** $\mathcal{C}_{loop\text{-}par}(\langle X, U, V \rangle)$
2. **else**
3.   **parallel:** $\mathcal{C}_{par}(\langle X_{11}, U_{11}, V_{11} \rangle)$, $\mathcal{C}_{par}(\langle X_{12}, U_{11}, V_{12} \rangle)$, $\mathcal{C}_{par}(\langle X_{21}, U_{21}, V_{11} \rangle)$, $\mathcal{C}_{par}(\langle X_{22}, U_{21}, V_{12} \rangle)$
4.   **parallel:** $\mathcal{C}_{par}(\langle X_{11}, U_{12}, V_{21} \rangle)$, $\mathcal{C}_{par}(\langle X_{12}, U_{12}, V_{22} \rangle)$, $\mathcal{C}_{par}(\langle X_{21}, U_{22}, V_{21} \rangle)$, $\mathcal{C}_{par}(\langle X_{22}, U_{22}, V_{22} \rangle)$

Figure 2.6: An $\mathcal{R}\text{-}\mathcal{DP}$ algorithm for the parenthesis problem. Initial call to the algorithm is $\mathcal{A}_{par}(\langle C, C, C \rangle)$, where C is the DP table.

distinct functions. From Definition 9, the threshold level is upper bounded by m. Therefore, we should set the sample problem size to $n = 2^{m+k}$, where k is a problem-specific natural number.

Empirically, we have found that the number of functions required to represent an $\mathcal{R}\text{-}\mathcal{DP}$ algorithm for most problems is at most 4. Considering $m = 4$ and $k = 2$, we get $n = 64$. If we are unable to generate all the functions, we increase the value of k and build the algorithm-tree again. We continue this process until we generate functions that call no new functions. Such a threshold value of n is called the threshold problem size for the given DP problem.

2.3.2 The FRACTAL-DP class

In this section, we define a class of iterative DP algorithms called FRACTAL-DP. If an $\mathcal{I}\text{-}\mathcal{DP}$ belongs to the FRACTAL-DP class, then AUTOGEN can be applied on the $\mathcal{I}\text{-}\mathcal{DP}$ to get a correct $\mathcal{R}\text{-}\mathcal{DP}$.

We define two properties on an $\mathcal{I}\text{-}\mathcal{DP}$: (a) One-way sweep property, which means that cells of the given DP table are finalized like a wavefront and the wavefront travels through the entire table only once and that too, in a single direction; and (b) Fractal property, which means that cell dependencies in the given DP table display self-similar patterns.

> **Property 2** (*One-way sweep*). *An $\mathcal{I}\text{-}\mathcal{DP}$ for a DP table C is said to satisfy the one-way sweep property if the following holds: $\forall$ cells $x, y \in C$, if x depends on y, then y is fully updated before x reads from y.*

This rule is called one-way sweep because the wavefront of computations (order of cells getting fully updated) never comes back to a cell again once it is both written as well as
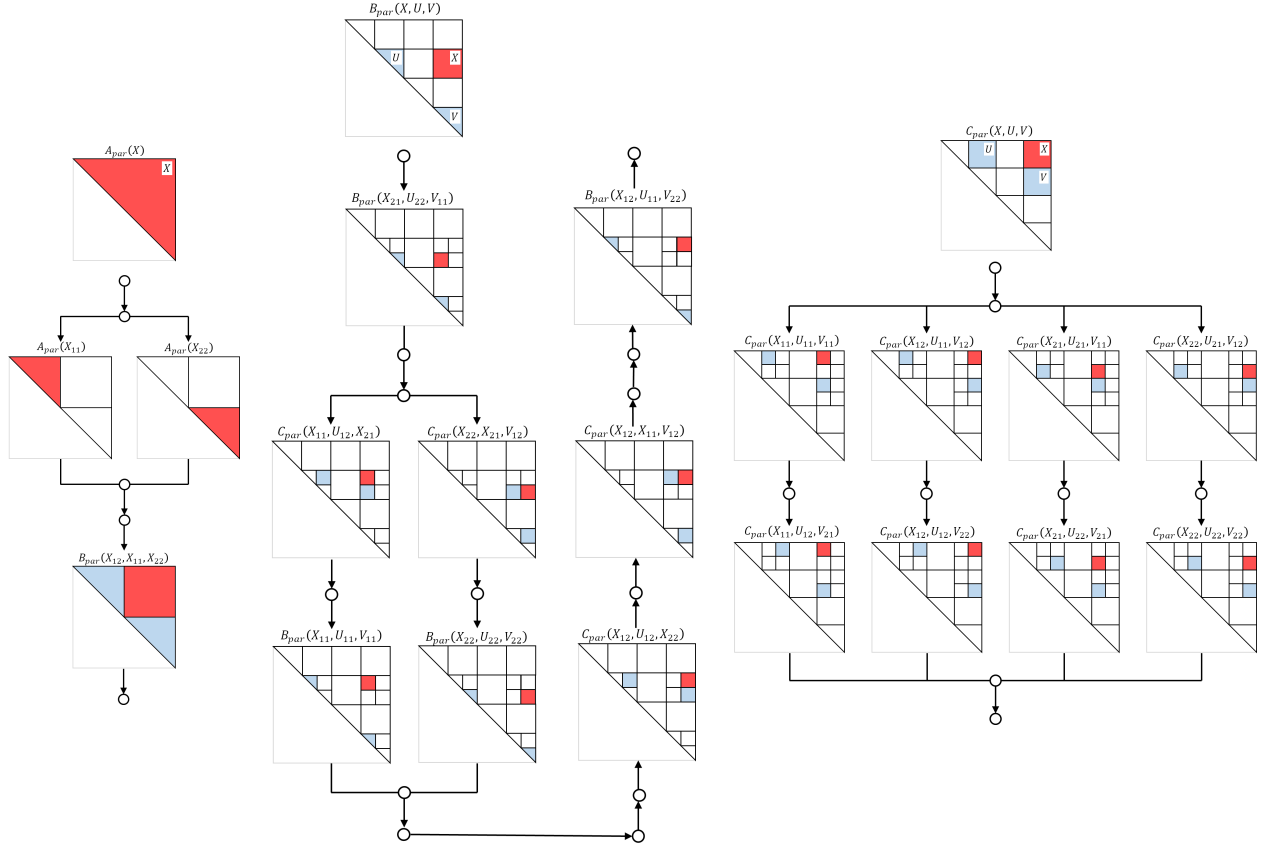
Figure 2.7: An $\mathcal{R}$-$\mathcal{DP}$ i.e., cache-oblivious parallel recursive divide-and-conquer algorithm for the parenthesis problem with functions $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. Entries in the dark red blocks are updated using data from light blue blocks. The initial call to the algorithm is $\mathcal{A}_{par}(\langle C, C, C \rangle)$, where C is the DP table. The function $\mathcal{B}_{par}$ contains two region tuples denoted by $\mathcal{B}_{par}(\langle X, U, X \rangle, \langle X, X, V \rangle)$, but because of space constraints we write it as $\mathcal{B}_{par}(\langle X, U, V \rangle)$.

read from (in the same order). AUTOGEN cannot be used on a loop-based DP implementation if it does not satisfy Property 2.

Define $G = \{$angle in radians made by line $xy \mid \forall x, y \in C$ and x depends on $y\}$. The set G can be rewritten as $G = \langle \theta_1, \theta_2, \ldots, \theta_k \rangle$ such that $\theta_1 < \theta_2 < \cdots < \theta_k$. We define δ as

$$\delta = 2\pi - \max \left(\max_{i \in [1, k-1]} (\theta_{i+1} - \theta_i), 2\pi - \theta_k + \theta_1 \right)$$

If $\delta \geq \pi$, then the $\mathcal{I}$-$\mathcal{DP}$ violates the one-way sweep property.

> **Property 3** (*Fractal property*)**.** *An $\mathcal{I}$-$\mathcal{DP}$ satisfies the fractal property if the following holds. Let S_n and S_{2n} be the cell-sets of the $\mathcal{I}$-$\mathcal{DP}$ for DP tables $[0..n-1]^d$ and $[0..2n-1]^d$, respectively, where $n \geq 2^k$ (see Section 2.3.1). Generate the cell-set S'_n from S_{2n} by replacing every coordinate value j with $\lfloor j/2 \rfloor$ and then retaining only the distinct tuples. Then, $S_n = S'_n$.*

Definition 13 (*FRACTAL-DP class*)**.** *An $\mathcal{I}$-$\mathcal{DP}$ is said to be in the FRACTAL-DP class if the following conditions hold:* (a) *the $\mathcal{I}$-$\mathcal{DP}$ satisfies the one-way sweep property (Property 2),* (b) *it satisfies the fractal property (Property 3), and* (c) *the number of input parameters given to its update function (i.e., s in Definition 4) is upper bounded by a constant.*

FRACTAL-DP is a wide class of dynamic programs. In Cormen et al.'s book [Cormen et al., 2009], 12 out of 16 (i.e., 75%) DP problems belong to FRACTAL-DP class. In Lew and Mauch's book [Lew and Mauch, 2006], approx. 29 out of 47 (i.e., 61%) DP problems belong to FRACTAL-DP class.

The problems that do not belong to FRACTAL-DP class are Viterbi algorithm (if the emission matrix is included), knapsack problem, sieve of Eratosthenes, some graph DP problems, some tree DP problems, some DAG DP problems and other problems where the update equation do not follow fractal patterns. E.g: in Viterbi algorithm a cell $P[i,j]$ depends on $B[i, Y[j]]$, where $Y[j]$ can take different values depending on the problem instance, in 0/1 knapsack problem, a cell $K[i,j]$ depends on $K[i-1, j-W[i]]$, in sieve of Eratosthenes, a cell $P[i]$ depends on $P[j]$, where j is a factor of i.

There might be some non-FRACTAL-DP problems for which AUTOGEN works. We are not aware of that more general class for which AUTOGEN works. Right now, the proof of correctness works for FRACTAL-DP problems. It does not say whether AUTOGEN fails for all non-FRACTAL-DP problems.

We prove in the next section that If an $\mathcal{I}$-$\mathcal{DP}$ belongs to the FRACTAL-DP class, then AUTOGEN can be applied on the $\mathcal{I}$-$\mathcal{DP}$ to get a functionally equivalent $\mathcal{R}$-$\mathcal{DP}$ as defined below.

Definition 14 (*Functional equivalence*). *An $\mathcal{R}$-$\mathcal{DP}$ $\mathcal{R}$ is said to be functionally equivalent to an $\mathcal{I}$-$\mathcal{DP}$ $\mathcal{I}$ provided for every input legal to $\mathcal{I}$, both $\mathcal{R}$ and $\mathcal{I}$ produce matching output.*

2.3.3 Proof of correctness

In this section, we present a proof of correctness for AUTOGEN.

> **Theorem 1** (**Correctness of AUTOGEN**). *Given an $\mathcal{I}$-$\mathcal{DP}$ from the FRACTAL-DP class as input, AUTOGEN generates an $\mathcal{R}$-$\mathcal{DP}$ that is functionally equivalent to the given $\mathcal{I}$-$\mathcal{DP}$.*

Proof. Let the $\mathcal{I}$-$\mathcal{DP}$ and $\mathcal{R}$-$\mathcal{DP}$ algorithms for a problem $\mathcal{P}$ be denoted by $\mathcal{I}$ and $\mathcal{R}$, respectively. We use mathematical induction to prove the correctness of AUTOGEN in d-D, assuming d to be a constant. First, we prove the correctness for the threshold problem size i.e., $n = 2^q$ for some $q \in \mathbb{N}$ (see Section 2.3.1) and then show that if the algorithm is correct for $n = 2^r$, for any $r \geq q$ then it is also correct for $n = 2^{r+1}$. The most complicated part of the proof is the one where we show that the generated $\mathcal{R}$-$\mathcal{DP}$ never violates the one-way sweep property (Property 2) which requires a case-by-case analysis of the order of updates of a pair of cell-tuples.

Basis. To prove that AUTOGEN is correct for $n = 2^q$, we have to show the following three:
  (a) Number of nodes in the algorithm-tree is $\mathcal{O}(1)$
  (b) Both $\mathcal{I}$ and $\mathcal{R}$ apply the same set of cell updates
  (c) $\mathcal{R}$ never violates the one-way sweep property (Property 2).

(a) *The size of the algorithm-tree is $\mathcal{O}(1)$.*
A node is a set of one or more region-tuples (see Rule 1). As per Rule 2, two nodes with the same input and output fingerprints are given the same function names. The maximum number of possible functions is upper bounded by the product of the maximum number of

possible nodes at a level $\left(\leq 2^d((2^d-1)^s+1)\right)$ and the maximum number of children a node can have $\left(\leq 2^{2^d((2^d-1)^s+1)}\right)$. This is because,

$$\text{Max \#region-tuples at a level} \leq 2^{d(s+1)}$$
$$\text{Max \#nodes at a level} \leq 2^d((2^d-1)^s+1)$$
$$\text{Max \#nodes with same i/p fingerprint} \leq 2^d((2^d-1)^s+1)$$
$$\text{Max \#children of a node} \leq 2^{2^d((2^d-1)^s+1)}$$
$$\text{Max \#functions} \leq 2^d((2^d-1)^s+1)2^{2^d((2^d-1)^s+1)}$$

The height of the tree is $\mathcal{O}(1)$ from Definition 9 and Assumption 1. The maximum branching factor (or the maximum number of children per node) of the tree is also upper bounded by a constant. Hence, the size of the algorithm-tree is $\mathcal{O}(1)$.

(b) *Both $\mathcal{I}$ and $\mathcal{R}$ perform the same set of cell updates.*
There is no cell-tuple of $\mathcal{I}$ that is not considered by $\mathcal{R}$. In Section 2.2.2, we split the entire cell-set into subsets of cell-tuples, subsubsets of cell-tuples and so on to represent the different region-tuples. As per the rules of construction of the algorithm-tree, all cell-tuples of $\mathcal{I}$ are considered by $\mathcal{R}$.

There is no cell-tuple of $\mathcal{R}$ that is not considered by $\mathcal{I}$. Let there be a cell-tuple T in $\mathcal{R}$ that is not present in $\mathcal{I}$. As the cell-tuples in $\mathcal{R}$ are obtained by splitting the cell-set into subsets of cell-tuples, subsubsets of cell-tuples and so on, the original cell-set should include T. This means that $\mathcal{I}$ should have generated the cell-tuple T, which contradicts our initial assumption. Hence, by contradiction, all the cell tuples of $\mathcal{R}$ are considered by $\mathcal{I}$.

(c) *$\mathcal{R}$ never violates the one-way sweep property (Property 2).*
We prove that for any two cell-tuples T_1 and T_2, the order of execution of T_1 and T_2 in $\mathcal{R}$ is exactly the same as that in $\mathcal{I}$ if changing the order may lead to violation of the one-way sweep property.

The relationship between the tuples T_1 and T_2 can be defined exhaustively as shown in Table 2.2 with the four conditions:
* $\boldsymbol{W}(T_1) \in (\text{or} \notin) \boldsymbol{R}(T_1)$.
* $\boldsymbol{W}(T_2) \in (\text{or} \notin) \boldsymbol{R}(T_2)$.
* $\boldsymbol{W}(T_1) \in (\text{or} \notin) \boldsymbol{R}(T_2)$.
* $\boldsymbol{W}(T_1) = (\text{or} \neq) \boldsymbol{W}(T_2)$.

A few cases do not hold as the cell-tuples cannot simultaneously satisfy paradoxical conditions, e.g., cases 3, 5, 11 and 13 in Tab. 2.2. The relation between T_1 and T_2 can be one of the following five:
* $T_1 = T_2$.
* $T_1 \rightarrow T_2$ i.e., T_1 is executed before T_2.
* $T_2 \rightarrow T_1$ i.e., T_2 is executed before T_1.
* $T_1 || T_2$ i.e., T_1 and T_2 can be executed in parallel.
* $T_1 \leftrightarrow T_2$ i.e., either $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$.

Columns $\mathcal{I}$ and $\mathcal{R}$ represent the ordering of the two cell-tuples in $\mathcal{I}$ and $\mathcal{R}$ algorithms, respectively. Column $\mathcal{I}$ is filled based on the one-way sweep property (Property 2) and column $\mathcal{R}$ is filled based on the four rules 3, 4, 5, and 6. It is easy to see that for every case in which changing the order of execution of T_1 and T_2 may lead to the violation of the

one-way sweep property, both $\mathcal{R}$ and $\mathcal{I}$ apply the updates in exactly the same order. Hence, $\mathcal{R}$ satisfies the one-way sweep property.

Case	$W(T_1) \in R(T_1)$	$W(T_2) \in R(T_2)$	$W(T_1) \in R(T_2)$	$W(T_1) = W(T_2)$	$\mathcal{I}$	Rule	$\mathcal{R}$
1	✓	✓	✓	✓	$T_1 = T_2$	−	$T_1 = T_2$
2	✓	✓	✓	✗	$T_1 \to T_2$	3	$T_1 \to T_2$
3	✓	✓	✗	✓	−	−	−
4	✓	✓	✗	✗	$T_1 \parallel T_2$	6	$T_1 \parallel T_2$
5	✓	✗	✓	✓	−	−	−
6	✓	✗	✓	✗	$T_1 \to T_2$	3	$T_1 \to T_2$
7	✓	✗	✗	✓	$T_2 \to T_1$	4	$T_2 \to T_1$
8	✓	✗	✗	✗	$T_1 \parallel T_2$	6	$T_1 \parallel T_2$
9	✗	✓	✓	✓	$T_1 \to T_2$	4	$T_1 \to T_2$
10	✗	✓	✓	✗	$T_1 \to T_2$	3	$T_1 \to T_2$
11	✗	✓	✗	✓	−	−	−
12	✗	✓	✗	✗	$T_1 \parallel T_2$	6	$T_1 \parallel T_2$
13	✗	✗	✓	✓	−	−	−
14	✗	✗	✓	✗	$T_1 \to T_2$	3	$T_1 \to T_2$
15	✗	✗	✗	✓	$T_1 \leftrightarrow T_2$	5	$T_1 \leftrightarrow T_2$
16	✗	✗	✗	✗	$T_1 \parallel T_2$	6	$T_1 \parallel T_2$

Table 2.2: T_1 and T_2 are two cell-tuples. Columns 2-5 represent the four conditions for the two cell-tuples. Columns $\mathcal{I}$ and $\mathcal{R}$ show the ordering of the cell-tuples for $\mathcal{I}$ and $\mathcal{R}$ algorithms, respectively. The order of cell updates of $\mathcal{R}$ is consistent with $\mathcal{I}$.

Induction. We show that if AUTOGEN is correct for a problem size of $n = 2^r$ for some $r \geq q \in \mathbb{N}$, it is also correct for $n = 2^{r+1}$.

From the previous arguments we obtained a correct algorithm $\mathcal{R}$ for $r = q$. Algorithm $\mathcal{R}$ is a set of dags for different functions. Let C^n and C^{2n} represent two DP tables of size n^d and $(2n)^d$, respectively, such that $n \geq 2^q$. According to Property 3, the dependencies among the regions C_{11}^n, C_{12}^n, C_{21}^n, C_{22}^n must be exactly same as the dependencies among the regions C_{11}^{2n}, C_{12}^{2n}, C_{21}^{2n}, C_{22}^{2n}. If they were different, then that would violate Property 3. Hence, the region-tuples for the two DP tables are the same. Arguing similarly, the region-tuples remain the same for the DP tables all the way down to the threshold level. In other words, the algorithm-trees for the two problem instances are exactly the same. Having the same algorithm-trees with the same dependencies implies that the dags for DP tables C^n and C^{2n} are the same. Therefore, if AUTOGEN is correct for $n = 2^r$ for some $r \geq q \in \mathbb{N}$, it is also correct for $n = 2^{r+1}$. □

2.4 Complexity analysis

In this section, we first analyze the space and time complexities of AUTOGEN itself, and then analyze the general cache complexity of a class of $\mathcal{R}$-$\mathcal{DP}$ algorithms generated by AUTOGEN.

2.4.1 Space/time complexity of AUTOGEN

We analyze the space and time complexities of AUTOGEN using the three parameters d, s, and l, where d is the number of dimensions, $1+s$ is the cell-tuple size, and l is the threshold level.

Let the size of the DP table be n^d. Assume that the maximum number of cells a cell depends on is upper bounded by Δ. The number of cell-tuples in d-D is $\mathcal{O}\left(n^d\Delta\right)$. Hence, the total space complexity of AUTOGEN is $\mathcal{O}\left(n^d\Delta ds\right)$. To construct an algorithm-tree, the cell-tuples have to be scanned $\mathcal{O}(l)$ times. The algorithm-dag construction consumes asymptotically less time as the number of functions is $\mathcal{O}(1)$ under Assumption 1. So, the total time complexity of AUTOGEN is $\mathcal{O}\left(n^d\Delta dsl\right)$.

2.4.2 Cache complexity of an $\mathcal{R}$-$\mathcal{DP}$

In this section, we analyze the cache complexity of an $\mathcal{R}$-$\mathcal{DP}$ under the *ideal-cache model* [Frigo et al., 1999]. We assume that the size of the DP table (or space complexity) is $\Theta\left(n^d\right)$, and $M = \Omega\left(B^d\right)$, where M = cache size and B = block size. We also assume that the $\mathcal{R}$-$\mathcal{DP}$ (i) satisfies Property 1, and (ii) includes at least one *dominating closed or semi-closed function* as defined below. Most well-known $\mathcal{R}$-$\mathcal{DP}$ algorithms (including all listed in Table 2.2) satisfy those two properties.

Definition 15 (*Closed / semi-closed function*). *A recursive function is closed provided it does not call any other recursive function but itself, and it is semi-closed provided it only calls itself and other closed functions.*

Definition 16 (*Domination*). *A closed (resp. semi-closed) function $\mathcal{G}$ is dominating provided no other closed (resp. semi-closed) function of the given $\mathcal{R}$-$\mathcal{DP}$ makes more self-recursive calls than made by $\mathcal{G}$ and every non-closed (resp. non-semi-closed) function makes strictly fewer such calls.*

Suppose the $\mathcal{R}$-$\mathcal{DP}$ algorithm consists of a set $\mathcal{F}$ of m recursive functions $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m$. For $1 \leq i, j \leq m$, let a_{ij} be the number of times $\mathcal{F}_i$ calls $\mathcal{F}_j$. Then for a suitable constant $\gamma_i > 0$, the cache complexity $Q_{\mathcal{F}_i}$ of $\mathcal{F}_i$ on an input of size n^d can be computed recursively as follows.

$$Q_{\mathcal{F}_i}(n) = \begin{cases} \mathcal{O}\left(\frac{n^d}{B} + n^{d-1}\right) & \text{if } n^d \leq \gamma_i M, \\ \sum_{j=1}^{m} a_{ij} Q_{\mathcal{F}_j}\left(\frac{n}{2}\right) + \mathcal{O}(1) & \text{otherwise.} \end{cases}$$

If $\mathcal{F}_k$ is a closed function, then $Q_{\mathcal{F}_k}(n) = a_{kk}Q_{\mathcal{F}_k}(n/2) + \mathcal{O}(1)$ for $n^d > \gamma_k M$. Solving the recurrence, we get the overall (for all values of n^d) cache complexity as $Q_{\mathcal{F}_k}(n) = \mathcal{O}\left(n^{l_k}/(BM^{(l_k/d)-1}) + n^d/B + 1\right)$, where $l_k = \log_2 a_{kk}$.

If $\mathcal{F}_k$ is a dominating semi-closed function, then $Q_{\mathcal{F}_k}(n) = a_{kk}Q_{\mathcal{F}_k}(n/2) + o\left(n^{l_k}/(BM^{(l_k/d)-1})\right)$ for $n^d > \gamma_k M$. For all sizes of the DP table this recurrence also solves to $\mathcal{O}\left(n^{l_k}/(BM^{(l_k/d)-1}) + n^d/B + 1\right)$.

If $\mathcal{F}_k$ is a dominating closed (resp. semi-closed) function then (i) $a_{kk} \geq a_{ii}$ for every closed (resp. semi-closed) function $\mathcal{F}_i$, and (ii) $a_{kk} > a_{jj}$ for every non-closed (resp. non-semi-closed) function $\mathcal{F}_j$. The algorithm-tree must contain at least one path $P = \langle \mathcal{F}_{r_1}, \mathcal{F}_{r_2}, \ldots, \mathcal{F}_{r_{|P|}} \rangle$ from its root $(= \mathcal{F}_{r_1})$ to a node corresponding to $\mathcal{F}_k (= \mathcal{F}_{r_{|P|}})$. Since $|P|$ is a

small number independent of n, and by definition $a_{r_i r_i} < a_{r_{|P|} r_{|P|}}$ holds for every $i \in [1, |P|-1]$, one can show that the cache complexity of every function on P must be $\mathcal{O}\left(Q_{\mathcal{F}_k}(n)\right)$. This result is obtained by moving upwards in the tree starting from $\mathcal{F}_{r_{|P|-1}}$, writing down the cache complexity recurrence for each function on this path, substituting the cache complexity results determined for functions that we have already encountered, and solving the resulting simplified recurrence. Hence, the cache complexity $Q_{\mathcal{F}_{r_1}}(n)$ of the $\mathcal{R}$-$\mathcal{DP}$ algorithm is $\mathcal{O}\left(Q_{\mathcal{F}_k}(n)\right)$.

> **Theorem 2 (Cache complexity of the $\mathcal{R}$-$\mathcal{DP}$ algorithms).** *If an $\mathcal{R}$-$\mathcal{DP}$ includes a dominating closed or semi-closed function $\mathcal{F}_k$ that calls itself recursively a_{kk} times, then the serial cache complexity of the $\mathcal{R}$-$\mathcal{DP}$ for a DP table of size n^d is*
>
> $$Q_1(n, d, B, M) = \mathcal{O}\left(\frac{T_1(n)}{BM^{(l_k/d)-1}} + \frac{S(n,d)}{B} + 1\right)$$
>
> *under the ideal-cache model, where $l_k = \log_2 a_{kk}$, $T_1(n) = \text{total work} = \mathcal{O}\left(n^{l_k}\right)$, $M = \text{cache size}$, $B = \text{block size}$, $M = \Omega\left(B^d\right)$, and $S(n,d) = \text{space complexity} = \mathcal{O}\left(n^d\right)$.*

It is important to note the serial cache complexity and the total work an $\mathcal{R}$-$\mathcal{DP}$ algorithm are related. The work is found by counting the total number of leaf nodes in the algorithm-tree. We using Master theorem repeatedly to find $T_1(n)$ of the $\mathcal{R}$-$\mathcal{DP}$ algorithm.

> **Theorem 3 (Cache complexity of the $\mathcal{R}$-$\mathcal{DP}$ algorithms).** *Let $\mathcal{F}_k$ be a dominating closed function that calls itself a_{kk} number of times and let $P = \langle \mathcal{F}_{r_1}, \mathcal{F}_{r_2}, \ldots, \mathcal{F}_{|P|}\rangle$ be a path in the algorithm-tree from its root $(= \mathcal{F}_{r_1})$ to a node corresponding to $\mathcal{F}_k(= \mathcal{F}_{|P|})$. Let q out of these $|P|$ functions call themselves a_{kk} times and q is maximized over all possible paths in the algorithm-tree. Then the serial cache complexity of the $\mathcal{R}$-$\mathcal{DP}$ for a DP table of size n^d is*
>
> $$Q_1(n, d, B, M) = \mathcal{O}\left(\frac{T_1(n)}{BM^{(l_k/d)-1}} + \frac{S(n,d)}{B} + 1\right)$$
>
> *under the ideal-cache model, where $l_k = \log_2 a_{kk}$, $T_1(n) = \text{total work} = \mathcal{O}\left(n^{l_k} \log^{q-1} n\right)$, $M = \text{cache size}$, $B = \text{block size}$, $M = \Omega\left(B^d\right)$, and $S(n,d) = \text{space complexity} = \mathcal{O}\left(n^d\right)$.*

2.4.3 Upper-triangular system of recurrence relations

Let $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m$ be the m functions of the autogenerated $\mathcal{R}$-$\mathcal{DP}$. Then, the work and cache complexity of the functions are computed as follows:

$$W_i(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{j=i}^{m} a_{ij} W_j(n/2) + \Theta(1) & \text{otherwise;} \end{cases} \tag{2.5}$$

$$Q_i(n) = \begin{cases} \mathcal{O}\left(n^d/B + n^{d-1}\right) & \text{if } n^d \leq \gamma_i M, \\ \sum_{j=i}^{m} a_{ij} Q_j(n/2) + \Theta(1) & \text{otherwise;} \end{cases} \tag{2.6}$$

where $W_i(n)$ is the total work of $\mathcal{F}_i(n)$, $Q_i(n)$ ($i \in [1, m]$) is the serial cache complexity of $\mathcal{F}_i(n)$, $\Theta\left(n^d\right)$ is the space complexity of the algorithm for $d \geq 1$, and a_{ij} is the number of times the function $\mathcal{F}_i(n)$ calls $\mathcal{F}_j(n)$.

We can represent this system of recurrence relations as an upper-triangular matrix with non-zero diagonal elements as follows:

$$
\begin{bmatrix} W_1(n) \\ W_2(n) \\ \vdots \\ W_m(n) \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ 0 & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{mm} \end{bmatrix} \begin{bmatrix} W_1(n/2) \\ W_2(n/2) \\ \vdots \\ W_m(n/2) \end{bmatrix} + \begin{bmatrix} \Theta(1) \\ \Theta(1) \\ \vdots \\ \Theta(1) \end{bmatrix}
$$

where

$$
H = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ 0 & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{mm} \end{bmatrix}
$$

is called the *algorithm-matrix*. It represents an autogenerated $\mathcal{R}$-$\mathcal{DP}$ that satisfies Assumption 1. For example, the algorithm-matrix for Floyd-Warshall's APSP divide-and-conquer DP algorithm is

$$
H = \begin{bmatrix} 2 & 2 & 2 & 2 \\ 0 & 4 & 0 & 4 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix}
$$

Representation using finite automaton

Figure 2.8 gives simple finite automata to represent the number of function calls to various functions in the $\mathcal{R}$-$\mathcal{DP}$s for Floyd-Warshall's APSP and parenthesis problem. In this graph-based representation, if there is a directed edge from node $\mathcal{F}_i$ to node $\mathcal{F}_j$ and its edge weight is a_{ij}, it means that function $\mathcal{F}_i$ calls $\mathcal{F}_j$, a_{ij} number of times. As we assume the one-way function generation property to hold, these graphs will always be DAGs as they do not have cycles (loops not included).

The finite automaton representation can also be used for *representing the spans and I/O complexities* of different recursive functions. This representation is a very powerful tool to intuitively get an idea to derive complicated theorems and lemmas related to the complexities (work, span, and I/O) of the $\mathcal{R}$-$\mathcal{DP}$s.

To optimize an $\mathcal{R}$-$\mathcal{DP}$ program, it is sufficient to optimize only those functions that are invoked the greatest number of times asymptotically, when the input parameter reaches a base case size. It is arguably simpler to find the exact number of calls to the functions than finding a tight asymptotic bound on the number of calls. First, we present techniques to find the exact number of calls to the functions $\mathcal{F}_1(1)$ to $\mathcal{F}_m(1)$ and then the strategies to find the number of function calls aymptotically.

Number of function calls

Here, we present two different methods to compute the number of calls to a function $\mathcal{F}_i$ at some level in the algorithm-tree. Throughout the section, we assume n and b are powers of two and b is a constant.

Let $R[i, j]$ represent the number of $\mathcal{F}_j(b)$ calls made from $\mathcal{F}_i(n)$. We are mainly interested in the values of $R[1, 1], R[1, 2], \ldots, R[1, m]$. The matrix R of size $m \times m$ can be computed using
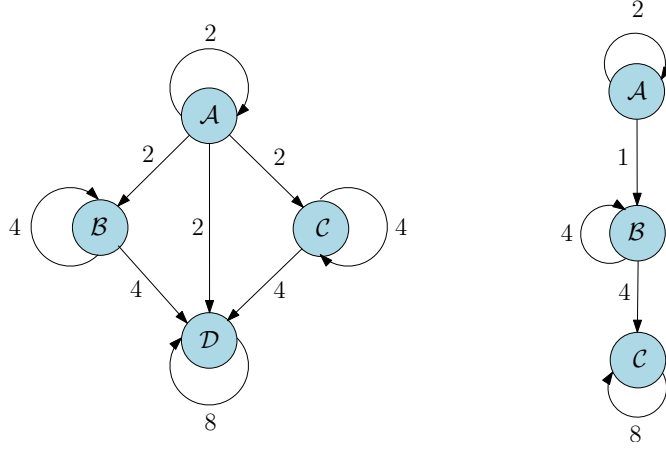
Figure 2.8: Finite automaton for representing the function calls for the $\mathcal{R}\text{-}\mathcal{DP}$ of: Left: Floyd-Warshall's APSP, and Right: parenthesis problem.

the formula

$$R = H^{\log(n/b)} \tag{2.7}$$

The time complexity to compute R is $\Theta\left(m^3 \log\log n\right)$ using the most common matrix multiplication algorithm and a standard technique called repeated squaring.

An another method to compute the number of function calls is as follows. Now, let $T[i,j]$ represent the number of calls to function $\mathcal{F}_i$ at a level j. We would like to compute the values of $T[1, \log(n/b)], T[2, \log(n/b)], \ldots, T[m, \log(n/b)]$. The matrix T of size $m \times (1 + \log(n/b))$ is computed using the recurrence

$$T[i,j] = \begin{cases} 1 & \text{if } i = 1 \text{ and } j = 0 \\ 0 & \text{if } i > 1 \text{ and } j = 0 \\ \sum_{k=1}^{i-1} \sum_{l=j-1}^{0} T[k,l] \cdot H[k,i] \cdot H[i,i]^{j-l-1} & \text{otherwise;} \end{cases} \tag{2.8}$$

The time complexity to compute R in the naive way is $\Theta\left(m^3 \log n\right)$.

Complexity of the number of function calls
Let $P = \langle \mathcal{F}_{r_1}, \mathcal{F}_{r_2}, \ldots, \mathcal{F}_{|P|} \rangle$ be a path in the algorithm-tree from its root $(= \mathcal{F}_{r_1})$ to a node corresponding to $\mathcal{F}_{r_i} (= \mathcal{F}_{|P|})$. Let $maxloop = \max\left(H[r_1, r_1], H[r_2, r_2] \ldots, H[r_i, r_i]\right)$. Let q out of these $|P|$ functions call themselves $maxloop$ times and q is maximized over all possible paths ending at $\mathcal{F}_{r_i}$. Using Master theorem repeatedly we can show that the complexity of the number of function calls denoted by S_i to a function $\mathcal{F}_i$ can be computed to be

$$S_i = \Theta\left(n^{\log maxloop} \log^{q-1} n\right) \tag{2.9}$$

2.5 Extensions of AUTOGEN

In this section, we discuss how to extend AUTOGEN to
  ✫ [*One-way sweep property violation.*] Find $\mathcal{R}\text{-}\mathcal{DP}$ algorithms for an important class of DP problems, given $\mathcal{I}\text{-}\mathcal{DP}$ algorithms that do not satisfy the one-way sweep property.

⋆ [*Space reduction*.] Sometimes reduce the space usage of the generated $\mathcal{R}$-$\mathcal{DP}$ algorithms thereby improving the cache complexity.

⋆ [*Non-orthogonal regions*.] Handle problems where dividing the DP table into non-orthogonal regions leads to the minimum number of recursive functions.

2.5.1 One-way sweep property violation

We describe a strategy to find an $\mathcal{R}$-$\mathcal{DP}$ indirectly for an $\mathcal{I}$-$\mathcal{DP}$ that violates the one-way sweep property (Property 2). The strategy works for dynamic programs for computing paths over a closed semiring in a directed graph [Aho et al., 1974]. Floyd-Warshall's algorithm for finding all-pairs shortest path (APSP) [Floyd, 1962, Warshall, 1962] belongs to this class, and will be used as an example. Our approach follows the following three steps.

We describe a strategy to find an $\mathcal{R}$-$\mathcal{DP}$ indirectly for an $\mathcal{I}$-$\mathcal{DP}$ that violates the one-way sweep property (Prop. 2). Floyd-Warshall's all-pairs shortest path (APSP) algorithm shall be used as an example. In fact, for all problems that come under a closed semiring sysem $(S, \oplus, \odot, \bar{0}, \bar{1})$, where S is a set of elements, $\oplus$ and $\odot$ are binary operations of S, and $\bar{0}$ and $\bar{1}$ are elements of S satisfying certain conditions, it can be proved (see [Cormen et al., 2009]) that the strategy works provided each of the three operations $T_\oplus, T_\odot$, and T_* takes $\mathcal{O}(1)$ time. If an $\mathcal{I}$-$\mathcal{DP}$ algorithm violates the one-way sweep property, we use the following strategy to find its $\mathcal{R}$-$\mathcal{DP}$:

While our AUTOGEN algorithm does not work on update functions that use values from partially updated cells, it turns out that handling such update functions in not difficult either. Our approach involves removing explicit dependences on partially updated cells from the code by allowing it to retain all intermediate values of each cell. This effectively lifts the DP to a higher dimensional space. Now our AUTOGEN algorithm can be applied on this modified DP to obtain an $\mathcal{R}$-$\mathcal{DP}$ algorithm. We then project this $\mathcal{R}$-$\mathcal{DP}$ algorithm back to the original lower dimensional space, and prove that the projected algorithm correctly implements the original $\mathcal{I}$-$\mathcal{DP}$ with dependences on partially updated cells. An $\mathcal{I}$-$\mathcal{DP}$ algorithm that violates the one-way sweep property (Property 2).

(*i*) [*Project $\mathcal{I}$-$\mathcal{DP}$ to higher dimension*.] Violation of the one-way sweep property means that some cells of the DP table are computed from cells that are not yet fully updated. By allocating space to retain each intermediate value of every cell, the problem is transformed into a new problem where the cells depend on fully updated cells only. The technique effectively projects the DP on to a higher dimensional space leading to a correct $\mathcal{I}$-$\mathcal{DP}$ that satisfies the one-way sweep property.

(*ii*) [*Autogenerate $\mathcal{R}$-$\mathcal{DP}$ from $\mathcal{I}$-$\mathcal{DP}$*.] AUTOGEN is applied on the higher dimensional $\mathcal{I}$-$\mathcal{DP}$ that satisfies Property 2 to generate an $\mathcal{R}$-$\mathcal{DP}$ in the same higher dimensional space.

(*iii*) [*Project $\mathcal{R}$-$\mathcal{DP}$ back to original dimension*.] The autogenerated $\mathcal{R}$-$\mathcal{DP}$ is projected back to the original dimensional space. One can show that the projected $\mathcal{R}$-$\mathcal{DP}$ correctly implements the original $\mathcal{I}$-$\mathcal{DP}$ [Cormen et al., 2009, Chowdhury and Ramachandran, 2010].

Example. Figure 2.9 shows the widely used quadratic-space $\mathcal{I}$-$\mathcal{DP}$ (LOOP-FLOYD-WARSHALL-APSP) for Floyd-Warshall's APSP. This algorithm violates the one-way sweep property, and hence does not belong to the FRACTAL-DP class. Therefore, AUTOGEN cannot be applied on this $\mathcal{I}$-$\mathcal{DP}$ directly to get an $\mathcal{R}$-$\mathcal{DP}$. However, the strategy described in Section 2.5.1 produces an $\mathcal{R}$-$\mathcal{DP}$ as follows.

(i) [*Project $\mathcal{I}$-$\mathcal{DP}$ to higher dimension*.] The 2-D version of the $\mathcal{I}$-$\mathcal{DP}$ in Figure 2.9(b) is projected on to 3-D space to get its 3-D version in Figure 2.9(a) that updates the 3-D matrix $D[1..n, 1..n, k]$, where $k \in [1, n]$. One can show that both the implementations produce the same final output, e.g., see Exercise 25.2-4 in Cormen et al. [Cormen et al., 2009].

(ii) [*Autogenerate $\mathcal{R}$-$\mathcal{DP}$ from $\mathcal{I}$-$\mathcal{DP}$*.] Autogen is applied on the 3-D $\mathcal{I}$-$\mathcal{DP}$ that satisfies Property 2, to get a 3-D cache-inefficient $\mathcal{R}$-$\mathcal{DP}$ as shown in Figure 2.9(a) that has a cache complexity of $\mathcal{O}\left(n^3/B\right)$.

(iii) [*Project $\mathcal{R}$-$\mathcal{DP}$ back to original dimension*.] Finally, the 3-D $\mathcal{R}$-$\mathcal{DP}$ is projected on to 2-D by projecting each $D[i, j, k]$, $i, j, k \in [1, n]$ onto $D[i, j, 0]$, and performing all reads/writes on $D[1..n, 1..n, 0]$. The resulting $\mathcal{R}$-$\mathcal{DP}$ is shown in Figure 2.9(b). It can be proved that the 2-D $\mathcal{R}$-$\mathcal{DP}$ is a correct implementation of the 2-D $\mathcal{I}$-$\mathcal{DP}$ [Chowdhury and Ramachandran, 2010]. The 2-D $\mathcal{R}$-$\mathcal{DP}$ has $\mathcal{O}\left(n \log^2 n\right)$ span and incurs $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ cache misses (assuming $M = \Omega\left(B^2\right)$).

2.5.2 Space reduction

Autogen can be extended to analyze and optimize the functions of an autogenerated $\mathcal{R}$-$\mathcal{DP}$ for a possible reduction in space usage. We explain through an example.

Example. The LCS problem [Hirschberg, 1975, Chowdhury and Ramachandran, 2006] asks one to find the longest of all common subsequences [Cormen et al., 2009] between two strings. LCS is a typical example of a class of DP problems known as *local dependency DP* [Chowdhury and Ramachandran, 2008]. , which in turn is a subset of a more general class known as *stencils* [Frigo and Strumpen, 2005, Frigo and Strumpen, 2007, Tang et al., 2011]. In LCS, a cell depends on its three adjacent cells. Here, we are interested in finding the length of the LCS and not the LCS itself. Starting from the standard $\Theta\left(n^2\right)$ space $\mathcal{I}$-$\mathcal{DP}$, we generate an $\mathcal{R}$-$\mathcal{DP}$ for the problem that contains four recursive functions. The autogenerated $\mathcal{R}$-$\mathcal{DP}$ still uses $\Theta\left(n^2\right)$ space and incurs $\mathcal{O}\left(n^2/B\right)$ cache misses. The $\mathcal{R}$-$\mathcal{DP}$ algorithm is cache-inefficient as it has no temporal locality. Indeed, without any asymptotic difference between the running time and the space usage (both are $\mathcal{O}\left(n^2\right)$ for the LCS $\mathcal{R}$-$\mathcal{DP}$) no cell in the DP table is reused more than $\mathcal{O}\left(1\right)$ times. Autogen can reason as follows in order to reduce the space usage of this $\mathcal{R}$-$\mathcal{DP}$ and thereby improving its cache performance.

The autogenerated $\mathcal{R}$-$\mathcal{DP}$ has two functions of the form $\mathcal{F}(n) \mapsto \{\mathcal{F}(n/2), \mathcal{F}(n/2), \mathcal{G}(n/2)\}$, where $\mathcal{G}$ is of the form $\mathcal{G}(n) \mapsto \{\mathcal{G}(n/2)\}$. Given their dependencies, it is easy that in $\mathcal{G}$, the top-left cell of bottom-right quadrant depends on the bottom-right cell of the top-left quadrant. Also, in $\mathcal{F}$, the leftmost (resp. topmost) boundary cells of one quadrant depends on the rightmost (resp. bottommost) quadrant of adjacent quadrant. When there is only a dependency on the boundary cells, we can copy the values of the boundary cells, which occupies $\mathcal{O}\left(n\right)$ space, between different function calls and we no longer require quadratic space. At each level of the recursion tree $\mathcal{O}\left(n\right)$ space is used, and the total space for the parallel $\mathcal{R}$-$\mathcal{DP}$ algorithm is $\mathcal{O}\left(n \log n\right)$. This new $\mathcal{R}$-$\mathcal{DP}$ algorithm will have a single function and its cache complexity improves to $\mathcal{O}\left(n^2/(BM)\right)$. Space usage can be reduced further to $\mathcal{O}\left(n\right)$ by simply reusing space between parent and child functions. Cache complexity remains $\mathcal{O}\left(n^2/(BM)\right)$.

LOOP-FLOYD-WARSHALL-APSP-3D(D, n)
1. **for** $k \leftarrow 1$ to n
2.   **for** $i \leftarrow 1$ to n
3.   **for** $j \leftarrow 1$ to n
4. $D[i,j,k] \leftarrow \min\left(D[i,j,k-1], D[i,k,k-1] + D[k,j,k-1]\right)$

$\mathcal{A}_{FW}^{3D}(\langle X,Y,X,X \rangle)$
1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}FW}^{3D}(\langle X,Y,X,X \rangle)$
  else
2. $\mathcal{A}_{FW}^{3D}(\langle X_{111}, Y_{112}, X_{111}, X_{111} \rangle)$
3.   **par:** $\mathcal{B}_{FW}^{3D}(\langle X_{121}, Y_{122}, X_{111}, X_{121} \rangle)$, $\mathcal{C}_{FW}^{3D}(\langle X_{211}, Y_{212}, X_{211}, X_{111} \rangle)$
4. $\mathcal{D}_{FW}^{3D}(\langle X_{221}, Y_{222}, X_{211}, X_{121} \rangle)$
5. $\mathcal{A}_{FW}^{3D}(\langle X_{222}, X_{221}, X_{222}, X_{222} \rangle)$
6.   **par:** $\mathcal{B}_{FW}^{3D}(\langle X_{212}, X_{211}, X_{222}, X_{212} \rangle)$, $\mathcal{C}_{FW}^{3D}(\langle X_{122}, X_{121}, X_{122}, X_{222} \rangle)$
7. $\mathcal{D}_{FW}^{3D}(\langle X_{112}, X_{111}, X_{122}, X_{212} \rangle)$

$\mathcal{B}_{FW}^{3D}(\langle X,Y,U,X \rangle)$
1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}FW}^{3D}(\langle X,Y,U,X \rangle)$
  else
2.   **par:** $\mathcal{B}_{FW}^{3D}(\langle X_{111}, Y_{112}, U_{111}, X_{111} \rangle)$, $\mathcal{B}_{FW}^{3D}(\langle X_{121}, Y_{122}, U_{111}, X_{121} \rangle)$
3.   **par:** $\mathcal{D}_{FW}^{3D}(\langle X_{211}, Y_{212}, U_{211}, X_{111} \rangle)$, $\mathcal{D}_{FW}^{3D}(\langle X_{221}, Y_{222}, U_{211}, X_{121} \rangle)$
4.   **par:** $\mathcal{B}_{FW}^{3D}(\langle X_{212}, X_{211}, U_{222}, X_{212} \rangle)$, $\mathcal{B}_{FW}^{3D}(\langle X_{222}, X_{221}, U_{222}, X_{222} \rangle)$
5.   **par:** $\mathcal{D}_{FW}^{3D}(\langle X_{112}, X_{111}, U_{122}, X_{212} \rangle)$, $\mathcal{D}_{FW}^{3D}(\langle X_{122}, X_{121}, U_{122}, X_{222} \rangle)$

$\mathcal{C}_{FW}^{3D}(\langle X,Y,X,V \rangle)$
1. **if** X is a small matrix **then** $\mathcal{C}_{loop\text{-}FW}^{3D}(\langle X,Y,X,V \rangle)$
  else
2.   **par:** $\mathcal{C}_{FW}^{3D}(\langle X_{111}, Y_{112}, X_{111}, V_{111} \rangle)$, $\mathcal{C}_{FW}^{3D}(\langle X_{211}, Y_{212}, X_{211}, V_{111} \rangle)$
3.   **par:** $\mathcal{D}_{FW}^{3D}(\langle X_{121}, Y_{122}, X_{111}, V_{121} \rangle)$, $\mathcal{D}_{FW}^{3D}(\langle X_{221}, Y_{222}, X_{211}, V_{121} \rangle)$
4.   **par:** $\mathcal{C}_{FW}^{3D}(\langle X_{122}, X_{121}, X_{122}, V_{222} \rangle)$, $\mathcal{C}_{FW}^{3D}(\langle X_{222}, X_{221}, X_{222}, V_{222} \rangle)$
5.   **par:** $\mathcal{D}_{FW}^{3D}(\langle X_{112}, X_{111}, X_{122}, V_{212} \rangle)$, $\mathcal{D}_{FW}^{3D}(\langle X_{212}, X_{211}, X_{222}, V_{122} \rangle)$

$\mathcal{D}_{FW}^{3D}(\langle X,Y,U,V \rangle)$
1. **if** X is a small matrix **then** $\mathcal{D}_{loop\text{-}FW}^{3D}(\langle X,Y,U,V \rangle)$
  else
2.   **par:** $\mathcal{D}_{FW}^{3D}(\langle X_{111}, Y_{112}, U_{111}, V_{111} \rangle)$, $\mathcal{D}_{FW}^{3D}(\langle X_{121}, Y_{122}, U_{111}, V_{121} \rangle)$,
    $\mathcal{D}_{FW}^{3D}(\langle X_{211}, Y_{212}, U_{211}, V_{111} \rangle)$, $\mathcal{D}_{FW}^{3D}(\langle X_{221}, Y_{222}, U_{211}, V_{121} \rangle)$
3.   **par:** $\mathcal{D}_{FW}^{3D}(\langle X_{112}, X_{111}, U_{122}, V_{212} \rangle)$, $\mathcal{D}_{FW}^{3D}(\langle X_{122}, X_{121}, U_{122}, V_{222} \rangle)$,
    $\mathcal{D}_{FW}^{3D}(\langle X_{212}, X_{211}, U_{222}, V_{212} \rangle)$, $\mathcal{D}_{FW}^{3D}(\langle X_{222}, X_{221}, U_{222}, V_{222} \rangle)$

(a)

LOOP-FLOYD-WARSHALL-APSP(D, n)
1. **for** $k \leftarrow 1$ to n
2.   **for** $i \leftarrow 1$ to n
3.   **for** $j \leftarrow 1$ to n
4. $D[i,j] \leftarrow \min\left(D[i,j], D[i,k] + D[k,j]\right)$

$\mathcal{A}_{FW}(\langle X,X,X \rangle)$
1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}FW}(\langle X,X,X \rangle)$
  else
2. $\mathcal{A}_{FW}(\langle X_{11}, X_{11}, X_{11} \rangle)$
3.   **par:** $\mathcal{B}_{FW}(\langle X_{12}, X_{11}X_{12} \rangle)$, $\mathcal{C}_{FW}(\langle X_{21}, X_{21}, X_{11} \rangle)$
4. $\mathcal{D}_{FW}(\langle X_{22}, X_{21}, X_{12} \rangle)$
5. $\mathcal{A}_{FW}(\langle X_{22}, X_{22}, X_{22} \rangle)$
6.   **par:** $\mathcal{B}_{FW}(\langle X_{21}, X_{22}, X_{21} \rangle)$, $\mathcal{C}_{FW}(\langle X_{12}, X_{12}, X_{22} \rangle)$
7. $\mathcal{D}_{FW}(\langle X_{11}, X_{12}, X_{21} \rangle)$

$\mathcal{B}_{FW}(\langle X,U,X \rangle)$
1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}FW}(\langle X,U,X \rangle)$
  else
2.   **par:** $\mathcal{B}_{FW}(\langle X_{11}, U_{11}, X_{11} \rangle)$, $\mathcal{B}_{FW}(\langle X_{12}, U_{11}, X_{12} \rangle)$
3.   **par:** $\mathcal{D}_{FW}(\langle X_{21}, U_{21}, X_{11} \rangle)$, $\mathcal{D}_{FW}(\langle X_{22}, U_{21}, X_{12} \rangle)$
4.   **par:** $\mathcal{B}_{FW}(\langle X_{21}, U_{22}, X_{21} \rangle)$, $\mathcal{B}_{FW}(\langle X_{22}, U_{22}, X_{22} \rangle)$
5.   **par:** $\mathcal{D}_{FW}(\langle X_{11}, U_{12}, X_{21} \rangle)$, $\mathcal{D}_{FW}(\langle X_{12}, U_{12}, X_{22} \rangle)$

$\mathcal{C}_{FW}(\langle X,X,V \rangle)$
1. **if** X is a small matrix **then** $\mathcal{C}_{loop\text{-}FW}(\langle X,X,V \rangle)$
  else
2.   **par:** $\mathcal{C}_{FW}(\langle X_{11}, X_{11}, V_{11} \rangle)$, $\mathcal{C}_{FW}(\langle X_{21}, X_{21}, V_{11} \rangle)$
3.   **par:** $\mathcal{D}_{FW}(\langle X_{12}, X_{11}, V_{12} \rangle)$, $\mathcal{D}_{FW}(\langle X_{22}, X_{21}, V_{12} \rangle)$
4.   **par:** $\mathcal{C}_{FW}(\langle X_{12}, X_{12}, V_{22} \rangle)$, $\mathcal{C}_{FW}(\langle X_{22}, X_{22}, V_{22} \rangle)$
5.   **par:** $\mathcal{D}_{FW}(\langle X_{11}, X_{12}, V_{21} \rangle)$, $\mathcal{D}_{FW}(\langle X_{21}, X_{22}, V_{12} \rangle)$

$\mathcal{D}_{FW}(\langle X,U,V \rangle)$
1. **if** X is a small matrix **then** $\mathcal{D}_{loop\text{-}FW}(\langle X,U,V \rangle)$
  else
2.   **par:** $\mathcal{D}_{FW}(\langle X_{11}, U_{11}, V_{11} \rangle)$, $\mathcal{D}_{FW}(\langle X_{12}, U_{11}, V_{12} \rangle)$,
    $\mathcal{D}_{FW}(\langle X_{21}, U_{21}, V_{11} \rangle)$, $\mathcal{D}_{FW}(\langle X_{22}, U_{21}, V_{12} \rangle)$
3.   **par:** $\mathcal{D}_{FW}(\langle X_{11}, U_{12}, V_{21} \rangle)$, $\mathcal{D}_{FW}(\langle X_{12}, U_{12}, V_{22} \rangle)$,
    $\mathcal{D}_{FW}(\langle X_{21}, U_{22}, V_{21} \rangle)$, $\mathcal{D}_{FW}(\langle X_{22}, U_{22}, V_{22} \rangle)$

(b)

Figure 2.9: (a) An autogenerated $\mathcal{R}$-$\mathcal{DP}$ algorithm from the cubic space Floyd-Warshall's APSP algorithm. In the initial call to $\mathcal{A}_{FW}^{3D}(\langle X,Y,X,X \rangle)$, X points to $D[1..n, 1..n, 1..n]$ and Y points to an n^3 matrix whose topmost plane is initialized with $D[1..n, 1..n, 0]$. (b) An $\mathcal{R}$-$\mathcal{DP}$ algorithm obtained by projecting the 3D matrix $D[1..n, 1..n, 0..n]$ accessed by the algorithm in column (a) to its 2D base $D[1..n, 1..n, 0]$.

2.5.3 Non-orthogonal regions

In this section, we generalize the definition of a region to include non-rectangular areas as well. As per Definition 2, the entire DP table was divided orthogonally into four equal quadrants and the quadrants were recursively subdivided into four equal subquadrants until each of the subquadrants contained only a single cell. Each quadrant was called a region at a particular level. This definition of a region can create many more functions than necessary. Though the number of functions are upper bounded by a constant as shown in Section 2.3.3, the number of functions can be reduced by generalizing the definition of a region.

We define a term called compute-shape that will be used subsequently.

Definition 17 (*Compute-cells, compute-shape*)**.** *The set of all cells that will be computed in a DP table for a given DP problem by its DP algorithm is called compute-cells. The geometric shape and area the compute-cells represents is called compute-shape.*

Given a DP problem that does not violate Rule 2, the fixed polygonal compute-shape S with q vertices $p_1, p_2, \ldots, p_q$ is found. The shape S is scaled down by a factor of two to S' having vertices $p'_1, p'_2, \ldots, p'_q$ such that $\forall i$, p'_i corresponds to p_i. The shape S' can move as opposed to S, which is fixed on the DP table. The shape S' is moved inside the fixed shape S without rotating such that S' is completely contained in S and for some i, $p_i = p'_i$ and that common area is denoted by S'_i. If all the cells inside S'_i, for some i, depend on cells inside S'_i alone, then the area S'_i is called a region, more specifically a self-dependent region. The disjoint areas when all self-dependent regions are removed from S are also called regions. If we obtain parallelograms for regions, then they can be divided into four equal parallelograms using lines parallel to the sides of the parallelogram and each of those smaller parallelograms will be regions at the next level.

(a) (b)
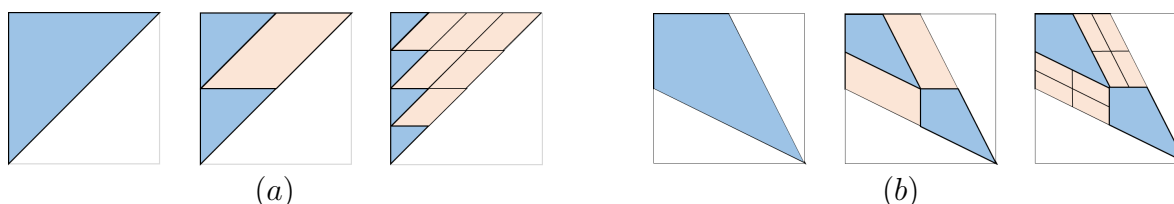
Figure 2.10: Non-orthogonal regions for the first three levels depicted for: (a) CYK algorithm, and (b) Spoken word recognition.

As an example, in the Cocke-Younger-Kasami (CYK) algorithm, the division of the compute-shape into regions at three levels, using the generalized definition of region, is shown in Figure 2.10.

2.6 The deductive AUTOGEN algorithm

In mathematical logic, we come across two types of arguments: deductive and inductive. A *deductive argument* is an argument in which if all premises are true, then the conclusion is true. On the other hand, an *inductive argument* is an argument in which if all premises are true, then it is likely that the conclusion is true, but it is not logically necessary that the conclusion be true. Good references to logic are [Hurley, 2014] and [Gensler, 2010].

Any kind of formal mathematical proof automatically is a deductive argument or deductive reasoning. Examples include direct proof, proof by mathematical induction, proof by contraposition, proof by construction, proof by exhaustion, probabilistic proof, combinatorial proof, non-constructive proof, and so on. On the other side, an inductive argument is like a guess, in fact an intelligent guess. It may or may not be true depending on circumstances. It is important to note the mathematical induction is not an inductive argument but a deductive argument because it is a formal mathematical proof.

47
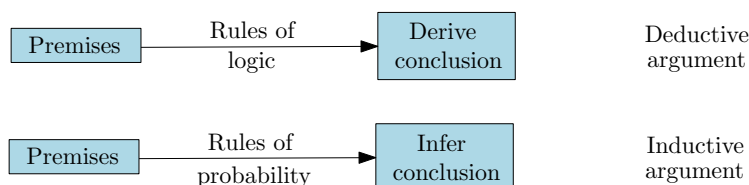
Deductive reasoning is very powerful. Starting with the premises, and using axioms of logic and mathematics, we can build a new theory or derive new conclusion(s) in a step-by-step fashion. Most of the mathematical logic field deals with deductive logic. Only in case if we cannot derive deductively, we can go for inductive arguments.

In Section 2.2, we developed the AUTOGEN algorithm using an inductive approach. In Section 2.3, we proved AUTOGEN correct using mathematical induction (deductive approach). Though, overall, the entire argument of AUTOGEN is deductive, it is not purely deductive. This is because, in the first stage (inductive) we constructed AUTOGEN from data dependency patterns and in the second stage (deductive) we proved that AUTOGEN indeed is correct.

The major limitations of the impure deductive AUTOGEN are:

* [*Space/time complexity.*] When the number of recursive functions in an $\mathcal{R}\text{-}\mathcal{DP}$ is very high, say 100, the problem parameter needed for the impure deductive AUTOGEN will be $q > 2^{100}$. Hence, both the space and time complexities of the AUTOGEN can be unimaginably high and hence impractical.
* [*Problem size.*] Initially we select a small problem with parameter, say, $q = 64$. If all the functions are not expressed in the algorithm-tree then we need to increase the problem size and this process continues.

In this section, we present a purely deductive method to construct the AUTOGEN algorithm. It addresses the limitations of the impure deductive AUTOGEN. Each step in this method logically follows from its previous step. We call this method deductive AUTOGEN algorithm and the algorithm we discussed in Section 2.2 as inductive AUTOGEN (though it is not completely inductive). The deductive AUTOGEN differs from inductive AUTOGEN in the first two steps only.

Algorithm

The four main steps of deductive AUTOGEN are:

(1) [*Generic iterative algorithm construction.*] A very general iterative algorithm is constructed from the given iterative algorithm. See Section 2.6.1.
(2) [*Algorithm-tree construction.*] An algorithm-tree is constructed from the generic iterative algorithm. See Section 2.6.2.
(3) [*Algorithm-tree labeling.*] Same as Section 2.2.3.
(4) [*Algorithm-DAG construction.*] Same as Section 2.2.4.

2.6.1   Generic iterative algorithm construction

In this step, we construct a generic iterative algorithm from the given iterative algorithm.

The intuition behind the generic kernel is that it can replace all the iterative kernels of all recursive functions in a standard 2-way $\mathcal{R}\text{-}\mathcal{DP}$. We explain in more details the meaning and use of the generic kernel.

Let the total work of the iterative algorithm be $\Theta\left(n^h\right)$. Then the iterative algorithm can be written in a generic structure as shown in Figure 2.11. The structure consists of h loops with loop parameters ℓ_1 to ℓ_h, each running from either 1 to n or from n to 1. In the innermost loop, we have a series of conditional and assignment statements. Each conditional statement can be a collection of simple inequalities involving the loop variables, e.g.: $condition_1$ can be ($n/2 + \ell_2 \geq \ell_1$ or $\ell_3 < n/8 - \ell_2$). If $condition_i$ is true, then $W[w]$, where w is the cell to be written of the submatrix W, is updated using the generic

update relation (see Definition 18) GENERIC-UPDATE$_i$. We call such an algorithm a generic iterative algorithm.

```
GENERIC-ITERATIVE-ALGORITHM(W, R¹, .., Rˢ, n)

1.  for [ℓ₁ ← 1 to n] or [ℓ₁ ← n to 1] do
2.  for [ℓ₂ ← 1 to n] or [ℓ₂ ← n to 1] do
        ⋮
3.  for [ℓₕ ← 1 to n] or [ℓₕ ← n to 1] do
4.  for every tuple (W, R¹, ..., Rˢ) do
5.      if condition₁ then
6.          GENERIC-UPDATE₁(W[w])
7.      elif condition₂ then
8.          GENERIC-UPDATE₂(W[w])
            ⋮
9.      elif condition_last then
10.         GENERIC-UPDATE_last(W[w])
```

```
GENERIC-LOOP-PARENTHESIS(X, U, V, n)

1.  for i ← n − 1 to 0 do
2.  for j ← 0 to n − 1 do
3.  for k ← 0 to n − 1 do
4.  for every tuple (X, U, V) do
5.      XI ← x_r + i; XJ ← x_c + j
6.      UI ← u_r + i; VJ ← v_c + j
7.      K ← u_c + k
8.      if XJ ≥ XI + 1 and K ≥ XI and XJ > K − 1 then
9.          C[XI, XJ] ← min{C[XI, XJ], C[UI, K] + C[K, VJ] + w(XI, XJ, K)}
```

Figure 2.11: Left: The generic iterative algorithm structure consisting of a set of loops and a series of conditional statements. Right: The generic iterative algorithm for the parenthesis problem.

We extend the definition of update relation from Definition 4 as follows.

Definition 18 (*Generic update relation*)**.** *A generic update function is an update that has any of the two forms:*

$$W[w_{d_1} + \ell_{d_1}, \ldots, w_{d_h} + \ell_{d_h}] \leftarrow f(R^1[r_{d_1}^1 + \mathbf{Aff}(\ell_1, \ldots, \ell_h)], \ldots, R^s[r_{d_1}^s + \mathbf{Aff}(\ell_1, \ldots, \ell_h)])$$
$$W[w_{d_1} + \ell_{d_1}, \ldots, w_{d_h} + \ell_{d_h}] \leftarrow W[w_{d_1} + \ell_{d_1}, \ldots, w_{d_h} + \ell_{d_h}]$$
$$\oplus f(R^1[r_{d_1}^1 + \mathbf{Aff}(\ell_1, \ldots, \ell_h)], \ldots, R^s[r_{d_1}^s + \mathbf{Aff}(\ell_1, \ldots, \ell_h)])$$

where, $W, R^1, \ldots, R^s$ are different matrices / submatrices, W means the write matrix and R means the read matrix; $d_1, \ldots, d_h$ are the dimensions where $d_i \in [1, h]$; $w, r^1, \ldots, r^s$ are the write and read regions; $(w_{d_1}, \ldots, w_{d_h})$ is the coordinate of the starting cell of submatrix W, $\mathbf{Aff}(\ell_1, \ldots, \ell_h)$ is the affine function involving the loop parameters $\ell_1, \ldots, \ell_h$; $\oplus$ is an associative operator; and f is any function.

From the structure of the generic iterative algorithm, as shown in Figure 2.11, it is clear that when all the loops execute, the program control touches all points on the n^h hypercube grid, W. At each point of the grid, we check whether the point satisfies any of the conditions $condition_1, \ldots, condition_{last}$. Each point is updated based on the condition it satisfies.

Consider the parenthesis problem as an example. Its generic kernel is shown in Figure 2.11. There is one write region X and two read regions U and V. There are three loops with indices i, j, and k that vary between 0 and $n − 1$ or between $n − 1$ and 0. There are three conditions and when they are satisfied, an assignment statement is run. The assignment statement depends on x_r, x_c, u_r, u_c, v_c and i, j, k, where (z_r, z_c) represents the first row and first column of submatrix Z.

The $\Theta(n^h)$ work required for a DP problem can be mapped to a $\Theta(n^h)$ hypercube grid. Then, the conditions that the indices satisfy are like the half-spaces in the hypercube grid.

The grid points that satisfy all the conditions form a polyhedron in the h-dimensional integer space. For all the points in the polyhedron, the assignment statement will be executed

For the parenthesis problem, we have a 3-D n^3 table i.e., $i, j, k \in [0, n-1]$ with three half-spaces:

- ✯ $x_c + j \geq x_r + i$
- ✯ $u_c + k \geq x_r + i$
- ✯ $x_c + j > u_c + k - 1$

where, (z_r, z_c) is the starting coordinate of region Z, and all coordinates are functions of n.

We can also consider that we have an infinite integer grid $\mathbb{Z}^d$ and we have nine half-spaces: $i \geq 0$, $i \leq n-1$, $j \geq 0$, $j \leq n-1$, $k \geq 0$, $k \leq n-1$, $x_c + j \geq x_r + i$, $u_c + k \geq x_r + i$, and $x_c + j > u_c + k - 1$.

2.6.2 Algorithm-tree construction

If there are h dimensions, we divide the n^h DP table into 2^h orthants, each of size $(n/2)^h$. Each region-tuple writes to one orthant and reads from s othants. Hence, there can be a total of $2^{h(1+s)}$ region-tuples. Assuming we can check whether each region-tuple dependency holds in $\mathcal{O}(1)$ time, we can check all the region-tuple dependencies in $\mathcal{O}\left(2^{h(1+s)}\right)$ time, which is upper bounded by a constant.

Consider the parenthesis problem. Let X denote the DP table. Let $\mathcal{A}(\langle X, X, X \rangle)$ be the first main function. Divide the DP table X into four quadrants: X_{11}, X_{12}, X_{21}, and X_{22}. In this problem, we use 2-D for analysis instead of 3-D. We could also have used 3-D with slight changes. The total number of possible quadrant dependencies are $4 \times 4 \times 4 = 64$, as shown in Figure 2.3. Among these 64 possible quadrant dependencies, only 4 of them exist.

The region-tuple dependencies can be recursively broken down into subregion-tuple dependencies. Also, when we want to analyze the functions called by another function, we combine region-tuples using Rule 1. Continuing in this way, we can create the entire algorithm-tree.

We will now focus on computing the existence of a region dependency. Consider the quadrant dependency $\langle X_{11}, X_{12}, X_{22} \rangle$. Let's say the DP table dimension length is n. Each quadrant dimension length will be $n/2$. Let's set $(X, U, V) = (X_{11}, X_{12}, X_{22})$. For this dependency, $(x_r, x_c) = (0, 0)$, $(u_r, u_c) = (0, n/2)$, and $(v_r, v_c) = (n/2, n/2)$. Now, the region dependency (X_{11}, X_{12}, X_{22}) can be written as code as shown in Figure 2.12. If there is at least one cell dependency that satisfies the three conditions in line 7, then we can say that the region dependency $\langle X_{11}, X_{12}, X_{22} \rangle$ exists. If the three conditions are not satisfied at all or in other words if line 8 is never executed, then the region dependency $\langle X_{11}, X_{12}, X_{22} \rangle$ does not exist. It can be verified that the line 8 will never be executed and hence we have a cross mark for the region dependency in Table 2.3.

Please note that to check the existence of a given region dependency for the parenthesis problem, we need to check whether all the three conditions are satisfied for some values of i, j, k. The algorithm as shown in Figure 2.12 can be written in an integer programming format as follows. Does there exist (i, j, k) such that:

1. $i, j, k \in [0, n/2 - 1]$; and
2. $j \geq i + 1$; $n/2 + k \geq i$; and $j > n/2 + k - 1$

Dependency	Exists?	Dependency	Exists?	Dependency	Exists?	Dependency	Exists?
$\langle X_{11}, X_{11}, X_{11}\rangle$	✓	$\langle X_{12}, X_{11}, X_{11}\rangle$	✗	$\langle X_{21}, X_{11}, X_{11}\rangle$	✗	$\langle X_{22}, X_{11}, X_{11}\rangle$	✗
$\langle X_{11}, X_{11}, X_{12}\rangle$	✗	$\langle X_{12}, X_{11}, X_{12}\rangle$	✓	$\langle X_{21}, X_{11}, X_{12}\rangle$	✗	$\langle X_{22}, X_{11}, X_{12}\rangle$	✗
$\langle X_{11}, X_{11}, X_{21}\rangle$	✗	$\langle X_{12}, X_{11}, X_{21}\rangle$	✗	$\langle X_{21}, X_{11}, X_{21}\rangle$	✗	$\langle X_{22}, X_{11}, X_{21}\rangle$	✗
$\langle X_{11}, X_{12}, X_{22}\rangle$	✗	$\langle X_{12}, X_{12}, X_{22}\rangle$	✓	$\langle X_{21}, X_{12}, X_{22}\rangle$	✗	$\langle X_{22}, X_{12}, X_{22}\rangle$	✗
$\langle X_{11}, X_{12}, X_{11}\rangle$	✗	$\langle X_{12}, X_{12}, X_{11}\rangle$	✗	$\langle X_{21}, X_{12}, X_{11}\rangle$	✗	$\langle X_{22}, X_{12}, X_{11}\rangle$	✗
$\langle X_{11}, X_{12}, X_{12}\rangle$	✗	$\langle X_{12}, X_{12}, X_{12}\rangle$	✗	$\langle X_{21}, X_{12}, X_{12}\rangle$	✗	$\langle X_{22}, X_{12}, X_{12}\rangle$	✗
$\langle X_{11}, X_{12}, X_{21}\rangle$	✗	$\langle X_{12}, X_{12}, X_{21}\rangle$	✗	$\langle X_{21}, X_{12}, X_{21}\rangle$	✗	$\langle X_{22}, X_{12}, X_{21}\rangle$	✗
$\langle X_{11}, X_{12}, X_{22}\rangle$	✗	$\langle X_{12}, X_{12}, X_{22}\rangle$	✗	$\langle X_{21}, X_{12}, X_{22}\rangle$	✗	$\langle X_{22}, X_{12}, X_{22}\rangle$	✗
$\langle X_{11}, X_{21}, X_{11}\rangle$	✗	$\langle X_{12}, X_{21}, X_{11}\rangle$	✗	$\langle X_{21}, X_{21}, X_{11}\rangle$	✗	$\langle X_{22}, X_{21}, X_{11}\rangle$	✗
$\langle X_{11}, X_{21}, X_{12}\rangle$	✗	$\langle X_{12}, X_{21}, X_{12}\rangle$	✗	$\langle X_{21}, X_{21}, X_{12}\rangle$	✗	$\langle X_{22}, X_{21}, X_{12}\rangle$	✗
$\langle X_{11}, X_{21}, X_{21}\rangle$	✗	$\langle X_{12}, X_{21}, X_{21}\rangle$	✗	$\langle X_{21}, X_{21}, X_{21}\rangle$	✗	$\langle X_{22}, X_{21}, X_{21}\rangle$	✗
$\langle X_{11}, X_{21}, X_{22}\rangle$	✗	$\langle X_{12}, X_{21}, X_{22}\rangle$	✗	$\langle X_{21}, X_{21}, X_{22}\rangle$	✗	$\langle X_{22}, X_{21}, X_{22}\rangle$	✗
$\langle X_{11}, X_{22}, X_{11}\rangle$	✗	$\langle X_{12}, X_{22}, X_{11}\rangle$	✗	$\langle X_{21}, X_{22}, X_{11}\rangle$	✗	$\langle X_{22}, X_{22}, X_{11}\rangle$	✗
$\langle X_{11}, X_{22}, X_{12}\rangle$	✗	$\langle X_{12}, X_{22}, X_{12}\rangle$	✗	$\langle X_{21}, X_{22}, X_{12}\rangle$	✗	$\langle X_{22}, X_{22}, X_{12}\rangle$	✗
$\langle X_{11}, X_{22}, X_{21}\rangle$	✗	$\langle X_{12}, X_{22}, X_{21}\rangle$	✗	$\langle X_{21}, X_{22}, X_{21}\rangle$	✗	$\langle X_{22}, X_{22}, X_{21}\rangle$	✗
$\langle X_{11}, X_{22}, X_{22}\rangle$	✗	$\langle X_{12}, X_{22}, X_{22}\rangle$	✗	$\langle X_{21}, X_{22}, X_{22}\rangle$	✗	$\langle X_{22}, X_{22}, X_{22}\rangle$	✓

Table 2.3: Dependencies of quadrants in the parenthesis problem.

```
REGION-DEPENDENCY (X₁₁, X₁₂, X₂₂, n)

1.  for i ← n/2 − 1 to 0 do
2.  for j ← 0 to n/2 − 1 do
3.  for k ← 0 to n/2 − 1 do
4.     XI ← i; XJ ← j
5.     UI ← i; VJ ← n/2 + j
6.     K ← n/2 + k
7.     if XJ ≥ XI + 1 and K ≥ XI and XJ > K − 1 then
8.        C[XI, XJ] ← min{C[XI, XJ], C[UI, K] + C[K, VJ] + w(XI, XJ, K)}
```

Figure 2.12: The looping code for the region-dependency $\langle X_{11}, X_{12}, X_{22}\rangle$.

There are methods solve the problem in time independent of n, i.e., $\mathcal{O}(1)$ time. In this way, we can verify whether a region dependency exists or not in $\mathcal{O}(1)$ time.

2.6.3 Space/time complexity of deductive AUTOGEN

We analyze the space and time complexities of the deductive AUTOGEN using the three parameters h, s, and t, where h is the number of dimensions, $1 + s$ is the cell-tuple size, and t is the threshold level. Let the total number of functions in the output $\mathcal{R}$-$\mathcal{DP}$ be m.

The number of subregion dependencies for a given region dependencies at any level is $2^{h(1+s)}$. If we consider a total of t threshold levels, then the number of region dependencies will be $\Theta\left(m2^{h(1+s)}\right)$. The time taken to check one region dependency is $g(h)$ for some function g. Hence, the total time taken to construct the algorithm tree is $\Theta\left(m2^{h(1+s)}g(h)\right)$, which is a constant. The time taken to construct DAGs is lesser compared to the time taken to create the algorithm tree. Therefore, the total time taken for a deductive AUTOGEN algorithm to generate an $\mathcal{R}$-$\mathcal{DP}$ is $\Theta\left(m2^{h(1+s)}g(h)\right)$. If we run this algorithm serially, then the space usage is $\Theta\left(m2^{1+s}\right)$.

2.7 Experimental results

This section presents empirical results showing the performance benefits and robustness of AUTOGEN-discovered algorithms.

AUTOGEN **prototype.** Given an $\mathcal{I}\text{-}\mathcal{DP}$ implementation for a small problem size of $n = 64$ as input, our prototype implementation outputs an $\mathcal{R}\text{-}\mathcal{DP}$ pseudocode. Note that developing such an $\mathcal{R}\text{-}\mathcal{DP}$ is a one-time process. Our prototype constructs correct $\mathcal{R}\text{-}\mathcal{DP}$s for several problems listed in Table 2.1.

We ran AUTOGEN on Intel Core i5-2410M 2.3 GHz machine with 6GB RAM. Several processes were running along with AUTOGEN. For the parenthesis problem, the time taken by AUTOGEN to generate $\mathcal{R}\text{-}\mathcal{DP}$ (not including printing) pseudocodes is as follows. For input DP table of size 32×32, time taken (average of 5 runs) to generate $\mathcal{R}\text{-}\mathcal{DP}$ was 0.04 seconds. For input DP table of size 64×64, time taken (average of 5 runs) to generate $\mathcal{R}\text{-}\mathcal{DP}$ was 0.292 seconds. The present AUTOGEN program is not optimized at all. With optimizations, AUTOGEN can be made to run even faster.
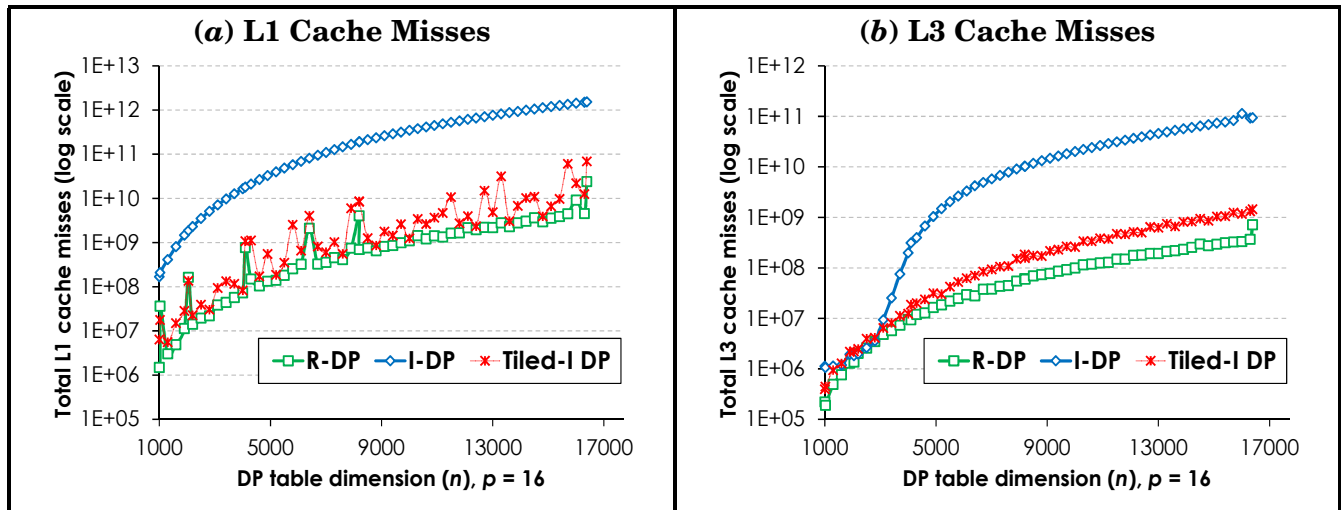
Figure 2.14: The plots show the L1 and L3 cache misses incurred by the three algorithms for solving the parenthesis problem. L2 cache misses are shown in Figure 6.5(b).

2.7.1 Experimental setup

The tiled $\mathcal{I}\text{-}\mathcal{DP}$ and the $\mathcal{R}\text{-}\mathcal{DP}$ algorithms were implemented by Jesmin Jahan Tithi. The implementations All our experiments were performed on a multicore machine with dual-socket 8-core 2.7 GHz[6] Intel Sandy Bridge processors ($2 \times 8 = 16$ cores in total) and 32 GB RAM. Each core was connected to a 32 KB private L1 cache and a 256 KB private L2 cache. All cores in a processor shared a 20 MB L3 cache. All algorithms were implemented in C++. We used Intel Cilk Plus extension to parallelize and Intel[6] C++ Compiler v13.0 to compile all implementations with optimzation parameters `-O3 -ipo -parallel -AVX -xhost`. PAPI 5.3 [PAP,] was used to count cache misses, and the MSR (Model-Specific Register) module

---

[6]All energy, adaptivity and robustness experiments were performed on a Sandy Bridge machine with a processor speed 2.00GHz.
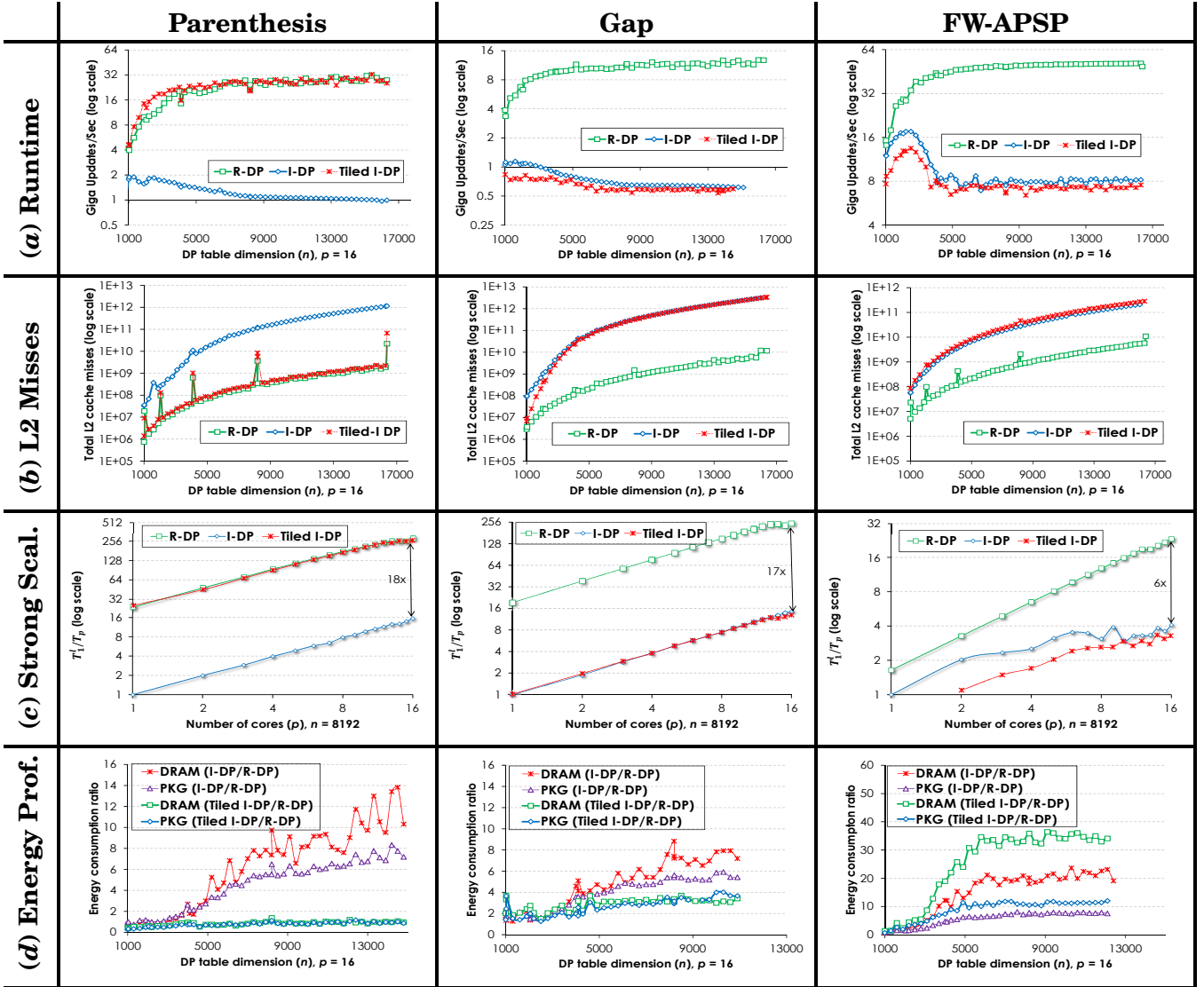
Figure 2.13: Performance comparison of $\mathcal{I}\text{-}\mathcal{DP}$, $\mathcal{R}\text{-}\mathcal{DP}$ and tiled $\mathcal{I}\text{-}\mathcal{DP}$: (*a*) runtime performance: giga updates per second achieved by all algorithms, (*b*) L2 cache misses for each program, (*c*) strong scalability with #cores, p when n is fixed at 8192 (in this plot T_1^l denotes the running time of $\mathcal{I}\text{-}\mathcal{DP}$ when $p = 1$), and (*d*) energy profile: ratios of total joule energy consumed by Package (PKG) and DRAM. Here, tiled $\mathcal{I}\text{-}\mathcal{DP}$ is an optimized version of the parallel tiled code generated by Pluto [Bondhugula et al., 2008].

and `likwid` [Treibig et al., 2010] were used for energy measurements. We used `likwid` for the adaptivity (Figure 2.15) experiments. All `likwid` measurements were end-to-end (i.e., captures everything from the start to the end of the program).

Given an iterative description of a DP in the FRACTAL-DP class, our AUTOGEN prototype generates pseudocode of the corresponding $\mathcal{R}\text{-}\mathcal{DP}$ algorithm in the format shown in Figure 3.1. We implemented such autodiscovered $\mathcal{R}\text{-}\mathcal{DP}$ algorithms for the parenthesis problem, gap problem, and Floyd-Warshall's APSP (2-D). In order to avoid overhead of recursion and increase vectorization efficiency the $\mathcal{R}\text{-}\mathcal{DP}$ implementation switched to an iterative kernel when the problem size became sufficiently small (e.g., when problem size reached 64×64). All our $\mathcal{R}\text{-}\mathcal{DP}$ implementations were the straightforward implementation of the pseudocode

with only *trivial hand-optimizations*, since we wanted to imitate what a potential future AUTOGEN compiler with the capability of implementing the algorithm would generate. With nontrivial hand-optimizations $\mathcal{R}\text{-}\mathcal{DP}$ algorithms can achieve even more speedup (see [Tithi et al., 2015]).

Trivial optimizations include:

✴ [*copy-optimization.*] – copying transpose of a column-major input matrix inside a basecase to a local array, so that it can be accessed in unit stride during actual computation.

✴ [*registers.*] using registers for variables that are accessed many times inside the loops.

✴ [*write optimization in the basecase.*] – if each iteration of an innermost loop updates the same location of the DP table we perform all those updates in a local variable instead of modifying the DP table cell over and over again, and update that cell only once using the updated local variable after the loop terminates.

✴ [*using #pragma directives.*] to auto-vectorize / auto-parallelize code.

Nontrivial optimizations that we did not apply include:

✴ [*using Z-morton row-major layout.*] (see [Tithi et al., 2015]) to store the matrices.

✴ [*using pointer arithmetic.*] and converting all multiplicative indexing to additive indexing.

✴ [*using explicit vectorization.*].

The major optimizations applied on $\mathcal{I}\text{-}\mathcal{DP}$ codes include the following: parallelization, use of pragmas (e.g., `#pragma ivdep` and `#pragma parallel`), use of 64 byte-aligned matrices, write optimizations, pointer arithmetic, and additive indexing.

We used Pluto [Bondhugula et al., 2008] – a state-of-the-art polyhedral compiler – to generate parallel tiled iterative codes for the parenthesis problem, gap problem, and Floyd-Warshall's APSP (2-D). Optimized versions of these codes are henceforth called tiled $\mathcal{I}\text{-}\mathcal{DP}$. After analyzing the autogenerated codes, we found that the parenthesis implementation had temporal locality as it was tiled across all three dimensions, but FW-APSP and gap codes did not as the dependence-based standard tiling conditions employed by Pluto allowed tiling of only two of the three dimensions for those problems. While both parenthesis and FW-APSP codes had spatial locality, the gap implementation did not as it was accessing data in both row- and column-major orders. Overall, for any given cache level the theoretical cache-complexity of the tiled parenthesis code matched that of parenthesis $\mathcal{R}\text{-}\mathcal{DP}$ assuming that the tile size was optimized for that cache level. But tiled FW-APSP and tiled gap had nonoptimal cache complexities. Indeed, the cache complexity of tiled FW-APSP turned out to be $\Theta\left(n^3/B\right)$ matching the cache complexity of its $\mathcal{I}\text{-}\mathcal{DP}$ counterpart. Similarly, the $\Theta\left(n^3\right)$ cache complexity of tiled gap matched that of $\mathcal{I}\text{-}\mathcal{DP}$ gap.

The major optimizations we applied on the parallel tiled codes generated by Pluto include (*i*) use of `#pragma ivdep`, `#pragma parallel`, and `#pragma min loop count(B)` directives; (*ii*) write optimizations (as was used for basecases of $\mathcal{R}\text{-}\mathcal{DP}$); (*iii*) use of empirically determined best tile sizes, and (*iv*) rigorous optimizations using pointer arithmetic, additive indexing, etc. The type of trivial copy optimization we used in $\mathcal{R}\text{-}\mathcal{DP}$ did not improve spatial locality of the autogenerated tiled $\mathcal{I}\text{-}\mathcal{DP}$ for the gap problem as the code did not have any temporal locality. The code generated for FW-APSP had only one parallel loop, whereas two loops could be parallelized trivially. In all our experiments we used two parallel loops for FW-APSP. The direction of the outermost loop of the autogenerated tiled code for the parenthesis problem had to be reversed in order to avoid violation of dependency constraints.

All algorithms we have tested are in-place, that is, they use only a constant number of extra memory/register locations in addition to the given DP table. The copy optimization requires the use of a small local submatrix per thread but its size is also independent of the input DP table size. None of our optimizations reduces space usage. The write optimization avoids directly writing to the same DP table location in the memory over and over again by collecting all those updates in a local register and then writing the final value of the register to the DP cell.

In the following part of the section, we first show performance of $\mathcal{R}$-$\mathcal{DP}$, $\mathcal{I}$-$\mathcal{DP}$ and tiled $\mathcal{I}$-$\mathcal{DP}$ implementations for all three problems when each of the programs runs on a dedicated machine. We show that $\mathcal{R}$-$\mathcal{DP}$ outperforms $\mathcal{I}$-$\mathcal{DP}$ in terms of runtime, scalability, cache-misses, and energy consumption. Next, we show the how the performance of $\mathcal{R}$-$\mathcal{DP}$, $\mathcal{I}$-$\mathcal{DP}$ and tiled $\mathcal{I}$-$\mathcal{DP}$ implementations change in a multiprogramming environment when multiple processes share cache space and bandwidth.

2.7.2 Single-process performance

Figure 6.5 shows detailed performance results of $\mathcal{I}$-$\mathcal{DP}$, tiled $\mathcal{I}$-$\mathcal{DP}$ and $\mathcal{R}$-$\mathcal{DP}$ implementations. For each of the three problems, our $\mathcal{R}$-$\mathcal{DP}$ implementations outperformed its $\mathcal{I}$-$\mathcal{DP}$ counterpart, and for $n = 8192$, the speedup factors w.r.t. parallel $\mathcal{I}$-$\mathcal{DP}$ on 16 cores were around 18, 17 and 6 for parenthesis, gap and Floyd-Warshall's APSP, respectively. For pathenthesis and gap problems $\mathcal{I}$-$\mathcal{DP}$ consumed 5.5 times more package energy and 7.4 times more DRAM energy than $\mathcal{R}$-$\mathcal{DP}$ when $n = 8192$. For Floyd-Warshall's APSP those two factors were 7.4 and 18, respectively.

For the parenthesis problem tiled $\mathcal{I}$-$\mathcal{DP}$ (i.e., our optimized version of Pluto-generated parallel tiled code) and $\mathcal{R}$-$\mathcal{DP}$ had almost identical performance for $n > 6000$. For $n \le 6000$, $\mathcal{R}$-$\mathcal{DP}$ was slower than tiled $\mathcal{I}$-$\mathcal{DP}$, but for larger n, $\mathcal{R}$-$\mathcal{DP}$ was marginally (1 - 2%) faster on average. Observe that though tiled $\mathcal{I}$-$\mathcal{DP}$ and $\mathcal{R}$-$\mathcal{DP}$ had almost similar L2 cache performance, Figure 2.14 shows that $\mathcal{R}$-$\mathcal{DP}$ incurred noticably fewer L1 and L2 cache misses than those incurred by tiled $\mathcal{I}$-$\mathcal{DP}$ which helped $\mathcal{R}$-$\mathcal{DP}$ to eventually fully overcome the overhead of recursion and other implementation overheads. This happened because the tile size of tiled $\mathcal{I}$-$\mathcal{DP}$ was optimized for the L2 cache, but $\mathcal{R}$-$\mathcal{DP}$ being cache-oblivious was able to adapt to all levels of the cache hierarchy simultaneously [Frigo et al., 1999].

As explained in Section 2.7.1 for the gap problem tiled $\mathcal{I}$-$\mathcal{DP}$ had suboptimal cache complexity matching that of $\mathcal{I}$-$\mathcal{DP}$. As a result, tiled $\mathcal{I}$-$\mathcal{DP}$'s performance curves were closer to those of $\mathcal{I}$-$\mathcal{DP}$ than $\mathcal{R}$-$\mathcal{DP}$, and $\mathcal{R}$-$\mathcal{DP}$ outperformed it by a wide margin. Similarly for Floyd-Warshall's APSP. However, in case of gap problem tiled $\mathcal{I}$-$\mathcal{DP}$ incurred significantly fewer L3 misses than $\mathcal{I}$-$\mathcal{DP}$ (not shown in the plots), and as a result, consumed less DRAM energy. The opposite was true for Floyd-Warshall's APSP.

2.7.3 Multi-process performance

$\mathcal{R}$-$\mathcal{DP}$ algorithms are more robust than both $\mathcal{I}$-$\mathcal{DP}$ and tiled $\mathcal{I}$-$\mathcal{DP}$. Our empirical results show that in a multiprogramming environment $\mathcal{R}$-$\mathcal{DP}$ algorithms are less likely to significantly slowdown when the available shared cache/memory space reduces (unlike tiled code with temporal locality), and less likely to suffer when the available bandwidth reduces (unlike standard $\mathcal{I}$-$\mathcal{DP}$ code and tiled $\mathcal{I}$-$\mathcal{DP}$ without temporal locality). Figures 2.15 and
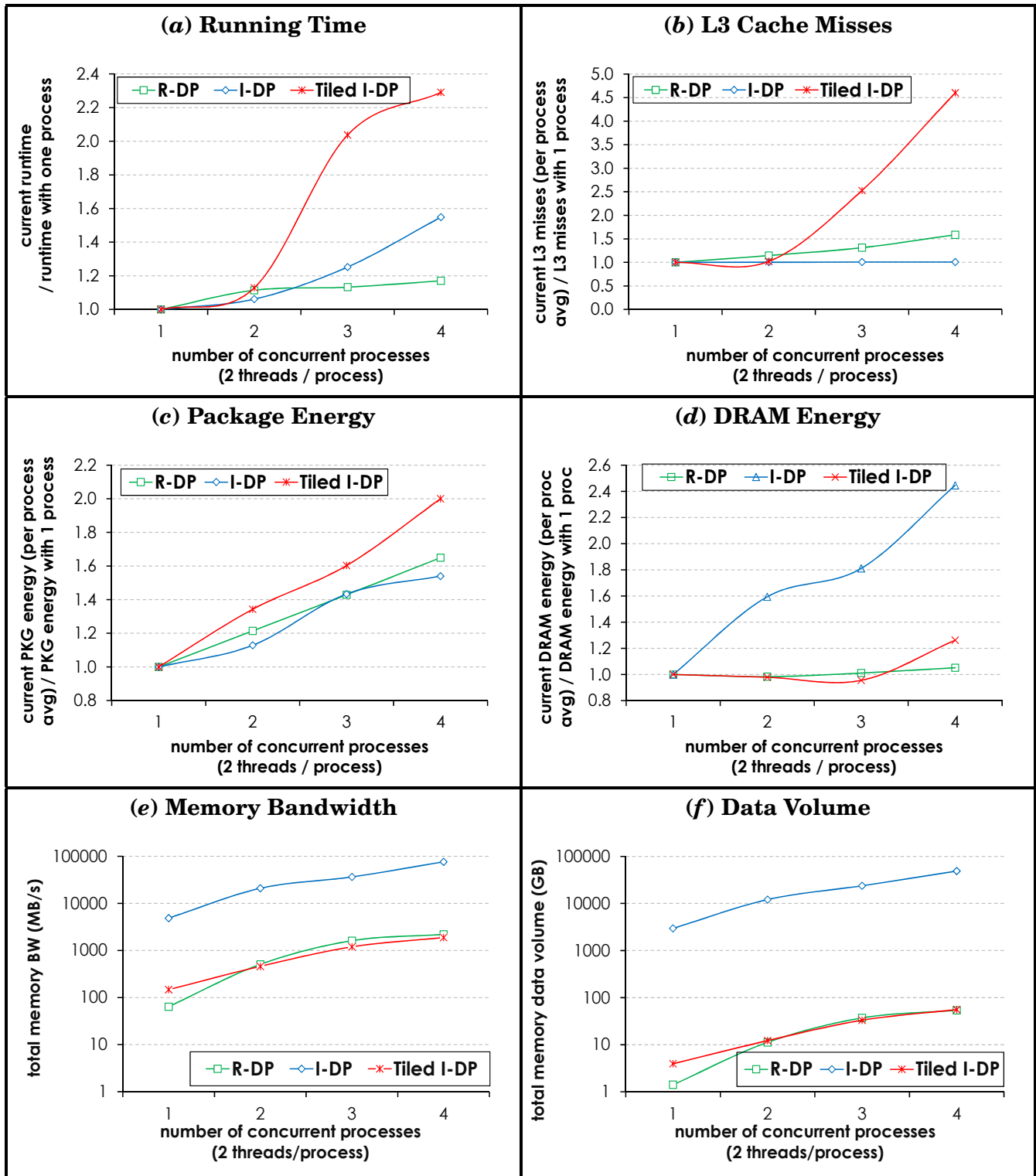
Figure 2.15: The plots show how the performances of standard looping, tiled looping and recursive codes for the parenthesis problem (for $n = 2^{13}$) are affected as multiple instances of the same program are run on an 8-core Intel Sandy Bridge with 20MB shared L3 cache.
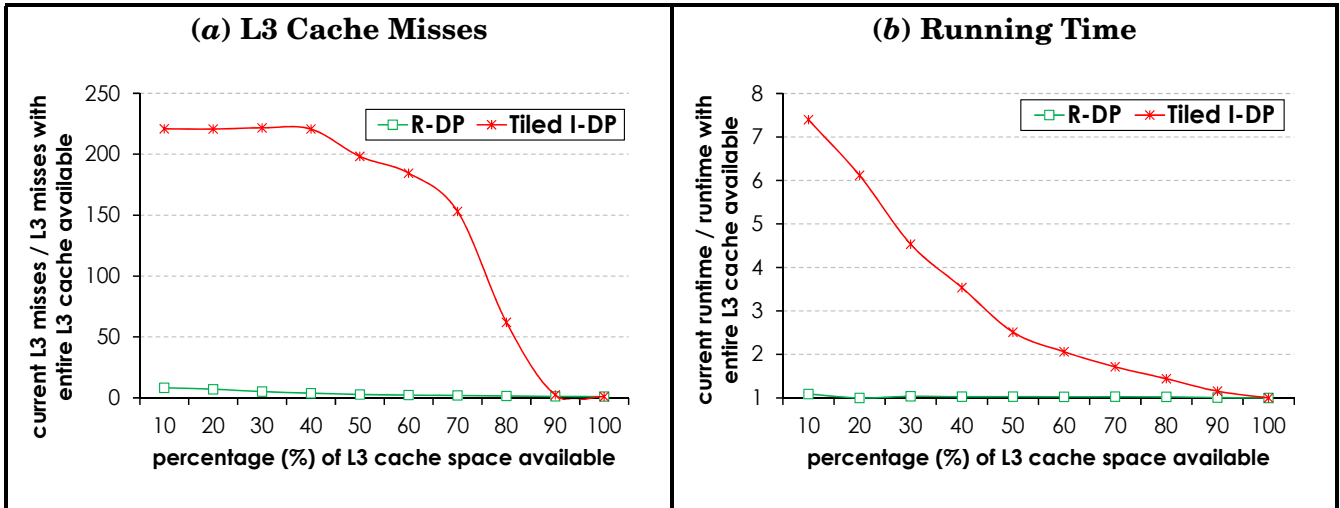
Figure 2.16: The plots show how changes in the available shared L3 cache space affect (a) the number of L3 cache misses, and (b) the serial running time of the tiled looping code and the recursive code solving the parenthesis problem for $n = 2^{13}$. The code under test was run on a single core of an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache. A multi-threaded `Cache Pirate` [Eklov et al., 2011] was run on the remaining cores.

2.16 show the results. For lack of space we have included only results for the parenthesis problem. We have seen similar trends for our other benchmark problems (e.g., FW-APSP).

We have performed experimental analyses of how the performance of a program ($\mathcal{R}$-$\mathcal{DP}$, $\mathcal{I}$-$\mathcal{DP}$, and tiled $\mathcal{I}$-$\mathcal{DP}$) changes if multiple copies of the same program are run on the same multicore processor (Figure 2.15). We ran up to 4 instances of the same program on an 8-core Sandy Bridge processor with 2 threads (i.e., cores) per process. The block size of the tiled code was optimized for best performance with 2 threads. With 4 concurrent processes $\mathcal{I}$-$\mathcal{DP}$ slowed down by 82% and tiled $\mathcal{I}$-$\mathcal{DP}$ by 46%, but $\mathcal{R}$-$\mathcal{DP}$ lost only 17% of its performance (see Figure 2.15). The slowdown of the tiled code resulted from its inability to adapt to the loss in the shared cache space which increased its L3 misses by a factor of 4 (see Figure 2.15). On the other hand, L3 misses incurred by $\mathcal{R}$-$\mathcal{DP}$ increased by less than a factor of 2.5. Since $\mathcal{I}$-$\mathcal{DP}$ does not have any temporal locality, loss of cache space did not significantly change the number of L3 misses it incurred. But $\mathcal{I}$-$\mathcal{DP}$ already incurred 90 times more L3 misses than $\mathcal{R}$-$\mathcal{DP}$, and with 4 such concurrent processes the pressure on the DRAM bandwidth increased considerably (see Figure 2.15) causing significant slowdown of the program.

We also report changes in energy consumption of the processes as the number of concurrent processes increases (Figure 2.15). Energy values were measured using `likwid-perfctr` (included in `likwid`) which reads them from the MSR registers. The energy measurements were end-to-end (start to end of the program). Three types of energy were measured: `package` energy which is the energy consumed by the entire processor die, `PP0` energy which is the energy consumed by all cores and private caches, and finally `DRAM` energy which is the energy consumed by the directly-attached DRAM. We omitted the PP0 energy since the curves almost always look similar to that of package energy. A single instance of tiled $\mathcal{I}$-$\mathcal{DP}$ consumed 5% less energy than an $\mathcal{R}$-$\mathcal{DP}$ instance while $\mathcal{I}$-$\mathcal{DP}$ consumed 9 times more energy. Average package and PP0 energy consumed by tiled $\mathcal{I}$-$\mathcal{DP}$ increased at

a faster rate than that by $\mathcal{R}$-$\mathcal{DP}$ as the number of processes increased. This happened because both its running time and L3 performance degraded faster than $\mathcal{R}$-$\mathcal{DP}$ both of which contribute to energy performance. However, since for $\mathcal{I}$-$\mathcal{DP}$ L3 misses did not change much with the increase in the number of processes, its package and PP0 energy consumption increased at a slower rate compared to $\mathcal{R}$-$\mathcal{DP}$'s when number of processes is less than 3. However, as the number of processes increases, energy consumption increases for $\mathcal{I}$-$\mathcal{DP}$ at a faster rate, and perhaps because of the DRAM bandwidth contention its DRAM energy increased significantly.

We have measured the effect on running times and L3 cache misses of serial $\mathcal{R}$-$\mathcal{DP}$ and serial tiled $\mathcal{I}$-$\mathcal{DP}$[7] when the available shared L3 cache space is reduced (shown in Figure 2.16). In this case, the serial tiled-$\mathcal{I}$-$\mathcal{DP}$ algorithm was running around 50% faster than the serial $\mathcal{R}$-$\mathcal{DP}$ code. The `Cache Pirate` tool [Eklov et al., 2011] was used to steal cache space[8]. When the available cache space was reduced to 50%, the number of L3 misses incurred by the tiled code increased by a factor of 22, but for $\mathcal{R}$-$\mathcal{DP}$ the increase was only 17%. As a result, the tiled $\mathcal{I}$-$\mathcal{DP}$ slowed down by over 50% while for $\mathcal{R}$-$\mathcal{DP}$ the slowdown was less than 3%. Thus $\mathcal{R}$-$\mathcal{DP}$ automatically adapts to cache sharing [Bender et al., 2014], but the tiled $\mathcal{I}$-$\mathcal{DP}$ does not. This result can be found in the second column of Figure 2.15.

2.8 Conclusion and open problems

We presented the AUTOGEN algorithm to autogenerate highly efficient recursive algorithms for a wide variety of DP problems from their iterative algorithms. We believe that AUTOGEN is the first step towards building a system to automate the process of developing efficient algorithms for DP problems.

Here are a few open problems:

✯ [*AUTOGEN for distributed systems*.] Design AUTOGEN to autogenerate $\mathcal{R}$-$\mathcal{DP}$ algorithms for distributed systems.

✯ [*AUTOGEN for planar graphs*.] The real-life graphs such as routing networks in Internet, transport networks of geographical information systems, and telephone networks are almost planar graphs and it seems possible that we can develop algorithms to exploit temporal locality by working on a localized set of nodes. Design AUTOGEN to autogenerate algorithms for planar graphs.

✯ [*AUTOGEN for r-way $\mathcal{R}$-$\mathcal{DP}$s*.] Design AUTOGEN to autogenerate r-way $\mathcal{R}$-$\mathcal{DP}$s. Note that, here, r is not a fixed constant but a parameter that can be set at the runtime.

✯ [*AUTOGEN for irregular DPs*.] Design AUTOGEN to handle irregular dependencies. E.g.: Viterbi algorithm, knapsack problem, and sieve of Eratosthenes.

✯ [*AUTOGEN for tiled $\mathcal{I}$-$\mathcal{DP}$s*.] Design AUTOGEN to autogenerate tiled-iterative DP algorithms.

✯ [*AUTOGEN for wavefront $\mathcal{I}$-$\mathcal{DP}$s*.] Design AUTOGEN to autogenerate iterative wavefront DP algorithms.

✯ [*AUTOGEN for partitioned global address space (PGAS) model*.] Design AUTOGEN to autogenerate algorithms for the PGAS model.

✯ [*AUTOGEN for graphics processing units (GPUs)*.] Design AUTOGEN to autogenerate algorithms that can be run on GPUs.

---

[7]with tile size optimized for best serial performance
[8]`Cache Pirate` allows only a single program to run, and does not reduce bandwidth.

✶ [*Autogen for MapReduce.*] Design Autogen to autogenerate algorithms that can use MapReduce.

✶ [*Autogen of Autogen.*] Autogen automatically discovers algorithms for a class of problems. Hence, it can be thought of as a level-2 algorithm. What we want is to design a level-2 Autogen or a level-3 algorithm. Is it possible to design an Autogen that can autogenerate Autogens for specific classes of problems and/or hardware?

Chapter 3

Semi-Automatic Discovery of Efficient Divide-&-Conquer Wavefront DP Algorithms

Iterative wavefront algorithms for evaluating dynamic programming recurrences exploit very high parallelism but show poor cache performance. Tiled-iterative wavefront algorithms achieve optimal cache complexity and high parallelism but are cache-aware and hence are not portable and not cache-adaptive. On the other hand, standard cache-oblivious recursive divide-and-conquer algorithms have optimal serial cache complexity but often have low parallelism due to artificial dependencies among subtasks.

Very recently we introduced cache-oblivious recursive divide-and-conquer wavefront (COW) algorithms that do not have any artificial dependencies, but they are too complicated to develop, analyze, implement, and generalize. Though COW algorithms are based on fork-join primitives, they extensively use atomic operations for ensuring correctness, and as a result, performance guarantees (i.e., parallel running time and parallel cache complexity) provided by state-of-the-art schedulers (e.g., the randomized work-stealing scheduler) for programs with fork-join primitives do not apply. Also extensive use of atomic locks may result in high overhead in implementation.

In this chapter, we show how to systematically transform standard cache-oblivious recursive divide-and-conquer algorithms into recursive divide-and-conquer wavefront algorithms to achieve optimal parallel cache complexity and high parallelism under state-of-the-art schedulers for fork-join programs. Unlike COW algorithms these new algorithms do not use atomic operations. Instead they use timing functions to compute at what time each divide-and-conquer function must be launched in order to achieve high parallelism without losing cache performance. The resulting implementations are arguably much simpler than those of known COW algorithms. We present theoretical analyses and experimental performance and scalability results showing superiority of these new algorithms over existing algorithms.

3.1   Introduction

For good performance on a modern multicore machine with a cache hierarchy, algorithms must have good parallelism and should be able to use the caches efficiently at the same

time. Iterative wavefront algorithms for solving DP problems have optimal parallelism but often suffer due to bad cache performance. On the other hand, though standard cache-oblivious [Frigo et al., 1999] recursive divide-and-conquer DP algorithms have optimal serial cache complexity, they often have low parallelism. The tiled-iterative wavefront algorithms achieve optimality in cache complexity and achieve high parallelism but are cache-aware, and hence are not portable and do not adapt well when available cache space fluctuates during execution in a multiprogramming environment. Very recently, the *cache-oblivious wavefront (COW) algorithms* [Tang et al., 2015] have been proposed that have optimal parallelism and optimal serial cache complexity. However, though those algorithms are based on fork-join primitives, they extensively use atomic operations for correctness. But current theory of scheduling nested parallel programs with fork-join primitives do not allow such atomic operations. As a result, no bounds on parallel running time and parallel cache complexity could be proved for those algorithms. Those algorithms are also very difficult to implement since they require hacking into a parallel runtime system. Extensive use of atomic locks causes too much overhead for very large and higher dimensional DPs.

In this chapter, we present a provably efficient method for scheduling cache-oblivious recursive divide-and-conquer wavefront algorithms on a multicore machine which optimizes parallel cache complexity and achieves high parallelism. Our algorithms are based on fork-join primitives, but do not use atomic operations. As a result, we are able to analyze their parallel running times and parallel cache complexities easily under state-of-the-art schedulers for fork-join based parallel programs. Our algorithms are also much simpler to implement compared to COW algorithms.

Iterative algorithms. Traditionally, DP algorithms are implemented iteratively using a series of (nested) loops and they can be parallelized easily. They are called $\mathcal{I}$-$\mathcal{DP}$s. These algorithms often have good spatial locality, no temporal locality, and standard implementations may not have optimal parallelism as well. For example, an iterative algorithm for the parenthesis problem (explained in Section 2.2) has $T_\infty(n) = \Theta(n^2)$ and $Q_1(n) = \Theta(n^3)$.

Iterative algorithms are also implemented as tiled loops, in which case the entire DP table is blocked or tiled and the tiles are executed iteratively. They are called $\mathcal{TI}$-$\mathcal{DP}$s. For example, for a tiled iterative algorithm for the parenthesis problem with $r \times r$ tile size, where $r \in [2, n]$, we have $T_\infty(n) = \Theta((n/r)^2) \cdot \Theta(r^2) = \Theta(n^2)$, and $Q_1(n, r) = (n/r)^3 \cdot \mathcal{O}(r^2/B + r) = \mathcal{O}(n^3/(rB) + n^3/r^2)$.

Fastest iterative DP implementations have the following wavefront-like property. Say a cell x in a DP table is written by reading from the cell $\langle y_1, y_2, \ldots, y_s \rangle$. When the cells $y_1, y_2, \ldots, y_s$ are completely updated, then the cell x can immediately get updated, either partially or fully. This property leads to the computation of a DP table in a wavefront manner. We call wavefront iterative algorithms $\mathcal{WI}$-$\mathcal{DP}$s. For example, for the parenthesis problem, an iterative wavefront algorithm has the shortest span of $T_\infty(n) = \Theta(n \log n)$, but has the worst possible cache complexity of $Q_1(n) = \mathcal{O}(n^3)$. The $\Theta(\log n)$ factor in the span comes from the parallel for construct.

Recursive divide-and-conquer algorithms. Cache-oblivious recursive divide-and-conquer parallel DP algorithms can overcome many of the limitations of their iterative counterparts. While iterative algorithms often have poor or no temporal locality, recursive algorithms have excellent and often optimal temporal locality. One problem with recursive divide-and-conquer algorithms is that they trade off parallelism for cache optimality, and

thus may end up with high parallelism.

For example, a 2-way recursive divide-and-conquer algorithm (where, each dimension of the subtask will be half the dimension of its parent task) for the parenthesis problem has a span of $T_\infty(n) = \Theta\left(n^{\log_2 3}\right)$ and $Q_1(n) = \Theta\left(n^3/(B\sqrt{M})\right)$, that is, it has optimal serial cache complexity but high span. For n-way recursive algorithm, $T_\infty(n) = \Theta(n\log n)$ and $Q_1(n) = \mathcal{O}(n^3)$. This time, the algorithm has very low span but worse serial cache complexity. Ideally we want to have a balance between cache complexity and span by choosing r-way recursive algorithm in which case both the span and the parallel cache complexity will be non-optimal, however will have best practical performance. For r-way recursive algorithm, we get (from Section A.2)

$$Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\left\lceil \frac{\sqrt{M}}{r}\right\rceil} + \frac{n^3}{\left\lceil \frac{\sqrt{M}}{r}\right\rceil^3} + \frac{n^2}{B} + \frac{n^2}{\left\lceil \frac{\sqrt{M}}{r}\right\rceil^2} + \frac{n}{B}\left\lceil \frac{\sqrt{M}}{r}\right\rceil + \frac{n}{\left\lceil \frac{\sqrt{M}}{r}\right\rceil}\right)$$

$$T_\infty(n) = \mathcal{O}\left(rn^{\log_r\left(2-\frac{1}{r}\right)} + \frac{n^{\log_r(2r-1)}}{r} + nr\right)$$

Limitation(s) of the existing algorithms. The standard iterative algorithms often have worse serial cache complexity and not high parallelism. The wavefront iterative algorithms have highest parallelism but worse serial cache complexity. The tiled iterative algorithms have excellent and often optimal serial cache complexity and very high parallelism. However, the tiled iterative algorithms are cache-aware and hence they are not portable across different machine architectures.

The recursive divide-and-conquer algorithms have excellent and often optimal serial cache complexity and have good but not very high parallelism. The algorithms and their limitations are summarized in Table 3.1.

In Table 3.1, some bounds are excellent, which may or may not mean "optimal". Proving optimality require rigorous mathematical proofs.

Limitation of the recursive divide-and-conquer algorithms. We want to develop algorithms that simultaneously achieves excellent serial cache complexity, excellent parallelism, and portability. To this end, we either need to improve the existing algorithms inheriting their structures or develop new algorithms that have totally different structures than the existing ones.

By careful observation it seems possible to develop new algorithms by improving upon our standard 2-way recursive divide-and-conquer algorithms. If we can analyze the prob-

Algorithm	Algorithm	$Q_1(n)$	$T_1(n)/T_\infty(n)$	Portability
$\mathcal{I}$-$\mathcal{DP}$	Standard iterative	Worse	Good	Yes
$\mathcal{WI}$-$\mathcal{DP}$	Wavefront iterative	Worse	Excellent	Yes
$\mathcal{TI}$-$\mathcal{DP}$	Tiled iterative	Excellent	Good	No
$\mathcal{R}$-$\mathcal{DP}$	Standard recursive	Excellent	Good	Yes
$\mathcal{WR}$-$\mathcal{DP}$	Wavefront recursive	Excellent	Excellent	Yes

Table 3.1: Limitations of existing DP algorithms. Note that excellent does not necessarily mean optimal.

lem behind not very high parallelism for recursive divide-and-conquer algorithms and solve that problem, it seems that we can develop new algorithms.

The low parallelism in 2-way recursive divide-and-conquer algorithms results from artificial dependencies among subproblems that are not implied by the underlying DP recurrence [Tang et al., 2015].

A 2-way recursive divide-and-conquer algorithm for the LCS problem splits the DP table X into four equal quadrants: X_{11} (top-left), X_{12} (top-right), X_{21} (bottom-left), and X_{22} (bottom-right). It then recursively computes the quadrants in the following order: X_{11} first, then X_{12} and X_{21} in parallel, and finally X_{22}. As per the recursive structure, the top-left quadrant of X_{12} and X_{21} i.e., $X_{12,11}$ and $X_{21,11}$, respectively, can only start executing when the execution of the bottom-right quadrant of X_{11} i.e., $X_{11,11}$ completes. This dependency between subproblems or subtasks is not defined by the underlying DP recurrence but defined by the recursive structure of the algorithm. Such dependencies in a recursive algorithm are called *artificial dependencies*. There are artificial dependencies at several different granularities. Most often, these artificial dependencies asymptotically increase the span thereby reducing parallelism. By removing such artificial dependencies but retaining the recursive structure we can design algorithms to achieve high parallelism.

Recursive divide-and-conquer wavefront algorithms. The 2-way recursive algorithms have optimal serial cache complexity and not very high parallelism. On the other hand, the wavefront iterative algorithms have high parallelism and worse cache complexity. Is it possible to inherit the best features from both the worlds? Yes.

By removing artificial dependencies from the recursive algorithms, it is possible to develop algorithms that simultaneously achieve excellent cache-locality, excellent parallelism, and cache-obliviousness. Such algorithms are called *recursive wavefront* or *divide-and-conquer wavefront* or *cache-oblivious wavefront algorithms*.

Recursive wavefront algorithms were introduced in [Tang et al., 2015]. Such algorithms keep several processors busy-waiting and hence wastes CPU resources. Also, those algorithms are too complicated to develop, analyze, implement, and generalize. Atomic instructions were used extensively to identify and launch ready tasks, and implementations required hacking into Cilk's runtime system. No bounds on parallel cache complexities of those algorithms are known.

The AUTOGEN-WAVE framework and scheduling algorithms. In this chapter, we present a framework called AUTOGEN-WAVE to discover recursive wavefront algorithms based on timing functions. These algorithms have a structure similar to the standard recursive divide-and-conquer algorithms, but each recursive function call is annotated with start-time and end-time hints that are passed to the scheduler. Timestamps represent the relative time at which a task starts or ends execution. Closed-form formulas or functions to get the exact timestamps at which different recursive functions write to different regions of the DP table are found. Such closed-form formulas or functions to compute the timestamps are plugged into the recursive algorithm for all the function invocations to derive a recursive wavefront algorithm.

The performance of a parallel algorithm does not depend on the algorithm alone, it also depends on the scheduler that schedules different threads to work on different processors. The recursive wavefront algorithms make use of timestamps and existing schedulers do not understand timestamps. Hence, we present a space-efficient scheduling method to attain

$\mathcal{A}(X,X,X)$
1. **if** X is a cell **then** $\mathcal{A}_{cell}(X,X,X)$
2. **else**
3.   **parallel:** $\mathcal{A}(X_{11},X_{11},X_{11})$, $\mathcal{A}(X_{22},X_{22},X_{22})$
4. $\mathcal{B}(X_{12},X_{11},X_{22})$

$\mathcal{B}(X,U,V)$
1. **if** X is a cell **then** $\mathcal{B}_{cell}(X,U,V)$
2. **else**
3. $\mathcal{B}(X_{21},U_{22},V_{11})$
4.   **parallel:** $\mathcal{C}(X_{11},U_{12},V_{21})$, $\mathcal{C}(X_{22},X_{21},V_{12})$
5.   **parallel:** $\mathcal{B}(X_{11},U_{11},V_{11})$, $\mathcal{B}(X_{22},X_{22},V_{22})$
6. $\mathcal{C}(X_{12},U_{12},X_{22})$
7. $\mathcal{C}(X_{12},X_{11},V_{12})$
8. $\mathcal{B}(X_{12},U_{11},V_{22})$

$\mathcal{C}(X,U,V)$
1. **if** X is a cell **then** $\mathcal{C}_{cell}(X,U,V)$
2. **else**
3.   **parallel:** $\mathcal{C}(X_{11},U_{11},V_{11})$, $\mathcal{C}(X_{12},U_{11},V_{12})$,
          $\mathcal{C}(X_{21},U_{21},V_{11})$, $\mathcal{C}(X_{22},U_{21},V_{12})$
4.   **parallel:** $\mathcal{C}(X_{11},U_{12},V_{21})$, $\mathcal{C}(X_{12},U_{12},V_{22})$,
          $\mathcal{C}(X_{21},U_{22},V_{21})$, $\mathcal{C}(X_{22},U_{22},V_{22})$

Programmer computes the timing functions

$\mathcal{S}_{\mathcal{A}}(X,X,X)$
1. **return** $\mathfrak{C}(x_r,x_c)$

$\mathcal{S}_{\mathcal{B}}(X,U,V)$
1. **return** $\mathfrak{C}(x_r+n-1,x_c)$

$\mathcal{S}_{\mathcal{C}}(X,U,V)$
1. $m \leftarrow (x_r+n-1+x_c)/2$; $\hat{u} \leftarrow u_c+n-1$
2. **if** $u_c > m$ **then return** $\max\{\mathfrak{C}(u_r+n-1,u_c),\mathfrak{C}(u_c,x_c)\}+1$
3. **elif** $\hat{u} < m$ **then return** $\max\{\mathfrak{C}(u_r+n-1,\hat{u}),\mathfrak{C}(\hat{u},x_c)\}+1$
4. **else return**
      $(\max\{\mathfrak{C}(u_r+n-1,m),\mathfrak{C}(m,x_c)\}+1).[u_c > (x_r+x_c)/2]$

$\mathcal{E}_{\mathcal{A}}(X,X,X)$
1. **return** $\mathfrak{C}(x_r,x_c+n-1)$

$\mathcal{E}_{\mathcal{B}}(X,U,V)$
1. **return** $\mathfrak{C}(x_r,x_c+n-1)$

$\mathcal{E}_{\mathcal{C}}(X,U,V)$
1. $lval \leftarrow \max\{\mathfrak{C}(u_r,u_c),\mathfrak{C}(u_c,x_c+n-1)\}$
2. $rval \leftarrow \max\{\mathfrak{C}(u_r,u_c+n-1),\mathfrak{C}(u_c+n-1,x_c+n-1)\}$
3. **return** $(\max\{lval,rval\}+1).[u_c > (x_r+x_c)/2]$

$\mathfrak{C}(i,j)$
1. **if** $(j-i) \leq 1$ **then return** $(j-i)$ **else return** $2\times(j-i)-1$

Transformation by the scheduler / programmer

RECURSIVE-WAVEFRONT-PARENTHESIS()
1. $w \leftarrow 0$
2. **while** $w < \infty$ **do** $w \leftarrow \mathcal{A}(G,G,G,w)$

$\mathcal{A}(X,X,X,w)$
1. $v_i \leftarrow \infty$ for all $i \in [1,3]$
2. **if** X is an $n' \times n'$ matrix **then**
3.   **if** $w = \mathcal{S}_{\mathcal{A}}(X,X,X)$ **then** $\mathcal{A}_{chunk}(X,X,X)$
4. **else**
5. $\mathcal{F}_{1..3} \leftarrow \{\mathcal{A},\mathcal{A},\mathcal{B}\}$
6. $arg_{1..3} \leftarrow \{(X_{11},X_{11},X_{11}),(X_{22},X_{22},X_{22}),\}$
          (X_{12},X_{11},X_{22})
7.   **parallel for** $i \leftarrow 1$ **to** 3 **do**
8.   **if** $w < \mathcal{S}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{S}_{\mathcal{F}_i}(arg_i)$
9.   **elif** $w \leq \mathcal{E}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{F}_i(arg_i,w)$
10.   **sync**
11. **return** $\min v_i$ for all $i \in [1,3]$

$\mathcal{B}(X,U,V,w)$
1. $v_i \leftarrow \infty$ for all $i \in [1,8]$
2. **if** X is an $n' \times n'$ matrix **then**
3.   **if** $w = \mathcal{S}_{\mathcal{B}}(X,U,V)$ **then** $\mathcal{B}_{chunk}(X,U,V)$
4. **else**
5. $\mathcal{F}_{1..8} \leftarrow \{\mathcal{B},\mathcal{C},\mathcal{C},\mathcal{B},\mathcal{B},\mathcal{C},\mathcal{C},\mathcal{B}\}$
6. $arg_{1..8} \leftarrow \{(X_{21},U_{22},V_{11}),(X_{11},U_{12},V_{21}),$
          $(X_{22},X_{21},V_{12}),(X_{11},U_{11},V_{11}),$
          $(X_{22},X_{22},V_{22}),(X_{12},U_{12},X_{22}),$
          $(X_{12},X_{11},V_{12}),(X_{12},U_{11},V_{22})\}$
7.   **parallel for** $i \leftarrow 1$ **to** 8 **do**
8.   **if** $w < \mathcal{S}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{S}_{\mathcal{F}_i}(arg_i)$
9.   **elif** $w \leq \mathcal{E}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{F}_i(arg_i,w)$
10.   **sync**
11. **return** $\min v_i$ for all $i \in [1,8]$

$\mathcal{C}(X,U,V,w)$
1. $v_i \leftarrow \infty$ for all $i \in [1,8]$
2. **if** X is an $n' \times n'$ matrix **then**
3.   **if** $w = \mathcal{S}_{\mathcal{C}}(X,U,V)$ **then** $\mathcal{C}_{chunk}(X,U,V)$
4. **else**
5. $\mathcal{F}_{1..8} \leftarrow \{\mathcal{C},\mathcal{C},\mathcal{C},\mathcal{C},\mathcal{C},\mathcal{C},\mathcal{C},\mathcal{C}\}$
6. $arg_{1..8} \leftarrow \{(X_{11},U_{11},V_{11}),(X_{12},U_{11},V_{12}),$
          $(X_{21},U_{21},V_{11}),(X_{22},U_{21},V_{12}),$
          $(X_{11},U_{12},V_{21}),(X_{12},U_{12},V_{22}),$
          $(X_{21},U_{22},V_{21}),(X_{22},U_{22},V_{22})\}$
7.   **parallel for** $i \leftarrow 1$ **to** 8 **do**
8.   **if** $w < \mathcal{S}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{S}_{\mathcal{F}_i}(arg_i)$
9.   **elif** $w \leq \mathcal{E}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{F}_i(arg_i,w)$
10.   **sync**
11. **return** $\min v_i$ for all $i \in [1,8]$

Figure 3.1: Left: The programmer derives the timing functions from a given standard 2-way recursive divide-and-conquer DP algorithm for the parenthesis problem. A region Z has its top-left corner at (z_r, z_c) and is of size $n \times n$. Right: A recursive divide-and-conquer wavefront algorithm is generated for the parenthesis problem. The programmer derives the algorithm if work-stealing scheduler (see Section 3.4.1) is used and the scheduler derives the algorithm if W-SB scheduler (see Section 3.4.2) is used. The algorithm makes use of the timing functions derived by the programmer.

excellent parallel cache complexity and very high parallelism. Our approach makes use of a combination of ideas from space-bounded scheduler and scheduling based on timestamps.

We also analyze the complexities under a randomized work-stealing scheduler.

Our task schedulers make sure that the algorithms are executed in a wavefront fashion using the timing functions. Indeed, the transformation the scheduler is expected to do based on the timing functions is straightforward, and a programmer may choose to do that herself and use a scheduler that do not accept hints. The transformed code is still purely based on fork-join parallelism, and the performance bounds (e.g., parallel running time and parallel cache complexity) guaranteed by any scheduler supporting fork-join parallelism apply. The recursive wavefront algorithm for the parenthesis problem has $T_\infty(n) = \Theta\left(n \log n\right)$ and $Q_1(n) = \mathcal{O}\left(n^3/(B\sqrt{M})\right)$. The bounds on T_p and Q_p can be obtained from the scheduler guarantees.

The theoretical performance of several algorithms for the parenthesis problem are summarized in Table 3.2. In the table, $\mathcal{I}$-$\mathcal{DP}$ represents standard iterative, $\mathcal{TI}$-$\mathcal{DP}$ represents tiled iterative, $\mathcal{WI}$-$\mathcal{DP}$ represents iterative wavefront, $\mathcal{R}$-$\mathcal{DP}$ represents recursive divide-and-conquer, and $\mathcal{WR}$-$\mathcal{DP}$ recursive divide-and-conquer wavefront algorithms.

Related work. The tiled iterative algorithms [Wolf, 1992, Darte et al., 1997, Panda et al., 1999, Sarkar and Megiddo, 2000, Goumas et al., 2001, Renganarayanan et al., 2012] have been studied extensively as tiling is the traditional way of implementing dynamic programming and other matrix algorithms. There are several frameworks to automatically produce tiled codes such as PLuTo [Bondhugula et al., 2008], Polly [Grosser et al., 2012], and PoCC [Pouchet et al., 2010]. However, these softwares are not designed to generate correct parallel tiled code for non-trivial DP recurrences. The major concerns with tiled programs are that they are cache-aware and sometimes processor-aware that sacrifices portability across machines. Another disadvantage of being cache-aware is that the algorithms are not cache-adaptive [Bender et al., 2014], i.e., the algorithms do not adapt to changes in available cache/memory space during execution and hence may run slower when multiple programs run concurrently in a shared-memory environment [Chowdhury et al., 2016b]. For example in applications such as cloud technologies and virtual networks in data centers, available memory changes continuously.

Several existing systems such as Bellman's GAP compiler [Giegerich and Sauthoff, 2011], semi-automatic synthesizer [Pu et al., 2011], EasyPDP [Tang et al., 2012], EasyHPS [Du et al., 2013], pattern-based system [Liu and Schmidt, 2004], and parallelizing plug-ins [Reitzig, 2012] can be used to generate iterative and tiled loop programs. Parallel task graph execution systems such as Nabbit [Agrawal et al., 2010] and BDDT [Tzenakis et al., 2013] execute the DP tasks during runtime using unrolling. Due to this they might lose cache efficiency.

The classic 2-way recursive divide-and-conquer algorithms having optimal serial cache complexity and good (but, not always optimal) parallelism have been developed, analyzed, and implemented in [Chowdhury and Ramachandran, 2008, Chowdhury and Ramachandran, 2010], [Tithi et al., 2015]. Hybrid r-way algorithms are considered in [Chowdhury and Ramachandran, 2008] but they are either cache- or processor-aware and complicated to program. Pochoir [Tang et al., 2011] is used to generate cache-oblivious implementations to stencils. However, the recursive algorithms often have low parallelism due to artificial dependencies among subtasks. Recently Aga et al. [Aga et al., 2015] proposed a speculation approach to alleviate the concurrency constraints imposed by the artificial dependencies in standard parallel recursive divide-and-conquer programs and reported a speedup up to $1.6\times$ on 30 cores over their baseline.

Algorithm	Sch.	$Q_1(n)$	$T_\infty(n)$	$Q_1^{opt}(n)$	$Q_p^{opt}(n)$	High par.
$\mathcal{I}\text{-}\mathcal{DP}$	WS	$\mathcal{O}\left(n^3\right)$	$\Theta\left(n^2\right)$	✗	✗	✗
r-size $\mathcal{TI}\text{-}\mathcal{DP}$	WS	$\mathcal{O}\left(n^3/(rB) + n^3/r^2\right)$	$\Theta\left(n^2\right)$	✗	✗	✗
$\sqrt{M}$-size $\mathcal{TI}\text{-}\mathcal{DP}$	WS	$\mathcal{O}\left(n^3/(B\sqrt{M}) + n^3/M\right)$	$\Theta\left(n^2\right)$	✓	✗	✗
$\mathcal{WI}\text{-}\mathcal{DP}$	WS	$\mathcal{O}\left(n^3\right)$	$\Theta\left(n\log n\right)$	✗	✗	✓
2-way $\mathcal{R}\text{-}\mathcal{DP}$	WS	$\mathcal{O}\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log_2 3}\right)$	✓	✗	✗
r-way $\mathcal{R}\text{-}\mathcal{DP}$	WS	$\mathcal{O}\left(\begin{array}{l}n^3/\left(B\lfloor\sqrt{M}/r\rfloor\right)\\ + n^3/\left(\lfloor\sqrt{M}/r\rfloor^3\right)\end{array}\right)$	$\mathcal{O}\left(\begin{array}{l}rn^{\log_r\left(2-\frac{1}{r}\right)} + nr\\ + (1/r)n^{\log_r(2r-1)}\end{array}\right)$	✗	✗	✗
$\mathcal{WR}\text{-}\mathcal{DP}$	WS	$\mathcal{O}\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n\log n\right)$	✓	−	✓
*$\mathcal{WR}\text{-}\mathcal{DP}$	WS	$\mathcal{O}\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n\log n\right)$	✓	✓	✓
*$\mathcal{WR}\text{-}\mathcal{DP}$	W-SB*	$\mathcal{O}\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n\log n\right)$	✓	✓	✓

Table 3.2: Different DP algorithms to solve the parenthesis problem using various schedulers. In the r-way $\mathcal{R}\text{-}\mathcal{DP}$, r is a constant. The second column represents the schedulers: WS means work stealing, W-SB means modified space bounded. * represents the algorithms presented in this chapter. The last column means excellent parallelism. In the last three columns, we use ✓ to denote optimality or near-optimality and "−" to denote non-applicability.

The recursive wavefront algorithms were introduced in [Tang et al., 2015] but they are too complicated to develop, analyze, implement, and generalize. They make extensive use of atomic instructions, and standard analysis model of fork-join parallelism does not apply. In this paper we try to address these issues.

Our contributions. Our major contributions are as follows:

(1) [*Framework*.] We present a framework called Autogen-Wave to discover recursive divide-and-conquer wavefront algorithms based on timing functions. The recursive wavefront algorithms are superior to all existing algorithms for DP problems.

(2) [*Divide-and-conquer wavefront algorithms*.] We present several recursive divide-and-conquer wavefront algorithms derived using the Autogen-Wave framework.

(3) [*Schedulers*.] We also present two scheduling methods to schedule a a recursive wavefront algorithm: (i) the algorithm passes timing functions and space usage info to modified version of a hint-accepting space-bounded scheduler, (ii) the programmer appropriately transforms the algorithm to use the timing functions, and uses a standard randomized work-stealing scheduler to run the program.

Organization of the chapter. In Section 3.2, we present our Autogen-Wave framework that can be used to discover recursive wavefront algorithms based on timing functions. Two schedulers to schedule recursive wavefront algorithms and their analysis are presented in Section 3.4.

3.2 The Autogen-Wave framework

In this section, we present a framework called Autogen-Wave that can be used to derive a recursive divide-and-conquer wavefront algorithm from a standard recursive divide-and-conquer DP algorithm. The method involves augmenting all recursive function calls with timing functions to launch them as early as possible without violating any dependency

| | Work | $\mathcal{R}$-$\mathcal{DP}$ | | $\mathcal{WR}$-$\mathcal{DP}$ | |
| | | Serial cache | Span | Best serial cache | Best span |
Problem	(T_1)	comp. (Q_1)	(T_∞)	comp. (Q_1)	(T_∞)
Parenthesis problem	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n\log n)$
Floyd-Warshall's APSP 3-D	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta\left(n\log^2 n\right)$	$\Theta(n^3/B)$	$\Theta(n\log n)$
Floyd-Warshall's APSP 2-D	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n\log^2 n\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n\log n)$
LCS / Edit distance	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta(n^2/(BM))$	$\Theta(n\log n)$
Gap problem	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n\log n)$
3-point stencil	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta(n^2/(BM))$	$\Theta(n\log n)$
Protein accordion folding	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n\log n)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n\log n)$
Spoken-word recognition	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta(n^2/(BM))$	$\Theta(n\log n)$
Function approximation	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n\log n)$
Binomial coefficient	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta(n^2/(BM))$	$\Theta(n\log n)$
Bitonic traveling salesman	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta(n\log n)$	$\Theta(n^2/(BM))$	$\Theta(n\log n)$

Table 3.3: Work (T_1), serial cache complexity (Q_1), and span (T_∞) of $\mathcal{R}$-$\mathcal{DP}$ and $\mathcal{WR}$-$\mathcal{DP}$ algorithms for several DP problems. Here, n = problem size, M = cache size, B = block size, and p = #cores. We assume that the DP table is too large to fit into the cache, and $M = \Omega\left(B^d\right)$ when $\Theta\left(n^d\right)$ is the size of the DP table. On p cores, the running time is $T_p = \mathcal{O}\left(T_1/p + T_\infty\right)$. The $\mathcal{R}$-$\mathcal{DP}$ algorithms have a parallel cache complexity is $Q_p = \mathcal{O}\left(Q_1 + p(M/B)T_\infty\right)$ with high probability when run under the randomized work-stealing scheduler on a parallel machine with private caches. The $\mathcal{WR}$-$\mathcal{DP}$ algorithms have a parallel cache complexity of $Q_p = \mathcal{O}\left(Q_1\right)$ when run with the modified space-bounded scheduler.

constraints implied by the DP recurrence. The timing functions are derived analytically, and do not employ locks or atomic instructions.

Our transformation allows the updates to the DP table proceed in an order close to iterative wavefront, but from within the structure of a recursive divide-and-conquer algorithm. The goal is to reach the higher parallelism of an iterative wavefront algorithm while retaining the better cache performance (i.e., efficiency and adaptivity) and portability (i.e., cache- and processor-obliviousness) of a recursive algorithm.

We had defined $\mathcal{I}$-$\mathcal{DP}$ and $\mathcal{R}$-$\mathcal{DP}$ in Section 2.2. Here, we define a more terms that we will use throughout the chapter.

Definition 19 ($\mathcal{TI}$-$\mathcal{DP}$/$\mathcal{WI}$-$\mathcal{DP}$). *Let $\mathcal{P}$ be a given DP problem. Then, a $\mathcal{TI}$-$\mathcal{DP}$ is a parallel tiled iterative algorithm for solving $\mathcal{P}$ and $\mathcal{WI}$-$\mathcal{DP}$ is a parallel wavefront iterative algorithm for $\mathcal{P}$.*

Typically a $\mathcal{TI}$-$\mathcal{DP}$ is autogenerated from a polyhedral compiler (parallelizer and locality optimizer) such as Pluto. These $\mathcal{TI}$-$\mathcal{DP}$s will not have the highest parallelism. On the other hand, $\mathcal{WI}$-$\mathcal{DP}$s have the highest parallelism.

Definition 20 ($\mathcal{WR}$-$\mathcal{DP}$/**Autogen-Wave**). *Let $\mathcal{P}$ be a given DP problem. A $\mathcal{WR}$-$\mathcal{DP}$ is a recursive wavefront algorithm for $\mathcal{P}$. Autogen-Wave is our framework that can be used to derive a $\mathcal{WR}$-$\mathcal{DP}$ from a given $\mathcal{R}$-$\mathcal{DP}$ and its corresponding DP recurrence.*

$\mathcal{R}$-$\mathcal{DP}$ $\longrightarrow$ Autogen-Wave $\longrightarrow$ $\mathcal{WR}$-$\mathcal{DP}$

Let us first define the *wavefront order* of applying updates to a DP table. Each update writes to one DP table cell by reading values from other cells. An update becomes *ready* when all cells it reads from are fully updated. We assume that only ready updates can be applied and each such update can only be applied once. A wavefront order of updates proceeds in discrete timesteps. In each step all ready updates to distinct cells are applied in parallel. However, if a cell has multiple ready updates only one of them is applied, and the rest are retained for future. A wavefront order does not have any artificial dependencies.

The AUTOGEN-WAVE framework. The AUTOGEN-WAVE framework consists of three major steps:

(1) [*Completion-time function construction.*] A closed-form formula or function (that can be evaluated in constant time) is derived based on the original DP recurrence that gives the timestep at which each DP cell is fully updated in wavefront order. See Section 3.2.1.

(2) [*Start- and end-time functions construction.*] Cell completion times are used to derive closed-form formulas that give the timesteps in wavefront order at which each recursive function call should start and end execution. See Section 3.2.2.

(3) [*Divide-and-conquer wavefront algorithm derivation.*] Each recursive function call in the standard recursive algorithm is augmented with its start- and end-time functions so that the algorithm can be used to apply only the updates in any given timestep in wavefront order. We then use a variant of iterative deepening on top of this recursive algorithm to execute all timesteps efficiently. See Section 3.2.3.

We describe our transformation for arbitrary d-dimensional $(d \geq 1)$ DP in which each dimension of the DP table is of the same length and is a power of 2.

Example. We explain our approach by applying it on an $\mathcal{R}$-$\mathcal{DP}$ algorithm for the parenthesis problem [Cherng and Ladner, 2005], which is defined as follows. Let $G[i, j]$ denote the minimum cost of parenthesizing $s_i \cdots s_j$. Then the 2D DP table $G[0 : n, 0 : n]$ is filled up using the following recurrence:

$$G[i,j] = \begin{cases} \infty & \text{if } 0 \leq i = j \leq n, \\ v_j & \text{if } 0 \leq i = j - 1 < n, \\ \min_{i \leq k \leq j} \{G[i,k] + G[k,j] + w(i,k,j)\} & \text{if } 0 \leq i < j - 1 < n; \end{cases} \tag{3.1}$$

where the v_j's and function $w(\cdot, \cdot, \cdot)$ are given. The recurrence is evaluated by the recursive algorithm [Tithi et al., 2015] given at the top of Figure 3.1. In the rest of the section, we show how a recursive wavefront algorithm (shown in Figure 3.1) can be derived from the given algorithm.

Fundamentals. We need to understand the fundamental concepts of timestamps, time steps and races before understanding the AUTOGEN-WAVE framework.

Definition 21 (Timestamp). *A timestamp, denoted by $\mathcal{F}_t(x, y_1, \ldots, y_s)$, is an update by a problem-specific function $\mathcal{F}$ writing to the cell x reading information from the cells $y_1, \ldots, y_s$ at time t.*

Sometimes we also specify time in the form of $t.t'$, where the term t' is used to avoid race conditions. If there are multiple updates that read information from different sets of cells and write to the same cell at the same time there will be race. Each such update will be performed at the same t value, but different t' values. We do not require t' for standard

68

iterative and recursive algorithms. For fast iterative algorithm it becomes easier to find patterns if we use t'.

For the parenthesis problem, the timestamps (without t' term) for $\mathcal{I}\text{-}\mathcal{DP}$, $\mathcal{R}\text{-}\mathcal{DP}$, $\mathcal{WI}\text{-}\mathcal{DP}$, and $\mathcal{WR}\text{-}\mathcal{DP}$ algorithms are given in Table 3.4. The right column of the table gives the last updated timestep for each cell of the DP table for different algorithms.

Consider the standard 2-way recursive algorithm for the parenthesis problem given in the top-left corner of Figure 3.1. It has three functions that update the DP table. Initially, function $\mathcal{A}(G, G, G)$ is called, where G is the entire DP table. Then the computation progresses by recursively breaking the table into quadrants, and calling functions $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ on these smaller regions of G. At the base case (i.e., a 1×1 region of G), each function updates a cell. When x is a cell, function $\mathcal{A}(x, x, x)$ updates x by reading x itself which corresponds to the case $i = k = j$ in the recurrence. Similarly, function $\mathcal{B}(x, u, v)$ updates cell x by reading x itself and two other cells u and v which correspond to cases $i = k \neq j$ and $i \neq k = j$. Finally, function $\mathcal{C}(x, u, v)$ updates the cell x by reading the two cells u and v which corresponds to $i \neq k \neq j$.

The middle part of Table 3.4 shows how the standard 2-way recursive algorithm with 1×1 base case updates $G[1 : n, 1 : n]$ when $n = 8$. We use $\mathcal{F}_t$ in a cell to denote that function $\mathcal{F}$ updates the cell at timestep t, where $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$. Using an unbounded number of processors the standard recursive algorithm updates the entire table in 31 timesteps. In contrast, the bottom part of Table 3.4 shows that an iterative wavefront algorithm will update G in only 18 timesteps.

The top part of the Table 3.4 shows how $\mathcal{I}\text{-}\mathcal{DP}$ updates an 8×8 DP table in 28 timesteps. And the $\mathcal{R}\text{-}\mathcal{DP}$ takes 31 timesteps to update the DP table. However, the span of $\mathcal{I}\text{-}\mathcal{DP}$ is $\Theta\left(n^2\right)$ and that of $\mathcal{R}\text{-}\mathcal{DP}$ is $\Theta\left(n^{\log 3}\right)$. For much larger DP tables it becomes clear that $\mathcal{I}\text{-}\mathcal{DP}$ takes many more timesteps than $\mathcal{R}\text{-}\mathcal{DP}$ to update the entire DP table.

It is important to note the we have used three functions $\mathcal{A}, \mathcal{B}, \mathcal{C}$ for the updates of the iterative algorithms. Each function has a specific functionality as described in the preceding paragraphs.

With a 1×1 base case the $\mathcal{WR}\text{-}\mathcal{DP}$ algorithms will perform the updates in exactly the same order as the $\mathcal{WI}\text{-}\mathcal{DP}$ algorithm, and terminate in 18 steps. The timestamps are shown in the bottom part of Table 3.4.

Our goal is to derive a recursive divide-and-conquer wavefront algorithm that has timestamps identical to a fast iterative algorithm.

3.2.1 Completion-time function construction

In this section, we define completion-time, and show how to compute it in $\mathcal{O}(1)$ time for any cell.

Definition 22 (*Completion-time*). *The completion-time for a particular cell x, denoted by $\mathfrak{C}(x)$, is the timestep in wavefront order at which x is fully updated. More formally,*

$$\mathfrak{C}(x) = \max \ t \mid \textit{for all } \mathcal{F}_t(x, \ldots); \tag{3.2}$$

where $\mathcal{F}_t(x, \ldots)$ means that cell x is updated by function $\mathcal{F}$ at timestep t.

Completion-time of a cell computed from the given DP recurrence is as follows.

$\mathcal{I}$-$\mathcal{DP}$

$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_9\mathcal{B}_{10}$	$\mathcal{C}_{11}\mathcal{C}_{12}\mathcal{C}_{13}\mathcal{C}_{14}\mathcal{B}_{15}$	$\mathcal{C}_{16}\mathcal{C}_{17}\mathcal{C}_{18}\mathcal{C}_{19}\mathcal{C}_{20}\mathcal{B}_{21}$	$\mathcal{C}_{22}\mathcal{C}_{23}\mathcal{C}_{24}\mathcal{C}_{25}\mathcal{C}_{26}\mathcal{C}_{27}\mathcal{B}_{28}$
−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_9\mathcal{B}_{10}$	$\mathcal{C}_{11}\mathcal{C}_{12}\mathcal{C}_{13}\mathcal{C}_{14}\mathcal{B}_{15}$	$\mathcal{C}_{16}\mathcal{C}_{17}\mathcal{C}_{18}\mathcal{C}_{19}\mathcal{C}_{20}\mathcal{B}_{21}$
−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_9\mathcal{B}_{10}$	$\mathcal{C}_{11}\mathcal{C}_{12}\mathcal{C}_{13}\mathcal{C}_{14}\mathcal{B}_{15}$
−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_9\mathcal{B}_{10}$
−	−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$
−	−	−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$
−	−	−	−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$
−	−	−	−	−	−	−	$\mathcal{A}_0$

0	1	3	6	10	15	21	28
−	0	1	3	6	10	15	21
−	−	0	1	3	6	10	15
−	−	−	0	1	3	6	10
−	−	−	−	0	1	3	6
−	−	−	−	−	0	1	3
−	−	−	−	−	−	0	1
−	−	−	−	−	−	−	0

$\mathcal{R}$-$\mathcal{DP}$

$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_3\mathcal{B}_4$	$\mathcal{C}_5\mathcal{C}_6\mathcal{B}_7$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{17}\mathcal{B}_{18}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{19}\mathcal{C}_{20}\mathcal{B}_{21}$	$\mathcal{C}_{22}\mathcal{C}_{23}\mathcal{C}_{24}\mathcal{C}_{25}\mathcal{C}_{27}\mathcal{B}_{28}$	$\mathcal{C}_{22}\mathcal{C}_{23}\mathcal{C}_{24}\mathcal{C}_{25}\mathcal{C}_{29}\mathcal{C}_{30}\mathcal{B}_{31}$
−	$\mathcal{A}_0$	$\mathcal{B}_2$	$\mathcal{C}_3\mathcal{B}_4$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{B}_{16}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{17}\mathcal{B}_{18}$	$\mathcal{C}_{22}\mathcal{C}_{23}\mathcal{C}_{24}\mathcal{C}_{25}\mathcal{B}_{26}$	$\mathcal{C}_{22}\mathcal{C}_{23}\mathcal{C}_{24}\mathcal{C}_{25}\mathcal{C}_{27}\mathcal{B}_{28}$
−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_9\mathcal{B}_{10}$	$\mathcal{C}_{11}\mathcal{C}_{12}\mathcal{B}_{13}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{17}\mathcal{B}_{18}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{19}\mathcal{C}_{20}\mathcal{B}_{21}$
−	−	−	$\mathcal{A}_0$	$\mathcal{B}_8$	$\mathcal{C}_9\mathcal{B}_{10}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{B}_{16}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{17}\mathcal{B}_{18}$
−	−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_3\mathcal{B}_4$	$\mathcal{C}_5\mathcal{C}_6\mathcal{B}_7$
−	−	−	−	−	$\mathcal{A}_0$	$\mathcal{B}_2$	$\mathcal{C}_3\mathcal{B}_4$
−	−	−	−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$
−	−	−	−	−	−	−	$\mathcal{A}_0$

0	1	4	7	18	21	28	31
−	0	2	4	16	18	26	28
−	−	0	1	10	13	18	21
−	−	−	0	8	10	16	18
−	−	−	−	0	1	4	7
−	−	−	−	−	0	2	4
−	−	−	−	−	−	0	1
−	−	−	−	−	−	−	0

$\mathcal{WI}$-$\mathcal{DP}$ & $\mathcal{WR}$-$\mathcal{DP}$

$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_4\mathcal{C}_7\mathcal{C}_8\mathcal{B}_9$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{B}_{12}$	$\mathcal{C}_7\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{C}_{13}\mathcal{C}_{14}\mathcal{B}_{15}$	$\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{C}_{13}\mathcal{C}_{14}\mathcal{C}_{16}\mathcal{C}_{17}\mathcal{B}_{18}$
−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_4\mathcal{C}_7\mathcal{C}_8\mathcal{B}_9$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{B}_{12}$	$\mathcal{C}_7\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{C}_{13}\mathcal{C}_{14}\mathcal{B}_{15}$
−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_4\mathcal{C}_7\mathcal{C}_8\mathcal{B}_9$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{B}_{12}$
−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_4\mathcal{C}_7\mathcal{C}_8\mathcal{B}_9$
−	−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$
−	−	−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$
−	−	−	−	−	−	$\mathcal{A}_0$	$\mathcal{B}_1$
−	−	−	−	−	−	−	$\mathcal{A}_0$

0	1	3	6	9	12	15	18
−	0	1	3	6	9	12	15
−	−	0	1	3	6	9	12
−	−	−	0	1	3	6	9
−	−	−	−	0	1	3	6
−	−	−	−	−	0	1	3
−	−	−	−	−	−	0	1
−	−	−	−	−	−	−	0

Table 3.4: Left: Timesteps at which each DP table cell is updated ($\mathcal{F}_t$ means function $\mathcal{F}$ updates at timestep t). Right: Timesteps at which each cell becomes fully updated (on the right) for the parenthesis problem on a DP table of size 8×8. Top: standard $\mathcal{I}$-$\mathcal{DP}$ which has a span of $\Theta\left(n^2\right)$. Middle: standard 2-way $\mathcal{R}$-$\mathcal{DP}$ algorithm which has a span of $\Theta\left(n^{\log 3}\right)$. Bottom: $\mathcal{WI}$-$\mathcal{DP}$ and $\mathcal{WR}$-$\mathcal{DP}$ algorithms which has a span of $\Theta\left(n \log n\right)$ overall (including traversing the recursion tree) but just $\Theta\left(n\right)$ to update the DP table. All recursive algorithms use a 1×1 base case. We assume that the number of processors is unbounded.

$\mathcal{A}_{0.0}$	$\mathcal{B}_{1.0}$	$\mathcal{C}_{2.0}\mathcal{B}_{3.0}$	$\mathcal{C}_{4.0}\mathcal{C}_{4.1}\mathcal{B}_{5.0}$	$\mathcal{C}_{4.0}\mathcal{C}_{6.0}\mathcal{C}_{6.1}\mathcal{B}_{7.0}$	$\mathcal{C}_{6.0}\mathcal{C}_{6.1}\mathcal{C}_{8.0}\mathcal{C}_{8.1}\mathcal{B}_{9.0}$	$\mathcal{C}_{6.0}\mathcal{C}_{8.0}\mathcal{C}_{8.1}\mathcal{C}_{10.0}\mathcal{C}_{10.1}\mathcal{B}_{11.0}$	$\mathcal{C}_{8.0}\mathcal{C}_{8.1}\mathcal{C}_{10.0}\mathcal{C}_{10.1}\mathcal{C}_{12.0}\mathcal{C}_{12.1}\mathcal{B}_{13.0}$
−	$\mathcal{A}_{0.0}$	$\mathcal{B}_{1.0}$	$\mathcal{C}_{2.0}\mathcal{B}_{3.0}$	$\mathcal{C}_{4.0}\mathcal{C}_{4.1}\mathcal{B}_{5.0}$	$\mathcal{C}_{4.0}\mathcal{C}_{6.0}\mathcal{C}_{6.1}\mathcal{B}_{7.0}$	$\mathcal{C}_{6.0}\mathcal{C}_{6.1}\mathcal{C}_{8.0}\mathcal{C}_{8.1}\mathcal{B}_{9.0}$	$\mathcal{C}_{6.0}\mathcal{C}_{8.0}\mathcal{C}_{8.1}\mathcal{C}_{10.0}\mathcal{C}_{10.1}\mathcal{B}_{11.0}$
−	−	$\mathcal{A}_{0.0}$	$\mathcal{B}_{1.0}$	$\mathcal{C}_{2.0}\mathcal{B}_{3.0}$	$\mathcal{C}_{4.0}\mathcal{C}_{4.1}\mathcal{B}_{5.0}$	$\mathcal{C}_{4.0}\mathcal{C}_{6.0}\mathcal{C}_{6.1}\mathcal{B}_{7.0}$	$\mathcal{C}_{6.0}\mathcal{C}_{6.1}\mathcal{C}_{8.0}\mathcal{C}_{8.1}\mathcal{B}_{9.0}$
−	−	−	$\mathcal{A}_{0.0}$	$\mathcal{B}_{1.0}$	$\mathcal{C}_{2.0}\mathcal{B}_{3.0}$	$\mathcal{C}_{4.0}\mathcal{C}_{4.1}\mathcal{B}_{5.0}$	$\mathcal{C}_{4.0}\mathcal{C}_{6.0}\mathcal{C}_{6.1}\mathcal{B}_{7.0}$
−	−	−	−	$\mathcal{A}_{0.0}$	$\mathcal{B}_{1.0}$	$\mathcal{C}_{2.0}\mathcal{B}_{3.0}$	$\mathcal{C}_{4.0}\mathcal{C}_{4.1}\mathcal{B}_{5.0}$
−	−	−	−	−	$\mathcal{A}_{0.0}$	$\mathcal{B}_{1.0}$	$\mathcal{C}_{2.0}\mathcal{B}_{3.0}$
−	−	−	−	−	−	$\mathcal{A}_{0.0}$	$\mathcal{B}_{1.0}$
−	−	−	−	−	−	−	$\mathcal{A}_{0.0}$

Table 3.5: The timestamps for the parenthesis problem for a DP table size 8×8 using a $\mathcal{WR}$-$\mathcal{DP}$ algorithm.

$$\mathfrak{C}(x) = \begin{cases} \text{initial values} & \text{initial conditions,} \\ smax(x) + flag(x) + su(x) & \text{otherwise;} \end{cases} \qquad (3.3)$$

where $smax(x)$ is the maximum completion time of the cells on which x directly depends, i.e., $smax(x) = \max_{\mathcal{F}(x,\dots,y,\dots)} \mathfrak{C}(y)$ and $y \neq x$; $flag(x) = 0$ if the cell-tuple (with maximum completion-time as $smax(x)$) is inflexible and $flag(x) = 1$ if the cell-tuple is flexible; and the term $su(x)$ is 1 if there is a self-update function that reads from itself; and 0, otherwise.

In simple words, the last timestep at which a cell is updated/written is called the completion-time of that cell. The term $smax(x)$ is the maximum completion-time of all

possible read cells of the cell x. The letter s in $smax$ means start-time. It means that the cell x gets completed only after all of its read cells are started and finish their execution. This is also the reason to include the term 1. The term $su(x)$ is required to avoid collisions or race conditions. The idea for self-update function was given by Jesmin Jahan Tithi.

We need to find closed-form formulas to find the completion-time of the $\mathcal{WR}$-$\mathcal{DP}$ algorithm. The timestamps of the $\mathcal{WR}$-$\mathcal{DP}$ algorithm for all cells of an 8×8 DP table is shown in Table 3.5. The completion-time for any cell (i, j) in the DP table can be found as follows.

$$\mathfrak{C}(i,j) = \begin{cases} 0 & \text{if } i = j, \\ \mathfrak{C}(i, j-1) + 0 + 1 & \text{if } i = j - 1, \\ \mathfrak{C}(i, j-1) + 1 + 1 & \text{if } i <= j - 2; \end{cases} \tag{3.4}$$

because $smax(i,j) = \mathfrak{C}(i, j-1) = \mathfrak{C}(i+1, j)$; if $i \leq j - 1$, then $su(i, j) = 1$ as the self-update function $\mathcal{B}$ updates the cell (i, j) reading from itself. When $i = j - 1$, there is no cell-tuple that does not read from (i, j) and when $i \leq j - 1$, there is at least one flexible tuple. Solving the recurrence, we get the following:

$$\mathfrak{C}(i,j) = \begin{cases} j - i & \text{if } (j - i) = 0, \\ 2(j - i) - 1 & \text{if } (j - i) \geq 1. \end{cases} \tag{3.5}$$

3.2.2 Start-time and end-time functions construction

In this section, we define start-time and end-time for a recursive function call, and show how to derive them from completion-times.

In an $\mathcal{R}$-$\mathcal{DP}$, every function invokes one or more recursive functions. Each function call will only be executed when all its read regions are completely updated. On the other hand, in a $\mathcal{WR}$-$\mathcal{DP}$, a function will be executed when as soon as a single cell of the write region is ready to execute. The relative timestep at which a function starts execution is called start-time and the relative timestep when the function completes execution is termed end-time. Closed-form formulas / mathematical functions are constructed to find the start- and end-times.

Definition 23 (*Start-time and end-time*). *The start-time (resp. end-time) of a recursive function call in a $\mathcal{WR}$-$\mathcal{DP}$ algorithm is the earliest (resp. latest) timestep in wavefront order at which one of the updates to be applied by that function call (either directly or through a recursive function call) becomes ready.*

Let $\mathcal{F}(X, Y_1, \ldots, Y_s)$ be a function call that writes to a region X by reading from regions $Y_1, \ldots, Y_s$ of the DP table. Its start- and end-times, denoted by $\mathcal{S}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)$ and $\mathcal{E}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)$, respectively, are computed as follows.

$$
\underbrace{\mathcal{S}_{\mathcal{F}}\left(X, Y_1, \ldots, Y_s\right)}_{X \in \{Y_1, \ldots, Y_s\}} =
\begin{cases}
(\mathfrak{C}(X)).0 & \text{if } X \text{ is a cell,} \\
\min \ \mathcal{S}_{\mathcal{F}'}\left(X', Y_1', \ldots, Y_s'\right) & \text{otherwise;}
\end{cases}
\tag{3.6}
$$

$$
\underbrace{\mathcal{S}_{\mathcal{F}}\left(X, Y_1, \ldots, Y_s\right)}_{X \notin \{Y_1, \ldots, Y_s\}} =
\begin{cases}
(\min_{1 \le i \le s}\{\mathfrak{C}(Y_i)\} + 1).ra(X, Y_1, \ldots, Y_s) & \text{if } X \text{ is a cell,} \\
\min \ \mathcal{S}_{\mathcal{F}'}\left(X', Y_1', \ldots, Y_s'\right) & \text{otherwise;}
\end{cases}
\tag{3.7}
$$

$$
\underbrace{\mathcal{E}_{\mathcal{F}}\left(X, Y_1, \ldots, Y_s\right)}_{X \in \{Y_1, \ldots, Y_s\}} =
\begin{cases}
(\mathfrak{C}(X)).0 & \text{if } X \text{ is a cell,} \\
\max \ \mathcal{E}_{\mathcal{F}'}\left(X', Y_1', \ldots, Y_s'\right) & \text{otherwise;}
\end{cases}
\tag{3.8}
$$

$$
\underbrace{\mathcal{E}_{\mathcal{F}}\left(X, Y_1, \ldots, Y_s\right)}_{X \notin \{Y_1, \ldots, Y_s\}} =
\begin{cases}
(\max_{1 \le i \le s}\{\mathfrak{C}(Y_i)\} + 1).ra(X, Y_1, \ldots, Y_s) & \text{if } X \text{ is a cell,} \\
\max \ \mathcal{E}_{\mathcal{F}'}\left(X', Y_1', \ldots, Y_s'\right) & \text{otherwise;}
\end{cases}
\tag{3.9}
$$

where, in the non-cellular case minimization/maximization is taken over all functions $\mathcal{F}'(X', Y_1', \ldots, Y_s')$ recursively called by $\mathcal{F}(X, Y_1, \ldots, Y_s)$. Also, $ra(X, Y_1, \ldots, Y_s)$ is the problem-specific race avoidance condition used when two functions write to the same region. Though we use real-valued timesteps for simplicity, the total number of distinct timesteps remain exactly the same as that in the $\mathcal{WI}\text{-}\mathcal{DP}$ algorithm.

For the parenthesis problem, the start-times for the three functions $\mathcal{A}, \mathcal{B}$, and $\mathcal{C}$ are computed as below. Let $(x_r, x_c), (u_r, u_c)$, and (v_r, v_c) denote the positions of the top-left cells of regions X, U and V, respectively. Then,

$$
\mathcal{S}_{\mathcal{A}}\left(X, X, X\right) =
\begin{cases}
\mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\
\mathcal{S}_{\mathcal{A}}\left(X_{11}, X_{11}, X_{11}\right) & \text{otherwise;}
\end{cases}
\tag{3.10}
$$

$$
\mathcal{S}_{\mathcal{B}}\left(X, U, V\right) =
\begin{cases}
\mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\
\mathcal{S}_{\mathcal{B}}\left(X_{21}, U_{22}, V_{11}\right) & \text{otherwise;}
\end{cases}
\tag{3.11}
$$

$$
\mathcal{S}_{\mathcal{C}}\left(X, U, V\right) =
\begin{cases}
(\max\{\mathfrak{C}(U), \mathfrak{C}(V)\} + 1).[u_c > \frac{x_r + x_c}{2}] & \text{if } X \text{ is a cell,} \\
\min\{\mathcal{S}_{\mathcal{C}}\left(X_{21}, U_{21}, V_{11}\right), \mathcal{S}_{\mathcal{C}}\left(X_{21}, U_{22}, V_{21}\right)\} & \text{otherwise;}
\end{cases}
\tag{3.12}
$$

where $[\ ]$ is the Iverson bracket [Iverson, 1962] which denotes 1 if the condition in the bracket is true and 0 otherwise. The idea for using the Iverson function to handle race avoidance was given by Jesmin Jahan Tithi.

The start time for $\mathcal{A}$ follows directly from Definition 23 and instead of $\mathcal{S}_{\mathcal{A}}\left(X_{11}, X_{11}, X_{11}\right)$ we could have instead chosen $\mathcal{S}_{\mathcal{A}}\left(X_{22}, X_{22}, X_{22}\right)$ because the completion-times of some of the cells in X_{11} and X_{22} are the minimum.

Function $\mathcal{B}$ actually reads from and writes to X. This is because it writes to X reading the pairs from U, X and from X, V, denoted by $\langle X, U, X \rangle$ and $\langle X, X, V \rangle$, respectively. Hence, the start-time recurrence for $\mathcal{B}$ matches with that of the first recurrence in Definition 23.

Function $\mathcal{C}$ follows the second recurrence from the definition. Among all the functions invoked by $\mathcal{C}$, the function call having the least start-time can be either $\mathcal{C}(\langle X_{21}, U_{21}, V_{11}\rangle)$ or $\mathcal{C}(\langle X_{21}, U_{22}, V_{21}\rangle)$. As the function $\mathcal{C}$ writes to the same region twice, there is a race and to avoid it we use the condition $[u_c > (x_r + x_c)/2]$ derived manually.

Similarly, the end-times are as follows.

$$\mathcal{E}_\mathcal{A}\left(X, X, X\right) = \begin{cases} \mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\ \mathcal{E}_\mathcal{B}\left(X_{12}, X_{11}, X_{22}\right) & \text{otherwise;} \end{cases}$$

$$\mathcal{E}_\mathcal{B}\left(X, U, V\right) = \begin{cases} \mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\ \mathcal{E}_\mathcal{B}\left(X_{12}, U_{11}, V_{22}\right) & \text{otherwise;} \end{cases}$$

$$\mathcal{E}_\mathcal{C}\left(X, U, V\right) = \begin{cases} \left(\max\left\{\mathfrak{C}(U), \mathfrak{C}(V)\right\} + 1\right).[u_c > \frac{x_r + x_c}{2}] & \text{if } X \text{ is a cell,} \\ \max\left\{\mathcal{E}_\mathcal{C}\left(X_{12}, U_{11}, V_{12}\right), \mathcal{E}_\mathcal{C}\left(X_{12}, U_{12}, V_{22}\right)\right\} & \text{otherwise.} \end{cases}$$

Solving the recurrences for the start-times and end-times above, we obtain the timing functions shown in Figure 3.1.

3.2.3 Divide-and-conquer wavefront algorithm derivation

In this section, we describe how to use timing functions to derive a $\mathcal{WR}$-$\mathcal{DP}$ algorithm from a given standard $\mathcal{R}$-$\mathcal{DP}$ algorithm. We use the parenthesis problem as an example.

Each recursive function makes $\mathcal{O}\left(1\right)$ function calls and each function call $\mathcal{F}(X, Y_1, \ldots, Y_s)$ writes to a region and reads from one or more regions $Y_1, \ldots, Y_s$. The function call $\mathcal{F}$ has to wait until all its read regions are completely updated and no other function call is updating $\mathcal{F}$'s write-region X. This means, the function call $\mathcal{F}(X, Y_1, \ldots, Y_s)$ has to wait for its execution until the start-time $\mathcal{S}_\mathcal{F}\left(X, Y_1, \ldots, Y_s\right)$. Also, the function completes its execution at $\mathcal{E}_\mathcal{F}\left(X, Y_1, \ldots, Y_s\right)$. Therefore, a $\mathcal{WR}$-$\mathcal{DP}$ algorithm can be easily derived from an $\mathcal{R}$-$\mathcal{DP}$ algorithm by simply specifying the start-times and end-times for each function invocation. Such start- and end-time information is a hint to the task scheduler to execute the tasks at the specified times.

A standard $\mathcal{R}$-$\mathcal{DP}$ for the parenthesis problem is shown in the top-left corner of Figure 3.1. We modify it as follows, and the modified algorithm is shown on the right hand side of the same figure.

First, we modify each function $\mathcal{F}$ to include a switching point $n' \geq 1$, and switch to the original non-wavefront recursive algorithm by calling $\mathcal{F}_{chunk}$ when the size of each input submatrix drops to $n' \times n'$ or below.

We augment each function to accept a timestep parameter w. We remove all serialization among recursive function calls by making sure that all functions that are called are launched in parallel. We do not launch a function unless w lies between its start-time and end-time which means that a function is not invoked if we know that it does not have an update to apply at timestep w in wavefront order. Observe that the function $\mathcal{F}_{chunk}$ at switching does not accept a timestep parameter, but if we reach it we know that it has an update to apply at timestep w. However, once we enter that function we do not stop until we apply all updates that function can apply at all timesteps $\geq w$.

Each function is also modified to return the smallest timestep above w for which it may have at least one update that is yet to be applied. It finds that timestep by checking the start-time of each function that was not launched because the start-time was larger than w, and the timestep returned by each recursive function that was launched, and taking the smallest of all of them.

Finally, we add a loop (see RECURSIVE-WAVEFRONT-PARENTHESIS in Figure 3.1) to execute all timesteps of the wavefront using the modified functions. We start with timestep $w = 0$, and invoke the main function $\mathcal{A}(G, G, G, w)$ which applies all updates at timestep

w and depending on the value chosen for n' possibly some updates above timestep w, and returns the smallest timestep above w for which there may still be some updates that are yet to be applied. We next call function $\mathcal{A}$ with that new timestep value, and keep iterating in the same fashion until we are able to exhaust all timesteps.

Though the $\mathcal{WR}\text{-}\mathcal{DP}$ algorithm has a recursive structure, it is really executed like a wavefront. All the real computations in the algorithm will be performed in the $\mathcal{F}_{chunk}$ kernels. Assume that the input DP table is subdivided into $(n/n') \times (n/n')$ matrix each unit of size $n' \times n'$. Possibly several different recursive functions write to an $n' \times n'$ chunk multiple times.

It is easy to prove that if two $\mathcal{F}_{chunk}$ functions can be executed in parallel, then they must have the same start-times. Using this idea, the variant of iterative deepening logic makes sure that in the first iteration, the $\mathcal{F}_{chunk}$ functions that have the smallest start-time will be executed and the second smallest start-time (among all chunks) is found. In the second iteration, all $\mathcal{F}_{chunk}$ functions that start at the computed start-time are executed and the third smallest start-time (among all $\mathcal{F}_{chunk}$ functions) is found. This process continues until all the tasks are complete.

3.3   Correctness

We prove the correctness of the principle used in deriving the recursive wavefront algorithms in the following theorem.

> **Theorem 4 (Correctness).** *Given an $\mathcal{R}\text{-}\mathcal{DP}$ for a DP problem $\mathcal{P}$, the AUTOGEN-WAVE framework can be used to design a $\mathcal{WR}\text{-}\mathcal{DP}$ algorithm that is functionally equivalent to the given $\mathcal{R}\text{-}\mathcal{DP}$, assuming the following:*
> (i) *Initial values of the completion-time recurrence is correct.*
> (ii) *Race avoidance function $ra()$ is correct.*
> (iii) *Update function $su(x)$ is correct.*
> (iv) *Completion-time, start-time, and end-time functions can be computed in $\mathcal{O}(1)$ time.*

Proof. We prove the theorem in three parts: (i) Completion-time function is correct, (ii) Start-time and end-time functions are correct, (iii) $\mathcal{WR}\text{-}\mathcal{DP}$ algorithm is correct. The assumptions listed in the theorem are important because they are manually computed and the results are problem-specific.

(i) *Completion-time function is correct.* The completion-time recurrence is based on the original DP recurrence of $\mathcal{P}$ and is given in Definition 22. A cell x can only be updated when all its read cells are completely updated. Hence, $\mathfrak{C}(x) \geq smax(x) + 1$.

Consider all timestamps $\mathcal{F}_t(x, \ldots, y_i, \ldots)$ that write to cell x. When the cell x has been updated by strictly reading from other cells (i.e., $y_i \neq x$ for all i), the timestamp will be $\mathcal{F}_{smax(x)+1}(x, \ldots)$ for some recursive function $\mathcal{F}$. We do not consider decimal values of timesteps for completion-time calculation. Due to Rule 1 of Section 2.2 there can be at most one function that reads from and writes to the same region which is called the self-update function. Therefore, there can be at most one more timestamp that updates cell x reading from x. This means that $\mathfrak{C}(x) \leq smax(x) + flag(x) + su(x)$.

Thus, $\mathfrak{C}(x) = smax(x) + flag(x) + su(x)$ is correct.

(*ii*) *Start-time and end-time functions are correct.* We prove the correctness of start-time functions. Similar arguments hold for end-time functions.

Start-time recurrences are given in Definition 23. A recursive function $\mathcal{F}(X, Y_1, \ldots, Y_s)$ invokes $\mathcal{O}(1)$ number of child functions $\mathcal{F}'(X', Y_1', \ldots, Y_s')$ such that X' and Y_i' are d-D or-thants of X and Y_i, respectively. The start-time of $\mathcal{F}$ must be the minimum among child functions $\mathcal{F}'$ of $\mathcal{F}$. Hence, the recurrence case that $\mathcal{S}_\mathcal{F}(X, \ldots, Y_i, \ldots) = \min \mathcal{S}_{\mathcal{F}'}(X', \ldots, Y_i', \ldots)$ is correct.

To compute the base cases of the start-time recurrence, we need to analyze two cases: (*i*) $\mathcal{F}$ is an inflexible function i.e., $x \in \{y_1, \ldots, y_s\}$, and (*ii*) $\mathcal{F}$ is a flexible function i.e., $x \notin \{y_1, \ldots, y_s\}$.

In the first case, the start-time of the inflexible function is same as the completion-time of the write-cell, because inflexible function represents complete updates. In the second case, the flexible function $\mathcal{F}(x, y_1, \ldots, y_s)$ can immediately start once all the read cells $y_1, \ldots, y_s$ have been updated completely i.e., $\mathcal{S}_\mathcal{F}(x, y_1, \ldots, y_s) = \max_{1 \leq i \leq s}\{\mathfrak{C}(y_i)\} + 1$. To avoid race conditions, we add manually computed race avoidance condition $ra(x, y_1, \ldots, y_s)$. Thus, the start-time function (resp. end-time function) is correct.

(*iii*) *Recursive divide-and-conquer wavefront algorithm is correct.* For all possible recursive functions $\mathcal{F}$ and regions $X, Y_1, \ldots, Y_s$ such that each region has a dimension length of n', let $w_0 < w_1 < \cdots < w_{L-1}$ be all the start-times of $\mathcal{F}(X, Y_1, \ldots, Y_s)$. As per the fundamental definition of a wavefront algorithm, initially, all those chunks X for which $\mathcal{S}_\mathcal{F}(X, Y_1, \ldots, Y_s) = w_0$ will be executed. Then, chunks for which the start-times are w_1 will be executed. This process continues till w_{L-1}. We need to prove the recursive wavefront algorithm too follows this strategy.

Once the timing functions are found, we simply plug in the start-time and end-time functions for every single recursive function call, as shown in an example in Figure 3.1, in such a way that every recursive function is invoked at its start-time and finishes at its end-time and this is true assuming that the timing functions are correct. Using contradiction it is straightforward to prove that a function cannot start (resp. end) execution before or later than its start-time (resp. end-time). It is up to the scheduler to schedule the recursive functions based on the timing functions.

The generic scheduling method to schedule a hybrid recursive wavefront algorithm uses a variant of iterative deepening technique to execute the base case tasks in a wavefront manner satisfying the DP dependencies and retaining the recursive structure, as depicted in an example in Figure 3.1. The scheduler (or programmer) adds conditional checks before every invocation of a recursive function call to check whether the executing wavefront levels are in fact between the start-time and end-time of the function. If they are, then the corresponding function is invoked to execute. As we do not execute a recursive function before its start-time or after its end-time, we are not violating any original DP dependencies. □

3.4 Scheduling algorithms

In this section, we show how to schedule recursive wavefront algorithms to achieve provably good bounds (optimal or near-optimal) for both parallelism and cache performance.

Recall that our recursive wavefront algorithm switches to the original non-wavefront recursive algorithm when the input parameter n drops to a value $\leq n'$. While both recursive (wavefront and non-wavefront) algorithms have the same serial work complexity T_1 and same serial cache complexity Q_1 (as they reduce to the same serial algorithm), their spans are different. We use $T_\infty^R(n')$ to denote the span of the non-wavefront algorithm for a problem of size n'.

In this section, we present a provably efficient method of scheduling recursive wavefront algorithms on a parallel machine based on iterative deepening. We analyze the scheduling of the recursive wavefront algorithms using two schedulers: (a) work-stealing scheduler, and (b) a variant of the space-bounded scheduler.

A scheduler maps parallel tasks of a multithreaded program to different processors in a parallel machine. Several schedulers have been proposed, analyzed, and implemented in literature. Table 3.6 gives a summary of several schedulers. Optimal algorithms scheduled using bad schedulers does not lead to high-performance and similarly slow algorithms scheduled using the best schedulers also do not lead to great performance. For the best performance, we need parallel algorithms with good cache efficiency and good parallelism and also scheduling algorithms (or schedulers) that exploit these characteristics.

It is not clear how to develop truly efficient cache-oblivious algorithms that have both optimal parallel cache performance and optimal parallelism. Even if we have high-performing algorithms, developing a scheduler that exploits the best from the two worlds is non-trivial. The fundamental difficulty for solving the two problems above comes from two seemingly paradoxical reasons:

(i) to get optimal parallel cache efficiency (majorly due to temporal locality), a task should work on a localized data as much as possible and migration of tasks to different caches should be minimized.

(ii) to get optimal parallelism or at least good parallelism, the task migration should be encouraged because more migration means more parallelism.

We prove that the span of the pure recursive wavefront algorithms is at most linear (w.r.t the input parameter) in Lemma 1. We use the notations $T_\infty^R(n)$ and $T_\infty^W(n)$ to denote the span of recursive and recursive wavefront algorithms, respectively. Consider a recursive wavefront algorithm that switches to recursive kernels when the input parameter drops from n to less than or equal to n'. Such algorithms are called *hybrid recursive wavefront algorithms* and in Sections 3.4.1 and 3.4.2 complexities of those algorithms when scheduled using different schedulers are discussed.

> **Lemma 1 (Worst-case linear span of the recursive wavefront algorithms (without scheduling overhead)).** *Given a DP problem, if the completion-time (see Definition 22) of every cell $(i_1, \ldots, i_m)$ can be written in the form*
>
> $$\mathfrak{C}(i_1, \ldots, i_m) \leftarrow \mathfrak{C}(i_1 - i_1', \ldots, i_m - i_m') + \mathcal{O}(1) \qquad (3.13)$$
>
> *where m is a constant; $i_1, \ldots, i_m, i_1', \ldots, i_m' \in [0, n-1]$ and at least one among $i_1', \ldots, i_m'$ is greater than or equal to 1. Then the number of wavefront levels in the recursive wavefront algorithm is at most linear, i.e., $N_\infty^W(n) = \mathcal{O}(n)$.*

Proof. Let n denote the input parameter. Let a cell to be computed be represented using m-dimensions (m is fixed). Let the cells $(0, \ldots, 0)$, $(n-1, \ldots, n-1))$, $(i_1, \ldots, i_m)$, and $(i_1 -$

$i'_1, \ldots, i_m - i'_m)$ be denoted by $first$, $last$, x, and y, respectively. Let y be the cell such that $\mathfrak{C}(y) = rmax(x)$. Here, $i_1, \ldots, i_m, i'_1, \ldots, i'_m \in [0, n-1]$ and at least one among $i'_1, \ldots, i'_m$ is greater than or equal to 1. Without loss of generality, we assume that $first$ and $last$ cells are the initial and final cells to be fully updated, respectively and the cell x depends on y. We know that $\mathfrak{C}(x)$ is more than $\mathfrak{C}(y)$ by some non-negative constant. We need to prove that the number of wavefront levels in the algorithm is at most linear i.e., $\mathfrak{C}(last)$ is $\mathcal{O}(n)$.

Given that the cell x depends on y, it is easy to see there is difference in the coordinates of x and y in at least one dimension and the index of that dimension decreases from cell x to cell y by at least 1. The maximum number of times the reduction in index in a single dimension can happen is n. As there are m dimensions, in the worst case, the maximum number of times reduction in any dimension can occur is nm. In other words, starting from $final$, it takes $nm = \mathcal{O}(n)$ time steps to reach $first$ cell and in each timestep there is a difference of a constant value in the completion times of the write and the read cells. Therefore, $\mathfrak{C}(last)$ is $\mathcal{O}(n)$ and hence $N_\infty^W(n) = \mathcal{O}(n)$. $\square$

3.4.1 Work-stealing scheduler

In this section, we analyze the complexity of the hybrid recursive wavefront algorithms when scheduled using a randomized work-stealing (WS) scheduler [Blumofe and Leiserson, 1999].

The machine model for WS is a machine with several levels of distributed private caches. The randomized WS presented and analyzed in [Blumofe and Leiserson, 1999] is a *distributed cache-oblivious scheduler* that can schedule hybrid recursive wavefront algorithms to get good parallel cache performance and good parallelism. We prove the following theorem to analyze the complexities of the schedule.

Scheduler	Q_p	T_p	Comments
Greedy scheduler	$-$	$\mathcal{O}\left(\frac{T_1}{p} + T_\infty\right)$	
Random work-stealing	$\mathcal{O}\left(Q_1 + p\frac{M}{B}T_\infty\right)$	$\mathcal{O}\left(\frac{T_1}{p} + T_\infty\right)$	Non-optimal Q_p
Priority work-stealing	$\mathcal{O}\left(Q_1 + p\frac{M}{B}T_\infty\right)$	$\mathcal{O}\left(\frac{T_1}{p} + T_\infty\right)$	Non-optimal Q_p
Parallel depth-first	$\Omega(Q_1)$	$\mathcal{O}\left(\frac{T_1}{p} + T_\infty\right)$	$M_p = \Omega(M_1 + \Theta(pT_\infty))$
Space-bounded	$\mathcal{O}(Q_1)$	$\mathcal{O}\left(\frac{T_1}{p} + T_\infty\right)$	Non-optimal T_∞
W-SB	$\mathcal{O}(Q_1)$	$\mathcal{O}\left(\frac{T_1}{p} + T_\infty\right)$	Optimal Q_p and near-optimal T_∞

Table 3.6: Parallel cache complexity (Q_p) and parallel running time (T_p) of several schedulers.

> **Theorem 5 (WS complexity).** *Suppose a DP recurrence is evaluated by the $\mathcal{WI}$-$\mathcal{DP}$ in $N_\infty(n)$ parallel steps with ∞ processors. When the $\mathcal{WR}$-$\mathcal{DP}$ algorithm with switching point $n' \leq n$ (switching to n $\mathcal{R}$-$\mathcal{DP}$) evaluates the recurrence, we achieve the following bounds while using a randomized work-stealing (WS) scheduler:*
> * *work, $T_1(n) = \mathcal{O}\left(T_1^R(n)\right)$,*
> * *span, $T_\infty(n) = \mathcal{O}\left(N_\infty(\frac{n}{n'})\left(\mathcal{O}(\log n) + T_\infty^R(n')\right)\right)$,*
> * *parallel time, $T_p(n) = \mathcal{O}\left(\frac{T_1(n)}{p} + T_\infty(n)\right)$ w.h.p. in n,*
> * *serial cache complexity, $Q_1(n) = T_1(n/n')Q_1^R(n')$,*
> * *parallel cache complexity, $Q_p(n) = \mathcal{O}\left(Q_1(n) + p \cdot \frac{M}{B} \cdot T_\infty(n)\right)$ w.h.p. in n, and*
> * *extra space, $S_p(n) = \mathcal{O}(p \log n)$.*
> *We assume that $N_\infty(n)$ and $T_1(n)$ (work) are polynomials of n, and choose n' such that $T_1(n') = \Omega(\log n)$.*

Proof. The theorem is proved in different parts.

Span. The total span is found by summing up the spans at every wavefront level. The span is bounded using a simple formula: total span $\leq$ #(wavefront levels) $\times$ (worst-case span of executing a chunk + worst-case span to reach a chunk). We know that there are $N_\infty(n/n')$ wavefront levels in the algorithm as the outer loop (e.g., the loop inside RECURSIVE-WAVEFRONT-PARENTHESIS of Figure 3.1) in the recursive wavefront algorithm will iterate $N_\infty(n/n')$ times. The worst-case span of executing a chunk is $T_\infty^R(n')$. Also, the worst-case span to reach a chunk is $\mathcal{O}(\log(n/n'))$. Simplifying, we have the total span as $N_\infty(n/n')(\mathcal{O}(\log n) + T_\infty^R(n'))$.

Total work. Similar to span, the total work can be found by summing up the work done at every wavefront level. The total work can be bounded using a simple formula: total work = #(chunks) $\times$(#computations inside a chunk + #computations required to reach a chunk). The expression computes to $T_1(n/n')\left(T_1(n') + \mathcal{O}(\log(n/n'))\right)$. We assume that the work is a polynomial function of n, which implies $T_1(n/n') = T_1(n)/T_1(n')$. To not increase the work compared to the original wavefront algorithm, we assume that every chunk does $\Omega(\log n)$ work, which implies $T_1(n') + \Theta(\log n) = T_1(n')$. After substitutions and simplification, the total work remains $\Theta(T_1(n))$ and does not increase.

Parallel running time. The parallel running time at a specific wavefront level is found using the theorem from [Blumofe and Leiserson, 1999]. The parallel running time $T_p(n)$ of the entire algorithm is found by summing up the parallel running time at each wavefront level. Let $T_1^{(i)}$, $T_\infty^{(i)}$ and $Q_1^{(i)}$ be the work, span and serial cache complexity, respectively, of the ith iteration of the outer loop in the recursive wavefront algorithm. Then the parallel running time of that iteration under the WS scheduler is $\mathcal{O}\left(1 + T_1^{(i)}(n)/p + T_\infty^{(i)}(n)\right)$ (w.h.p.). We sum up over all i, and obtain the claimed bound for $T_p(n)$. Thus, we have

$$
T_p(n) = \mathcal{O}\left(\sum_{\text{level } i}\left(1 + \frac{\text{work at iteration } i}{p} + \text{span at iteration } i\right)\right)
$$

$$
= \mathcal{O}\left(\#(\text{wavefront levels}) + \frac{1}{p} \cdot \sum_{\text{level } i} T_1^{(i)}(n) + \sum_{\text{level } i} T_\infty^{(i)}(n)\right)
$$

$$
= \mathcal{O}\left(\#(\text{wavefront levels}) + \frac{1}{p} \cdot T_1(n) + T_\infty(n)\right) = \mathcal{O}\left(\frac{T_1(n)}{p} + T_\infty(n)\right)
$$

Serial cache complexity. The total serial cache complexity can be found using the formula: $Q_1(n) = \#(\text{chunks}) \times (\text{serial cache complexity of a chunk} + \text{serial cache complexity of reaching a chunk}) = T_1(n/n')\left(Q_1^R(n') + \Theta\left((\log n)/B\right)\right)$. As $T_1(n') = \Omega(\log n)$, we have $Q_1^R(n') = \Omega\left((\log n)/B\right)$ and on simplification we get $Q_1(n) = T_1(n/n')Q_1^R(n')$.

Parallel cache complexity. The parallel cache complexity at a particular level is found by a result from [Acar et al., 2000] in terms of serial cache complexity. Similar to the argument of parallel running time, the total parallel cache complexity $Q_p(n)$ comprises of the parallel cache complexity at all levels. The parallel cache complexity of the i-th iteration of the outer loop is $\mathcal{O}\left(Q_1^{(i)}(n) + p(M/B)T_\infty^{(i)}(n)\right)$ (w.h.p.) under the WS scheduler. Summing up over all i gives us the claimed bound for $Q_p(n)$. We have

$$Q_p(n) = \mathcal{O}\left(\sum_{\text{level } i}\left(Q_1^{(i)}(n) + p \cdot \frac{M}{B} \cdot T_\infty^{(i)}(n)\right)\right)$$
$$= \mathcal{O}\left(Q_1(n) + p(M/B) \cdot T_\infty(n)\right)$$

To achieve $Q_p(n)$ as described above w.h.p. we set $\epsilon = 1/n^{2+\lambda}$ for any $\lambda > 0$ in Lemma 16 of [Acar et al., 2000].

Extra-space complexity. For p processors, there can be at most p tasks running in parallel. When the tasks reach the levels of the leaves of the recursion tree, they execute the logic in the leaf functions. As each executing task is a leaf node in the recursion tree, we might require a total extra space all along the path from the root to the p executing leaves of the recursion tree. Hence, $S_p(n) = \mathcal{O}(p\log n)$.

$\square$

If n' is a polynomial function of n with degree less than 1 and Lemma 1 is satisfied, then we have the following corollary derived from Theorem 5.

> **Corollary 1 (WS complexity).** *In the hybrid recursive wavefront algorithm, if $n' = n^\alpha$ for some constant $\alpha \in [0,1)$ and if Lemma 1 is satisfied, then WS schedules the algorithm on distributed cache machine to have $S_p = \mathcal{O}(p\log n)$, $T_\infty(n) = \mathcal{O}\left(n^{1-\alpha}T_\infty^R(n^\alpha)\right)$, $Q_1(n) = T_1(n^{1-\alpha})Q_1^R(n^\alpha)$, $T_p(n) = \mathcal{O}(T_1(n)/p + T_\infty(n))$, and $Q_p(n) = \Theta\left(Q_1(n) + p(M/B)T_\infty(n)\right)$ and the latter two bounds are satisfied w.h.p. For the $T_p(n)$ and $Q_p(n)$ equations to hold, we assume that $T_1(n)$ is a polynomial of n, and choose n' such that $T_1(n') = \Omega(\log n)$.*

3.4.2 Modified space-bounded scheduler

In this section, we show how to modify a space-bounded scheduler(W-SB) [Chowdhury et al., 2013] so that it can execute a recursive wavefront algorithm cache-optimally with near-optimal parallelism.

The *parallel memory hierarchy (PMH)* [Blelloch et al., 2011] model models the memory hierarchy of many real parallel systems. It is described in Section 1.4.2 and we use this memory model for the scheduler. The W-SB scheduler combines two ideas: timing functions presented in Section 3.2 and space-bounded scheduler introduced in [Chowdhury et al., 2010, Chowdhury et al., 2013].

For each recursive function call, our W-SB scheduler accepts three hints: start-time, end-time and working set size (i.e., total size of all regions in the DP table accessed by the function call). Given an implementation of a standard recursive algorithm with each function call annotated with those three hints, the W-SB can automatically generate a recursive wavefront implementation (similar to the one on the right hand side of Figure 3.1). From the given start-times, the scheduler determines the lowest start-time and executes the tasks that can be executed at that lowest start-time. Since the scheduler knows all the cache sizes, as soon as the working set size of any function executing on a processor under a cache fits into that cache, the scheduler anchors the function to that cache in the sense that all recursive function calls made by that function and its descendants will only be executed by the processors under that anchored cache. This approach of limiting migration of tasks ensures cache-optimality [Chowdhury et al., 2013, Blelloch et al., 2011].

The W-SB scheduler is centralized and cache-aware. It schedules recursive wavefront algorithms in which the number of recursive function calls made by a function is upper bounded by a constant. The complete information related to a function call is given by by $\mathcal{T} = [\mathcal{S}_{\mathcal{F}}(X, Y_1, \ldots, Y_s), \mathcal{E}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)] : \mathcal{F}(X, Y_1, \ldots, Y_s)$. That is, this scheduler accepts start-time, end-time and problems size for each function call. From the given start-times, the scheduler determines the lowest start-time and executes the tasks that can be executed at that lowest start-time. If there are p processors in the machine, then at any point in time a maximum of p tasks can be executed in parallel. The subtask $\mathcal{T}$ of a task $\mathcal{T}'$ is said to be *anchored* to a cache at some level if $\mathcal{T}$ fits into the cache but not $\mathcal{T}'$. When a task $\mathcal{T}$ is anchored to a cache $\mathcal{M}$, the task and all of its subtasks at all granularities will be in the same cache and they will not be migrated to any other cache that is not part of the subtree of caches rooted at $\mathcal{M}$. A similar argument holds for all cache levels and hence when a task fits into a cache as much useful work is done as possible.

A *task* is a subproblem that writes to a region of an input DP table and reads from one or more regions. The complete information related to a task is denoted by $\mathcal{T} = [\mathcal{S}_{\mathcal{F}}(X, Y_1, \ldots, Y_s),$ $\mathcal{E}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)] : \mathcal{F}(X, Y_1, \ldots, Y_s)$. That is, this scheduler accepts start-time, end-time and problems size for each function call/task. From the given start-times, the scheduler determines the lowest start-time and executes the tasks that can be executed at that lowest start-time. If there are p processors in the machine, then at any point in time a maximum of p tasks can be executed in parallel. The subtask $\mathcal{T}$ of a task $\mathcal{T}'$ is said to be *anchored* to a cache at some level if $\mathcal{T}$ fits into the cache but not $\mathcal{T}'$. When a task $\mathcal{T}$ is anchored to a cache $\mathcal{M}$, the task and all of its subtasks at all granularities will be in the same cache and they will not be migrated to any other cache that is not part of the subtree of caches rooted at $\mathcal{M}$. A similar argument holds for all cache levels and hence when a task fits into a cache as much useful work is done as possible.

Example. For simplicity of exposition, we describe here the process by which W-SB schedules a recursive wavefront algorithm for the LCS problem on a 16×16 DP table X with a chunk size of 4×4 after which it switches to standard 2-way recursive algorithm. In the PMH model, we assume $h = 2$, $f_2 = 2$, $f_1 = 1$, and M_1 has just enough memory to execute 4×4 subproblem. As per the scheduling method shown in Figure 3.1, the base case size n' will be chosen by the scheduler automatically such that it is equal to M_1 as LCS can be implemented in linear space. In the first iteration the wavefront level is 0 and the only task that would be executed is $\{[0, 6] : \mathcal{A}(X_{11,11})\}$. As this task fits in M_1, it will be executed completely before it is knocked off from the cache. During the execution of first

iteration the second smallest start-time i.e., 4 will be computed. In the second iteration all tasks that have start-time 4 will be executed. The two tasks that will be executed by the two processors are $\{[4, 10] : \mathcal{A}(X_{11,12}); [4, 10] : \mathcal{A}(X_{11,21})\}$. Following the process, one possible scheduling of the tasks is shown in Table 3.7.

Iter.	Processor 1	Processor 2
1	$[0, 6] : \mathcal{A}(X_{11,11})$	−
2	$[4, 10] : \mathcal{A}(X_{11,12})$	$[4, 10] : \mathcal{A}(X_{11,21})$
3	$[8, 14] : \mathcal{A}(X_{11,22})$	$[8, 14] : \mathcal{A}(X_{21,11})$
4	$[8, 14] : \mathcal{A}(X_{12,11})$	−
5	$[12, 18] : \mathcal{A}(X_{21,21})$	$[12, 18] : \mathcal{A}(X_{12,21})$
6	$[12, 18] : \mathcal{A}(X_{21,12})$	$[12, 18] : \mathcal{A}(X_{12,12})$
7	$[16, 22] : \mathcal{A}(X_{21,22})$	$[16, 22] : \mathcal{A}(X_{22,11})$
8	$[16, 22] : \mathcal{A}(X_{12,22})$	−
9	$[20, 26] : \mathcal{A}(X_{22,21})$	$[20, 26] : \mathcal{A}(X_{22,12})$
10	$[24, 30] : \mathcal{A}(X_{22,22})$	−

1	2	4	6
2	3	5	7
3	6	7	9
5	8	9	10

Table 3.7: W-SB scheduled LCS recursive wavefront algorithm on a 2-processor machine. Left: Tasks executed by different processors. Right: Depiction of the tasks on the DP table.

> **Theorem 6 (W-SB complexity).** *Suppose a DP recurrence is evaluated by the $\mathcal{WI}$-$\mathcal{DP}$ in $N_\infty(n)$ parallel steps with ∞ processors. When the $\mathcal{WR}$-$\mathcal{DP}$ with switching point $n' \leq n$ (switching to a 2-way $\mathcal{R}$-$\mathcal{DP}$) evaluates the recurrence, we achieve the following bounds under the modified space-bounded (W-SB) scheduler:*
> * *span, $T_\infty(n) = \mathcal{O}\left(N_\infty(n/n')\left(\mathcal{O}(\log n) + T_\infty^R(n')\right)\right)$,*
> * *parallel cache complexity, $Q_p(n) = \mathcal{O}(Q_1(n))$, and*
> * *extra space, $S_p(n) = \mathcal{O}(p \log n)$.*
> *We assume that $N_\infty(n)$ and $T_1(n)$ (work) are polynomials of n, and choose n' such that $T_1(n') = \Omega(\log n)$.*

Proof. The arguments for $T_\infty(n)$ and $S_p(n)$ are the same as those given in the proof of Theorem 5. The parallel cache complexity is found as follows.

When the working set size of a function call fits into a cache $\mathcal{M}$ the W-SB scheduler does not allow any recursive function calls made by that function or its descendants to migrate to other caches that are not a part of the subtree of caches rooted at $\mathcal{M}$. This implies the data is read completely and as much work as possible is done on this loaded cache data blocks in $\mathcal{M}$ before kicking them out of the cache. Hence, temporal cache locality is fully exploited at $\mathcal{M}$. As shown in [Chowdhury et al., 2013, Blelloch et al., 2011] being able to achieve cache-optimality for working set sizes that are smaller than the cache size by at most a constant factor guarantees $Q_p(n) = \Theta(Q_1(n))$ for our algorithms.   □

> **Corollary 2 (W-SB complexity).** *If Lemma 1 is satisfied, if $\Theta\left(n^d\right)$ is the size of the input DP table, then the W-SB scheduler schedules the hybrid recursive algorithm algorithm on a PMH machine model to achieve the following bounds:*
> * *If $n' = n^\alpha$ for some constant $\alpha \in [0,1)$, then $T_\infty(n) = \mathcal{O}\left(n^{1-\alpha}T_\infty^R(n^\alpha)\right)$.*
> * *If $n' = \Theta\left(M^{1/d}\right)$, where M is a cache-size, then $T_\infty(n) = \mathcal{O}\left((n/M^{1/d})(\log n + T_\infty^R(M^{1/d}))\right)$.*
> *In both the cases, we have $Q_p(n) = \Theta\left(Q_1(n)\right)$ and $S_p(n) = \mathcal{O}\left(p \log n\right)$ or $S_p(n) = \mathcal{O}\left(pS_1(n)\right)$.*

3.5 Further improvement of parallelism

We develop algorithms to asymptotically improve the parallelism of the divide-and-conquer wavefront ($\mathcal{WR}$-$\mathcal{DP}$) algorithms. We explain the algorithm in simple words (without pseudocode) and then analyze it for its complexities.

The new algorithm works as follows. We set the switching point as $n' = f(n)$ where $f(n) = o(n)$. The standard $\mathcal{WR}$-$\mathcal{DP}$ algorithms switches to an $\mathcal{R}$-$\mathcal{DP}$ chunk when the problem parameter n reduces to less than or equal to n'. In the new algorithm, when the problem parameter n reduces to less than or equal to $f(n)$, instead of calling the $\mathcal{R}$-$\mathcal{DP}$ algorithm we call the same divide-and-conquer wavefront algorithm. Now the divide-and-conquer wavefront algorithm works on a subproblem with subproblem parameter $f(n)$ and switches the algorithm when the subprobem parameter reduces to less than or equal to $f(f(n))$. This process continues recursively.

Complexity analysis. Let $T_\infty(n, f(n))$ denote the span of this new algorithm. Let $T_\infty(n)$ denote the span of the pure $\mathcal{WR}$-$\mathcal{DP}$ algorithm with switch-point $n' = 1$. Then,

$$T_\infty(n, f(n)) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \Theta\left(\frac{n}{f(n)}\right) \cdot (T_\infty(f(n), f(f(n))) + \mathcal{O}(\log n) + \Theta(1)) & \text{if } n > 1. \end{cases}$$

Let $T_1(n, f(n))$ denote the work of this new algorithm. Let $T_1(n)$ denote the work of the pure $\mathcal{WR}$-$\mathcal{DP}$ algorithm with switch-point $n' = 1$. Then,

$$T_1(n, f(n)) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T_1\left(\frac{n}{f(n)}\right) \cdot (T_1(f(n), f(f(n))) + \mathcal{O}(\log n) + \Theta(1)) & \text{if } n > 1. \end{cases}$$

When $f(n) = \log n$, we have

$$\begin{aligned}
T_\infty(n, \log n) &\leq c\left(\frac{n}{\log n}\right) \cdot (T_\infty(\log n, \log \log n) + c' \log n) \\
&\leq c\left(\frac{n}{\log n}\right)\left(c\left(\frac{\log n}{\log \log n}\right) \cdot (T_\infty(\log \log n, \log \log \log n) + c' \log \log n) + c' \log n\right) \\
&= c^2\left(\frac{n}{\log \log n}\right) \cdot T_\infty(\log \log n, \log \log \log n) + cc'(c+1)n \\
&= c^{\log^* n} \cdot n + cc'\left(c^{(\log^* n)-1} + c^{(\log^* n)-2} + \cdots + 1\right)n \leq c^{\log^* n} \cdot n + c^{(\log^* n)+1}c'n \\
&= \Theta\left(c^{\log^* n} \cdot n\right)
\end{aligned}$$

It is easy to see that this span is asymptotically better (or smaller) than $\Theta(n \log n)$ as $\Theta\left(c^{\log^* n}\right) = o(\log n)$.

Let $L_i(n) = \log \log \cdots \log n$ Assuming $T_1(n)$ is a polynomial function of n, the total work $T_1(n, \log n)$ can be computed as

$$T_1(n, \log n) = T_1\left(\frac{n}{\log n}\right)(T_1(\log n, \log \log n) + c \log n)$$

$$= T_1\left(\frac{n}{\log n}\right) \cdot T_1(\log n, \log \log n) + cT_1\left(\frac{n}{\log n}\right) \cdot \log n$$

$$= T_1\left(\frac{n}{\log n}\right) \cdot T_1\left(\frac{\log n}{\log \log n}\right) \cdot (T_1(\log \log n, \log \log \log n) + c \log \log n) + cT_1\left(\frac{n}{\log n}\right) \cdot \log n$$

$$= T_1\left(\frac{n}{\log n}\right) \cdot T_1\left(\frac{\log n}{\log \log n}\right) \cdot T_1(\log \log n, \log \log \log n)$$

$$\quad + cT_1\left(\frac{n}{\log \log n}\right) \cdot \log \log n + cT_1\left(\frac{n}{\log n}\right) \cdot \log n$$

$$= T_1\left(\frac{n}{\log n}\right) \cdot T_1\left(\frac{\log n}{\log \log n}\right) \cdots \Theta(1) + c'T_1(n)$$

$$= \frac{T_1(n)}{T_1(\log n)} \cdot \frac{T_1(\log n)}{T_1(\log \log n)} \cdots \Theta(1) + c'T_1(n)$$

$$= \Theta(T_1(n))$$

3.6 Experimental results

In this section we present experimental results showing performance of recursive wavefront algorithms for the parenthesis and the 2D FW-APSP problems. We also compare performance of those algorithms with the corresponding standard 2-way recursive divide-and-conquer and the original cache-oblivious wavefront (COW) algorithms [Tang et al., 2015]. The recursive wavefront algorithms were implemented by Jesmin Jahan Tithi.

We used C++ with Intel Cilk™ Plus extension to implement all algorithms presented in this section. Therefore, all implementations basically used the work-stealing scheduler provided by Cilk™ runtime system. All programs were compiled with `-O3 -ip -parallel -AVX -xhost` optimization parameters. To measure cache performance we used PAPI-5.3 [PAP,]. Table 3.8 lists the systems on which we ran our experiments.

3.6.1 Projected parallelism

Since we still do not have shared-memory multi-core machines with thousands of cores, we have used the Intel Cilkview scalability analyzer to compute the ideal parallelism and burdened span of the following implementations:
- (i) recursive wavefront algorithm that does not switch to the 2-way non-wavefront recursive algorithm and instead directly uses an iterative basecase (`wave`),
- (ii) recursive wavefront algorithm that switches to the 2-way recursive divide-and-conquer at some point (`wave-hybrid`),
- (iii) standard 2-way recursive divide-and-conquer algorithm (`CO_2Way`).

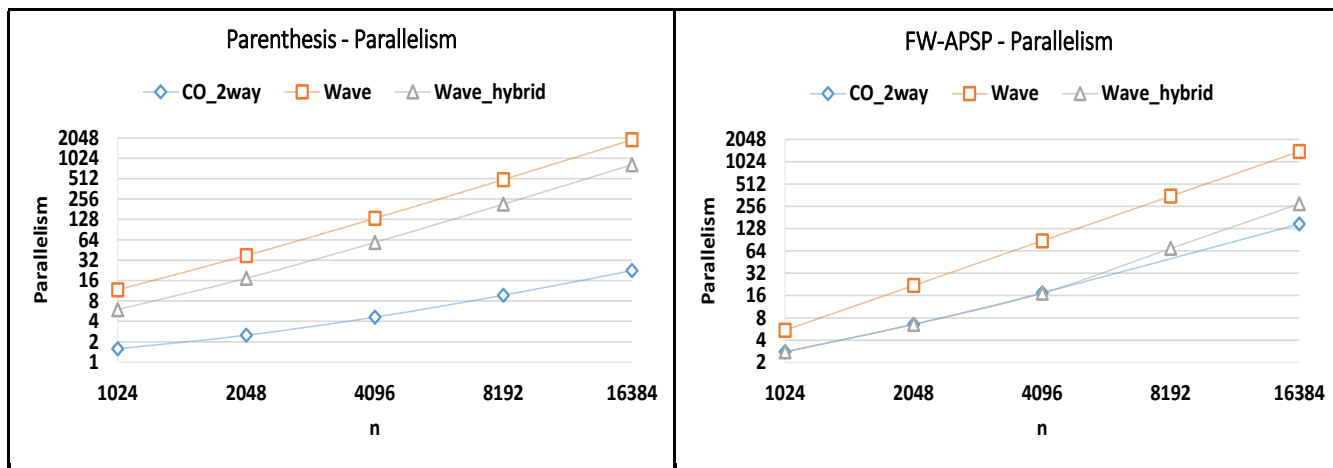| Model	E5-2680	E5-4650	E5-2680
Cluster	Stampede [Sta,]	Stampede [Sta,]	Comet [Com,]
#Cores	2x8	4x8	2x12
Frequency	2.70GHz	2.70GHz	2.50GHz
L1	32K	32K	32K
L2	256K	256K	256K
L3	20480K	20480K	30720K
Cache-line size	64B	64B	64B
Memory	64GB	1TB	64GB
Compiler	15.0.2	15.0.2	15.2.164
OS	CentOS 6.6	CentOS 6.6	CentOS 6.6

Table 3.8: System specifications.

Figure 3.2: Projected scalability of new recursive wavefront algorithms by Cilkview Scalability Analyzer. The numbers basically denote till how many cores the implementation should scale linearly.

For wave-hybrid, we have used $n' = \max\{256, \text{power of 2 closest to } n^{2/3}\}$. In order to reduce overhead of recursion and to take advantage of vectorization we switch to an iterative kernel when n becomes sufficiently small (e.g., 64 for wave, wave-hybrid and CO_2Way).

Figure 3.2 shows the scalability results reported by Cilkview for algorithms solving the parenthesis problem and Floyd-Warshall's APSP. These parallelism numbers show that recursive wavefront algorithms scale much better than standard 2-way recursive divide-and-conquer algorithms.

3.6.2 Running time and cache performance

Figure 3.3 shows performance of the following on a 16-core Sandy Bridge machine: (i) wave, (ii) wave-hybrid, (iii) CO_2Way, and (iv) our original cache-oblivious wavefront (COW) algorithms with atomic locks from [Tang et al., 2015]. For wave-hybrid, we have used $n' = \max\{256, \text{power of 2 closest to } n^{2/3}\}$. In order to reduce overhead of recursion and to take advantage of vectorization we switch to an iterative kernel when n becomes sufficiently small (e.g., 64 for wave, wave-hybrid and CO_2Way). It is clear from the figures that wave and wave-hybrid algorithms perform better than CO_2Way and the COW algorithms for all cases. For parenthesis problem, wave is $2.6\times$, and wave-hybrid is $2\times$ faster

than `CO_2Way`. Similarly, number of cache misses of `CO_2Way` is slightly higher than that of both `wave` and `wave-hybrid`. For Floyd-Warshall's APSP, `wave` is 18%, and `wave-hybrid` is 10% faster than `CO_2Way`. Therefore, even with 16 cores, the impact of improvements in parallelism and cache-misses is visible on the running time. On the other hand, though COW algorithms have excellent theoretical parallelism, their implementations heavily use atomic locks, which may have impacted their performance negatively for large n and for DP dimension $d > 1$.
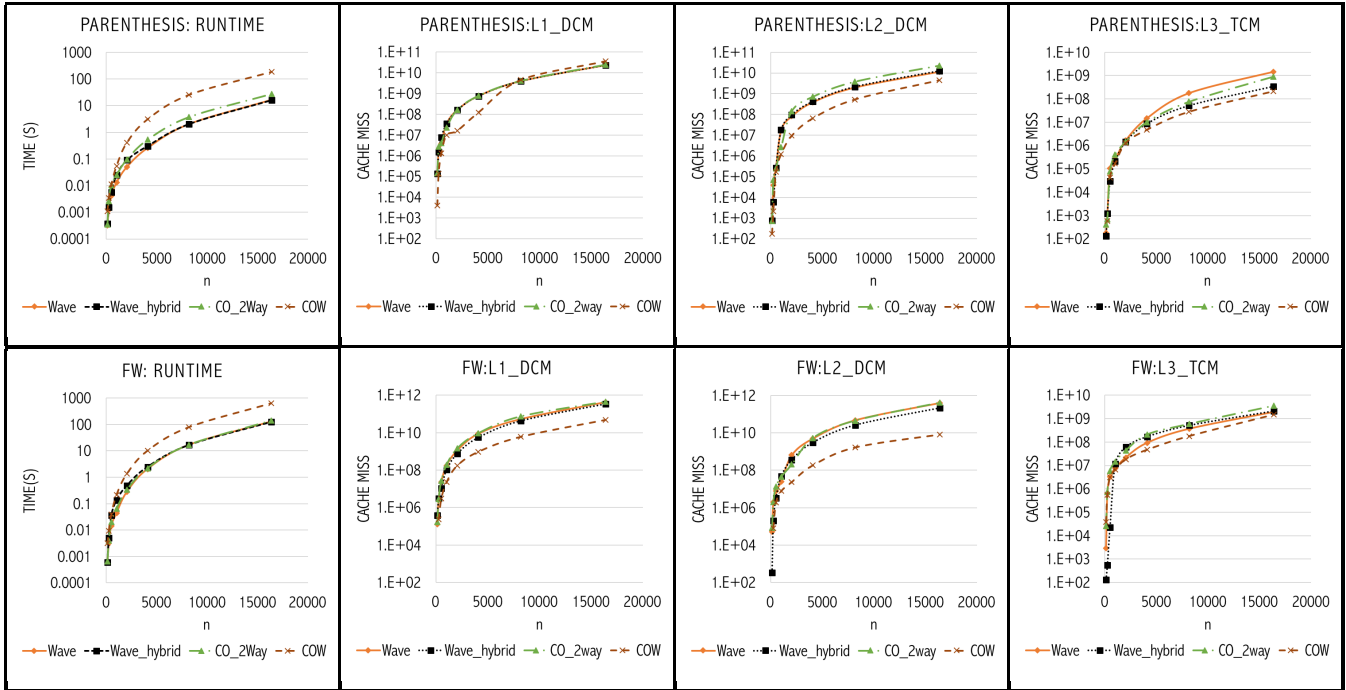

Figure 3.3: Runtime and cache misses in three levels of caches for classic 2-way recursive divide-and-conquer, COW and recursive wavefront algorithms for Parenthesis and 2D FW-APSP Problems. All programs were run on 16 core machines in Stampede. All implementations used Cilk Plus's work-stealing scheduler.

We have obtained performance results on a 24-core Haswell machine (shown in Figure 3.4). Value of n' and size of iterative kernel were determined in the same way as we did on Stampede. For FW-APSP, `wave` is 15% and `wave-hybrid` is 10% faster than `CO_2Way`. Although we see improvement in L1 and L2 cache misses, number of L3 misses is worse probably due to the increased parallelism. For parenthesis problem, `wave` is 16% and `wave-hybrid` is 18% faster than `CO_2Way`, and we see only improvement in the L3 cache misses.

On a 32-core Sandy Bridge machine, `wave` for FW-APSP runs 73% faster and `wave-hybrid` runs 69% faster than `CO_2Way`. On the other hand, for the parenthesis problem both `wave` and `wave-hybrid` are 2.1× faster than `CO_2Way`.

3.7   Conclusion and open problems

In this chapter we presented a framework to semi-automatically discover divide-and-conquer algorithms that achieve extremely high parallelism retaining excellent cache-efficiency
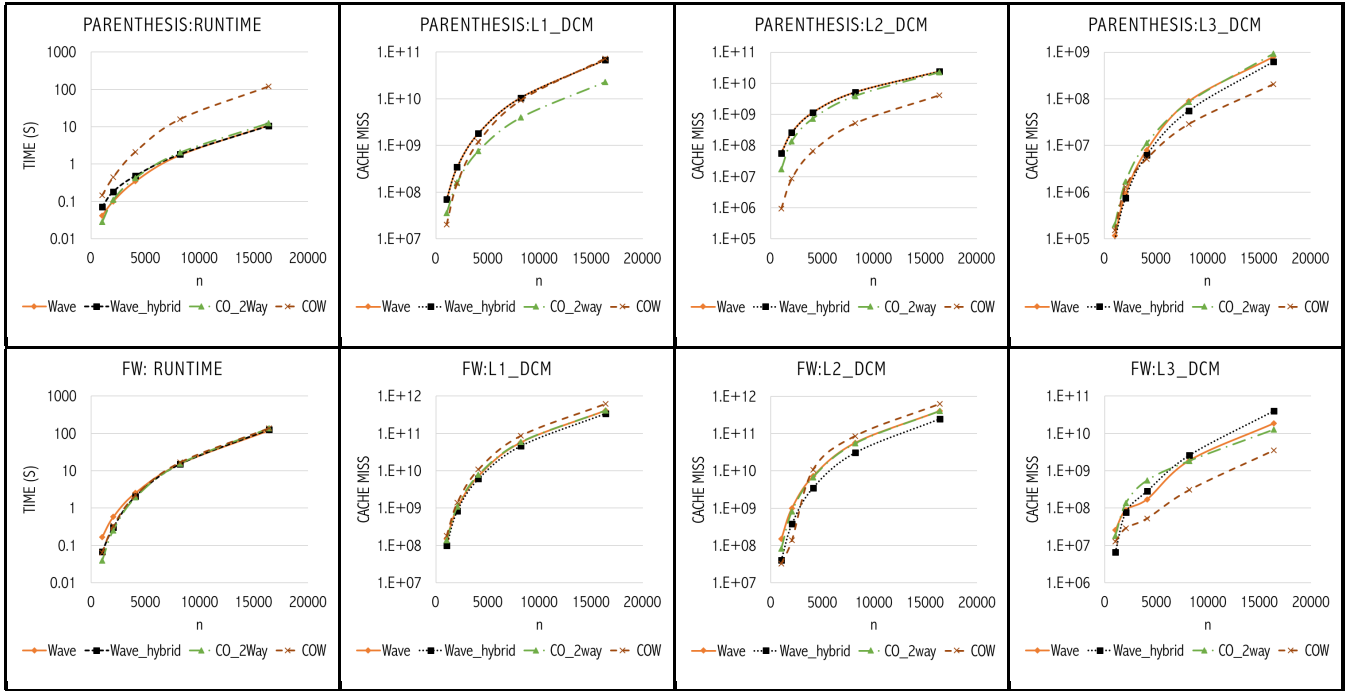
Figure 3.4: Runtime and cache misses in three levels of caches for classic 2-way recursive divide-and-conquer, COW and recursive wavefront algorithms for Parenthesis and 2D FW-APSP Problems. All programs were run on 24 core machines in Comet. All implementations used cilk plus's work-stealing scheduler.

(i.e., temporal locality). The presented $\mathcal{WR}$-$\mathcal{DP}$ algorithms are theoretically fastest dynamic programming implementations.

Some open problems that could be investigated are as follows:

⋆ [*Full automation.*] Completely automate the discovery of $\mathcal{WR}$-$\mathcal{DP}$ algorithms (then the AUTOGEN-WAVE framework will be transformed to AUTOGEN-WAVE algorithm) including computing completion-time functions and computing $ra()$ − race avoidance condition.

⋆ [*Improve span to* $\Theta(n)$.] The $\mathcal{WR}$-$\mathcal{DP}$ algorithms have the best-case span of $\Theta(n \log n)$. Is it possible to improve this best-case span to $\Theta(n)$?

⋆ [$\mathcal{WR}$-$\mathcal{DP}$ *algorithms for matrix problems.*] Design $\mathcal{WR}$-$\mathcal{DP}$ algorithms for several non-DP matrix problems (e.g.: QR decomposition, SVD decomposition, etc). A very good source of matrix algorithms is [Golub and Van Loan, 2012].

Chapter 4

An Efficient Divide-&-Conquer Viterbi Algorithm

The Viterbi algorithm is used to find the most likely path through a hidden Markov model given an observed sequence, and has numerous applications. Due to its importance and high computational complexity, several algorithmic strategies have been developed to parallelize it on different parallel architectures. However, none of the existing algorithms for the Viterbi decoding problem is simultaneously cache-efficient and cache-oblivious. Being oblivious of machine resources (e.g., caches and processors) while also being efficient promotes portability.

In this chapter, we present an *efficient cache- and processor-oblivious Viterbi algorithm* based on *rank convergence* solving the two decade old open problem. The algorithm builds upon the parallel Viterbi algorithm of Maleki et al. (PPoPP 2014). We provide empirical analysis of our algorithm by comparing it with Maleki et al.'s algorithm. To the best of our knowledge, this is the first work that presents a provably cache-efficient cache-oblivious parallel Viterbi algorithm.

4.1 Introduction

The Viterbi algorithm [Viterbi, 1967, Viterbi, 1971, Forney Jr, 1973, Omura, 1969, Lou, 1995] which was proposed by Andrew J. Viterbi (co-founder of Qualcomm) in 1967 is a dynamic programming algorithm that finds the most probable sequence of hidden states from a given sequence of observed events. As each observation is a probabilistic function of a hidden state, the discrete-time finite-state Markov process is modeled using a hidden Markov model (HMM). An excellent tutorial on HMM can be found in [Rabiner, 1989].

The Viterbi algorithm has numerous real world applications. Viterbi developed his algorithm as a decoding method for convolutional codes in noisy communication channels. Since then, the algorithm has found wide applications in Qualcomm's CDMA technology [Gilhousen et al., 1991, Kand and Willson, 1998, Feldman et al., 2002], TDMA system for GSM [Costello et al., 1998], television sets [Nam and Kwak, 1998], satellite and space communication [Heller and Jacobs, 1971], speech recognition [Rabiner, 1989, Soong and Huang, 1991, Franzini et al., 1990], handwritten word recognition [Kundu et al., 1988], hand gesture recognition [Chen et al., 2003], modems [Ungerboeck, 1982], magnetic recording systems [Kobayashi, 1971a, Kobayashi, 1971b], biological sequence analysis [Hender-

87

son et al., 1997, Durbin et al., 1998], parsing context-free grammars [Klein and Manning, 2003, Schmid, 2004], and part-of-speech tagging [Cutting et al., 1992, Taylor and Black, 1998]. In 2006, Forney [Forney Jr, 2005] asserted that the Viterbi decoder was being used in about a billion cellphones and that approximately 10^{15} bits of data were being decoded by the Viterbi algorithm in the digital television sets per second. Due to the immense significance of the algorithm and other contributions, Viterbi received the National Medal of Technology (US) in 2007 and the National Medal of Science (US) in 2008.

The 2^{14}-state big Viterbi decoder (BVD) [Collins, 1992] built by Jet Propulsion Laboratory (JPL) in 1992 for Galileo space mission is the largest Viterbi decoder in use. It is important to develop fast *parallel* Viterbi algorithms for gigantic number of states and timesteps for multicore and many core machines. When the input data of an algorithm is too large to fit into a cache, the time spent by the algorithm in block transfers (or IO) between adjacent levels of caches becomes more significant than the time taken for the CPU computations. In such cases, a *cache-efficient* Viterbi algorithm that minimizes the number of cache misses is desired. Though there have been a lot of efforts and successes in parallelizing the Viterbi algorithm, there is little work in the realm of designing cache-efficient algorithms for the Viterbi problem. To the best of our knowledge, we present the first cache-efficient Viterbi algorithm.

An algorithm that can be easily ported to different computing platforms from cellphones to supercomputers should be *cache-oblivious* [Frigo et al., 1999] and *processor-oblivious*. A cache-oblivious algorithm need not know the cache parameters such as cache size and block size. Similarly, a processor-oblivious algorithm need not know the number of processors on the machine it runs on. Several cache- and processor-oblivious algorithms for dynamic programs that are majorly based on *recursive divide-and-conquer* have been developed, analyzed, and implemented in [Cherng and Ladner, 2005, Chowdhury and Ramachandran, 2006, Chowdhury and Ramachandran, 2008, Chowdhury and Ramachandran, 2010, Tan et al., 2006, Bille and Stckel, 2012].

Our contributions. The major contributions of this chapter are summarized as follows:
  (1) We present an efficient cache- and processor-oblivious Viterbi algorithm based on rank convergence.
  (2) We present an efficient cache- and processor-oblivious recursive divide-and-conquer Viterbi algorithm for multiple instances of the problem.
  (3) We present experimental results comparing our algorithms with the existing fastest Viterbi algorithms.

Organization of the chapter. In Section 4.2, we give a simple cache-inefficient Viterbi algorithm based on recursive divide-and-conquer. In Section 4.3, we present a cache-efficient Viterbi algorithm for multiple instances of the problem. A parallel Viterbi algorithm based on rank convergence is described in Section 4.4. In Section 4.5, a cache-efficient parallel Viterbi algorithm based on rank convergence is presented. Finally we conclude.

4.2 Cache-inefficient Viterbi algorithm

In this section, we give a formal specification of the Viterbi algorithm, and describe a simple cache-inefficient Viterbi algorithm based on recursive divide-and-conquer.

Formal specification

The Viterbi algorithm is described as follows. We are given an observation space $O = \{o_1, o_2, \ldots, o_m\}$, state space $S = \{s_1, s_2, \ldots, s_n\}$, observations $Y = \{y_1, y_2, \ldots, y_t\}$, transition matrix A of size $n \times n$, where $A[i, j]$ is the transition probability of transiting from s_i to s_j, emission matrix B of size $n \times m$, where $B[i, j]$ is the probability of observing o_j at s_i, and initial probability vector (or initial solution vector) I, where $I[i]$ is the probability that $x_i = s_i$. Let $X = \{x_1, x_2, \ldots, x_t\}$ be a sequence of hidden states that generates $Y = \{y_1, y_2, \ldots, y_t\}$. Then the matrices P and P' of size $n \times t$, where $P[i, j]$ is the probability of the most likely path of getting to state s_i at observation o_j and $P'[i, j]$ stores the hidden state of the most likely path are computed as follows.

$$P[i, j] = \begin{cases} I[i] \cdot B[i, y_1] & \text{if } j = 1, \\ \max_{k \in [1,n]}(P[k, j-1] \times A[k, i] \times B[i, y_j]) & \text{if } j > 1. \end{cases}$$

$$P'[i, j] = \begin{cases} 0 & \text{if } j = 1, \\ \text{argmax}_{k \in [1,n]}(P[k, j-1] \times A[k, i] \times B[i, y_j]) & \text{if } j > 1. \end{cases}$$

There are two phases in the Viterbi algorithm. In the first phase called *forward phase*, the matrices P and P' are computed. In the second phase called *backward phase*, the sequence of hidden states X is computed from P' by traversing from the tth timestep of P' to its first timestep and recursively computing the argument max at each timestep. The time complexity of the algorithm implemented naively is $\Theta(n^2 t)$.

Quite often, the forward phase dominates the execution time. Hence, in this paper, we focus only on the forward phase of the algorithm, i.e., computing P and/or P'. Also, for all our algorithms, we assume that the matrices P, P', A, and B are stored in either *column-major order* or *z-Morton order*.

Irregular data dependency

Viterbi algorithm is a dynamic programming (DP) algorithm having *irregular dependency* as shown in Figure 4.1. It means that the data dependency does not follow a particular pattern for all cells in the matrix. The irregular dependency is due to the memory access $B[i, y_j]$ that depends on the value of y_j, which in turn depends on a problem instance. We believe the Viterbi problem is the first DP problem with irregular dependency for which an efficient cache-oblivious algorithm is being presented.

The elements accessed in B changes depending on the value of y_j. It is possible to convert the problem with irregular data dependency to a problem with regular dependency by transforming the matrix B of size $n \times m$ into another matrix C of size $n \times t$. An algorithm to construct matrix C from matrix B is given in Figure 4.2.
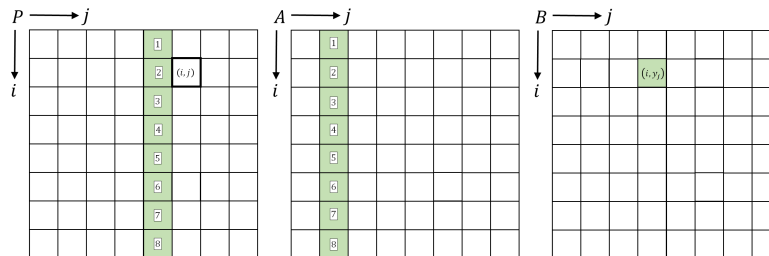

Figure 4.1: Dependency structure of the Viterbi DP with irregular dependency: cell (i, j) depends on the green cells.
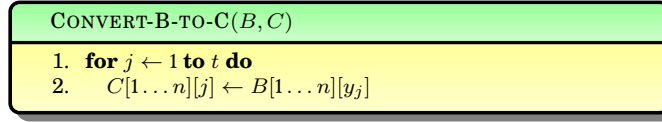
```
CONVERT-B-TO-C(B, C)
1.  for j ← 1 to t do
2.      C[1 . . . n][j] ← B[1 . . . n][y_j]
```

Figure 4.2: An algorithm to convert B matrix to C matrix.

After converting the B matrix to C, the Viterbi algorithm recurrence having regular dependency can be written as

$$P[i, j] = \begin{cases} I[i] \times C[i, 1] & \text{if } j = 1, \\ \max_{k \in [1,n]}(P[k, j-1] \times A[k, i] \times C[i, j]) & \text{if } j > 1. \end{cases} \quad (4.1)$$

The new dependency structure is given in Figure 4.3. The serial cache complexity to construct C matrix is $\Theta(nt/B)$.

Cache-inefficient algorithm

An iterative parallel and a recursive divide-and-conquer-based parallel Viterbi algorithms are given in Figure 4.4. The recursive divide-and-conquer algorithm was co-designed with Vivek Pradhan. A visual depiction of the recursive algorithm is given in Figure 4.4. As per the Viterbi recurrence, each cell (i, j) of matrix P depends on all cells of P at column $j - 1$, all cells of A at column i, and the cell (i, y_j) of B. The function $\mathcal{A}_{vit}$ fills jth column of P denoted by X using $(j - 1)$th column denoted by U using a divide-and-conquer approach. To compute each column of P, the entire matrix of A should be read. Hence the recursive algorithm is cache-inefficient. In both algorithms, the cells in each stage (or timestep) are computed in parallel and the stages are computed sequentially.

Complexity analysis. The serial cache complexity of the iterative algorithm is computed as $\sum_{j=1}^{t} \sum_{i=1}^{n} \mathcal{O}(n/B) = \mathcal{O}(n^2 t/B)$ and that of the divide-and-conquer algorithm is computed as follows. Let $Q_{\mathcal{A}}(n)$ denote the serial cache complexity of $\mathcal{A}_{vit}$ on a matrix of size $n \times n$. Then

$$Q_{\mathcal{A}}(n) = \begin{cases} \mathcal{O}(n^2/B + n) & \text{if } n^2 \leq \gamma_{\mathcal{A}} M, \\ 4Q_{\mathcal{A}}(n/2) + \mathcal{O}(1) & \text{otherwise.} \end{cases}$$

where, $\gamma_{\mathcal{A}}$ is a suitable constant. Solving, $Q_{\mathcal{A}}(n) = \mathcal{O}(n^2/B + n)$. Thus, the serial cache complexity of the recursive algorithm is $\mathcal{O}(n^2 t/B + nt)$ when n^2 is too large to fit in cache.

Both the iterative and recursive algorithms have spatial locality, but they do not have any temporal locality. Hence, these algorithms are not cache-efficient.

The span of the iterative algorithm is $\Theta(nt)$, as there are t time steps and it takes n time steps to update a cell of P. The span of the recursive algorithm is computed as follows. Let
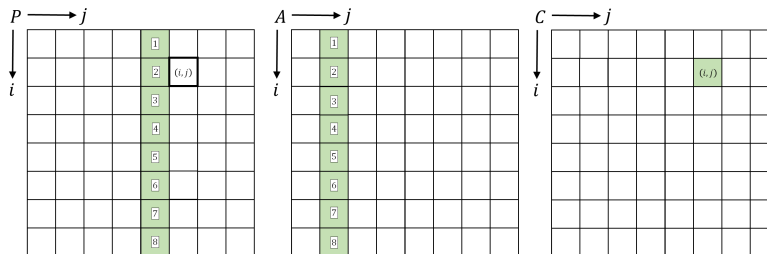

Figure 4.3: Dependency structure of the Viterbi DP with regular dependency: cell (i, j) depends on the green cells.
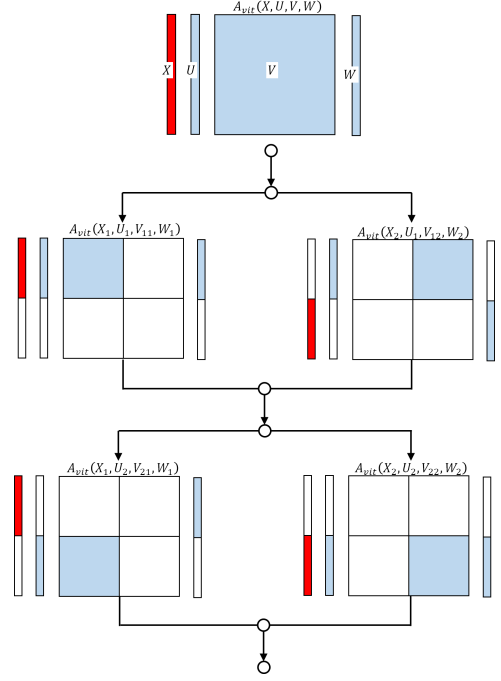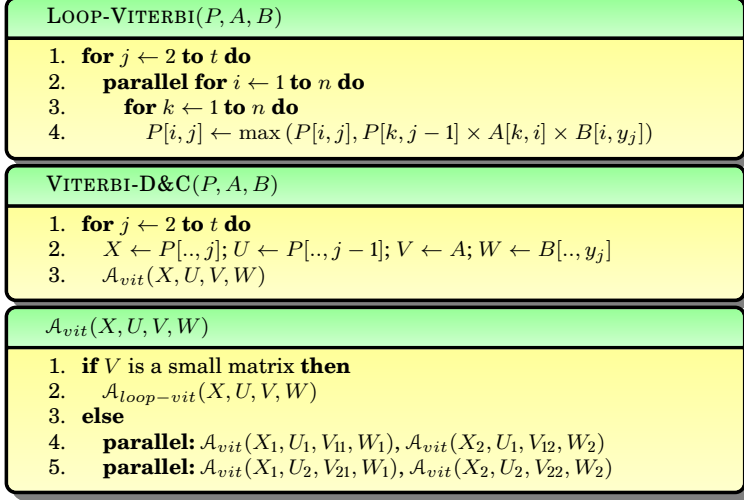
Figure 4.4: (a) Parallel iterative Viterbi algorithm and cache-inefficient parallel recursive divide-and-conquer-based Viterbi algorithm. Note that we need not store the entire matrix P. We can keep a single column vector to compute the last column of matrix P. (b) A parallel cache-inefficient recursive divide-and-conquer procedure for the Viterbi algorithm. The initial call to the function is $\mathcal{A}_{vit}(X, U, V, W)$. The first array is the array of matrix P at column j, the second array is the $(j-1)$th column of P, the third matrix is the matrix A and the last array is the array of B at column y_j. The algorithm updates the red regions using data from the blue regions.

$T_A(n)$ denote the span of $\mathcal{A}_{vit}$ on a matrix of size $n \times n$. Then

$$T_A(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_A(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

Solving, $T_A(n) = \Theta(n)$, which implies the span of the divide-and-conquer-based Viterbi algorithm is $\Theta(nt)$.

The divide-and-conquer algorithm does not seem to improve the cache complexity over the iterative algorithm. This is due to the fact that to update one column of matrix P, we have to read the entire matrix A, which costs us $\Theta\left(\frac{n^2}{B}\right)$ cache misses. If matrix A is taken out of the Viterbi recurrence, then the total cache complexity would have been $\mathcal{O}\left(\frac{n^2}{BM}t\right)$. One might think that we can read the matrix A once and update many rows of P at once. But, this is not possible because until the entire first column of P is updated, the second column of P cannot be updated and until the entire second column of P is updated, the third column of P cannot be updated. We cannot partially update the future columns of P as in the case of Floyd-Warshall APSP algorithm, because in the Floyd-Warshall APSP algorithm, the partial values are useful but here, the partial values are not useful and the cells have to be updated again and again increasing the work asymptotically.

4.3 Cache-efficient multi-instance Viterbi algorithm

In this section, we present a cache-efficient Viterbi algorithm for multiple instances of the problem.

We saw in Section 4.2 that the recursive algorithm has no temporal locality because to compute each column of P ($\Theta\left(n^2\right)$ work), we have to scan the entire matrix A ($\Theta\left(n^2\right)$ space). It is unclear how exactly to get or if at all we can get temporal locality from the time dimension. Therefore, we use a different approach to exploit temporal cache locality from the divide-and-conquer Viterbi algorithm. Instead of solving only one instance of the Viterbi problem, we solve q instances of the problem simultaneously. Two problems that have the same transition matrix A and emission matrix B are termed two instances of the same problem. The core idea of the algorithm comes from the fact that by scanning the transition matrix A only once, a particular column of matrix P can be computed for n instances of the problem. Figure 4.6 gives a cache-efficient and cache- and processor-oblivious recursive divide-and-conquer-based parallel Viterbi algorithm that can be used to solve q instances of the problem at once.

In Figure 4.6, in the function $\mathcal{A}_{vit}(X, U, V, W)$, the matrix U is an $n \times q$ matrix obtained by concatenating $(j-1)$th columns of q matrices $P_1, P_2, \ldots, P_q$, where P_i is the most likely path probability matrix of problem instance i. The algorithm computes X, which is a concatenation of jth columns of the q problem instances. Each problem instance i has a different observations vector $Y_i = \{y_{i1}, y_{i2}, \ldots, y_{it}\}$. The matrix W is a concatenation of y_jth columns of matrix B obtained from different observations i.e., W is a concatenation of $B[y_{1j}], B[y_{2j}], \ldots, B[y_{qj}]$. We use $X_T, X_B, X_L,$ and X_R to represent the top half, bottom half, left half, and right half of the matrix X, respectively. Executing the divide-and-conquer algorithm once computes the second column of all matrices P_1 to P_q. Executing the algorithm again computes the third column of the q matrices. Executing the algorithm t times, the last column of all problem instances would be filled. Note that for each time step, the matrix W should be constructed again and again.

It is important to note that the structure of the function $\mathcal{A}_{vit}$ is similar to the recursive divide-and-conquer-based in-place matrix multiplication algorithm. When $q = n$, both the algorithms have 8 recursive function calls in two parallel steps and the base case consist of three loops. Therefore, the complexity analysis of the multi-instance Viterbi algorithm will be similar to that of the matrix multiplication algorithm.

Complexity analysis. The serial cache complexity of the multi-instance recursive algorithm given in Figure 4.6 is computed as follows. Let $Q_A(n, q)$ denote the serial cache complexity of $\mathcal{A}_{vit}$ on a matrix of size $n \times q$, and let n and q be powers of two. Then

$$Q_A(n, q) = \begin{cases} \mathcal{O}\left(n^2/B + n\right) & \text{if } n^2 + nq \leq \gamma_A M, \\ 8Q_A\left(n/2, q/2\right) + \mathcal{O}\left(1\right) & \text{if } n = q, \\ 2Q_A\left(n, q/2\right) + \mathcal{O}\left(1\right) & \text{if } n < q, \\ 4Q_A\left(n/2, q\right) + \mathcal{O}\left(1\right) & \text{if } n > q. \end{cases}$$

where, γ_A is a suitable constant. Solving, we get

$$Q_A(n, q) = \mathcal{O}\left(\frac{n^2 q}{B\sqrt{M}} + \frac{n^2 q}{M} + \frac{n(n+q)}{B} + 1\right)$$

Thus, the serial cache complexity of the divide-and-conquer Viterbi algorithm for q problem instances consisting of t timesteps is $\mathcal{O}\left(n^2 qt/(B\sqrt{M}) + n^2 qt/M + n(n+q)t/B + t\right)$. As

```
VITERBI-MULTI-INSTANCE-D&C(P_1, P_2, ..., P_q, A, B, t)
```

1. **for** $j \leftarrow 2$ **to** t **do**
2. $X \leftarrow [P_1[.., j], P_2[.., j], ..., P_q[.., j]]; U \leftarrow [P_1[.., j-1], P_2[.., j-1], ..., P_q[.., j-1]]$
3. $V \leftarrow A; W \leftarrow [B[.., y_{1j}], B[.., y_{2j}], ..., B[.., y_{qj}]]$
4. $\mathcal{A}_{vit}(X, U, V, W)$

```
A_vit(X_{n×q}, U_{n×q}, V_{n×n}, W_{n×q})
```

1. **if** X and V are small matrices **then**
2. $\mathcal{A}_{loop-vit}(X, U, V, W)$
3. **else if** $q > n$ **then**
4.   **parallel:** $\mathcal{A}_{vit}(X_L, U_L, V, W_L), \mathcal{A}_{vit}(X_R, U_R, V, W_R)$
5. **else if** $q < n$ **then**
6.   **parallel:** $\mathcal{A}_{vit}(X_T, U_T, V_{11}, W_T), \mathcal{A}_{vit}(X_B, U_T, V_{12}, W_B)$
7.   **parallel:** $\mathcal{A}_{vit}(X_T, U_B, V_{21}, W_T), \mathcal{A}_{vit}(X_B, U_B, V_{22}, W_B)$
8. **else**
9.   **parallel:** $\mathcal{A}_{vit}(X_{11}, U_{11}, V_{11}, W_{11}), \mathcal{A}_{vit}(X_{12}, U_{12}, V_{11}, W_{12}), \mathcal{A}_{vit}(X_{21}, U_{11}, V_{12}, W_{21}), \mathcal{A}_{vit}(X_{22}, U_{12}, V_{12}, W_{22})$
10.   **parallel:** $\mathcal{A}_{vit}(X_{11}, U_{21}, V_{21}, W_{11}), \mathcal{A}_{vit}(X_{12}, U_{22}, V_{21}, W_{12}), \mathcal{A}_{vit}(X_{21}, U_{21}, V_{22}, W_{21}), \mathcal{A}_{vit}(X_{22}, U_{22}, V_{22}, W_{22})$

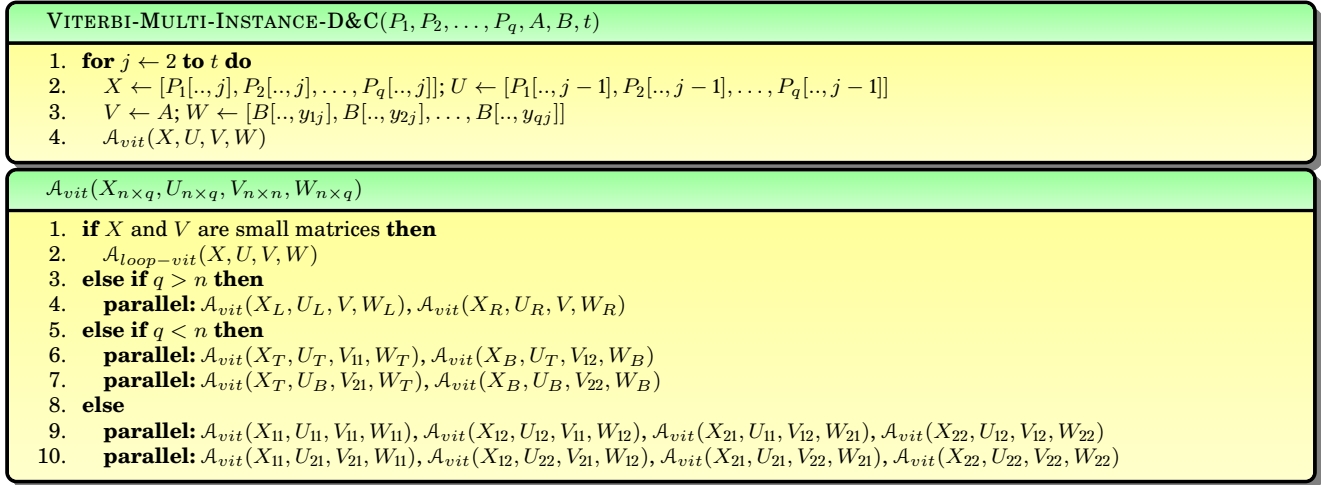Figure 4.5: Cache-efficient parallel recursive divide-and-conquer-based Viterbi algorithm. The matrix U is constructed by combining column $j-1$ of each of the q problem instances $P_1, ..., P_q$. Note that we need not store the entire matrices $P_1, ..., P_q$.
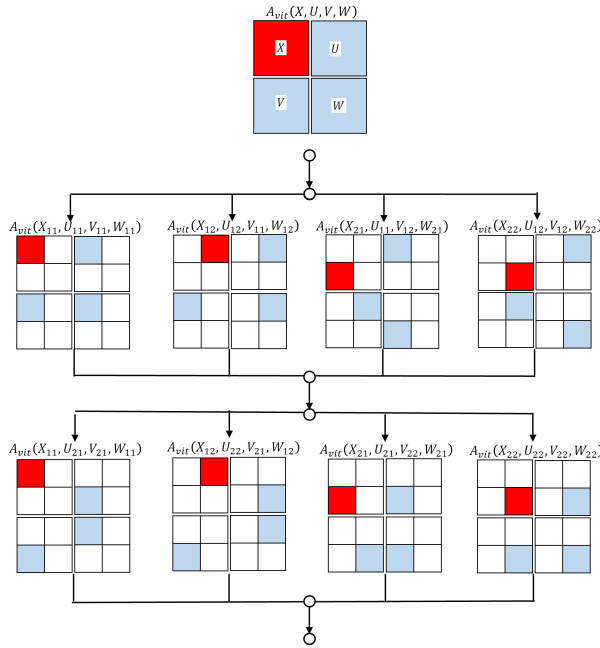
Figure 4.6: A multi-instance cache-efficient parallel recursive divide-and-conquer-based procedure for the Viterbi algorithm that can be used to solve q instances of the problem. We have chosen $q = n$ for simplicity. The initial call to the function is $\mathcal{A}_{vit}(X, U, V, W)$. The first matrix is a collection of jth columns of matrix P of n instances of the problem, the second matrix is a collection of $(j-1)$th columns of P of n instances, the third matrix is the matrix A that is assumed same for every instance of the problem and the last matrix W is a matrix of column y_j for n different instances. The algorithm updates the red regions using data from the blue regions.

the algorithm exploits temporal locality, it is cache-efficient.

The span of the algorithm is still $\Theta(nt)$, and its analysis is similar to that in Section 4.2.

4.4 Viterbi algorithm using rank convergence

In this section, we briefly describe and improve Maleki et al.'s approach [Maleki et al., 2014] to parallelize the Viterbi algorithm.

There have been several approaches to exploit the intra-stage (inside a timestep) parallelism, also called wavefront parallelism, from the Viterbi recurrence using different algorithms and architectures. Recently, Maleki et al. [Maleki et al., 2014] gave the first technique to exploit the inter-stage (across different timesteps) parallelism from the Viterbi recurrence using rank convergence.

Preliminaries. Before describing the algorithm, we provide a few important definitions from Maleki et al. paper. In the original Viterbi algorithm, if we compute the logarithm of all probabilities called *log-probabilities* initially then we can use additions instead of multiplications. The Viterbi recurrence can be rewritten as

$$P[i,j] = \begin{cases} I[i] + B[i, y_1] & \text{if } j = 1, \\ \max_{k \in [1,n]}(P[k, j\text{-}1] + A[k,i] + B[i, y_j]) & \text{if } j > 1. \end{cases}$$

The above recurrence can be modified and written as

$$s[t-1] = s[0] \odot A_1 \odot A_2 \odot \cdots \odot A_{t-1}$$

where $s[j]$ is the jth solution vector (or column vector $P[..,j]$) of matrix P, the $n \times n$ matrix A_i is a suitable combination of A and B, and $\odot$ is a matrix product operation defined between two matrices $R_{n \times n}$ and $S_{n \times n}$ as

$$(R \odot S)[i,j] = \max_{k \in [1,n]}(R[i,k] + S[k,j])$$

The *rank* of a matrix $A_{m \times n}$ is r if r is the smallest number such that A can be written as a product of two matrices $C_{m \times r}$ and $R_{r \times n}$, i.e., $A_{m \times n} = C_{m \times r} \odot R_{r \times n}$. Two vectors v_1 are v_2 are *parallel* if v_1 and v_2 differ by a constant offset.

4.4.1 Original algorithm

The algorithm (see Figure 4.7) consists of two phases: (i) parallel forward phase, and (ii) fix up phase. In the forward phase, the t stages are divided into p segments, where p is the number of processors, each segment having $\lceil n/p \rceil$ stages (except possibly the last stage). The stages in the ith segment are from l_i to r_i. The initial solution vector of the entire problem is the initial vector of the first segment and it is known. The initial solution vectors of each of the other segments are initialized to non-zero random values. A sequential Viterbi algorithm is run in all the segments in parallel. A stage i is said to converge if the computed solution vector $s[i]$ is parallel to the actual solution vector s_i. A segment i is said to converge if $\text{rank}(A_{l_i} \odot A_{l_i+1} \odot \cdots \odot A_j)$ is 1 for $j \in [l_i, r_i - 1]$.

In the fix up phase, as in the forward phase a sequential Viterbi algorithm is executed for all segments simultaneously. The solution vectors computed in different segments (except the first) might be wrong. But, eventually they will become parallel to the actual solution vectors if *rank convergence* occurs. If rank convergence occurs at every segment then the solution vectors at every stage will be parallel to the actual solution vectors. On
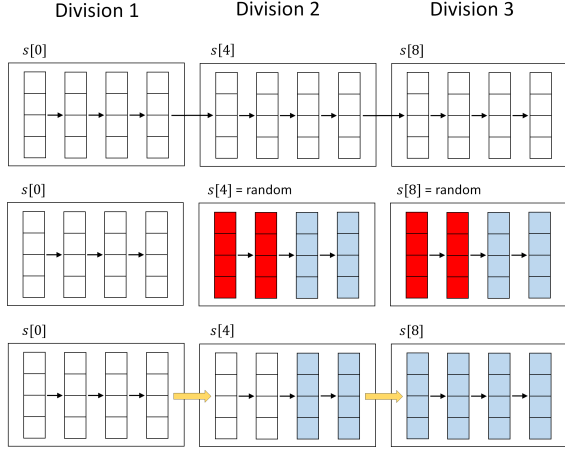
Figure 4.7: The pictorial representation of the rank convergence appproach. Color coding: white: correct solution, dark red: incorrect solution, light blue: parallel to correct.

Figure 4.8: Processor-aware parallel Viterbi algorithm using rank convergence as given in Maleki et al. paper [Maleki et al., 2014]. The algorithm is not cache-efficient.

the other hand, if the rank convergence does not happen at every segment, the fix up phase is run again and again until rank convergence occurs at some point. In the worst case, which rarely happens in practice, if rank convergence does not occur, then the fix up phase will be executed a total of $p-1$ times because in each run of a fix up phase, all the stages in exactly one segment is fixed up.

Please refer to Maleki et al.'s paper [Maleki et al., 2014] for a proof of why the method works.

4.4.2 Improved algorithm

The algorithm described above is processor-aware and can be made processor-oblivious by setting p to some constant, 10 or 100. The problem with setting a particular value to p is as follows. If p is set to a small value, the segment size will be large and if the rank of the matrix $A_1 \odot A_2 \odot \cdots \odot A_t$ is small, the algorithm does not exploit the full parallelism inherent in the problem. On the other hand, if p is large, there will be many segments of small size and if rank convergence does not happen the total work of the algorithm increases to at least pn^2t, where p is large.

We can improve the algorithm by making it processor-oblivious and also solving the problem mentioned above, using the following technique as shown in Figure 4.9. Instead of dividing the t stages into p segments, we divide the stages into $t/2^i$ segments each of size 2^i where $i \in [\log c, \log t]$ and c is a small fixed value, say 256. In the forward phase, all solution vectors at positions multiples of c, except the 0th solution vector are initialized to non-zero random values and serial Viterbi algorithm is run in each segment of size $c+1$. In the fix up phase, in the first iteration, from the first solution vector of every segment of size $2c$ the first solution vector of its next segment is computed through the serial Viterbi algorithm running for $2c+1$ timesteps. Generalizing, when the segment size is 2^i, from the first
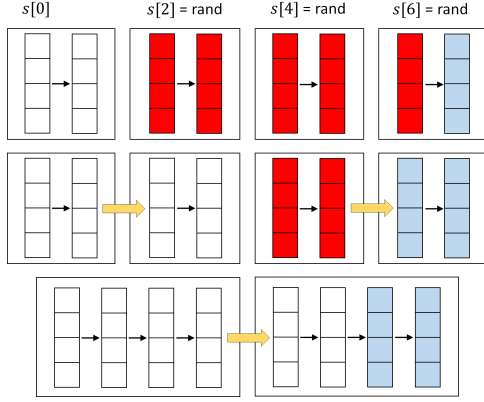
s[0] s[2] = rand s[4] = rand s[6] = rand

Figure 4.9: The pictorial representation of the improved rank convergence algorithm. Color coding: white: correct solution, dark red: incorrect solution, light blue: parallel to correct.

$\textsc{Viterbi-Rank-Improved}(s[0..t-1], A, B)$

1. $n \leftarrow 2^k; t \leftarrow 2^{k+k'}; c \leftarrow 2^8$
2. $\langle$ *Forward phase* $\rangle$ _____
3. $size \leftarrow c; q \leftarrow t/size$
4. **parallel for** $i \leftarrow 0$ **to** $q-1$ **do**
5. $l_i \leftarrow i \times size, r_i \leftarrow l_i + size - 1$
6.   **if** $i > 0$ **then** $s[l_i] \leftarrow$ random vector
7.   **for** $j \leftarrow l_i$ **to** r_i **do**
8. $s[j+1] \leftarrow \textsc{Viterbi}(s[j], A, B[.., y_{j+1}])$
9. $\langle$ *Fixup phase* $\rangle$ _____
10. $u[0..t-1] \leftarrow s[0..t-1]; converged \leftarrow$ false
11. **for** $(j \leftarrow \log c$ **to** $(\log t) - 1)$ and $!converged$ **do**
12. $size \leftarrow 2^j; q \leftarrow t/(2 \times size)$
13.   **parallel for** $i \leftarrow 0$ **to** $q-1$ **do**
14. $l_i \leftarrow (2i+1) \times size$
15. $r_i \leftarrow l_i + size - 1; conv_i \leftarrow$ false
16.   **for** $j \leftarrow l_i$ **to** r_i **do**
17. $u[j+1] \leftarrow \textsc{Viterbi}(u[j], A, B[.., y_{j+1}])$
18.   **if** $u[j+1]$ is parallel to $s[j+1]$ **then**
19. $conv_i \leftarrow$ true; break
20. $s[j+1] \leftarrow u[j+1]$
21.   **for** $i \leftarrow 0$ **to** $q-1$ **do**
22. $converged \leftarrow converged \wedge conv_i$
23.   **if** $converged =$ true **then** break

Figure 4.10: Processor-oblivious parallel Viterbi algorithm using rank convergence. The algorithm is not cache-efficient.

solution vector of every segment the first solution vector of its next segment is computed through the standard serial Viterbi algorithm running for a total of $2^i + 1$ timesteps.

The algorithm can be optimized to cut the total computations by a factor of two through a technique as follows. Assume that the numbering of the different segments starts from 0. We say that a set of stages $s[i \ldots j]$ is fixed when a serial Viterbi algorithm is run with $s[i]$ as the initial vector and the results have been propagated till $s[j]$. In the fix up phase, to fix up an ith segment of size 2^{j+1}, it is enough to fix the $(2i+1)$th segment of size 2^j because the $(2i)$th segment has already been fixed. The improved parallel Viterbi algorithm along with the optimization described above is given in Figure 4.10.

Complexity analysis. For $F \in \{O, I\}$, let $T_1^F(n,t), Q_1^F(n,t), T_\infty^F(n,t)$, and $S^F(t)$ denote the work, serial cache complexity, span, and the steps for convergence, respectively, of the F algorithm. The symbol O represents the original rank convergence algorithm and I denotes the modified algorithm. In the O algorithm, let there are $f(t)$ segments. Let the number of times the fix up phase is executed in O and I be λ_O and λ_I, respectively. Then $\lambda^O \in [1, f(t)]$ and $\lambda^I \in [1, \log(t/c)]$.

Work. $T_1^O(n,t) = \Theta(n^2 t \cdot \lambda_O)$. In the worst case, $T_1^O(n,t)$ is $\Theta(n^2 t \cdot f(t))$. On the other hand, $T_1^I(n,t) = \Theta(n^2 t \cdot \lambda_I)$. In the worst case, $T_\infty^I(n,t)$ is $\Theta(n^2 t \cdot \log t)$.

Serial cache complexity. It is easy to see that as there is no temporal locality, $Q_1^O(n,t) = \mathcal{O}\left(T_1^O(n,t)/B\right)$ and $Q_1^I(n,t) = \mathcal{O}\left(T_1^I(n,t)/B\right)$, when n^2 does not fit in cache. The analysis is similar to that of Section 4.2.

Span. $T_\infty^O(n,t) = \Theta(n(t/f(t)) \cdot \lambda_O)$, as the span of executing all the stages is $\Theta(t/f(t))$ and the span of executing each stage is $\Theta(n)$. In the worst case, $T_\infty^O(n,t)$ is $\Theta(nt)$. $T_\infty^I(n,t)$ is computed as follows. When the fix up phase is executed for the ith iteration, the number of

stages in each segment is 2^i. This implies the span of executing all stages for λ_I iterations in the fix up phase is $\Theta\left(\sum_{i=\log c}^{(\log c)+\lambda_I} 2^i\right) = \Theta\left(2^{\lambda_I}\right)$. Then, $T_\infty^I(n,t) = \Theta\left(n2^{\lambda_I}\right)$. In the worst case, $T_\infty^I(n,t)$ is $\Theta(nt)$.

Steps for convergence. Let the rank of the matrix $A_1 \odot A_2 \odot \cdots \odot A_t$ be k. For the original algorithm, $(S^O(t) - 1) \times (t/f(t)) < k \le S^O(t) \times (t/f(t))$, which implies $S^O(t) = \lceil kf(t)/t \rceil$. In the worst case, when $f(t) = t$ and $k \ge t$, $S^O(t) = t$. Similarly, for the improved algorithm, $2^{S^I(t)-1+\log c} < k \le 2^{S^I(t)+\log c}$, which implies $S^I(t) = \lceil k/c \rceil$. In the worst case, $S^I(t) = \lceil \log(t/c) \rceil$.

Conditions for no rank convergence. If we do not want rank convergence to happen for a particular problem instance, then we want rank $(A_1 \odot A_2 \odot \cdots \odot A_t) > 1$. For simplicity, assume $A_1 = A_2 = \cdots = A_t$. This means, we want rank$((A_1)^t) > 1$. If the two properties of rank$(A_1) > 1$ and rank$((A_1)^2) = $ rank(A_1) are satisfied, then we can recursively apply these properties to show that rank$((A_1)^t) > 1$. It is easy to see that one solution to rank$((A_1)^2) = $ rank(A_1) is when $(A_1)^2 = A_1$, which implies $A_1 = I_n$, where I_n is the identity matrix of size $n \times n$. When we say A_1, it really means $A_1 = A \odot B[k]$, where A is the transition probability matrix and $B[k]$ is a vector in the emission matrix B for some $k \in [1, m]$. If we want $A \odot B[k] = I_n$, we can have $A = I_n$ and $B = J_{n,m}$, where $J_{n,m}$ is the unit matrix of size $n \times m$. Also, rank$(I_n) = n > 1$ for $n > 1$. Therefore, when $A = I_n$ and $B = J_{n,m}$, the rank convergence does not happen.

4.5   Cache-efficient Viterbi algorithm

In this section, we present a cache-efficient cache-oblivious processor-oblivious recursive divide-and-conquer based parallel Viterbi algorithm derived by combining ideas from the cache-efficient multi-instance Viterbi algorithm (see Section 4.3) and the improved parallel Viterbi algorithm (see Section 4.4) based on rank convergence. The beautiful idea that the different segments of a single-instance Viterbi problem can be considered as the multiple instances of the same problem was given by Vivek Pradhan.

The multi-instance Viterbi algorithm works on the ith solution vectors, $s[i]$, of different instances of the problem and generates the $(i+1)$th solution vectors, $s[i+1]$, of the instances cache-efficiently. The improved parallel Viterbi algorithm (see Figure 4.9) that uses rank convergence divides the stages into $t/2^i$ segments each of size 2^i in the ith iteration. Let the base case segment consist of c stages. As each base case segment is run independently, we can assume that these segments are different instances of the same problem. Divide the t stages into t/c base case segments each of size c. The first solution vectors of all except the first segment are initialized to non-zero random values.

In the forward phase, a multi-instance Viterbi algorithm is run on the first solution vectors of t/c segments to generate the remaining solution vectors. In the fix up phase, in the $\log c$ iteration, the first solution vectors of each of the $t/c - 1$ segments (excluding the first segment) is found by applying the multi-instance Viterbi algorithm on the last solution vectors of the $t/c - 1$ segments (excluding the last segment). In general, in the ith iteration, where $i \in [\log c, \log t]$, each segment will be of size 2^i and there will be $t/2^i$ segments. In every iteration, a multi-instance Viterbi algorithm is used to find the initial solution vectors of all segments (except the first) from the final solution vectors of their previous segments. Then, a multi-instance Viterbi algorithm is run on the initial solution

97

```
VITERBI-CACHE-EFFICIENT(s[0..t − 1], A, B)

 1. n ← 2^k; t ← 2^{k+k'}; c ← 2^8
 2. ⟨ Forward phase ⟩ _____
 3. size ← c; q ← t/size
 4. parallel for i ← 0 to q − 1 do
 5.    l_i ← i × size, r_i ← l_i + size
 6.    if i > 0 then s[l_i] ← random vector
 7.    VITERBI-MULTI-INSTANCE-D&C(s[l_0..r_0], s[l_1..r_1], .., s[l_{q-1}..r_{q-1}], A, B, c + 1)

 8. ⟨ Fixup phase ⟩ _____
 9. u[0..t − 1] ← s[0..t − 1]; converged ← false
10. for (j ← log c to (log t) − 1) and !converged do
11.    size ← 2^j, q ← t/(2 × size)
12.    parallel for i ← 0 to q − 1 do
13.       l_i ← (2i + 1) × size; r_i ← l_i + size; conv_i ← false
14.    VITERBI-MULTI-INSTANCE-D&C(u[l_0..r_0], u[l_1..r_1], .., u[l_{q-1}..r_{q-1}], A, B, size + 1)
15.    parallel for i ← 0 to q − 1 do
16.       r_i ← 2(i + 1) × size − 1
17.       if u[r_i] is parallel to s[r_i] then conv_i ← true
18.       else s[r_i] ← u[r_i]
19.    for i ← 0 to q − 1 do
20.       converged ← converged ∧ conv_i
```

Figure 4.11: An efficient cache- and processor-oblivious parallel Viterbi algorithm using rank convergence. VITERBI-MULTI-INSTANCE-D&C is the algorithm presented in Section 4.3.

vectors of the $t/2^i$ segments to generate the last solution vectors of those segments. The fix up phase is similar to that of Section 4.4. The fix up phase is executed for λ ($\in [1, \log(t/c)]$) number of iterations depending on whether rank convergence happens or not.

Complexity analysis. Let $T_1(n, t), Q_1(n, t)$, and $T_\infty(n, t)$ be the work, serial cache complexity, and span of the cache-efficient Viterbi algorithm, respectively. Let $\lambda \in [1, \log(t/c)]$ be the number of iterations the fix up phase is executed.

$T_1(n, t) = \Theta(n^2 t \cdot \lambda)$. In the worst case, $T_1(n, t) = \Theta(n^2 t \cdot \log t)$. $T_\infty(n, t) = \Theta(n 2^\lambda)$. The analysis for computing the span is similar as in Section 4.4. The serial cache complexity $Q_1(n, t)$ is computed as follows.

$$Q_1(n, t) = \mathcal{O}\left(\sum_{i=\log c}^{(\log c)+\lambda} \left(Q_A\left(n, \frac{t}{2^i}\right) \cdot 2^i \right) \right)$$

$$= \mathcal{O}\left(\sum_{i=\log c}^{(\log c)+\lambda} \left(\frac{n^2 t}{B\sqrt{M}} + \frac{n^2 t}{M} + \frac{n(n2^i + t)}{B} + 2^i \right) \right)$$

$$= \mathcal{O}\left(\frac{n^2 t \lambda}{B\sqrt{M}} + \frac{n^2 t \lambda}{M} + \frac{n(n2^\lambda + t\lambda)}{B} + 2^\lambda \right)$$

If $n^2, t = \Omega\left(\sqrt{M}\right)$ and convergence happens after $\lambda = \mathcal{O}(1)$ iterations of the fix up phase, $Q_1(n, t)$ reduces to $\mathcal{O}\left(\frac{n^2 t \lambda}{B\sqrt{M}} + \frac{n^2 t \lambda}{M}\right)$ which further reduces to $\mathcal{O}\left(\frac{n^2 t \lambda}{B\sqrt{M}}\right)$ when the cache is tall (i.e., $M = \Omega(B^2)$).

On the other hand, the worst case serial cache complexity of the iterative Viterbi algorithm is $\mathcal{O}(n^2 t/B)$.

> **Theorem 7** (*Complexity analysis of the Viterbi algorithm*)**.** *The cache-efficient single-instance parallel Viterbi algorithm that uses rank convergence has the work of $T_1(n, t) = \Theta\left(n^2 t \cdot \lambda\right)$, span of $T_\infty(n, t) = \Theta\left(n 2^\lambda\right)$ and a serial cache complexity of*
>
> $$Q_1(n, t, B, M) = \mathcal{O}\left(\frac{n^2 t \lambda}{B\sqrt{M}} + \frac{n^2 t \lambda}{M} + \frac{n(n 2^\lambda + t\lambda)}{B} + 2^\lambda\right) \qquad (4.2)$$
>
> *where $\lambda \in [1, \log(t/c)]$ is the steps for rank convergence and c is a power of 2 and constant, such as, 256.*

New cache-efficient Viterbi algorithms

The intuition behind the cache-efficient algorithm described above leads to an important observation. Any variant of the Viterbi algorithm that exploits parallelism across stages by dividing the t stages into p segments and running them in parallel can be made cache-efficient by considering the segments as different instances of the same problem and executing the multi-instance Viterbi algorithm.

4.6   Lower bound

In this section, we prove the lower bound on the number of cache misses incurred by any algorithm to compute a general Viterbi recurrence that satisfies Property 4.

Property 4 (*One-way sweep*)**.** *An algorithm for a DP table C is said to satisfy the one-way sweep property if the following holds: $\forall$ cells $x, y \in C$, if x depends on y, then y is fully updated before x reads from y.*

An algorithm is cache-efficient when it has both spatial and temporal locality, among which temporal locality is harder to achieve. To exploit temporal locality, there must be asymptotic difference between work (serial running time) and space complexity of the algorithm. For the Viterbi algorithm, work is $\Theta\left(n^2 t\right)$ and space complexity is $\Omega\left(n^2\right)$ where the asymptotic difference comes from the time dimension. This means, if there is temporal locality, it should come from the time dimension only. In other words, when we load a few DP table cells to cache, we must be able to perform $\omega(1)$ number of updates to those cells before evicting them from cache, then we can get temporal locality.

Assume that all the cells at timestep (or stage) i are fully updated. Let all cells except one at timestep $i + 1$ be fully updated. Denote the only cell that is not updated at stage $i + 1$ by x. None of the cells at timestep $i + 2$ can be fully updated because of x, which implies none of the cells at timestep $i + j$ for $j \geq 3$ can be computed as per Property 4. As we cannot fully update a set of cells for non-constant number of timesteps without evicting the cells from the cache, we cannot make use of the time dimension to get temporal locality. Reading the entire state-transition matrix A incurs $\Omega\left(n^2/B\right)$ cache misses using any algorithm. For $\Theta(t)$ time steps, the cache complexity will be $\Omega\left(n^2 t/B\right)$.

> **Theorem 8** (*Lower bound for the Viterbi algorithm*)**.** *The number of cache misses incurred by any algorithm to solve the Viterbi recurrence that satisfies Property 4 i.e., the one-way sweep property is $\Omega\left(n^2 t/B\right)$.*

The cache-efficient algorithm described in Section 4.5 has a better cache complexity than the lower bound presented in this section. This is because the algorithm does not satisfy the one-way sweep property and hence the lower bound does not apply to the cache-efficient algorithm.

4.7   Experimental results

In this section, we briefly describe our implementation details and performance results. The multi-instance Viterbi algorithm was implemented by Yunpeng Xiao and Jesmin Jahan Tithi. The single-instance Viterbi algorithm was implemented by Jesmin Jahan Tithi.

We implemented all algorithms presented in the paper in C++ with Intel Cilk Plus [Int,] extension and compiled them using Intel C++ Compiler v13.0. We used PAPI 5.2 [PAP,] to count the cache misses and `likwid` [Treibig et al., 2010] to measure energy and power consumption of the program. We used a hybrid recursive divide-and-conquer algorithm where the recursive implementation switched to an iterative kernel when the problem size became smaller than a predefined base case size (e.g., 64×64) to amortize the overhead of recursion. All programs were compiled with `-O3 -parallel -AVX -ansi-alias -opt-subscript-in-range` optimization parameters and were auto vectorized by the compiler.
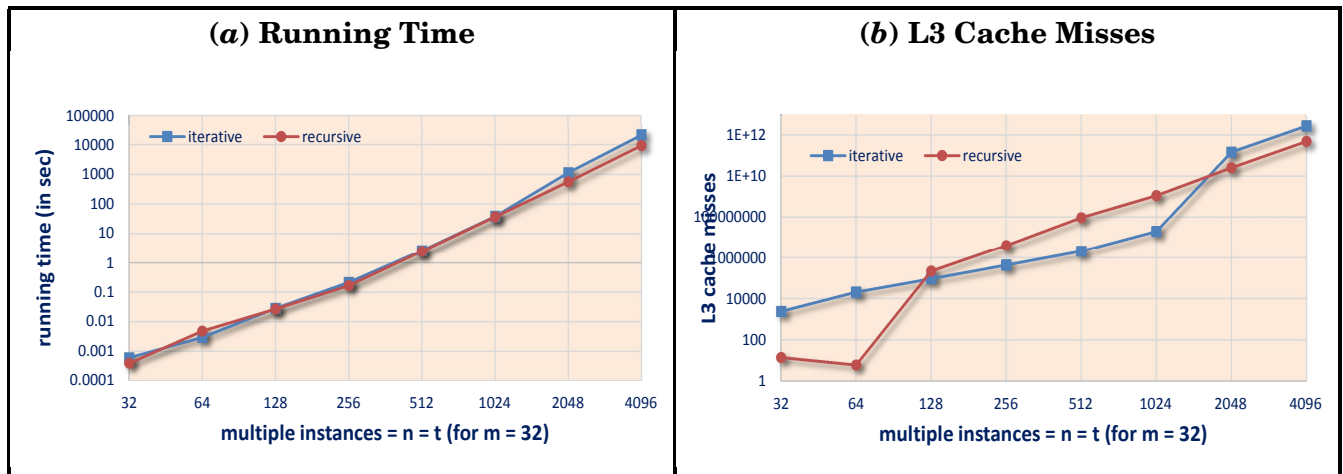
Figure 4.12: Running time and L3 miss of our cache-efficient multi-instance Viterbi algorithm along with the multi-instance iterative Viterbi algorithm.

We used a dual socket 16-core ($= 2 \times 8$-cores) 2 GHz Intel Sandy Bridge machine to run all experiments presented in the paper. Each core of this machine was connected to a 32 KB private L1 cache and a 256 KB private L2 cache. All the cores in a socket shared a 20 MB L3 cache, and the machine had 32 GB RAM shared by all cores. The matrices A, B, and I were initialized to random probabilities. We used log-probabilities in all implementations and hence used addition instead of multiplication in the Viterbi recurrence. All matrices were stored in column-major order. We performed two sets of experiments to compare our cache-efficient algorithms with the iterative and the fastest known Viterbi (Maleki et al.'s) algorithms. They are as follows.

4.7.1 Multi-instance Viterbi algorithm

We compared our cache-efficient multi-instance recursive Viterbi algorithm with the multi-instance iterative Viterbi algorithm. Both algorithms were optimized and parallelized. To construct matrix $W_{n \times q}$ (we chose q to be n in this case), instead of copying all the relevant columns of B, only the pointers to the respective columns were used. Wherever possible, pointer swapping was used to interchange previous solution vector (or matrix) and current solution vector (or matrix).

The running time and the L3 cache misses for the two algorithms are plotted in Figure 4.12. The number of stages n, which is also the number of instances was varied from 32 to 4096. Note that in the cache-efficient multi-instance Viterbi algorithm, the number of stages does not need to be the same as the number of instances. The variable m was fixed to 32 and the number of timesteps t was also kept the same as n (hence overall complexity is $O(n^4)$). The recursive algorithm ran slightly faster than the iterative algorithm in most cases when the number of instances increase. When n was 4096, our recursive algorithm ran around 2.26 times faster than the iterative algorithm.

For a big difference between the performance of the multi-instance recursive and multi-instance iterative algorithms, the algorithms should be run for large n and large t. But, if n and t are too large the time taken to run experiments also increases by several orders of magnitude as the total work of the multi-instance algorithm is $\Theta(n^3 t)$, when the number of instances is n. We believe that for applications that can use multiple instance Viterbi algorithm and need to compute on large data (e.g., multiple sequence alignment problems), using cache-efficient multi-instance Viterbi algorithm will be very beneficial.

4.7.2 Single-instance Viterbi algorithm

We compared our cache-efficient parallel Viterbi algorithm with Maleki et al.'s parallel Viterbi algorithm. Both implementations were optimized and parallelized and the reported statistics are average of 4 independent runs. In all our experiments, the number of processors p was set to 16. The plots of Figure 4.13 shows the graphs of the running time and L3 cache misses for the two algorithms for $n = 4096$.

When $n = 4096$, the number of timesteps t was varied from 2^{12} to 2^{18} and m was set to 32. Our algorithm ran faster than Maleki et al.'s original rank convergence algorithm throughout, and for $t = 2^{18}$ our algorithm ran approximately 33% faster. Our algorithm's L3 cache misses were also lower by a significant amount, and for $t = 2^{18}$, Maleki et al.'s algorithm incurred 6 times more cache misses than ours. The impact shows up in DRAM energy consumption and bandwidth utilization.

Energy consumption. We also ran experiments to analyze the energy consumption (taking average over three runs) of our cache-efficient recursive and Maleki et. al.'s algorithm. Our algorithm consumed relatively less DRAM energy compared to the other algorithm.

We used the `likwid-perfctr` tool to measure CPU, Power Plane 0 (PP0), DRAM energy, and DRAM power consumption during the execution of the programs. The energy measurements were end-to-end, i.e., included all costs during the entire program execution. Note that the DRAM energy consumption is somewhat related to the L3 cache miss of a program as each L3 cache miss results in a DRAM access. Similarly, since CPU energy gives the energy consumed by the entire package (all cores, on chip caches, registers and

Figure 4.13: Running time, L3 miss and energy/power consumption of our cache-efficient Viterbi algorithm along with the existing algorithms.

their interconnections), it is related to a program's running time. PP0 is basically a subset of CPU energy since it captures energy consumed by only the cores and their private caches.

For $n = 2048$, the timesteps was increased from 2048 to 16384 keeping $m = 32$. Figure 4.13 shows that the DRAM energy as well as power consumption of our algorithm was significantly less because of the reduced L3 cache misses. When $t = 16384$, Maleki et al.'s algorithm consumed 60% more DRAM energy and 30% more DRAM power than ours.

4.8   Conclusion and open problems

This chapter presented the first provably cache-efficient cache- and processor-oblivious parallel Viterbi algorithm solving the two decade old open problem. The algorithm combines the ideas of our cache-efficient multi-instance Viterbi algorithm with Maleki et al.'s parallel Viterbi algorithm. The significance of our algorithm lies mainly in its improved cache complexity (exploiting temporal locality), and cache- and processor-obliviousness.

Some open problems are:

✶ [*Cache-efficient cache-oblivious algorithms.*] Is it possible to design other cache-efficient cache-oblivious parallel Viterbi algorithms?

✯ [*Lower bound.*] What is the tight lower bound for serial cache complexity for the Viterbi algorithm?

Chapter 5

Semi-Automatic Discovery of Efficient Divide-&-Conquer Tiled DP Algorithms

Recursive divide-and-conquer is a very powerful design technique to develop efficient cache-oblivious algorithms. But what if the computer architectures (e.g.: GPUs) do not implement automatic page replacement scheme, or if the computer architectures do not support recursion, or if the programming languages do not support recursion? In such cases, cache-aware tiling might be the only way to write efficient programs. Hence, it is important to design a generic framework to design efficient tiled algorithms.

In this chapter, we present a generic framework for designing high-performing efficient cache-aware fractiled[1] algorithms for a class of fractal dynamic programming (DP) problems. These algorithms are based on recursive r-way divide and conquer, where r (≥ 2) varies based on the current depth of the recursion. In addition to providing strong theoretical guarantees for the parallel running time of these algorithms, we show that they perform asymptotically optimal number of data transfers between every two consecutive levels of the memory hierarchy. If the computer architectures support recursion and implements automatic page replacement scheme, then the algorithms can be oblivious of the sizes of the memories in those machines. On the other hand, if the computer architectures do not implement automatic page replacement scheme or if recursion is not supported, then the algorithms have to be aware of the memory sizes. It is possible that the algorithms is oblivious to a few levels of caches where automatic page replacement is implemented and aware of the memory sizes of those levels where automatic page replacement is not implemented.

We give the first general framework for designing provably efficient algorithms for solving a whole class of DP problems on GPUs. To the best of our knowledge, we present the first GPU algorithms for solving DP problems I/O-optimally when the DP table does not fit into the RAM. The major advantage of having a generic framework is that it can potentially lead to automation of algorithm design for an entire class of problems.

We have implemented GPU algorithms derived using our framework for four DP/DP-like problems: Floyd-Warshall's APSP, parenthesis problem, gap problem, and Gaussian elimination without pivoting. Our GPU implementations run significantly faster than all

---

[1]fractile – recursively TILEd algorithms for FRACtile-DP problems

internal-memory multicore CPU implementations as well as all existing iterative tiled GPU implementations. Also, our I/O-optimal external-memory implementations are compute-bound for state-of-the-art GPUs, and as a result can effectively harness the superior computing power of those GPUs to achieve significantly faster running times.

5.1   Introduction

Computer architecture is always changing. Designing algorithms for a specific architecture requires expertise in various domains such as algorithms, data structures, parallel programming, computer design, compiler design, and so on. While efficient algorithms for important problems are being designed and implemented, new machine architectures would have hit the market and programmers have to redesign and reimplement the key algorithms. Furthermore, new problems are being encountered everyday. This necessitates the development of generic frameworks to design fast algorithms for new machine architectures.

In the initial years of the computing era, CPUs were used as processing elements for general purpose programming tasks. Other processing elements such as graphics processing units (GPUs) were used as coprocessors for parallel graphics computations. But, over time these coprocessors are being used more for general purpose parallel computing tasks. Typically, CPUs are used for complicated control logic and GPUs are used for highly data-parallel and compute-intensive tasks.

The high performance computing (HPC) community is quickly moving towards heterogeneous computing, where each compute node consists of both multicore CPUs and many-core GPUs connected through Peripheral Component Interconnect (PCI) express bus. The hardware accelerators such as GPUs are being used [Cheng et al., 2014] in seismic processing, biochemistry simulations, weather and climate modeling, signal processing, computational finance, computer-aided engineering, computational fluid dynamics, and data analysis. To aid in writing applications to use this heterogeneous architecture (CPUs and GPUs) efficiently, simpler programming models and application programming interfaces (APIs) are developed. Currently, the effective use of such heterogeneous systems is limited by the increased complexity of efficient algorithm design.

Performance of a parallel algorithm can be measured using two major parameters: *I/O complexity* and *parallelism* as described in Chapter 1. I/O complexity, used to measure memory locality, represents the total number of I/O / memory / data transfers of an algorithm. I/O complexity can be measured between external memory and CPU RAM, or two adjacent levels of memories in a CPU, or CPU RAM and GPU global memory, or two adjacent memory levels in a GPU. Often, better I/O complexity implies lower execution time.

We use the ideal-cache model [Frigo et al., 1999] to compute serial I/O complexity between consecutive memory levels inside a CPU, denoted by $Q_1(n)$, where n is the problem parameter. We use Least Recently Used (LRU) page replacement scheme for data transfer between external memory and CPU RAM, which is 2-competitive to optimal page replacement scheme. Also, between CPU RAM and GPU global memory and between memory levels in GPU, there is no automatic page replacement scheme provided by the hardware or the runtime system and hence, we explicitly copy the submatrices of the input DP table.

To measure parallelism, we use dynamic multithreading model [Cormen et al., 2009].

Assuming a multithreaded program is modeled as a directed acyclic graph, the work $T_1(n)$ and span $T_\infty(n)$ of the parallel program can be defined as the runtime of the program on one and an unbounded number of processor(s), respectively. Then, the parallelism is theoretically computed as $T_1(n) \div T_\infty(n)$, which is defined as the maximum speedup achievable running the program an any parallel machine.

Algorithm design in the ever-changing world of computer architecture must also take *portability* into consideration. Portability is important because the underlying parallel machine architecture keeps changing continually. If the algorithms are not portable, the programmers have to keep redesigning and reimplementing their algorithms with the arrival of every new machine architecture. Even if the algorithms are not completely portable at least they must be easier to port to different architectures. The most common way to design portable algorithms is to design resource-oblivious, where resources include caches, processors, etc.

In this chapter, we present a generic framework to develop high-performing (having excellent I/O complexity and good parallelism) and easily portable algorithms to be run on GPU systems for a wide class of dynamic programming problems using r-way divide-and-conquer.

Tiled iterative GPU algorithms. GPUs are one of the major architectures that do not implement automatic page replace schemes. Hence, we concentrate most of our discussion towards designing tiled algorithms for GPU systems. However, our approach can be used for multicore systems and manycore architectures such as Intel MIC.

Most existing GPU implementations are based on tiling or blocking iterative algorithms for global and shared memories of a GPU to exploit memory locality. There are several limitations with this approach:

* [*Generic approach is not developed*.] No generic method exists for designing tiled-iterative algorithms for dynamic programming problems.
* [*Extremely complicated*.] Existing standard tiling techniques are typically used for tiling 1 or at most 2 levels of memories. When the number of memory levels to be tiled is greater than 2, also called multi-level or hierarchical tiling, the process of tiling becomes extremely complicated for algorithm design, complexity analysis, and program implementation.
  Existing work on tiled iterative algorithms do not analyze theoretical performance guarantees for the I/O complexity or the parallelism involved.
* [*External-memory implementations not developed*.] Almost all existing implementations assume that the entire problem fits in the global memory of a GPU and do not work for large data which fits in RAM / external memory.

Tiled recursive GPU algorithms. GPU implementations of a few DP problems are based on tiling techniques derived from recursive divide-and-conquer algorithms that use only matrix-matrix multiplications on a semiring, also called MM-like kernels. For example, R-Kleene's algorithm [Sibeyn, 2004, D'Alberto and Nicolau, 2007, Buluc et al., 2010] to solve the Floyd-Warshall's APSP problem can be both memory efficient and highly parallel because of MM-like kernels. However, there are limitations of this recursive approach.

* The approach typically uses closure property and it is not clear how to generalize this approach of transforming most recursive functions to MM-like kernels to a wide variety of DP problems.

✶ The R-Kleene algorithm works for only closed semirings where addition operation is *idempotent* [Buluc et al., 2010, Tiskin, 2004] e.g.: tropical and Boolean semirings. Due to this constraint, the approach cannot be used if we have arbitrary function (see Figure 3 of [Chowdhury and Ramachandran, 2010]) in the innermost loop of the iterative algorithm as in the case of Gaussian elimination without pivoting (or LU decomposition) and other Floyd-Warshall-type problems.

The AUTOGEN-FRACTILE framework. In this chapter, we present the AUTOGEN-FRACTILE framework that uses the r-way recursive divide-and-conquer method to design efficient tiled algorithms. The r-way divide-and-conquer method, introduced in [Chowdhury and Ramachandran, 2008], where $r \in [2, n]$, is a generalization of the 2-way divide-and-conquer. In this method, a d-dimensional hypercubic DP table of size n^d for a given DP problem is divided into r^d hypercubic orthants each of size $(n/r)^d$. Different recursive functions are defined as per the read-write DP dependencies among the hypercubic orthants. The functions are then invoked recursively to compute the entire DP table.
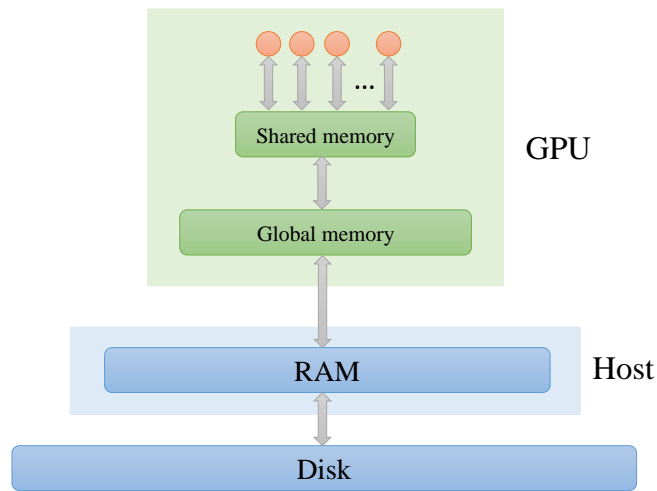
Figure 5.1: A simple representation of the CPU-GPU memory hierarchy.

The r-way divide-and-conquer method can overcome many limitations of the tiled iterative method.

✶ [*Generic framework*.] We develop a generic framework to design tiled algorithms for a wide class of dynamic programs based on recursive divide-and-conquer.

✶ [*Strong theoretical bounds*.] It is possible to give strong theoretical performance bounds for both serial I/O complexity and parallelism for r-way divide-and-conquer algorithms (see Section 5.4). The r-way algorithms, where r is set appropriately at each recursion level based on the memory sizes, often have optimal serial I/O complexity.

✶ [*Easily extensible to any number of memory levels*.] The r-way divide-and-conquer algorithms are so generic that they work for any number of levels of memories. For example, Figure 5.1 shows a simple version of the memory hierarchy of a GPU system abstracting many intricate details. We can extend this system to include multiple GPUs, multiple shared memories per GPU global memory, or multiple levels of memories inside both CPU and GPU. In all these cases, the r-way algorithms can be easily ported to the GPU systems.

The r-way divide-and-conquer method can also be used to overcome the limitations of the recursive approach that uses the closure property and MM-like kernels [Tithi et al., 2015].

  ✯ Our method can be used for a wide variety of DP problems.
  ✯ For Floyd-Warshall-type problems, using $(n^2 + n)$ extra space the r-way algorithms can be made to work for arbitrary functions instead of only closed semirings where the addition operator is idempotent [Chowdhury and Ramachandran, 2010].

Designing r-way divide-and-conquer algorithms for different dynamic programs is complicated. However, this method is a generic and powerful approach to design efficient and easily portable algorithms for heterogeneous systems with deep memory hierarchy.

Our contributions. The major contributions of this chapter are:

  1. [*Algorithmic*.] We present a generic framework called AUTOGEN-FRACTILE to develop tiled algorithms using r-way recursive divide-and-conquer approach for a wide class of DP problems. We present new r-way divide-and-conquer algorithms for Gaussian elimination without pivoting and the gap problems.
  2. [*Experimental*.] We implement GPU algorithms (derived from AUTOGEN-FRACTILE framework) for both internal and external memory and achieve an order of magnitude speedup compared with the existing CPU and GPU tiled implementations.

Related work. Recursive cache-oblivious algorithms for DP and other matrix problems exist: Floyd-Warshall's-APSP-type problems [Sibeyn, 2004, D'Alberto and Nicolau, 2007, Ullman and Yannakakis, 1990, Ullman and Yannakakis, 1991, Park et al., 2004], Gaussian elimination without pivoting [Blumofe et al., 1996a], Cholesky, LDLT, LU with pivoting, and QR [Gustavson, 1997, Elmroth and Gustavson, 2000, Elmroth et al., 2004], parenthesis problem [Cherng and Ladner, 2005, Chowdhury and Ramachandran, 2008], longest common subsequence [Chowdhury and Ramachandran, 2006], sequence alignment with gap penalty [Chowdhury, 2007], and protein folding [Tithi et al., 2015]. An algorithm for a paradigm of Floyd-Warshall-type algorithms (including Floyd-Warshall's APSP, transitive closure, LU decomposition, Gaussian elimination without pivoting, parenthesis problem, etc) is presented in [Chowdhury and Ramachandran, 2007, Chowdhury and Ramachandran, 2010].

Several GPU implementations have been developed to solve data-intensive matrix problems: Floyd-Warshall's APSP [Diament and Ferencz, 1999, Venkataraman et al., 2003, Harish and Narayanan, 2007, Katz and Kider Jr, 2008, Matsumoto et al., 2011, Buluc et al., 2010, Lund and Smith, 2010, Solomonik et al., 2013, Djidjev et al., 2014], parenthesis problem family (includes chain matrix multiplication, CYK algorithm, optimal polygon triangulation, RNA folding, etc) [Steffen et al., 2009, Rizk and Lavenier, 2009, Solomon and Thulasiraman, 2010, Nishida et al., 2011, Wu et al., 2012, Nishida et al., 2012], sequence alignment [Liu et al., 2006, Liu et al., 2007, Manavski and Valle, 2008, Striemer and Akoglu, 2009, Xiao et al., 2009].

It is important to note that the existing GPU implementations are based on tiled-iterative or closure property and therefore suffer from several limitations as discussed before. Our r-way divide-and-conquer method overcomes these limitations.

Organization of the chapter. In Section 5.2, we give the importance, types, and meaning of r-way recursive divide-and-conquer algorithms. In Section 5.3 we present several approaches for the AUTOGEN-FRACTILE framework to design tiled algorithms for DP prob-

lems using r-way divide-and-conquer. The complexities of the r-way divide-and-conquer algorithms are presented in Section 5.4. In Section 5.5, we give empirical results comparing our implementations with the existing algorithms.

5.2 r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms

In this section, we define a few important terms, give the importance, types, and meaning of r-way divide-and-conquer algorithms.

We define a few terms such as $\mathcal{I}$-$\mathcal{DP}$, 2-way $\mathcal{R}$-$\mathcal{DP}$, r-way $\mathcal{R}$-$\mathcal{DP}$, and $\mathcal{TR}$-$\mathcal{DP}$ to represent different algorithms. We will use these terms throughout the chapter.

Definition 24 (*$\mathcal{I}$-$\mathcal{DP}$ / 2-way $\mathcal{R}$-$\mathcal{DP}$ / r-way $\mathcal{R}$-$\mathcal{DP}$ / $\mathcal{TR}$-$\mathcal{DP}$*)**.** *Let $\mathcal{P}$ be a given DP problem specified through a DP recurrence. An iterative (or loop-based) algorithm for $\mathcal{P}$ is called $\mathcal{I}$-$\mathcal{DP}$. A standard recursive divide-and-conquer algorithm for $\mathcal{P}$ that divides a d-dimensional DP of size $n \times \cdots \times n$ into $\frac{n}{2} \times \cdots \times \frac{n}{2}$ orthants and then processes these orthants recursively is called a 2-way $\mathcal{R}$-$\mathcal{DP}$. A recursive divide-and-conquer algorithm for $\mathcal{P}$ that divides a d-dimensional DP of size $n \times \cdots \times n$ into r^d submatrices each of size $\frac{n}{r} \times \cdots \times \frac{n}{r}$ and then processes these submatrices recursively is called an r-way $\mathcal{R}$-$\mathcal{DP}$. A cache-aware parallel tiled algorithm (if it exists) for $\mathcal{P}$, derived from r-way $\mathcal{R}$-$\mathcal{DP}$ is called a $\mathcal{TR}$-$\mathcal{DP}$.*

In this chapter, we present a framework called AUTOGEN-FRACTILE that can be used to derive $\mathcal{TR}$-$\mathcal{DP}$ algorithms.

Definition 25 (*AUTOGEN-FRACTILE*)**.** *Let $\mathcal{P}$ be a given DP problem specified through a DP recurrence. The framework that can be used to semi-automatically discover a $\mathcal{TR}$-$\mathcal{DP}$ given any implementation for $\mathcal{P}$ is called AUTOGEN-FRACTILE.*

The input to AUTOGEN-FRACTILE is a DP recurrence or an $\mathcal{I}$-$\mathcal{DP}$. As the simplest implementation of a DP recurrence is an iterative algorithm or $\mathcal{I}$-$\mathcal{DP}$, in this chapter we assume $\mathcal{I}$-$\mathcal{DP}$s as inputs to AUTOGEN-FRACTILE.

5.2.1 Importance of r-way $\mathcal{R}$-$\mathcal{DP}$s

Before delving into the process of derivation of r-way $\mathcal{R}$-$\mathcal{DP}$s, let's understand the meaning and importance of r-way $\mathcal{R}$-$\mathcal{DP}$s. An r-way $\mathcal{R}$-$\mathcal{DP}$ is a generalization of a standard 2-way $\mathcal{R}$-$\mathcal{DP}$. Though designing a 2-way $\mathcal{R}$-$\mathcal{DP}$ is now in the realm of science for a large class of dynamic programs, designing an r-way $\mathcal{R}$-$\mathcal{DP}$ is still an art. Both 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s have exactly the same recursive functions. In the case of an r-way $\mathcal{R}$-$\mathcal{DP}$, we divide a d-dimensional n^d-sized hypercubic DP table into r^d hypercubic submatrices each of size $(n/r)^d$, analyze the dependencies between the submatrices, and recursively define functions based on such dependencies.

The major reasons to use r-way $\mathcal{R}$-$\mathcal{DP}$s over 2-way $\mathcal{R}$-$\mathcal{DP}$s are:

⋆ [*More parallelism.*] r-way $\mathcal{R}$-$\mathcal{DP}$s typically have more parallelism than their 2-way counterparts (see Figure 3 in [Tang et al., 2014] for an example).

✯ [*Computer architectures that do not implement automatic page replacement or do not support recursion.*] r-way $\mathcal{R}$-$\mathcal{DP}$s can be used to design highly efficient cache-aware algorithms for computer architectures with deep memory hierarchy that do not implement automatic page replacement (e.g.: GPUs) or do not support recursion.

5.2.2 Types of r-way $\mathcal{R}$-$\mathcal{DP}$s

In the r-way $\mathcal{R}$-$\mathcal{DP}$s, the term r can be set to any of the three values as given below:
  ✯ r can be a constant.
  ✯ r can be a function of n.
  ✯ r can be the largest value $tilesize[d]$ at every recursion level d based on a particular cache or memory size such that the tile exactly fits in the memory.

Typically, r is set to $tilesize[d]$. If there are h levels of memories or caches, then there are h levels of recursion and hence there are h tile sizes $tilesize[1\ldots h]$. When a subproblem fits into a memory of the smallest size, we execute an iterative kernel.

If r is a constant or $tilesize[d]$, then typically an r-way $\mathcal{R}$-$\mathcal{DP}$ will have optimal serial I/O complexity (see Theorem 9). On the other hand, if r is a function of n and r is super-constant, then the r-way $\mathcal{R}$-$\mathcal{DP}$ will have non-optimal serial I/O complexity. If r is a constant or a function of n, then the $\mathcal{R}$-$\mathcal{DP}$ is cache-oblivious. If r is set to $tilesize[d]$, then the $\mathcal{R}$-$\mathcal{DP}$ is cache-aware. See Table 5.1 for a summary. *In this paper, unless explicitly mentioned r-way $\mathcal{R}$-$\mathcal{DP}$ means $\mathcal{R}$-$\mathcal{DP}$ where r is set to $tilesize[d]$.*

Feature	$r = \Theta\left(1\right)$	$r = f(n)$ & $r = \omega\left(1\right)$	$r = tilesize[d]$
Cache-obliviousness	Yes	Yes	No
Serial I/O-optimality	Yes (often)	No	Yes (often)

Table 5.1: Comparison of features of r-way $\mathcal{R}$-$\mathcal{DP}$s for different values of r.

5.2.3 GPU computing model

The r-way $\mathcal{R}$-$\mathcal{DP}$s are ideally suited for computer architectures that do not implement automatic page replacement schemes. The best example of such an architecture are the GPUs. Here, we give a brief overview of the GPU architecture, its programming model, and the challenges encountered when writing high-performing programs for the current generation GPUs.

General purpose computing on GPUs. GPUs attached to CPU using PCI bus are used as hardware accelerators. They have a many-core architecture having cores in the range of tens to hundreds. They are designed to have thousands of light-weight threads compared with those of the CPUs, are optimized for highly data-parallel and compute-intensive tasks, and are designed to maximize the throughput of the parallel programs. GPUs typically have multiple types of parallelism: multithreading, single-instruction multiple-data (SIMD), and instruction-level parallelism. The GPU architecture is naturally suitable for data-parallel scientific problems involving matrices of dimensions 1, 2, 3, or 4.

Figure 1 gives a simple representation of the memory hierarchy of the current generation GPUs. The CPU is connected to a disk (or external-memory) of a very large size and a RAM (or internal-memory) of a decent size. The GPU consists of a global memory and

several shared memories. Each shared memory is connected to a constant number of cores that have their own registers. A GPU is connected to the CPU through a PCI bus. Multiple GPUs can be connected to a single CPU.

Several application programming interfaces (APIs) have been developed so that programmers can use GPUs for general purpose computing without being experts in computer graphics. The APIs are used to accelerate specific set of parallel computations on GPUs. Some of the most commonly used APIs are Open Computing Language (OpenCL), NVIDIA's Compute Unified Device Architecture (CUDA), Microsoft's DirectCompute, AMD's Accelerated Parallel Processing (APP) SDK, and Open Accelerators (OpenACC). To exploit the GPU architecture fully, the compute kernels must make use of the memory efficiently and the organization of the threads must be good.

GPU programming challenges. Writing good GPU programs is hard. The difficulty in GPU programming is due to several factors.

- ✫ [*Automatic page replacement is not implemented*.] Automatic page replacement is not implemented in GPUs. Recursive divide-and-conquer is a powerful tool to design efficient (I/O-efficient, energy-efficient, and highly parallel), portable (cache- and processor-oblivious) and robust (cache- and processor-adaptive) algorithms. However these design techniques involve complicated control logic and hence they are either unsupported / unsuitable for GPUs, which forces programmers to search for other approaches.
- ✫ [*GPU optimization is hard*.] Optimization on GPUs is harder than that on CPUs. The key factors that have big influence on the performance on GPUs are: thread organization (threads can be organized in blocks of different dimensions with different dimension lengths), warp size (the granularity at which the streaming multiprocessors can execute computations), memory coalescing (consecutive numbered threads access consecutive memory locations), and streams and events (overlapping compute kernel execution and data transfers). Depending on the mixture of optimizations used the performance can be drastically different.

5.3   The Autogen-Fractile framework

In this section, we present the Autogen-Fractile framework to design efficient tiled algorithms for a wide class of dynamic programs. From hereon, we call a recursive divide-and-conquer algorithm as an $\mathcal{R}\text{-}\mathcal{DP}$ algorithm. The 2-way and r-way algorithms are called 2-way and r-way $\mathcal{R}\text{-}\mathcal{DP}$s, respectively.

There are three major approaches in which the Autogen-Fractile framework could be defined.

Example. For simplicity of exposition we explain the working of Autogen-Fractile by designing a $\mathcal{TR}\text{-}\mathcal{DP}$ algorithm for Floyd-Warshall's all-pairs shortest path (APSP) from its recurrence. The APSP is solved using the DP recurrence as described in Equation A.4 in Appendix A.3.

5.3.1   Approach 1: Generalization of 2-way $\mathcal{R}\text{-}\mathcal{DP}$

This is the first approach that uses the generalization of 2-way $\mathcal{R}\text{-}\mathcal{DP}$s to get r-way $\mathcal{R}\text{-}\mathcal{DP}$s.

The four main steps of AUTOGEN-FRACTILE are:

1. [*2-way $\mathcal{R}$-$\mathcal{DP}$ derivation.*] A 2-way $\mathcal{R}$-$\mathcal{DP}$ is constructed from a given $\mathcal{I}$-$\mathcal{DP}$.
2. [*3-way, 4-way, ... $\mathcal{R}$-$\mathcal{DP}$ derivation and visualizaton.*] We derive r-way $\mathcal{R}$-$\mathcal{DP}$, where r is a constant, and then visualize their working.
3. [*r-way $\mathcal{R}$-$\mathcal{DP}$ derivation.*] An r-way $\mathcal{R}$-$\mathcal{DP}$ algorithm is derived with the aid of visualization.
4. [*$\mathcal{TR}$-$\mathcal{DP}$ derivation.*] The $\mathcal{TR}$-$\mathcal{DP}$ is derived for two consecutive memory levels from the r-way $\mathcal{R}$-$\mathcal{DP}$ by setting r based on the memory sizes.

2-way $\mathcal{R}$-$\mathcal{DP}$ derivation

In this step, a 2-way $\mathcal{R}$-$\mathcal{DP}$ algorithm is derived from a given $\mathcal{I}$-$\mathcal{DP}$. The major reason to derive 2-way $\mathcal{R}$-$\mathcal{DP}$ is to identify the recursive functions that will be present in the 2-way $\mathcal{R}$-$\mathcal{DP}$ as *the recursive functions in both 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s are exactly the same*.

The structure and design of 2-way $\mathcal{R}$-$\mathcal{DP}$s gives clues to the design of r-way $\mathcal{R}$-$\mathcal{DP}$s. The standard 2-way $\mathcal{R}$-$\mathcal{DP}$s have been presented in [Chowdhury and Ramachandran, 2006, Chowdhury and Ramachandran, 2008, Chowdhury and Ramachandran, 2010] and other work. Such algorithms split a d-dimensional n^d-sized hypercubic DP table into 2^d hypercubic orthants each of size $(n/2)^d$. They consist of a fixed number of distinct recursive functions and the functions are defined based on the dependencies between different orthants. Manually deriving 2-way $\mathcal{R}$-$\mathcal{DP}$s for several dynamic programs is difficult.

The AUTOGEN [Chowdhury et al., 2016b] algorithm can be used to automatically discover standard 2-way (also r-way for small constant r) $\mathcal{R}$-$\mathcal{DP}$s for a wide class of dynamic programs.[2] By using AUTOGEN, we can easily determine the number of recursive functions that are present in the 2-way $\mathcal{R}$-$\mathcal{DP}$ of a given dynamic program. Each recursive function is different from each other in the order in which it writes and reads from submatrices of the input DP table.

For example, we use AUTOGEN to derive a 2-way $\mathcal{R}$-$\mathcal{DP}$ for the Floyd-Warshall's APSP problem as described in Section 2.5.1. The 2-way $\mathcal{R}$-$\mathcal{DP}$ is also described with diagrams in Section A.3. We see that there are four recursive functions $\mathcal{A}, \mathcal{B}, \mathcal{C},$ and $\mathcal{D}$. Each of the recursive functions take three parameters (X, U, V). They three regions / submatrices are related to each other depending on which function calls them, as shown below.

⋆ [*Function $\mathcal{A}$.*] The relation is $X = U$ and $X = V$.
⋆ [*Function $\mathcal{B}$.*] The relation is $X \neq U$ and $X = V$.
⋆ [*Function $\mathcal{C}$.*] The relation is $X = U$ and $X \neq V$.
⋆ [*Function $\mathcal{D}$.*] The relation is $X \neq U$ and $X \neq V$ and $U \neq V$.

3-way, 4-way, ... $\mathcal{R}$-$\mathcal{DP}$ derivation and visualization

The AUTOGEN algorithm can be used to automatically discover 3-way, 4-way, so on up till any r-way $\mathcal{R}$-$\mathcal{DP}$, where r is a constant. The number of unique recursive functions in each of the r-way $\mathcal{R}$-$\mathcal{DP}$ will be the same. We use visualization to analyze the pattern behind how the recursive functions updates different regions / submatrices of the DP table.

---

[2]Other types of cache-oblivious recursive algorithms [Frigo et al., 1999, Sibeyn, 2004, Cherng and Ladner, 2005, D'Alberto and Nicolau, 2007] require problem-specific algorithm design logic and hence the approach might not be generic enough to solve a large class of dynamic programs.

Consider Figure 5.2. It shows the order in which the $\mathcal{A}$ function of the r-way $\mathcal{R}$-$\mathcal{DP}$ for FW-APSP problem updates an $n \times n$ DP table for $r = 2, 3$, and 4. Similarly, we can write the updates of the $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ functions. In the figure, the light pink, light red, light yellow, and light blue regions represent the submatrices updated by $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ functions, respectively. The numbers represent the relative order in which the functions write to regions. If two functions have the same numbers, it means they can write to their respective regions at the same time. If the time of one function is lesser than the another, then the former function must finish execution before the second function can start.

In the figure, when $r = 2$, then the ranges of k are $\left[0, \frac{n}{2} - 1\right]$, and $\left[\frac{n}{2}, n - 1\right]$. Similarly, when $r = 3$, the ranges of k will be $\left[0, \frac{n}{3} - 1\right]$, $\left[\frac{n}{3}, \frac{2n}{3} - 1\right]$, and $\left[\frac{2n}{3}, n - 1\right]$. Similarly, k ranges are defined for other values of r.

The visualization of the function updates makes algorithm design easy. By analyzing the regions (denoted by colors) written by different recursive functions and times at which different functions are invoked for different values of k and r, we can see the pattern.

	$r = 2$	$r = 3$	$r = 4$
$k \in \left[0, \dfrac{n}{r} - 1\right]$	$\begin{array}{cc} A_0 & B_1 \\ C_1 & D_2 \end{array}$	$\begin{array}{ccc} A_0 & B_1 & B_1 \\ C_1 & D_2 & D_2 \\ C_1 & D_2 & D_2 \end{array}$	$\begin{array}{cccc} A_0 & B_1 & B_1 & B_1 \\ C_1 & D_2 & D_2 & D_2 \\ C_1 & D_2 & D_2 & D_2 \\ C_1 & D_2 & D_2 & D_2 \end{array}$
$k \in \left[\dfrac{n}{r}, \dfrac{2n}{r} - 1\right]$	$\begin{array}{cc} D_5 & C_4 \\ B_4 & A_3 \end{array}$	$\begin{array}{ccc} D_5 & C_4 & D_5 \\ B_4 & A_3 & B_4 \\ D_5 & C_4 & D_5 \end{array}$	$\begin{array}{cccc} D_5 & C_4 & D_5 & D_5 \\ B_4 & A_3 & B_4 & B_4 \\ D_5 & C_4 & D_5 & D_5 \\ D_5 & C_4 & D_5 & D_5 \end{array}$
$k \in \left[\dfrac{2n}{r}, \dfrac{3n}{r} - 1\right]$		$\begin{array}{ccc} D_8 & D_8 & C_7 \\ D_8 & D_8 & C_7 \\ B_7 & B_7 & A_6 \end{array}$	$\begin{array}{cccc} D_8 & D_8 & C_7 & D_8 \\ D_8 & D_8 & C_7 & D_8 \\ B_7 & B_7 & A_6 & B_7 \\ D_8 & D_8 & C_7 & D_8 \end{array}$
$k \in \left[\dfrac{3n}{r}, \dfrac{4n}{r} - 1\right]$			$\begin{array}{cccc} D_{11} & D_{11} & D_{11} & C_{10} \\ D_{11} & D_{11} & D_{11} & C_{10} \\ D_{11} & D_{11} & D_{11} & C_{10} \\ B_{10} & B_{10} & B_{10} & A_9 \end{array}$

Figure 5.2: The execution of the different functions invoked by the $\mathcal{A}$ function in r-way $\mathcal{R}$-$\mathcal{DP}$ of FW-APSP for $r = 2$, $r = 3$, and $r = 4$.

Figure 5.2 gives a visualization of the updates of different functions invoked by the $\mathcal{A}$ function of r-way $\mathcal{R}$-$\mathcal{DP}$ in FW-APSP. We can derive an r-way $\mathcal{R}$-$\mathcal{DP}$ from this visualization by setting $n = r$ and then analyzing. When we set $r = 4$, the updates of the four functions are as shown in Figure 5.3.

In Figure 5.3, $\mathcal{A}, \mathcal{B}, \mathcal{C}$, and $\mathcal{D}$ calls are shown in light pink, light red, light yellow, and light blue. Consider the updates of the DP table by function $\mathcal{A}$ as shown in column 1 of the entire figure. When we look at all the pink cells, we see a pattern. The position of the pink cells clearly depend on the value of k. To be exact, function $\mathcal{A}$ writes to cells at position (k, k). Immediately after $\mathcal{A}$ function call, functions $\mathcal{B}$ and $\mathcal{C}$ are invoked. Again, there is a clear-cut pattern. The $\mathcal{B}$ function calls are called on cells on the same row as that of $\mathcal{A}$ i.e., (k, j), where $j \neq k$. Also, the $\mathcal{C}$ functions are called on cells on the same column as that of $\mathcal{A}$ i.e., (i, k), where $i \neq k$. The function $\mathcal{D}$ is called on cells which have a different row and/or column than k. That is the position of the $\mathcal{D}$ function calls are (i, j), where $i \neq k$ and $j \neq k$.

Figure 5.3: The updates of the four functions in r-way $\mathcal{R}$-$\mathcal{DP}$ of FW-APSP for $n = r = 4$.

We show in Figure A.7 an r-way $\mathcal{R}$-$\mathcal{DP}$ for Floyd-Warshall's APSP with 4 functions

$\mathcal{A}, \mathcal{B}, \mathcal{C}$, and $\mathcal{D}$. The initial invocation to the r-way $\mathcal{R}$-$\mathcal{DP}$ is $\mathcal{A}(C, C, C, 1)$, where C is the DP table for the APSP problem. The term m in all the functions represents the dimension length at a particular recursion level. For function $\mathcal{A}$, $X = U = V$. For function $\mathcal{B}$, $X \neq U$ and $X = V$. Similarly, for function $\mathcal{C}$, $X = U$ and $X \neq V$. For function $\mathcal{D}$, $X \neq U$, $X \neq V$, and $U \neq V$. The keyword **parallel** means that the functions can be invoked in parallel. In this way, we design an r-way $\mathcal{R}$-$\mathcal{DP}$ for Floyd-Warshall's APSP.

$\mathcal{TR}$-$\mathcal{DP}$ **derivation**

In this section, we explain how to port an r-way $\mathcal{R}$-$\mathcal{DP}$ algorithm to a PMH model (see Section 1.4.2) as shown in Figure 1.1. A similar kind of porting can be done for several dynamic programs from the corresponding r-way $\mathcal{R}$-$\mathcal{DP}$s.

Let the r-way $\mathcal{R}$-$\mathcal{DP}$ for a problem $\mathcal{P}$ consist of functions $\mathcal{F}_1, \ldots, \mathcal{F}_m$ such that any function $\mathcal{F}_i$, where $i \in [1, m]$ calls other functions $\mathcal{F}_j$, where $j \in [i, m]$.

Consider Figure 1.1 in Section 1.4.2. Let the input DP table be present in the memory at level h. This means that the data from the DP table has to pass through the memory levels at $h-1, h-2, \ldots, 1$ to reach the processors at level 0. We define the following functions: `M[h]_F[1]_P`, `M[h-1]_F[1]_P`, $\ldots$, `M[h-1]_F[m]_P`, `M[h-2]_F[1]_P`, $\ldots$, `M[h-2]_F[m]_P`, so on till `M[1]_F[1]_P`, $\ldots$, `M[1]_F[m]_P`. The brackets [ ] are simply used to demarcate numbers from alphabet and they are not considered as part of the function names. The terms `F[1]`, $\ldots$, `F[m]` correspond to the m recursive functions. The functions with keyword `M[i]`, where $i \in [1, h]$ denotes that the input and the output matrices of the functions are present in memory at level i.

Initially, the function `M[h]_F[1]_P` is invoked with the entire DP table as input. The function splits the entire d-dimensional $n \times \cdots \times n$ DP table into $r_h \times \cdots \times r_h$ submatrices each of size $(n/r_h) \times \cdots \times (n/r_h)$, assuming r_h divides n for simplicity (and without loss of generality). The term r_h is chosen such that the input submatrices for the function exactly fits a memory at level $h-1$. The function copies the relevant submatrices to a $(h-1)$-level memory and then invokes m other functions `M[h-1]_F[1]_P`, $\ldots$, `M[h-1]_F[m]_P` as per the r-way $\mathcal{R}$-$\mathcal{DP}$ algorithm and sends the desired submatrices as input parameters to those child functions. Note that we need not define functions `M[h]_F[2]_P`, $\ldots$, `M[h]_F[m]_P` as they will never be invoked.

The functions `M[h-1]_F[1]_P`, $\ldots$, `M[h-1]_F[m]_P` split each of the $(n/r_h) \times \cdots \times (n/r_h)$ sized matrices present in level-$(h-1)$ memory into $r_{h-1} \times \cdots \times r_{h-1}$ submatrices each of size $(n/(r_h r_{h-1})) \times \cdots \times (n/(r_h r_{h-1}))$, assuming r_{h-1} divides (n/r_h) for simplicity. The functions copy the relevant matrices to the $(h-2)$-level memory and invoke the functions `M[h-2]_F[1]_P`, $\ldots$, `M[h-2]_F[m]_P`. This process continues till we reach the functions: `M[1]_F[1]_P`, $\ldots$, `M[1]_F[m]_P`. In these functions we run the iterative or looping kernels and make use of threads to perform several computations. This is a high-level overview of the division of work across recursive functions and memory levels. In this way, we can design $\mathcal{TR}$-$\mathcal{DP}$ algorithms from r-way $\mathcal{R}$-$\mathcal{DP}$s for a PMH model.

$\mathcal{TR}$-$\mathcal{DP}$ **for a GPU system.** We explain how to port the r-way Floyd-Warshall's APSP algorithm given in Figure A.7 to a GPU system. For simplicity, we assume that the memory hierarchy of our GPU system is as shown in Figure 5.1. It is easy to design algorithms for multiple levels, multiple GPUs, and multiple shared memories connected to a global memory.

Let the input DP table be present in the external memory. This means that the data from the DP table has to pass through three levels of memory: CPU RAM, GPU global memory, and GPU shared memory. We define the following functions: `host_disk_A_FW`, `host_RAM_A_FW`, ..., `host_RAM_D_FW`, `device_global_A_FW`, ..., `device_global_D_FW`, `device_shared_A_FW` ..., `device_shared_D_FW`. The terms `A`, `B`, `C`, and `D` correspond to the four recursive functions. The functions with keywords `host` and `device` represent that the code for such functions run on the CPU and GPU, respectively. The functions with keywords `disk`, `RAM`, `global`, and `shared` denote that the input and the output matrices of the functions are present in CPU disk, CPU RAM, GPU global memory, and GPU shared memory, respectively.

Initially, the function `host_disk_A_FW` is invoked with the entire DP table as input. The function splits the entire $n \times n$ DP table into $r_d \times r_d$ submatrices each of size $(n/r_d) \times (n/r_d)$, assuming r_d divides n for simplicity (and without loss of generality). The term r_d is chosen such that the input submatrices for the function exactly fits the RAM. The function copies the relevant submatrices to RAM and then invokes four other functions `host_RAM_A_FW`, ..., `host_RAM_D_FW` as per the r-way $\mathcal{R}\text{-}\mathcal{DP}$ algorithm and sends the desired submatrices as input parameters to those child functions. Note that we need not define functions `host_disk_B_FW`, ..., `host_disk_D_FW` as they will never be invoked.

The functions `host_RAM_A_FW`, ..., `host_RAM_D_FW` split each of the $(n/r_d) \times (n/r_d)$ sized matrices present in RAM into $r_m \times r_m$ submatrices each of size $(n/(r_d r_m)) \times (n/(r_d r_m))$, assuming r_m divides (n/r_d) for simplicity. The functions copy the relevant matrices to the GPU global memory and invoke the four functions `device_global_A_FW`, ..., `device_global_D_FW`. This process continues till we reach the functions: `device_shared_A_FW`, ..., `device_shared_D_FW`. In these functions we run the iterative or looping kernels and make use of threads to perform several computations. This is a high-level overview of the division of work across recursive functions and memory levels. In this way, we can design CPU-GPU algorithms from r-way $\mathcal{R}\text{-}\mathcal{DP}$s.

Now let's assume that we do not know the size of the CPU RAM, but it is maintained as fully associative memory with an automatic LRU page replacement policy. The input DP table is stored in either the external memory or the RAM. Then instead of `host_disk_A_FW` and `host_RAM_A_FW` we will only have `host_A_FW`, and similarly `host_B_FW`, `host_C_FW` and `host_D_FW`. Initially, the function `host_A_FW` is invoked with the entire DP table as input. The function splits the entire $n \times n$ DP table into 2×2 subtables each of size $(n/2) \times (n/2)$, assuming n is divisible by 2 for simplicity. Now if a $(n/2) \times (n/2)$ subtable fits into the GPU global memory we invoke `device_global_A_FW`, ..., `device_global_D_FW`, otherwise we recursively invoke `host_A_FW`, ..., `host_D_FW`.

5.3.2   Approach 2: Generalization of parallel iterative base cases

The five main steps of Autogen-Fractile are:

1. [*2-way $\mathcal{R}\text{-}\mathcal{DP}$ derivation*.] A 2-way $\mathcal{R}\text{-}\mathcal{DP}$ is constructed from a given $\mathcal{I}\text{-}\mathcal{DP}$.
2. [*Serial iterative base cases construction*.] Serial iterative base cases are constructed for the recursive functions of the 2-way $\mathcal{R}\text{-}\mathcal{DP}$.
3. [*Parallel iterative base cases visualization*.] Serial iterative base cases are constructed for the recursive functions of the 2-way $\mathcal{R}\text{-}\mathcal{DP}$.
4. [*r-way $\mathcal{R}\text{-}\mathcal{DP}$ derivation*.] An r-way $\mathcal{R}\text{-}\mathcal{DP}$ algorithm is derived with the aid of visualization.

5. [$\mathcal{TR}$-$\mathcal{DP}$ *derivation*.] The $\mathcal{TR}$-$\mathcal{DP}$ is derived for two consecutive memory levels from the r-way $\mathcal{R}$-$\mathcal{DP}$ by setting r based on the memory sizes.

2-way $\mathcal{R}$-$\mathcal{DP}$ derivation

We derive a 2-way $\mathcal{R}$-$\mathcal{DP}$ as explained in Section 5.3.1.

Serial iterative base cases construction

In this step, we construct the serial base cases for the recursive functions of the derived 2-way $\mathcal{R}$-$\mathcal{DP}$. *The serial base cases of both 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s are exactly the same*. Also, the serial base cases give vital information regarding the recursive structure of the r-way $\mathcal{R}$-$\mathcal{DP}$.

For example, for the FW-APSP problem, the serial base cases of the four recursive functions $\mathcal{A}, \mathcal{B}, \mathcal{C}$, and $\mathcal{D}$ of the 2-way $\mathcal{R}$-$\mathcal{DP}$ are shown in Figure 5.4. We assume that the initial distance matrix consists of only non-negative values. It happened by luck that the serial base cases for the four functions are the same. But, in general, the serial base cases of the recursive functions can be very different from each other.

The function $\mathcal{F}_{loop\text{-}FW}$ takes several parameters. The parameters $(xrow, xcol)$ are the starting row and starting column of write region X. Similarly, $(urow, ucol)$ and $(vrow, vcol)$ are the starting rows and columns of read submatrices U and V, respectively. The value n is the dimension length of the submatrix X (or U or V).

$\mathcal{F}_{loop\text{-}FW}(xrow, xcol, urow, ucol, vrow, vcol, n)$

1. **for** $k \leftarrow 0$ **to** $n-1$ **do**
2.   **for** $i \leftarrow 0$ **to** $n-1$ **do**
3.   **for** $j \leftarrow 0$ **to** $n-1$ **do**
4. $XI \leftarrow xrow + i; XJ \leftarrow xcol + j$
5. $UI \leftarrow urow + i; VJ \leftarrow vcol + j; K \leftarrow ucol + k$
6. $D[XI, XJ] \leftarrow \min\{D[XI, XJ], D[UI, K] + D[K, VJ]\}$

Figure 5.4: The serial base case of all recursive functions $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$ of 2-way $\mathcal{R}$-$\mathcal{DP}$ of FW-APSP assuming non-negative values for the initial distance matrix.

Constructing serial base cases for the recursive functions is majorly an art. By analyzing relationship between regions of a recursive function, we could attempt to come up with iterative kernels for updating the write regions from the read regions.

Parallel iterative base cases visualization

In this step, we visualize the execution of parallel iterative base cases of the recursive functions of the derived 2-way $\mathcal{R}$-$\mathcal{DP}$. Finding parallel iterative base cases is important as we want to find parallel r-way $\mathcal{R}$-$\mathcal{DP}$.

We design parallel iterative base cases using the help of computers. Through good visualization we find the parallel versions of serial iterative base cases. Let every update is of the form as given in Definition 4 in Section 2.2.1. Now, for each serial iterative base case, for a specific value of n, we write the updates in the $n \times \cdots \times n$ DP table using function names and time steps. If a cell x is written by reading u and v cells, then the write $\mathcal{F}_t$ at cell x, where $\mathcal{F}$ is the appropriate recursive function and t is the earliest time step at which $\mathcal{F}$

can write to cell x. More details of writing functions with timesteps can be found in Section 3.2.

For example, Figure 5.3 shows the execution of the parallel iterative base cases of the four functions in r-way $\mathcal{R}\text{-}\mathcal{DP}$ for FW-APSP for $n = 4$. For simplicity and space constraints, only the regions where the data are updated are shown and not the regions from where the data is read. We can extend the diagram with more details for better analysis. Similar diagrams can be written for $n = 5, 6, 7$, etc and we can easily see the pattern of how functions are called.

r-**way** $\mathcal{R}\text{-}\mathcal{DP}$ **derivation**

In this step, we derive an r-way $\mathcal{R}\text{-}\mathcal{DP}$ from parallel iterative base case kernels.

We explain the design of r-way $\mathcal{R}\text{-}\mathcal{DP}$ for the FW-APSP example. Figure 5.3 shows the parallel execution of the four function calls for the FW-APSP $\mathcal{R}\text{-}\mathcal{DP}$. Columns are for different functions and rows are for different values of $k \in [0, 3]$, where each value of k represents a plane number. The updates are shown for $n \times n$ DP table, where $n = 4$.

Once we have the visualization for the parallel base cases as shown in Figure 5.3, we can derive an r-way $\mathcal{R}\text{-}\mathcal{DP}$ as described in Section 5.3.1.

$\mathcal{A}_{FW}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{A}_{loop\text{-}FW}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4. $\mathcal{A}_{FW}(X_{kk}, U_{kk}, V_{kk}, d+1)$
5.   **parallel:** $\mathcal{B}_{FW}(X_{kj}, U_{kk}, V_{kj}, d+1)$, $\mathcal{C}_{FW}(X_{ik}, U_{ik}, V_{kk}, d+1)$ for $i, j \in [1, r]$, $i \neq k$, and $j \neq k$
6.   **parallel:** $\mathcal{D}_{FW}(X_{ij}, U_{ik}, V_{kj}, d+1)$ for $i, j \in [1, r]$, $i \neq k$, and $j \neq k$

$\mathcal{B}_{FW}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{B}_{loop\text{-}FW}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel:** $\mathcal{B}_{FW}(X_{kj}, U_{kk}, V_{kj}, d+1)$ for $j \in [1, r]$
5.   **parallel:** $\mathcal{D}_{FW}(X_{ij}, U_{ik}, V_{kj}, d+1)$ for $i, j \in [1, r]$ and $i \neq k$

$\mathcal{C}_{FW}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{C}_{loop\text{-}FW}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel:** $\mathcal{C}_{FW}(X_{ik}, U_{ik}, V_{kk}, d+1)$ for $i \in [1, r]$
5.   **parallel:** $\mathcal{D}_{FW}(X_{ij}, U_{ik}, V_{kj}, d+1)$ for $i, j \in [1, r]$ and $j \neq k$

$\mathcal{D}_{FW}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{D}_{loop\text{-}FW}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel:** $\mathcal{D}_{FW}(X_{ij}, U_{ik}, V_{kj}, d+1)$ for $i, j \in [1, r]$

Figure 5.5: An r-way $\mathcal{R}\text{-}\mathcal{DP}$ for Floyd-Warshall's APSP.

$\mathcal{TR}$-$\mathcal{DP}$ **derivation**

We derive the $\mathcal{TR}$-$\mathcal{DP}$ from the $\mathcal{R}$-$\mathcal{DP}$ as explained in Section 5.3.1.

5.4 Complexity analysis

In this section, we present the serial I/O complexity of r-way $\mathcal{R}$-$\mathcal{DP}$s on a parallel memory hierarchy (PMH) model, I/O complexity of GPU algorithms on a typical GPU architecture, and the parallel running time of $\mathcal{TR}$-$\mathcal{DP}$s on the PMH model.

The parallel memory hierarchy (PMH) model was described in Section 1.4.2. Most of our results are shown in the PMH model. The PMH model can be used in multi-cores or many cores.

5.4.1 I/O complexity

We state the theorem for the I/O complexity of an r-way $\mathcal{R}$-$\mathcal{DP}$ on a PMH model.

> **Theorem 9 (I/O complexity of r-way $\mathcal{R}$-$\mathcal{DP}$s on a PMH model).** *The serial I/O complexity of an r-way $\mathcal{R}$-$\mathcal{DP}$ on a PMH model for a d-dimensional hypercubic DP table of size n^d, where the parameter r is set to the best tile size at every recursion level, is*
>
> $$Q_1(n) = \mathcal{O}\left(\frac{T_1(n)}{BM^{(w/d)-1}} + \frac{S(n)}{B} + 1\right)$$
>
> *where, $T_1(n) = $ total work $= \tilde{\mathcal{O}}\left(n^w\right)$, $M = $ memory size, $B = $ block size, $M = \Omega\left(B^d\right)$, and $S(n) = $ input DP table size $= \mathcal{O}\left(n^d\right)$.*

Proof. We prove the theorem in two stages:

(a) $Q_1(n)$ *for r-way $\mathcal{R}$-$\mathcal{DP}$ $= Q_1(n)$ for 2-way $\mathcal{R}$-$\mathcal{DP}$.*
The metric $Q_1(n)$ for an $\mathcal{R}$-$\mathcal{DP}$ is computed as the product of number of subproblems that exactly fits the cache of size M and the I/Os required to scan the input and output matrices for a subproblem. Say $r = 2^k$ for some $k \in \mathbb{N}$. Then, to compare the I/O complexities of the two algorithms, the r-way $\mathcal{R}$-$\mathcal{DP}$ can be considered as simply the 2-way $\mathcal{R}$-$\mathcal{DP}$ unrolled k times. The number of subproblems that exactly fit a memory of size M will be asymptotically same for both 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s. Also, the I/Os required to scan the matrices that exact fit the memory is asymptotically same for both 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s. Hence, the I/O complexity of 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s are the same.

Alternate argument. The claim can be proved in two steps. First, both 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s exploit temporal locality from some constant factor of M. The reason for this is that the r-way $\mathcal{R}$-$\mathcal{DP}$ makes sure that every subproblem takes a constant fraction of the memory i.e., tile size is αM for some $\alpha \in (0, 1]$. Similarly, the 2-way $\mathcal{R}$-$\mathcal{DP}$ too exploits temporal locality from constant fraction of the memory i.e., subproblem size is γM for some $\gamma \in (0, 1]$.

Second, both 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s are simply different scheduling of the same DP recurrence and no DP cell update is applied more than once. Hence, the total work of the

2-way and r-way $\mathcal{R}\text{-}\mathcal{DP}$s are the same. In both 2-way and r-way $\mathcal{R}\text{-}\mathcal{DP}$s the amount of work done for a subproblem when it fits into a memory of size M is asymptotically same due to the previous arguments. Hence the proposition.

(b) Compute $Q_1(n)$ for 2-way $\mathcal{R}\text{-}\mathcal{DP}$.
In a given r-way $\mathcal{R}\text{-}\mathcal{DP}$ if we set $r = 2$ at every recursion level we get a 2-way $\mathcal{R}\text{-}\mathcal{DP}$. Once we have a 2-way $\mathcal{R}\text{-}\mathcal{DP}$ we can simply use Theorem 2 from [Chowdhury et al., 2016b]. □

Theorem 2 from [Chowdhury et al., 2016b] is defined for the cache-oblivious model [Frigo et al., 1999]. The cache-oblivious model assumes automatic page replacement policy. The theoretically optimal page replacement algorithm replaces the page (or data block) that will be used farthest in the future (which is difficult to predict). In practice, the least recently used (LRU) and first-in-first-out (FIFO) are the most commonly used page replacement algorithms as they are within a factor of 2 of the optimal number of page replacements. The current operating systems do not support automatic page replacements between CPU RAM and GPU global memory or between two adjacent memory levels of GPU. Still, it is possible to implement customized LRU or FIFO algorithms and simulate the automatic page replacement policy. Hence, the serial I/O complexity bounds of the r-way $\mathcal{R}\text{-}\mathcal{DP}$s does not get affected without the automatic page replacement policies on CPU-GPUs.

Consider Floyd-Warshall's APSP as an example. For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, let $Q_f(n)$ denote the serial I/O complexity of f_{FW} of the 2-way $\mathcal{R}\text{-}\mathcal{DP}$ on a matrix of size $n \times n$. Also, γ be a constant such that $\gamma \in (0, 1]$. Then $Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) = Q_{\mathcal{D}}(n) = \mathcal{O}\left(n^2/B + n\right)$ if $n^2 \leq \gamma M$. On the other hand if $n^2 > \gamma M$, then

$$Q_{\mathcal{A}}(n) = 2\left(Q_{\mathcal{A}}\left(\frac{n}{2}\right) + Q_{\mathcal{B}}\left(\frac{n}{2}\right) + Q_{\mathcal{C}}\left(\frac{n}{2}\right) + Q_{\mathcal{D}}\left(\frac{n}{2}\right)\right) + \Theta(1)$$
$$Q_{\mathcal{B}}(n) = 4\left(Q_{\mathcal{B}}\left(\frac{n}{2}\right) + Q_{\mathcal{D}}\left(\frac{n}{2}\right)\right) + \Theta(1)$$
$$Q_{\mathcal{C}}(n) = 4\left(Q_{\mathcal{C}}\left(\frac{n}{2}\right) + Q_{\mathcal{D}}\left(\frac{n}{2}\right)\right) + \Theta(1)$$
$$Q_{\mathcal{D}}(n) = 8Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1)$$

Applying Theorem 9, the serial I/O complexity of the r-way $\mathcal{R}\text{-}\mathcal{DP}$ of Floyd-Warshall's APSP as shown in Figure A.7 is $Q_1(n) = \mathcal{O}\left(n^3/\left(B\sqrt{M}\right) + n^2/B + 1\right)$.

We define a few terms before we present a theorem for the serial I/O complexity of GPU algorithms. The theorem assumes that the core memory hierarchy of a GPU system is as shown in Figure 5.1, excluding intricate details. Let M_m, M_g, and M_s be the size of the CPU main memory, GPU global memory, and GPU shared memory, respectively. Let n, n_m, n_g, and n_s be the subproblem parameter (i.e., tile dimension) when the subproblem is present in the disk, RAM, global, and shared memories, respectively. Then $n_m^d = \Theta(M_m)$, $n_g^d = \Theta(M_g)$ and $n_s^d = \Theta(M_s)$. Let B, B_m, B_g, and B_s denote the block size between disk and RAM, RAM and global memory, global memory and shared memory, and shared memory and processor, respectively. All M's, n's, and B's are natural numbers.

> **Theorem 10 (I/O complexity of GPU algorithms).** *The I/O complexity (number of data blocks transferred) of an external-memory GPU algorithm when run on a GPU memory hierarchy as shown in Figure 5.1 between:*
>
> ⋆ *disk & RAM is* $\Theta\left(\dfrac{n^w}{BM_m^{\frac{w}{d}-1}} + \dfrac{n^w}{M_m^{\frac{w+1}{d}-1}}\right)$
>
> ⋆ *RAM & global memory is* $\Theta\left(\dfrac{n^w}{B_m M_g^{\frac{w}{d}-1}} + \dfrac{n^w}{M_g^{\frac{w+1}{d}-1}}\right)$
>
> ⋆ *global & shared memory is* $\Theta\left(\dfrac{n^w}{B_g M_s^{\frac{w}{d}-1}} + \dfrac{n^w}{M_s^{\frac{w+1}{d}-1}}\right)$
>
> *where, the total work of the GPU algorithm is* $\Theta\left(n^w\right)$.

Proof. First, we find the I/O complexity of the r-way $\mathcal{R}$-$\mathcal{DP}$ between two adjacent levels of memories and then we directly use this result to prove the theorem.

Let the larger and smaller adjacent memories be of sizes M_L and M_S, respectively. Let the largest subproblem parameters that fit into M_L and M_S be integers n_L and n_S, respectively. Let B_L denote the block size between larger and smaller memories. The number of write and read submatrices for each recursive function of the $\mathcal{R}$-$\mathcal{DP}$ is $1 + s$, where s is a upper bounded by a constant. This is because there is only one region to be written and there can be at most s distinct regions (not including the write region) to be read from. Without loss of generality, we assume that an element takes 1 byte. The space occupied by the $(1 + s)$ read and write regions in the smaller memory is $(1 + s)n_S^d \le M_S$, where $n_S \le (M_L/(1 + s))^{1/d} < n_S + 1$. Then the I/O complexity to fill the smaller memory once is computed as $\mathcal{O}\left(n_S^{d-1}(n_S/B_L + 1)\right)$. The smaller memory will be loaded $\Theta\left((n_L/n_S)^w\right)$ times. Hence, the I/O complexity between the larger and the smaller memories is $\mathcal{O}\left((n_L/n_S)^w n_S^{d-1}(n_S/B_L + 1)\right)$.

We now apply the result above to prove the theorem. The I/O complexity between disk and RAM is $\mathcal{O}((n/n_m)^w n_m^{d-1}\left(n_m/B + 1\right))$. The I/O complexity between RAM and global memory to work on all data present in RAM is $\mathcal{O}((n_m/n_g)^w n_g^{d-1}\left(n_g/B_m + 1\right))$. However, the RAM will be loaded $\Theta\left((n/n_m)^w\right)$ times. Hence, the total I/O complexity between RAM and global is $\mathcal{O}((n/n_g)^w n_g^{d-1}\left(n_g/B_m + 1\right))$. We use a similar reasoning to compute the total I/O complexity between global and shared memories. Substituting $n_m^d = \Theta\left(M_m\right)$, $n_g^d = \Theta\left(M_g\right)$ and $n_s^d = \Theta\left(M_s\right)$ we obtain the claimed bounds. $\qquad\square$

5.4.2 Parallel running time

We formalize a few ideas before we give a theorem on the parallel running time of the r-way $\mathcal{R}$-$\mathcal{DP}$s on a parallel machine with deep memory hierarchy using space-bounded scheduler.

Let the recursive functions of an r-way $\mathcal{R}$-$\mathcal{DP}$ be $\mathcal{F}_1, \dots, \mathcal{F}_m$. Let us consider the recursive function calls any $\mathcal{F}_j$ makes when the data is in level i and is tiled for level $i - 1$ of the PMH hierarchy. Suppose $\mathcal{F}_j$ calls $\mathcal{F}_k$ a total of $a_{j,k}^i$ times from level i when $p_i = 1$. If $p_i = \infty$ then $\mathcal{F}_j$ will call as many recursive functions as possible in parallel, and let the number of such parallel steps in which $\mathcal{F}_j$ makes at least one call to $\mathcal{F}_k$ is $b_{j,k}^i$. Let $\mathbf{A}^{(i)}$ and $\mathbf{B}^{(i)}$ denote the level-i coefficient matrices $(a_{j,k}^i)$ and $(b_{j,k}^i)$, respectively. Let $\mathbf{A}$ and $\mathbf{B}$ be computed as:

$$\mathbf{A} = \mathbf{A}^{(h)} \otimes \mathbf{A}^{(h-1)} \otimes \cdots \otimes \mathbf{A}^{(1)}$$

$$\mathbf{B} = \mathbf{B}^{(h)} \otimes \mathbf{B}^{(h-1)} \otimes \cdots \otimes \mathbf{B}^{(1)}$$

where, $\otimes$ is a matrix multiplication operator.

> **Theorem 11 (Parallel running time of r-way $\mathcal{R}$-$\mathcal{DP}$s on a PMH model using space bounded scheduler).** *Let the recursive functions of an r-way $\mathcal{R}$-$\mathcal{DP}$ be $\mathcal{F}_1, \ldots, \mathcal{F}_m$, and let $\boldsymbol{A}$ and $\boldsymbol{B}$ be as defined above. Then the parallel running time of the $\mathcal{R}$-$\mathcal{DP}$ on a DP table of size n^d under the PMH model with $h+1$ levels as defined above will be:*
>
> $$T_{PMH}^h(n) = \mathcal{O}\left(\frac{\sum_{i=1}^m \boldsymbol{A}[i,i]}{p} + \sum_{i=1}^m \boldsymbol{B}[i,i]\right) \cdot \Pi_{i=1}^h B_i$$

Proof. Suppose there are h levels of recursion. Let $r_h, \ldots, r_1$ be the best tile sizes for recursion levels h to 1, respectively. This implies that $n = r_h \times r_{h-1} \times \cdots \times r_1$. The total space complexity of the algorithm is assumed to be $S(n) = \Theta\left(n^d\right)$. Then,

$$r_i = \begin{cases} \Theta\left(\frac{n}{M_{h-1}^{\frac{1}{d}}}\right) & \text{if } i = h, \\ \Theta\left(\left(\frac{M_i}{M_{i-1}}\right)^{\frac{1}{d}}\right) & \text{if } i \in [2, h-1], \\ \Theta\left(M_1^{\frac{1}{d}}\right) & \text{if } i = 1. \end{cases}$$

We assume for generality that it takes B time to write cells to a block of size B if the cells are computed from different processors.

An $\mathcal{R}$-$\mathcal{DP}$ has only one recursive function. Let an r-way $\mathcal{R}$-$\mathcal{DP}$ consist of only one function. Let the work and span of the function be defined by: $T_1(n) = aT_1(n/r) + \Theta(1)$ and $T_\infty(n) = bT_\infty(n/r) + \Theta(1)$ for $a, b \geq 1$ and $n \geq r > 1$. The work and span both are $\Theta(1)$ if $n = 1$. Let $T_{\mathrm{PMH}}(n)$ be the parallel running time of the algorithm on a parallel machine with the PMH memory model. We define $T_{\mathrm{PMH}}^i(n)$ to be the parallel running time of the recursive function at level i. This implies $T_{\mathrm{PMH}}(n) = T_{\mathrm{PMH}}^h(n)$. Then, $T_{\mathrm{PMH}}^i(n)$ can be computed recursively as:

$$T_{\mathrm{PMH}}^i(n) = \begin{cases} \mathcal{O}\left(\left(\frac{T_1(r_1)}{p_1} + T_\infty(r_1)\right) \cdot B_1\right) & \text{if } i = 1, \\ \mathcal{O}\left(\left(\frac{T_1(r_i)}{p_i} + T_\infty(r_i)\right) \cdot B_i\right) \cdot T_{\mathrm{PMH}}^{i-1}\left(\frac{n}{r_i}\right) & \text{if } i \in [2, h]. \end{cases}$$

Solving the recurrence, we get:

$$T_{\mathrm{PMH}}^h(n) = \Pi_{i=1}^h \mathcal{O}\left(\left(\frac{T_1(r_i)}{p_i} + T_\infty(r_i)\right) \cdot B_i\right)$$

An $\mathcal{R}$-$\mathcal{DP}$ has multiple recursive functions. We extend the analysis above to multiple functions. Let the recursive functions of the r-way $\mathcal{R}$-$\mathcal{DP}$ be $\mathcal{F}_1, \ldots, \mathcal{F}_m$. Let the work and span of an r-way recursive function $\mathcal{F}_j$ at level i be defined as: $T_1^{i,\mathcal{F}_j}(n) = \sum_{k=1}^m a_{j,k}^i T_1^{i-1,\mathcal{F}_k}(n/r_i) + \Theta(1)$ and $T_\infty^{i,\mathcal{F}_j}(n) = \sum_{k=1}^m b_{j,k}^i T_\infty^{i-1,\mathcal{F}_k}(n/r_i) + \Theta(1)$. As base cases, $T_1^{1,\mathcal{F}_j}(r_1) = \sum_{k=1}^m a_{j,k}^1$ and $T_\infty^{1,\mathcal{F}_j}(r_1) = \sum_{k=1}^m b_{j,k}^1$. It is important to note that if the span of a function has $\max\{x, y\}$ term, then it is replaced with $x + y$ thereby this serves as an upper bound on span. Let the

parallel running time of the function $\mathcal{F}_j$ at level i be denoted by $T_{\text{PMH}}^{i,\mathcal{F}_j}(n)$. Then we compute $T_{\text{PMH}}^{i,\mathcal{F}_j}(n)$ recursively as:

$$
T_{\text{PMH}}^{i,\mathcal{F}_j}(n) = \begin{cases} \mathcal{O}\left(\left(\frac{T_1^{1,\mathcal{F}_j}(r_1)}{p_1} + T_\infty^{1,\mathcal{F}_j}(r_1) \right) \cdot B_1 \right) & \text{if } i = 1, \\ \sum_{k=1}^{m} \left(\mathcal{O}\left(\left(\frac{a_{j,k}^i}{p_i} + b_{j,k}^i \right) \cdot B_i \right) \cdot T_{\text{PMH}}^{i-1,\mathcal{F}_k}\left(\frac{n}{r_i} \right) \right) & \text{if } i \in [2, h]. \end{cases}
$$

Let $\mathbf{A}^{(i)}$ and $\mathbf{B}^{(i)}$ denote the level-i coefficient matrices $(a_{j,k}^i)$ and $(b_{j,k}^i)$, respectively. Let $\mathbf{A}$ and $\mathbf{B}$ be computed as:

$$
\mathbf{A} = \mathbf{A}^{(h)} \otimes \mathbf{A}^{(h-1)} \otimes \cdots \otimes \mathbf{A}^{(1)}
$$
$$
\mathbf{B} = \mathbf{B}^{(h)} \otimes \mathbf{B}^{(h-1)} \otimes \cdots \otimes \mathbf{B}^{(1)}
$$

where, $\otimes$ is a matrix multiplication operator. Solving the parallel running time recurrence, we get:

$$
T_{\text{PMH}}^h(n) = \mathcal{O}\left(\frac{\sum_{i=1}^{m} \mathbf{A}[i,i]}{p} + \sum_{i=1}^{m} \mathbf{B}[i,i] \right) \cdot \Pi_{i=1}^h B_i
$$

$\square$

5.5 Experimental results

In this section, we present empirical results showing the performance benefits of our GPU algorithms that are based on r-way $\mathcal{R}$-$\mathcal{DP}$. The internal-memory GPU algorithms are implemented by Stephen Tschudi and the external-memory GPU algorithms are implemented by Rathish Das.

5.5.1 Experimental setup

All our experiments were performed on a heterogeneous node of Stampede supercomputer. The multicore machine had a dual-socket 8-core 2.7 GHz Intel Sandy Bridge processors ($2 \times 8 = 16$ cores in total) and 32 GB RAM. Each core was connected to a 32 KB private L1 cache and a 256 KB private L2 cache. All cores in a processor shared a 20 MB L3 cache. For GPGPU processing the Stampede compute nodes were attached with a single NVIDIA K20 GPU on each node. Each GPU had an on-board GDDR5 memory of 5GB. The GPU machine had 2496 CUDA cores for parallel processing.

All our algorithms were implemented in C++. We used Intel Cilk Plus extension to parallelize and Intel⁶ C++ Compiler v13.0 to compile the CPU implementations with optimzation parameters `-O3 -ipo -parallel -AVX -xhost`. We used NVIDIA's CUDA platform to write our GPU programs. The programs were compiled with `nvcc` compiler with parameters `-O3 -gencode arch=compute_35,code=sm_35`.

5.5.2 Internal-memory GPU implementations

We focus on four DP problems: Floyd-Warshall's APSP, Gaussian elimination without pivoting, parenthesis problem, and the gap problem. The programs with `cpu` and `gpu` keywords

are run on CPU and GPU machines, respectively. The different types of programs we compare are:

(i) `cpu-idp`: iterative serial program,

(ii) `cpu-rdp`: standard 2-way parallel $\mathcal{R}$-$\mathcal{DP}$,

(iii) `gpu-rdp`: r-way parallel $\mathcal{R}$-$\mathcal{DP}$.

For the FW problem, we also have

(iv) `gpu-tidp-harish`: Harish and Narayanan's [Harish and Narayanan, 2007] tiled-iterative program,

(v) `gpu-tidp-lund`: Lund and Smith's [Lund and Smith, 2010] tiled-iterative program,

(vi) `gpu-tidp-katz`: Katz and Kider's [Katz and Kider Jr, 2008] tiled-iterative program,

(vii) `gpu-rec-buluc`: Buluc et al.'s implementation of the 2-way R-Kleene algorithm with Volkov and Demmel's optimization [Volkov and Demmel, 2008] for the MM kernel,

($viii$) `gpu-rdp-opt`: r-way $\mathcal{R}$-$\mathcal{DP}$ replaced with Buluc et al.'s MM-like[3] kernel for the MM-like functions of the $\mathcal{R}$-$\mathcal{DP}$.

Optimizations. The common optimizations used for the programs, except `cpu-idp`, are as follows.

(i) We use shared memory of the GPUs by setting `BLOCK_SIZE = 32` so that 1024 threads could work on square matrices of size 32×32 simultaneously. Also, two blocks with 1024 threads were run in parallel.

(ii) If a function kernel reads from submatrices it is not writing to (MM-like), then we do not use synchronization inside the kernel.

(iii) Row-major order was used for all submatrices instead of column-major order. Flipping a submatrix to column-major degrades performance. We used row-major order for the grid and also inside each block inside a grid. Using row-major across `BLOCK_SIZE × BLOCK_SIZE` worsens the performance.

(iv) Allocating memory using `gpuMalloc()` on GPU global memory is slow. Instead of invoking this function multiple times we simply malloc once and then copy the submatrices to the respective regions.

(v) We allocate directly in host's pinned memory using `cudaMallocHost()`. This reduces the block transfers between pageable host memory and pinned memory.

The `cpu-idp` was not optimized. The optimizations used for `cpu-rdp` include:

(i) #pragmas such as `#pragma parallel`, `#pragma ivdep`, and `min loop count(B)`,

(ii) using 64 byte-aligned matrices,

(iii) write optimizations,

(iv) using pointer arithmetic,

(v) Z-morton layout (only for the gap problem).

The optimizations used for the `gpu-rdp` programs are: (i) block-row-major order (similar as in the case of tiling) to reduce data transfers, (ii) GRID_SIZE was set to $\min\{n, 16384\}$, where 16384 was the maximum size such that our subproblems can exactly fit into the 5GB of global memory, (iii) The data copy from RAM to global memory and vice versa was synchronized. Using asynchronous copy (overlapping computations and data copy) is extremely complicated.

There are a few important points related to the programs of the FW problem. The three tiled-iterative implementations were blocked for 32×32 instead of 16×16 as set in the

---

[3]MM = Matrix Multiplication

original versions. The $\mathcal{A}, \mathcal{B}$, and $\mathcal{C}$ functions in the tiled implementations were optimized as much as possible, whereas the $\mathcal{D}$ function was already optimized.

Results. Figure 5.6 shows the speedup of various programs w.r.t. `cpu-idp` for four DP problems. For each program, the DP table dimension n is varied from 2^{10} to 2^{15}. When $n = 2^{15}$, a subproblem fits in RAM, and in all other cases the subproblems fit in GPU global memory. For FW-APSP, `gpu-rdp-opt` was the second fastest running program with a speedup of $219\times$, whereas `gpu-rec-buluc` had a speedup of $330\times$ for $n = 2^{15}$. This is because unlike `gpu-rec-buluc`, all the kernels of `gpu-rdp-opt` were not MM-like and hence it ran slower than Buluc et al.'s implementation.

For the Gaussian elimination, parenthesis and gap problems for $n = 2^{15}$, the speedup of our `gpu-rdp` programs were $216\times, 1762\times$, and $523\times$, respectively, way higher than their `cpu-rdp` counterparts which were $169\times, 162\times$, and $188\times$, respectively. The speedup of the GPU algorithms for the parenthesis and gap problems is more than that for FW-APSP / Gaussian elimination because of two reasons:

(i) `cpu-idp` for the former two problems do not have spatial locality whereas the latter two have spatial locality,

(ii) `gpu-rdp` for the former two problems have higher parallelism than the latter two.

5.5.3   External-memory GPU implementations

It is easy to extend our algorithms to work for external-memory (or disks). We could use either 2-way or r-way $\mathcal{R}$-$\mathcal{DP}$s for external-memory until a subproblem fits in GPU global memory, after which we use r-way $\mathcal{R}$-$\mathcal{DP}$s. To make our algorithms RAM-oblivious, we use 2-way $\mathcal{R}$-$\mathcal{DP}$s in the external-memory until a subproblem fits in GPU global memory. We have implemented algorithms for all four problems: Floyd-Warshall's APSP, parenthesis, gap, and Gaussian elimination. We use Standard Template Library for Extra Large Data Sets (STXXL) [STX,] 1.4.1 to implement our algorithms for external-memory. STXXL is a C++ library for implementing containers and algorithms that can process vast amounts of data that reside in disks. In STXXL, we set the external block size as 4MB, #pages as 1024, and #blocks per page as 1. This gives the RAM size as 4GB.

For each of the four DP problems we compare three programs:

(a) `cpu-rdp-1`: 2-way serial $\mathcal{R}$-$\mathcal{DP}$ running on CPU,

(b) `cpu-rdp-128`: 2-way parallel $\mathcal{R}$-$\mathcal{DP}$ running on CPU with 128 cores (the details of which is explained shortly),

(c) `gpu-rdp`: r-way parallel $\mathcal{R}$-$\mathcal{DP}$ running on the GPU machine.

The input DP table is stored in Z-morton-row-major layout in the external-memory such that when a submatrix reaches a size that fits in the GPU global memory it is stored in row-major order. Note that the input problem consists of a single matrix in external-memory. On the other hand, a subproblem can consist of multiple submatrices (e.g. up to 3 matrices for the $\mathcal{R}$-$\mathcal{DP}$ of FW-APSP) of the input DP table and they all have to fit into the GPU global memory and this is taken care by the recursive functions of the r-way $\mathcal{R}$-$\mathcal{DP}$s. Once we compute a DP table submatrix, we write the output to the same location in the DP table in the external-memory. For the CPU programs, the base case dimension length for the algorithms was set to 256 and we run iterative kernels inside each base case.

The running time of `cpu-rdp-1` and `cpu-rdp-128` are approximated as follows. The DP table is stored as a grid of blocks, each block is of size 16K×16K and it is stored in Z-morton

order. We use r-way $\mathcal{R}$-$\mathcal{DP}$[4] in external-memory and whenever a subproblem is brought to RAM, we use 2-way $\mathcal{R}$-$\mathcal{DP}$ to execute it on CPU.

Let $n_{base}, n_{base}^{128}, n_{chunk}, t_{base}, t_{chunk}$ represent the number of invocations to base case kernels, number of parallel steps of execution of the base case kernels when we assume 128 cores, number of times RAM (of size 16K $\times$ 16K) is loaded and unloaded, minimum time taken (among several runs) to execute a base case kernel, and time taken to copy data between external-memory and RAM as given in STXXL I/O statistics, respectively. Then, running time of `cpu-rdp-1` is $(n_{base} \cdot t_{base} + n_{chunk} \cdot t_{chunk})$. The running time of `cpu-rdp-128` is $(n_{base}^{128} \cdot t_{base} + n_{chunk} \cdot t_{chunk})$.

Results. Figure 5.7 shows the speedup of various programs w.r.t. `cpu-rdp-1` for four DP problems. For each program, the DP table dimension n is varied from 2^{15} to 2^{17}. In all cases, the input matrices are stored in the external-memory. For FW-APSP, Gaussian elimination, parenthesis and gap problems for $n = 2^{17}$, the speedup of our `gpu-rdp` programs were $3072\times, 1096\times, 3376\times,$ and $1122\times$, respectively, way higher than their `cpu-rdp-128` counterparts which were $126\times, 126\times, 112\times,$ and $122\times$, respectively.
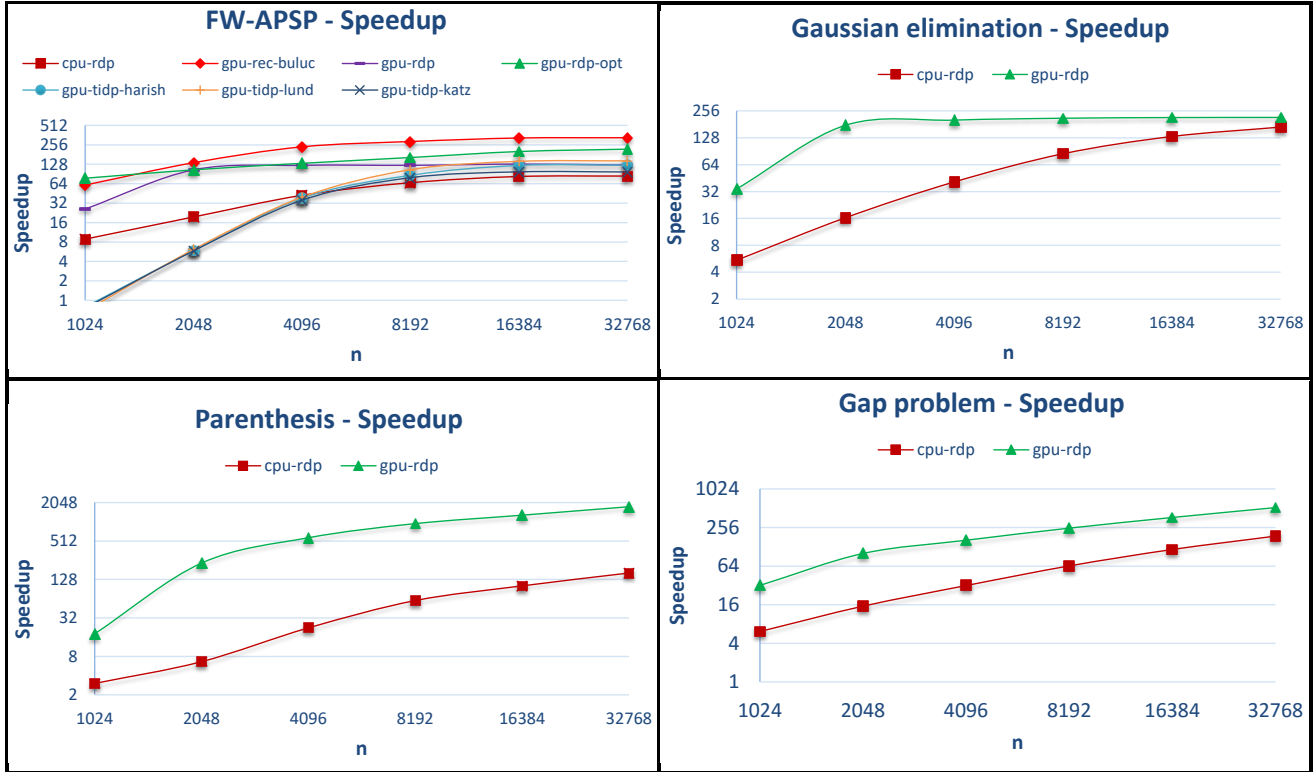
Figure 5.6: Speedup of `cpu-rdp` and `gpu-rdp` programs over `cpu-idp` for various dynamic programs. For FW-APSP, the speedup of `gpu-buluc-rec` and `gpu-rdp-opt` are also shown.

5.6 Conclusion and open problems

We presented a framework called AUTOGEN-FRACTILE to semi-automatically discover recursively tiled algorithms based on r-way recursive divide-and-conquer that are efficient in

---
[4]We did not make the program RAM-oblivious to make the benchmark program run fastest.
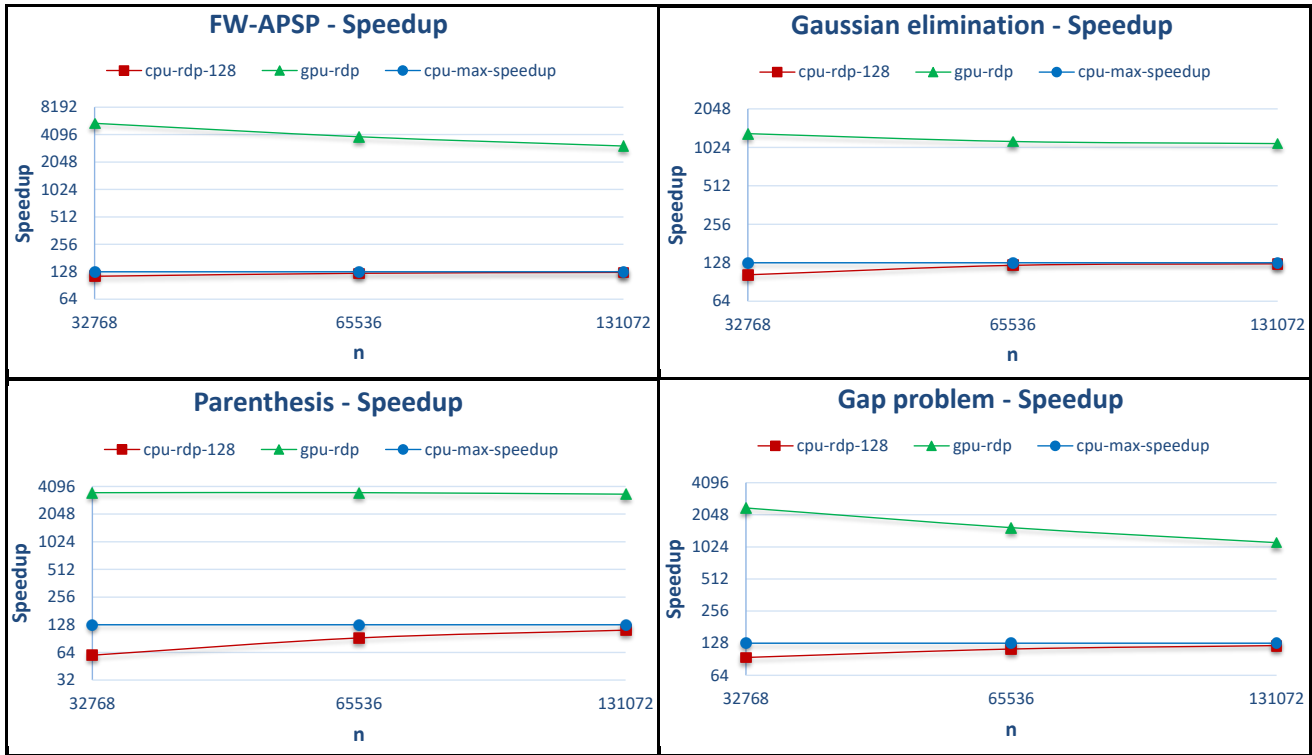
Figure 5.7: Speedup of `cpu-rdp-128` and `gpu-rdp` programs w.r.t. `cpu-rdp-1` for various dynamic programs in external-memory.

computer architectures that do not support recursion. We develop several efficient algorithms for GPUs using our framework and achieve high performance and beat almost all the existing implementations.

A few open problems are:

★ [*Automation of AUTOGEN-FRACTILE framework*.] Completely automate the derivation of formulas and looping constraints for designing the r-way $\mathcal{R}\text{-}\mathcal{DP}$s.

Chapter 6

Semi-Automatic Discovery of Divide-&-Conquer DP Algorithms with Space-Parallelism Tradeoff

In chapter 2, we presented the AUTOGEN algorithm to automatically discover divide-and-conquer DP algorithms. In chapter 3, we presented the AUTOGEN-WAVE framework to improve the parallelism of these auto-discovered algorithms to near-optimal. In this chapter, we will see a framework called AUTOGEN-TRADEOFF that can be used to improve the parallelism of certain class of the AUTOGEN-discovered algorithms by increasing the space usage.

The standard in-place divide-and-conquer parallel matrix multiplication algorithm uses $\Theta\left(p \log n\right)$ extra space (in the worst case) including stack space (where $p = \#$threads) and has $\Theta\left(n^2\right)$ parallelism. On the other hand, the not-in-place divide-and-conquer parallel matrix multiplication algorithm uses $\Theta\left(p^{1/3}n^2\right)$ extra space, but achieves high parallelism of $\Theta\left(n^3/\log^2 n\right)$. The more space we use, the more parallelism we can exploit. So, there exists a tradeoff between total space and parallelism.

We present the AUTOGEN-TRADEOFF framework that can be used to improve the parallelism of a class of divide-and-conquer cache-oblivious matrix algorithms by increasing the space. As an application of the framework, we present a hybrid matrix multiplication algorithm, which is a careful mix of the two algorithms above, to achieve parallelism $\omega\left(n^2\right)$ using total space of $\Theta\left(n^{2+\epsilon}\right)$ for some $\epsilon \in (0,1)$. Similarly, we give hybrid algorithms to asymptotically increase the parallelism for other algorithms such as multi-instance Viterbi algorithm, Floyd-Warshall's all-pairs shortest path, and protein folding problems.

Our experiments show that when we have enough space and thousands of processors, these hybrid algorithms will be able to harness more parallelism to achieve good performance.

6.1 Introduction

Multicore machines often have a tree-like cache hierarchy consisting of different-sized caches having various access times. The access times of caches farther from the processors are an order of magnitude times that of caches near to the processors. Reducing the number of data movements for an algorithm across different levels of the cache hierarchy

invariably reduces the running time of the algorithm. To achieve this, when programs bring data to cache, they must do as much work as possible before evicting the data out of cache. Hence, achieving good cache locality improves the overall performance of an algorithm.

Present-day multicore machines have more than one processing element (ranging from 2 to 80) to execute a program in parallel decreasing its overall running time. More the number of processors, the faster the algorithm runs. To exploit parallelism, it is not just enough to have more processing elements but the algorithm must have asymptotically more parallelism as well. Good cache locality and good parallelism are the two key factors to increase the performance of a parallel algorithm. Also, if a parallel algorithm has to be portable i.e., be able to run on machines from smart phones to compute nodes of supercomputers without modification then the algorithm has to be resource-oblivious (e.g.: cache- and processor-oblivious).

The cache-oblivious recursive divide-and-conquer parallel algorithms highly parallel, and exploit several optimization opportunities. By this virtue, if properly implemented, they can be both high-performing and portable at the same time. But, one important question remains. Can we increase the parallelism of these algorithms?

Space-parallelism tradeoff. We build on top of a simple space-parallelism tradeoff idea to show that with extra space we can increase the parallelism of the cache-oblivious divide-and-conquer parallel algorithms asymptotically. This idea works for several problems such as matrix multiplication, multi-instance Viterbi algorithm, Floyd-Warshall's all-pairs shortest path, and protein accordion folding.

Consider an example. Given an array of n numbers, its sum can be found in a couple of different ways without altering the input array (see Table 6.1):

1. [*Sequential sum.*] The sum is found by simply adding all the numbers using $\Theta(1)$ extra space and it has a span of $\Theta(n)$ (or a parallelism of $\Theta(1)$) as the algorithm is inherently sequential.

2. [*Parallel reduction.*] The sum is found using parallel reduction using $\Theta(n)$ extra space and it has a span of $\Theta(\log n)$ (or a parallelism of $\Theta(n/\log n)$). See Figure 6.1.

Algorithm	Work (T_1)	Span (T_∞)	Parallelism (T_1/T_∞)	Extra space (S_∞)
Sequential sum (in-place)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Parallel reduction (not-in-place)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n/\log n)$	$\Theta(n)$
Hybrid algorithm 1 (not-in-place)	$\Theta(n)$	$\Theta((n/r)\log r)$	$\Theta(r/\log r)$	$\Theta(r)$
Hybrid algorithm 2 (not-in-place)	$\Theta(n)$	$\Theta((n/r) + \log r)$	$\Theta(nr/(n + r\log r))$	$\Theta(r)$

Table 6.1: Space-parallelism tradeoff for sum of an array. For the hybrid algorithm, $r \in [2, n]$.

It is straightforward to see that we can have a space-parallelism tradeoff for the algorithm above. When we increase the extra space from $\Theta(1)$ to $\Theta(n)$, the span decreases from $\Theta(n)$ to $\Theta(\log n)$ or the parallelism increases from $\Theta(1)$ to $\Theta(n/\log n)$.

We can construct two types of hybrid algorithms using the basic algorithms. We assume that we are allowed $\Theta(r)$ extra space, where $r \in [2, n]$. The two hybrid algorithms using the allowed extra space are:
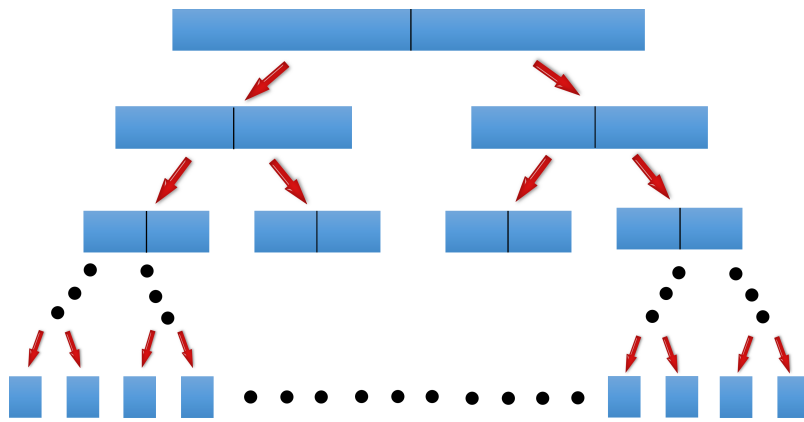
Figure 6.1: Computing sum of an array using parallel reduction.

1. [*Hybrid algorithm 1: Serial on parallel.*] We divide the array of size n into $\Theta(n/r)$ chunks. For each of the $\Theta(n/r)$ chunks we apply the parallel reduction method. The sum of numbers in each chunk is computed in parallel. However, the chunks are processed in a serial fashion. The span for computing sum in each chunk is $\Theta(\log r)$. There are $\Theta(n/r)$ chunks. Hence, the total span is $\Theta((n/r)\log r)$. Parallelism is $\Theta(r/\log r)$.

2. [*Hybrid algorithm 2: Parallel on serial.*] We divide the array of size n into $\Theta(r)$ chunks. For each of the $\Theta(r)$ chunks we apply the serial sum method. The sum of numbers in each chunk is computed serially. However, the chunks are processed in a parallel fashion using parallel reduction method. The span for computing sum in each chunk is $\Theta(n/r)$. There are $\Theta(r)$ chunks. Using a parallel reduction method, the total span will be $\Theta((n/r) + \log r)$. Parallelism is $\Theta(nr/(n + r\log r))$.
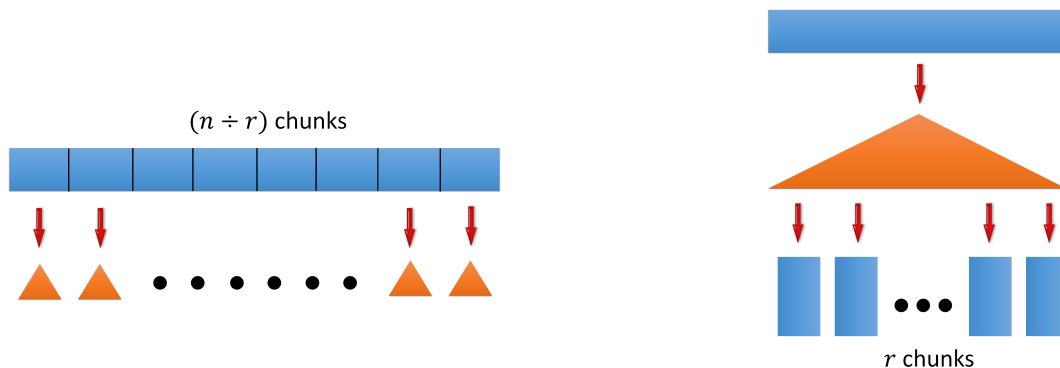
Figure 6.2: Pictorial representation of the two hybrid algorithms to compute the sum of elements of an array. Left: Hybrid algorithm 1: Serial on parallel. Right: Hybrid algorithm 2: Parallel on serial.

We extend this idea further to matrix multiplication. There are two major ways to multiply matrices using divide-and-conquer (see Figure 6.3):

1. In-place algorithm denoted by $\mathcal{A}_{MM}$
2. Not-in-place algorithm denoted by $\mathcal{A}_{MM\text{-}N}$

The $\mathcal{A}_{MM}$ algorithm has optimal $Q_1(n) = \Theta\left(n^3/(B\sqrt{M})\right)$, has decent span of $T_\infty(n) = \Theta(n)$, and uses extra space of $\Theta(p\log n)$ including stack space. On the other hand, the

$\mathcal{A}_{MM\text{-}N}$ algorithm has optimal $Q_1(n) = \Theta\left(n^3/(B\sqrt{M})\right)$ but, has a small span of $T_\infty(n) = \Theta\left(\log^2 n\right)$, and uses extra space of $\mathcal{O}\left(p^{1/3}n^2\right)$. Using the space-parallelism tradeoff idea, we can get hybrid algorithms for matrix multiplication and other problems.

We develop a hybrid algorithm $\mathcal{A}_{MM\text{-}H}$ following a careful combination of $\mathcal{A}_{MM}$ and $\mathcal{A}_{MM\text{-}N}$. We execute $\mathcal{A}_{MM\text{-}N}$ for a few levels in the recursion tree (say, k levels) until the problem size reduces to a threshold at which we switch and execute $\mathcal{A}_{MM}$. When we have enough processors and space, the hybrid algorithm $\mathcal{A}_{MM\text{-}H}$ is better than both $\mathcal{A}_{MM}$ and $\mathcal{A}_{MM\text{-}N}$ in parallelism by guaranteeing a complexity of $T_\infty(n) = \mathcal{O}\left(k(2\log n - k) + n/2^k\right)$, $S_p(n) = \mathcal{O}\left(\min\{p^{1/3}, 2^k\}n^2\right)$, where $S_p(n)$ denotes the total space when there are p processors. This tradeoff can be applied to other problems such as multi-instance Viterbi algorithm, Floyd-Warshall's all-pairs shortest path, and protein accordion folding. The complexities of all algorithms are summarized in Table 6.2.

In further sections, we present a framework called AUTOGEN-TRADEOFF that can be used to derive hybrid algorithms that have asymptotically higher parallelism than that for standard divide-and-conquer matrix algorithms.

Algorithm	Work (T_1)	Span (T_∞)	Space (S_p)
MM in-place	$\Theta\left(n^3\right)$	$\Theta\left(n\right)$	$\mathcal{O}\left(n^2\right)$
MM not-in-place	$\Theta\left(n^3\right)$	$\Theta\left(\log^2 n\right)$	$\mathcal{O}\left(p^{1/3}n^2\right)$
MM hybrid	$\Theta\left(n^3\right)$	$\Theta\left(k(2\log n - k) + \frac{n}{2^k}\right)$	$\mathcal{O}\left(\min\{p^{1/3}, 2^k\}n^2\right)$
VA in-place	$\Theta\left(n^3 t\right)$	$\Theta\left(tn\right)$	$\mathcal{O}\left(n^2\right)$
VA not-in-place	$\Theta\left(n^3 t\right)$	$\Theta\left(t\log^2 n\right)$	$\mathcal{O}\left(p^{1/3}n^2\right)$
VA hybrid	$\Theta\left(n^3 t\right)$	$\Theta\left(t\left(k(2\log n - k) + \frac{n}{2^k}\right)\right)$	$\mathcal{O}\left(\min\{p^{1/3}, 2^k\}n^2\right)$

Table 6.2: Work (T_1), serial cache complexity (Q_1), and span (T_∞) of in-place, not-in-place, and hybrid algorithms for matrix problems. and parallelism (T_1/T_∞) of $\mathcal{I}\text{-}\mathcal{DP}$ and $\mathcal{R}\text{-}\mathcal{DP}$ algorithms for several DP problems. Here, n = problem size, t = #timesteps, and p = #cores. We assume that the DP table is too large to fit into the cache. On p cores, the running time is $T_p = \mathcal{O}\left(T_1/p + T_\infty\right)$ with high probability when run under the randomized work-stealing scheduler.

Our contributions. The major contributions of this chapter are:

1. [*Algorithmic.*] We present a generic framework called AUTOGEN-TRADEOFF to develop recursive divide-and-conquer matrix algorithms that often have a tradeoff between space and parallelism. We derive these hybrid algorithms from standard cache-oblivious divide-and-conquer DP algorithms. We present four hybrid algorithms for four problems: matrix multiplication, Floyd-Warshall's all-pairs shortest path, and protein accordion folding that exploit the tradeoff. See Table 6.2.
2. [*Experimental.*] We present empirical results to show that the hybrid algorithms indeed increase parallelism with very small degradation in cache performance.

Organization of the chapter. Section 6.2 presents the space-parallelism tradeoff framework AUTOGEN-TRADEOFF. The empirical evaluations for the algorithms are presented in Section 6.3.

6.2 The AUTOGEN-TRADEOFF framework

In this section, we describe the tradeoff between space and parallelism for cache-oblivious recursive parallel algorithms considering matrix multiplication as an example. A similar technique can be applied to other problems to asymptotically increase parallelism using asymptotically more space.

Example. Consider matrix multiplication (MM). MM is one of the most fundamental and well-studied problems in mathematics and computer science. Due to its extensive use in scientific computing, several algorithms have been proposed to solve the problem in parallel and distributed systems. In this paper, we focus on the square MM problem with cubic computations.

The classic square MM problem is defined as follows. Given two $n \times n$ matrices A and B containing real numbers and stored in row- and column-major orders, respectively, the matrix product of A and B, denoted by C, is computed through $C[i,j] = \sum_1^n A[i,k] \times B[k,j]$. The naive cache-inefficient iterative algorithm is in-place has a span of $\Theta(n)$ or a parallelism of $\Theta(n^2)$. The tiled cache-efficient cache-aware algorithm with tile size $\sqrt{M}$ has $\Theta(n)$ span or $\Theta(n^2)$ parallelism.

Assumptions. We make the following assumptions:
   ⁂ The cache is tall i.e., $M = \Omega(B^2)$.
   ⁂ Data layout is row-major order.
   ⁂ Task scheduler is randomized work stealing.

Parallel cache complexity. If we use a randomized work stealing scheduler, the parallel cache complexity of a recursive divide-and-conquer algorithm on a p-processor machine can be found using the formula $Q_p(n) = \mathcal{O}(Q_1(n) + p(M/B)T_\infty(n))$ w.h.p. in the problem parameter. The maximum realistic value p can take is same as the parallelism i.e., $T_1(n) \div T_p(n)$. When we substitute the maximum value of p in Q_p equation, we get $Q_\infty(n) = \mathcal{O}(Q_1(n) + (T_1(n)/T_\infty(n))(M/B)T_\infty(n)) = \mathcal{O}(Q_1(n) + T_1(n)M/B) = \mathcal{O}(T_1(n)M/B) > Q_1(n)$. Hence, $Q_\infty(n)$ will always be greater than $Q_1(n)$ and is bounded by $\mathcal{O}(T_1(n)M/B)$.

Framework. The four main steps of AUTOGEN-TRADEOFF are:
   1. [*In-place algorithm construction.*] An in-place divide-and-conquer algorithm is generated using AUTOGEN for a given problem. See Section 6.2.1.
   2. [*Span analysis.*] If the span recurrence of the in-place algorithm satisfies a particular property then a hybrid algorthm can be found. See Section 6.2.2.
   3. [*Not-in-place algorithm construction.*] A not-in-place algorithm is derived from the in-place algorithm using extra space. See Section 6.2.3.
   4. [*Hybrid algorithm construction.*] A hybrid algorithm with asymptotic increase in parallelism is constructed with space usage somewhere between in-place and not-in-place algorithms. See Section 6.2.4.

6.2.1 In-place algorithm construction

In this step, we construct an in-place divide-and-conquer algorithm to a given matrix problem.

We can generate in-place 2-way recursive divide-and-conquer algorithms to a wide class

of dynamic programs and matrix problems, including matrix multiplication, using AUTO-GEN. Please refer to Chapter 2 for more details.

Consider the matrix multiplication problem. The in-place cache-efficient cache-oblivious recursive parallel algorithm for square matrices was developed in [Blumofe et al., 1996a] and then extended to rectangular matrices in [Prokop, 1999]. From hereon, we consider only square matrices unless explicitly mentioned. The in-place algorithm is shown in Figure 6.3, function $\mathcal{A}_{MM}$. The term ℓ in $\mathcal{A}_{MM}$ represents the plane number. This means that the algorithm updates the ℓth plane through the base case looping kernels $(\mathcal{A}_{loop-MM})$. The value of ℓ can be set to 0 when we are using the in-place algorithm independently. The usage of term ℓ becomes clear when we analyze the hybrid algorithm. Lemma 2 gives the complexity analysis of the in-place MM algorithm.

> **Lemma 2 (In-place MM).** *The in-place MM algorithm $\mathcal{A}_{MM}$, assuming a tall cache, has a complexity of $T_1(n) = \Theta(n^3)$, $Q_1(n) = \Theta\left(n^3/(B\sqrt{M}) + n^2/B + n\right)$, $T_\infty(n) = \Theta(n)$, $S_\infty(n) = \Theta(n^2)$, and $E_p(n) = \Theta(p\log n)$.*

Proof. The recurrences for the complexities are as follows:

$$T_1(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T_1(\frac{n}{2}) + \Theta(1) & \text{if } n > 1. \end{cases} \qquad Q_1(n) = \begin{cases} \Theta\left(\frac{n^2}{B} + n\right) & \text{if } n^2 \leq \alpha M, \\ 8Q_1(\frac{n}{2}) + \Theta(1) & \text{if } n^2 > \alpha M. \end{cases}$$

$$T_\infty(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_\infty(\frac{n}{2}) + \Theta(1) & \text{if } n > 1. \end{cases} \qquad S_\infty(n) = \begin{cases} \mathcal{O}(1) & \text{if } n = 1, \\ 4S_\infty(\frac{n}{2}) + \Theta(1) & \text{if } n > 1. \end{cases}$$

Also, for p processors, there can be at most p tasks running in parallel. When the tasks reach the basecase, they execute $\mathcal{A}_{loop-MM}$. As each executing task is a leaf node in the recursion tree, we might require a total extra space all along the path from the root to the p executing leaves of the recursion tree. Hence, $E_p(n) = \Theta(p\log n)$. Solving the recurrences and from $E_p(n)$ argument, we have the lemma. □

6.2.2 Span analysis

In this section, we give a theorem that relates to the span of the in-place algorithm. If the in-place algorithm satisfies a particular property as given in the span theorem (Theorem 12), then a hybrid algorithm can be designed to asymptotically increase the parallelism.

> **Theorem 12 (Span analysis).** *Let $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m$ be the m functions of the autogenerated in-place $\mathcal{R}$-$\mathcal{DP}$. Let the span of the functions be computed as follows:*
>
> $$T_{\mathcal{F}_i}(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{j=i}^{m} a_{ij} T_{\mathcal{F}_j}(n/2) + \Theta(1) & \text{otherwise;} \end{cases} \qquad (6.1)$$
>
> *where, $T_{\mathcal{F}_i}(n)$ is the span of $\mathcal{F}_i(n)$ and a_{ij} is the number of parallel steps in which the function $\mathcal{F}_i(n)$ calls $\mathcal{F}_j(n)$. Suppose the function $\mathcal{F}_m$ be like matrix-multiplication reading from submatrices it does not write to and also that $\mathcal{F}_m$ is dominating (see Definition 16). Also, let $maximum = \max\{a_{11}, a_{22}, \ldots, a_{mm}\}$.*
>
> *If $maximum = 2$, then there exist not-in-place and hybrid algorithms to asymptotically increase parallelism.*

Proof. Let $P = \mathcal{F}_{r_1}, \mathcal{F}_{r_2}, \ldots, \mathcal{F}_{|P|}$ be a path in the recursion tree of the root $(= \mathcal{F}_{r_1} = \mathcal{F}_1)$ to a node corresponding to $\mathcal{F}_m(= \mathcal{F}_{|P|})$. Let q out of these $|P|$ functions call themselves in $(maximum = 2)$ parallel steps and q is maximized over all possible paths in the recursion tree. Then the span of the in-place algorithm is

$$T_{\mathcal{F}_1}(n) = \Theta\left(n \log^{q-1} n\right) \qquad \text{(in-place algorithm)} \tag{6.2}$$

which is found by recursively applying the master theorem from the leaf of the recursion tree to the root.

We know that $\mathcal{F}_m$ is matrix multiplication algorithm (or any matrix-multiplication-like kernel) and there is a not-in-place algorithm for it whose span recurrence is

$$T_{\mathcal{F}_m}(n) = T_{\mathcal{F}_m}\left(\frac{n}{2}\right) + \Theta(1) \qquad \text{if } n > 1. \tag{6.3}$$

It is important to note that replacing the in-place MM-like algorithm $\mathcal{F}_m$ with a not-in-place algorithm, the value of q will be decremented by 1. This gives rise to the not-in-place algorithm with span

$$T_{\mathcal{F}_1}(n) = \begin{cases} \Theta(\log n) & \text{if } q = 1, \\ \Theta\left(n \log^{q-1} n\right) & \text{if } q > 1. \end{cases} \tag{6.4}$$

This implies that there is a not-in-place algorithm with asymptotic decrease in span (or asymptotic increase in parallelism). Hence, there is a hybrid algorithm with asymptotic increase in parallelism. $\qquad \square$

Consider the MM example. We see that the in-place MM has only one function and it calls itself in two parallel steps as shown in the proof of Lemma 2. It satisfies Theorem 1 and hence a not-in-place and a hybrid algorithm exist for MM.

6.2.3 Not-in-place algorithm construction

In this step, we show how to construct a not-in-place algorithm via the MM example.

For DP or DP-like divide-and-conquer algorithms, we can easily construct the not-in-place algorithms by simply replacing the in-place MM-like functions with the not-in-place versions of them.

Consider Figure 6.3. The $\mathcal{A}_{MM\text{-}N}$ algorithm is the not-in-place algorithm. Instead of computing the 8 matrix products in 2 parallel steps, we compute the 8 matrix products in 1 parallel step. This requires us to compute the products in a different matrix and once we have the 8 matrix products in 2 different tables, we combine them to form the original solution. This requires extra space that can be either dynamically allocated during execution or statically allocated at before the initial call to the algorithms. Dynamic memory allocation inside the MM function calls is not recommended because of memory contention. Lemma 3 gives the complexity analysis for the not-in-place MM.

> **Lemma 3 (Not-in-place MM).** *The not-in-place MM algorithm $\mathcal{A}_{MM\text{-}N}$, assuming a tall cache, has a complexity of $T_1(n) = \Theta(n^3)$, $Q_1(n) = \Theta\left(n^3/(B\sqrt{M}) + n^2/B + n\right)$, $T_\infty(n) = \Theta\left(\log^2 n\right)$, $S_p(n) = \Theta\left(n^2 p^{1/3}\right)$.*

$\mathcal{A}_{MM}(X, U, V, \ell)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}MM}(X, U, V, \ell)$
2. **else**
3.   **par:** $\mathcal{A}_{MM}(X_{11}, U_{11}, V_{11}, \ell)$, $\mathcal{A}_{MM}(X_{12}, U_{11}, V_{12}, \ell)$,
      $\mathcal{A}_{MM}(X_{21}, U_{21}, V_{11}, \ell)$, $\mathcal{A}_{MM}(X_{22}, U_{21}, V_{12}, \ell)$
4.   **par:** $\mathcal{A}_{MM}(X_{11}, U_{12}, V_{21}, \ell)$, $\mathcal{A}_{MM}(X_{12}, U_{12}, V_{22}, \ell)$,
      $\mathcal{A}_{MM}(X_{21}, U_{22}, V_{21}, \ell)$, $\mathcal{A}_{MM}(X_{22}, U_{22}, V_{22}, \ell)$

$\mathcal{A}_{MM\text{-}N}(X, U, V)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}MM\text{-}N}(X, U, V)$
2. **else**
3.   **par:** $\mathcal{A}_{MM\text{-}N}(X_{11}, U_{11}, V_{11})$, $\mathcal{A}_{MM\text{-}N}(X_{12}, U_{11}, V_{12})$,
      $\mathcal{A}_{MM\text{-}N}(X_{21}, U_{21}, V_{11})$, $\mathcal{A}_{MM\text{-}N}(X_{22}, U_{21}, V_{12})$,
      $\mathcal{A}_{MM\text{-}N}(Y_{11}, U_{12}, V_{21})$, $\mathcal{A}_{MM\text{-}N}(Y_{12}, U_{12}, V_{22})$,
      $\mathcal{A}_{MM\text{-}N}(Y_{21}, U_{22}, V_{21})$, $\mathcal{A}_{MM\text{-}N}(Y_{22}, U_{22}, V_{22})$
4. $X \leftarrow X + Y$

$\mathcal{A}_{MM\text{-}H}(X, U, V, s, m)$

1. **if** $m = 1$ **or** X is a small matrix **then** $\mathcal{A}_{MM}(X, U, V, s)$
2. **else**
3.   **par:** $\mathcal{A}_{MM\text{-}H}(X_{11}, U_{11}, V_{11}, s, m/2)$,
      $\mathcal{A}_{MM\text{-}H}(X_{12}, U_{11}, V_{12}, s, m/2)$,
      $\mathcal{A}_{MM\text{-}H}(X_{21}, U_{21}, V_{11}, s, m/2)$,
      $\mathcal{A}_{MM\text{-}H}(X_{22}, U_{21}, V_{12}, s, m/2)$,
      $\mathcal{A}_{MM\text{-}H}(X_{11}, U_{12}, V_{21}, s + m/2, m/2)$,
      $\mathcal{A}_{MM\text{-}H}(X_{12}, U_{12}, V_{22}, s + m/2, m/2)$,
      $\mathcal{A}_{MM\text{-}H}(X_{21}, U_{22}, V_{21}, s + m/2, m/2)$,
      $\mathcal{A}_{MM\text{-}H}(X_{22}, U_{22}, V_{22}, s + m/2, m/2)$

Figure 6.3: Cache-oblivious recursive divide-and-conquer parallel matrix multiplication algorithms. The terms ℓ, s, and m denote the plane number, starting plane number, and total number of planes that can be used, respectively.

Proof. The recurrences for the complexities are as follows:

$$T_1(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T_1(\frac{n}{2}) + \Theta(1) & \text{if } n > 1. \end{cases} \qquad Q_1(n) = \begin{cases} \Theta\left(\frac{n^2}{B} + n\right) & \text{if } n^2 \leq \alpha M, \\ 8Q_1(\frac{n}{2}) + \Theta(1) & \text{if } n^2 > \alpha M. \end{cases}$$

$$T_\infty(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T_\infty(\frac{n}{2}) + \mathcal{O}(\log n) & \text{if } n > 1. \end{cases} \qquad S_p(m) = \begin{cases} \mathcal{O}(m^2) & \text{if } \left(\frac{n}{m}\right)^3 \leq p, \\ 8S_p(\frac{m}{2}) + \Theta(m^2) & \text{if } \left(\frac{n}{m}\right)^3 > p. \end{cases}$$

Solving the recurrences, we have the lemma. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

6.2.4 Hybrid algorithm construction

In this section, we show how to construct a hybrid algorithm for a given in-place divide-and-conquer algorithm if it satisfies the span property (Theorem 12).

The $\mathcal{A}_{MM}$ algorithm uses least extra space from Lemma 2 and the $\mathcal{A}_{MM\text{-}N}$ algorithm has very high parallelism from Lemma 3. Is it possible to develop a hybrid algorithm that uses less space and has highest parallelism? We take a step in this direction to develop a hybrid algorithm combining both $\mathcal{A}_{MM}$ and $\mathcal{A}_{MM\text{-}N}$ that uses relatively lesser space than that of $\mathcal{A}_{MM\text{-}N}$ and has asymptotically more parallelism than that of $\mathcal{A}_{MM}$.

There are fundamentally two different ways to combine $\mathcal{A}_{MM}$ and $\mathcal{A}_{MM\text{-}N}$:

(i) [$\mathcal{A}_{MM}$ *on* $\mathcal{A}_{MM\text{-}N}$.] Execute $\mathcal{A}_{MM}$ until the size of the input submatrix drops to $n/2^k$ or below and then execute $\mathcal{A}_{MM\text{-}N}$.

(ii) [$\mathcal{A}_{MM\text{-}N}$ *on* $\mathcal{A}_{MM}$.] Execute $\mathcal{A}_{MM\text{-}N}$ until the size of the input submatrix drops to $n/2^k$ or below and then execute $\mathcal{A}_{MM}$.

The hybrid algorithm in case (ii) is interesting because it offers a tradeoff between space and parallelism. From hereon, we term this hybrid algorithm $\mathcal{A}_{MM\text{-}H}$, where H refers to hybrid. The increase in parallelism comes with increase in the extra space used. We call this tradeoff the *space-parallelism tradeoff*.

The hybrid MM algorithm $\mathcal{A}_{MM\text{-}H}$ is shown in Figures 6.4 and 6.3. The terms ℓ, s,
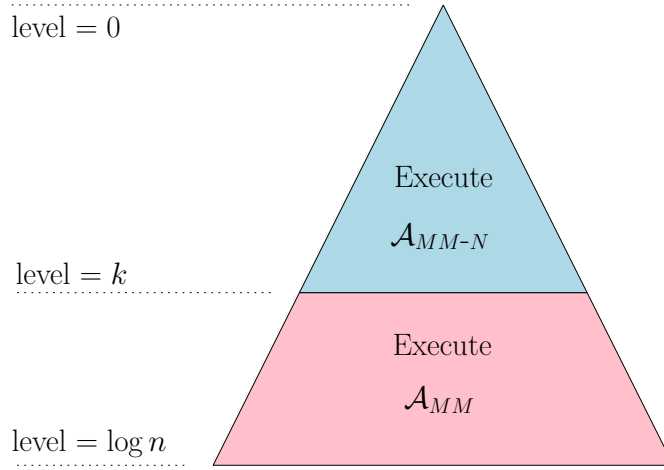
Figure 6.4: Hybrid matrix multiplication algorithm.

and m in $\mathcal{A}_{MM\text{-}H}$ denote the starting plane number and total number of planes that can be used, respectively. The way the algorithm works is that we execute the not-in-place algorithm (i.e., $\mathcal{A}_{MM\text{-}N}$) until the problem size or the task size reduces to a threshold value, say $(n/2^k) \times (n/2^k)$ for some k, and then we execute the in-place algorithm (i.e., $\mathcal{A}_{MM}$).

In $\mathcal{A}_{MM\text{-}H}$, the more number of levels we execute $\mathcal{A}_{MM\text{-}N}$ initially, the more space we consume, the more parallelism we have. Therefore, we need to carefully choose the level k in which we need to shift from one algorithm to another. The complexity of the $\mathcal{A}_{MM\text{-}H}$ algorithm is given in Theorem 13.

> **Theorem 13 (Hybrid MM).** *The hybrid MM algorithm $\mathcal{A}_{MM\text{-}H}$, assuming a tall cache, has a complexity of $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(n^3/(B\sqrt{M}) + n^2/B + n\right)$, $T_\infty(n) = \mathcal{O}\left(k(2\log n - k) + n/2^k\right)$, and $S_p(n) = \mathcal{O}\left(\min\{p^{1/3}, 2^k\}n^2\right)$.*

Proof. In the hybrid MM algorithm $\mathcal{A}_{MM\text{-}H}$, when the size of each input matrix drops to $n/2^k$ or below then we execute $\mathcal{A}_{MM}$. Hence, in the following recurrences, when the problem size reaches the base case (i.e., when $m = n/2^k$), the complexities are same as the complexities from Lemma 2 but n is replaced by $n/2^k$. Also, the recursion cases are from Lemma 3. Thus, we have:

$$T_1(m) = \begin{cases} \Theta\left(\left(\frac{n}{2^k}\right)^3\right) & \text{if } m = \frac{n}{2^k}, \\ 8T_1(\frac{m}{2}) + \Theta(1) & \text{if } m > \frac{n}{2^k}. \end{cases} \qquad Q_1(m) = \begin{cases} \Theta\left(\frac{n^3}{8^k B \sqrt{M}} + \frac{n^2}{4^k B} + \frac{n}{2^k}\right) & \text{if } m = \frac{n}{2^k}, \\ 8Q_1(\frac{m}{2}) + \Theta(1) & \text{if } m > \frac{n}{2^k}. \end{cases}$$

$$T_\infty(m) = \begin{cases} \Theta\left(\frac{n}{2^k}\right) & \text{if } m = \frac{n}{2^k}, \\ T_\infty(\frac{m}{2}) + \mathcal{O}(\log m) & \text{if } m > \frac{n}{2^k}. \end{cases} \quad S_p(m) = \begin{cases} \mathcal{O}(m^2) & \text{if } m = \frac{n}{2^k} \text{ or } \left(\frac{n}{m}\right)^3 \le p, \\ 8S(\frac{m}{2}) + \Theta(m^2) & \text{if } m > \frac{n}{2^k} \text{ and } \left(\frac{n}{m}\right)^3 > p. \end{cases}$$

Solving the recurrences and from $S_p(n)$ argument, we have the theorem. □

Theorem 13 may or may not be cache-efficient depending on the value of k.

6.3 Experimental results

In this section, we present empirical results[1] showing the increased parallelism of our hybrid algorithms.

Experimental setup. All our experiments were performed on a multicore machine with dual-socket 8-core 2.7 GHz Intel Sandy Bridge processors ($2 \times 8 = 16$ cores in total) and 32 GB RAM. Each core was connected to a 32 KB private L1 cache and a 256 KB private L2 cache. All cores in a processor shared a 20 MB L3 cache. All algorithms were implemented in C++. We used Intel Cilk Plus extension to parallelize and Intel$^{\delta}$ C++ Compiler v13.0 to compile all implementations with optimzation parameters `-O3 -ipo -parallel -AVX -xhost`. PAPI 5.3 [PAP,] was used to count cache misses.

Implementations. We implement algorithms for the matrix multiplication (MM) problem. For all programs, the base case size B was set to 32. For MM, the DP table dimension n varied from $2^6 = 64$ to $2^{12} = 2048$.

For each problem, we compared five algorithms: in-place algorithm `I`, not-in-place algorithm `N`, and three hybrid algorithms `H(2)`, `H(4)`, `H(8)`. The hybrid algorithm `H(m)` means that the algorithm uses a total of m planes each of size $n \times n$. In other words, $m - 1$ extra planes were used along with the input DP table. We can think of the in-place algorithm as `H(1)` and not-in-place algorithm as `H(n)`.

Matrix multiplication. The plots of MM are shown in Figure 6.5. The parallelism of `I` is $\Theta\left(n^2\right)$ and that of N is $\Theta\left(n^3/\log^2 n\right)$. Hence, the difference between the values of `I` and `N` increases significantly when n increases. The more number of planes we use in the hybrid algorithm, the more parallelism we can exploit. The cache misses incurred by a hybrid algorithm is almost the same as that of the in-place algorithm.

6.4 Conclusion and open problems

We presented a framework called AUTOGEN-TRADEOFF to semi-automatically design recursive divide-and-conquer algorithms with very high parallelism using extra space. The framework can be useful when we have enough number of processors and enough space.

A few open problems are:

⋆ [*Improve span analysis*.] Improve the framework such that given a DP recurrence, find whether the problem can have an algorithm with parallelism better than that of the standard 2-way $\mathcal{R}$-$\mathcal{DP}$ or not. If we can indeed get better parallelism, derive the faster algorithm.

⋆ [*Tight lower bounds for optimal parallelism*.] The *optimal parallelism* of an algorithm can be different depending on the constraints. Identify different constraints for an algorithm (e.g.: with and without using exra space), and prove lower bounds for optimal parallelism.

⋆ [*Compute optimal parallelism*.] For any given DP recurrence, compute the optimal parallelism achievable by an algorithm (irrespective of all constraints) that can be used to solve a DP recurrence.

---

[1]The matrix multiplication algorithm was implemented by Mohammad Mahdi Javanmard.
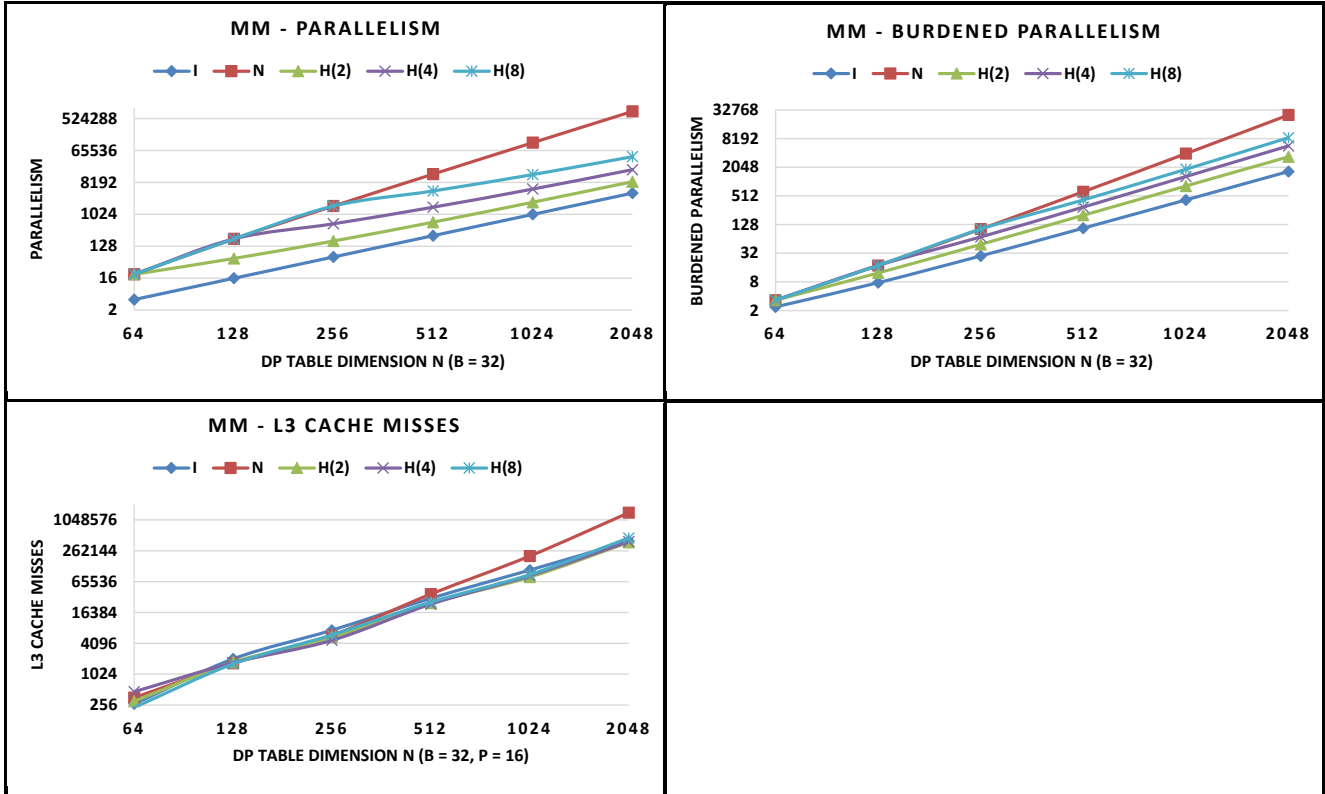
Figure 6.5: Performance comparison of in-place algorithm - I, not-in-place algorithm - N, hybrid algorithms with k planes H(k) where $k = 2, 4, 8$: (a) parallelism, (b) burdened parallelism, and (c) L3 cache misses; when DP table dimension increases and base case size B is 32.

Appendix A

Efficient Divide-&-Conquer DP Algorithms

In this section, we present several divide-and-conquer DP algorithms. By the virtue of divide-and-conquer (see Section 1.7), these algorithms are cache-efficient, cache-oblivious, processor-oblivious, and parallel. Standard 2-way divide-and-conquer algorithms for longest common subsequence, parenthesis problem, sequence alignment with gap penalty (often called the gap problem), and Floyd Warshall's all-pairs shortest path already exist and we include them here for completeness and we analyze them in more depth. The rest of the algorithms are new.

We also discuss the divide-and-conquer variants of elementary sorting algorithms. Though such algorithms are not DP algorithms, the core approach of designing those algorithms comes from AUTOGEN.

A.1 Longest common subsequence & edit distance

A sequence $Z = \langle z_1, z_2, \ldots z_k \rangle$ is called a subsequence of another sequence $X = \langle x_1, x_2, \ldots x_m \rangle$ if there exists a strictly increasing function $f : [1, 2, \ldots, k] \rightarrow [1, 2, \ldots, m]$ such that for all $i \in [1, k]$, $z_i = x_{f(i)}$. A sequence Z is a common subsequence of sequences X and Y if Z is a subsequence of both X and Y. In the longest common subsequence (LCS) problem, we are given two sequences X and Y, and we need to find a maximum-length common subsequence of X and Y. The LCS problem arises in a wide variety of applications, and it is especially important in computational biology in sequence alignment.

Given two sequences $X = \langle x_1, x_2, \ldots x_m \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$, we define $C[i, j]$ ($0 \leq i \leq m, 0 \leq j \leq n$) to be the length of an LCS of $\langle x_1, x_2, \ldots x_i \rangle$ and $\langle y_1, y_2, \ldots y_j \rangle$. Then $C[m, n]$ is the length of an LCS of X and Y, and can be computed using the following recurrence relation (see, e.g., [Cormen et al., 2009]):

$$C[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ C[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{C[i, j-1], C[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \quad \text{(A.1)}$$

The classic dynamic programming solution to the LCS problem is based on this recurrence relation, and computes the entries of $C[0 \ldots m, 0 \ldots n]$ in row-major order in $\Theta(mn)$ time and space.
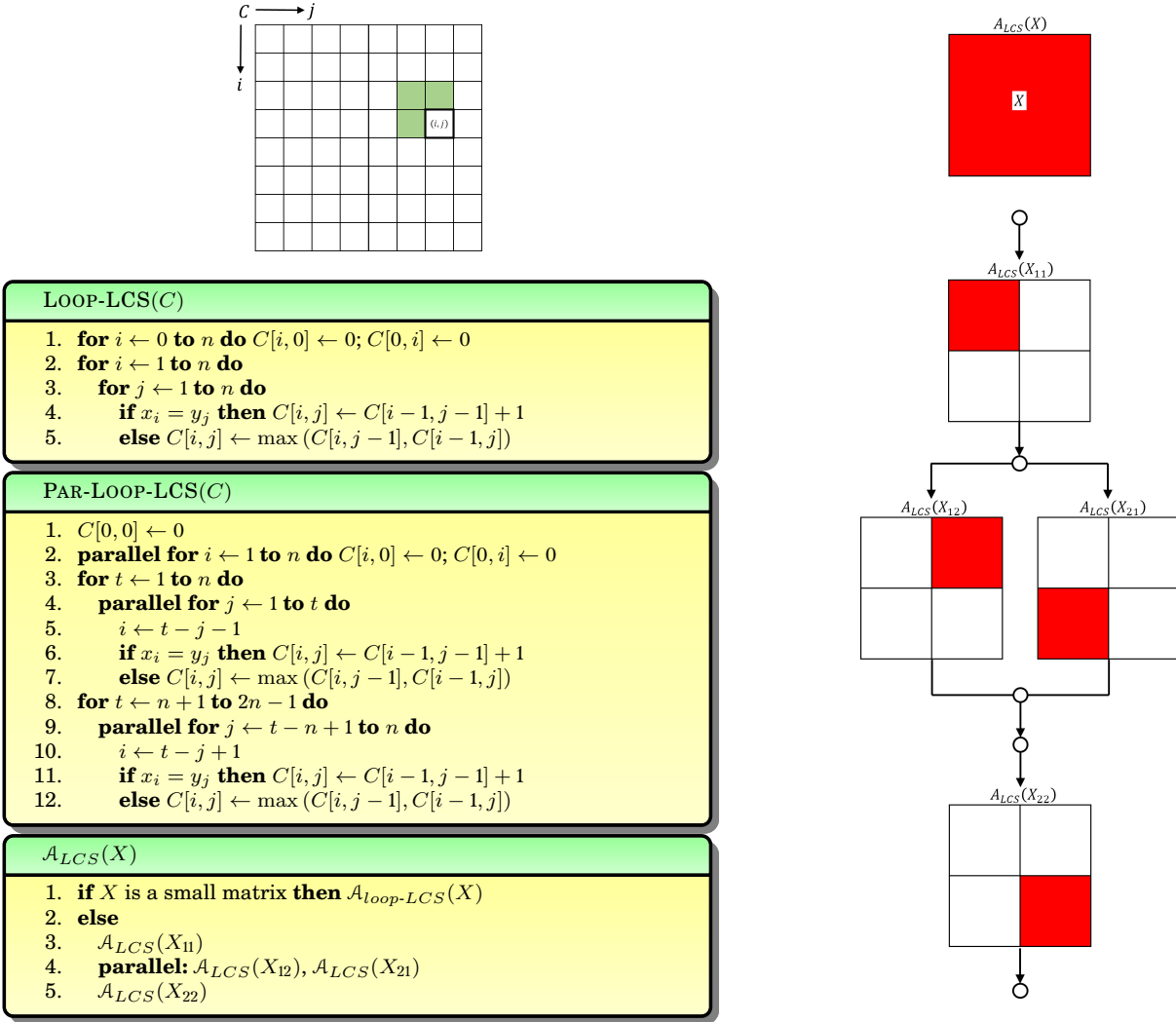
Figure A.1: The dependency graph, serial iterative algorithm, parallel iterative algorithm, and the divide-and-conquer algorithm for the LCS problem. Initial call to the algorithm is $\mathcal{A}_{LCS}(C)$, where C is the full DP table.

Please refer Figure A.1. It shows the dependency graph for the problem. A cell at position (i, j) depends on three light green cells as per the recurrence. The figure contains several algorithms for computing the LCS lengths. A simple iterative algorithm is given as LOOP-LCS. A highly parallel iterative algorithm is given as PAR-LOOP-LCS.

The divide-and-conquer algorithm $\mathcal{A}_{LCS}$ was presented in [Chowdhury and Ramachandran, 2006, Chowdhury and Ramachandran, 2008]. A picture representation of algorithm is shown in the right of Figure A.1. The algorithm splits the entire DP X table into four quadrants: X_{11} (top-left), X_{12} (top-right), X_{21} (bottom-left), and X_{22} (bottom-right). Computing the LCS lengths in each of these quadrants are the independent subproblems of our original problem (computing the entire DP table). Hence, we solve them recursively. First, we fill X_{11}. Then we fill quadrants X_{12} and X_{21} in parallel. Finally, we fill X_{22}. Note that the quadrants are filled recursively. More details on the algorithm can be found in the corresponding paper(s).

The problem of converting one string to another string using three operations: inserts,

deletes, and substitutions, and minimizing the overall cost is called the *edit distance problem* [Leiserson, 2010]. The applications of the edit distance problem include automatic spelling correction and the longest common subsequence. The LCS problem is in fact a special case of the edit distance problem. A quadratic-time algorithm to find the edit distance between two strings is given in [Wagner and Fischer, 1974]. A sub-quadratic time algorithm for the problem was first presented in [Masek and Paterson, 1980]. The divide-and-conquer algorithm presented for the LCS problem also works and retains its optimal serial cache complexity for the edit distance problem provided cost for inserts, deletes, and substitutions are fixed constants. If the costs for the three operations are extremely generic, then no cache-efficient divide-and-conquer algorithm exists for the problem.

Linear space. Note that if we use $\Theta(n^2)$ space for the DP table, as the total work (total number of computations) is also $\Theta(n^2)$, we cannot get temporal locality and the serial cache complexity will be $\Theta(n^2/B)$. To exploit temporal locality, the total work complexity must be asymptotically greater than the total space complexity. We can exploit temporal locality if we asymptotically reduce the space. As described in the papers [Chowdhury and Ramachandran, 2006, Chowdhury and Ramachandran, 2008], we store only the input boundaries for different parallel tasks (or quadrants of the DP table) that are to be executed. With this idea, the space can be reduced to $\Theta(n \log n)$. To reduce the space further to linear space i.e., $\Theta(n)$ space, we must reuse the input / output boundaries of the quadrants across different levels of the recursion tree.

Complexity analysis for the PAR-LOOP-LCS algorithm
The parallel iterative algorithm makes use of a 2-D matrix C for simplicity. The algorithm can be modified easily to make use of a 1-D array of size $\Theta(n)$.

For $f \in \{\text{PAR-LOOP-LCS}\}$, let $W_f(n), Q_f(n), T_f(n)$, and $S_f(n)$ denote the total work, serial cache complexity, span, and parallel space consumption of f_{LCS} for the parameter n. Then $W_f(n) = \Theta(n^2)$. $Q_f(n)$ is same as scanning n anti-diagonals n times i.e., $Q_f(n) = \Theta\left(n\left(\frac{n}{B}+1\right)\right) = \Theta\left(\frac{n^2}{B}+n\right)$. Each parallel for loop takes $\Theta(\log n)$ time to divide the loop to processors and there are $\Theta(n)$ such parallel for loops. Hence, $T_f(n) = \Theta(n \log n)$. Space $S_f(n)$ is simply $\Theta(n)$.

> For the edit distance and LCS problems, the PAR-LOOP-LCS algorithm achieves $T_1(n) = \Theta(n^2)$, $Q_1(n) = \Theta\left(\frac{n^2}{B}+n\right)$, $T_\infty(n) = \Theta(n \log n)$, **parallelism** $= \Theta\left(\frac{n}{\log n}\right)$, and $S_\infty(n) = \Theta(n)$.

Complexity analysis for the 2-way $\mathcal{A}_{LCS}$ algorithm
For $f \in \{\mathcal{A}\}$, let $W_f(n), Q_f(n), T_f(n)$, and $S_f(n)$ denote the total work, serial cache complexity, span, and parallel space consumption of f_{LCS} for the parameter n. Then

$$W_\mathcal{A}(n) = \begin{cases} \mathcal{O}(1) & \text{if } n = 1, \\ 4W_\mathcal{A}\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases} \qquad Q_\mathcal{A}(n) = \begin{cases} \mathcal{O}\left(\frac{n}{B}+1\right) & \text{if } n \leq \gamma_\mathcal{A} M, \\ 4Q_\mathcal{A}\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

$$T_\mathcal{A}(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 3T_\mathcal{A}\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise;} \end{cases} \qquad S_\mathcal{A}(n) = \begin{cases} \mathcal{O}(n) & \text{if } n \leq \gamma_\mathcal{A} M, \\ 2S_\mathcal{A}\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise;} \end{cases}$$

where, γ_A is a suitable constant. Solving, $W_A(n) = \Theta(n^2)$, $Q_A(n) = \Theta\left(\frac{n^2}{BM} + \frac{n^2}{M^2} + \frac{n}{B} + 1\right)$, $T_A(n) = \Theta\left(n^{\log 3}\right)$, and $S_A(n) = \Theta(n \log n)$.

The space complexity is $\Theta(n \log n)$ because there are $\Theta(\log n)$ levels in the recursion tree and at every level the input boundaries take $\Theta(n)$ space. In fact, we can reduce the space to $\Theta(n)$ if at every level we reuse the input boundaries of the parent function call.

> For the edit distance and LCS problems, the 2-way divide-and-conquer algorithm achieves $T_1(n) = \Theta(n^2)$, $Q_1(n) = \Theta\left(\frac{n^2}{BM} + \frac{n^2}{M^2} + \frac{n}{B} + 1\right)$, $T_\infty(n) = \Theta\left(n^{\log 3}\right)$, parallelism $= \Theta\left(n^{2-\log 3}\right)$, and $S_\infty(n) = \Theta(n)$.

Complexity analysis for the r-way $\mathcal{A}_{LCS}$ algorithm

In an r-way divide-and-conquer algorithm, we divide the entire $n \times n$ matrix into $r \times r$ submatrices each of size $(n/r) \times (n/r)$ and solve them recursively. In Figure A.1, for an r-way divide-and-conquer we will have r^2 function calls inside the $\mathcal{A}_{LCS}$ function. For simplicity of exposition we assume that n is a power of r. *When r increases, the number of cache misses increases, the span decreases, and it gets more difficult to implement the program.*

For $f \in \{\mathcal{A}\}$, let $W_f(n), Q_f(n), T_f(n)$, and $S_f(n)$ denote the total work, serial cache complexity, span, and parallel space consumption of r-way f_{LCS} for the parameter n. Then

$$W_A(n) = \begin{cases} \mathcal{O}(1) & \text{if } n = 1, \\ r^2 W_A\left(\frac{n}{r}\right) + \Theta(1) & \text{otherwise;} \end{cases} \qquad Q_A(n) = \begin{cases} \mathcal{O}\left(\frac{n}{B} + 1\right) & \text{if } n \le \gamma_A M, \\ r^2 Q_A\left(\frac{n}{r}\right) + \Theta(1) & \text{otherwise;} \end{cases}$$

$$T_A(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ (2r - 1) T_A\left(\frac{n}{r}\right) + \Theta(1) & \text{otherwise;} \end{cases} \qquad S_A(n) = \begin{cases} \mathcal{O}(n) & \text{if } n \le \gamma_A M, \\ r S_A\left(\frac{n}{r}\right) + \Theta(n) & \text{otherwise;} \end{cases}$$

where, γ_A is a suitable constant. Solving, $W_A(n) = \Theta(n^2)$, $Q_A(n) = \mathcal{O}\left(\frac{n^2}{B\lceil M/r \rceil} + \frac{n^2}{\lceil M/r \rceil^2} + \frac{n}{B} + 1\right)$, $T_A(n) = \mathcal{O}\left(n^{\log_r(2r-1)}\right)$, and $S_A(n) = \Theta(n \log_r n)$.

The serial cache complexity is found as below

$$Q_A(n) \le r^2 Q_A\left(\frac{n}{r}\right) + c = r^2\left(r^2 Q_A\left(\frac{n}{r^2}\right) + c\right) + c = (r^2)^2 Q_A\left(\frac{n}{r^2}\right) + cr^2 + c$$

$$= (r^2)^k Q_A\left(\frac{n}{r^k}\right) + c(r^2)^{k-1} + \cdots + c \qquad \left(\text{say } \frac{n}{r^k} \le \gamma_A M\right)$$

$$= c(r^2)^k\left(\frac{n}{r^k B} + 1\right) + c\left((r^2)^{k-1} + \cdots + 1\right) = cr^k \frac{n}{B} + c\left((r^2)^k + \cdots + 1\right)$$

$$= cr^k \frac{n}{B} + c\left(\frac{(r^2)^{k+1} - 1}{r^2 - 1}\right) \le cr^k \frac{n}{B} + c'r^{2k} \qquad \text{(for some constant } c')$$

We would like to find the upper bound for $Q_A(n)$. As $\left(n/r^k\right) \le \gamma_A M$, we have $\left(n/r^{k-1}\right) > \gamma_A M$, or $r^{k-1} < (n/(\gamma_A M))$. Substituting for the upper bound of r^{k-1} in the inequality above

we get

$$Q_A(n) \leq c\frac{n}{B} \cdot r^{k-1} \cdot r + c' \left(r^{k-1}\right)^2 \cdot r^2 \leq c\frac{n}{B} \cdot \frac{n}{\gamma_A M} \cdot r + c' \left(\frac{n}{\gamma_A M}\right)^2 \cdot r^2$$

$$= \mathcal{O}\left(\frac{n^2 r}{BM} + \frac{n^2 r^2}{M^2}\right) = \mathcal{O}\left(\frac{n^2}{B\left\lceil\frac{M}{r}\right\rceil} + \frac{n^2}{\left\lceil\frac{M}{r}\right\rceil^2}\right)$$

When r increases, the temporal cache locality decreases, and hence the number of cache misses increases. The amount of memory from which we can exploit temporal locality is $\lfloor M/r \rfloor$ for the r-way divide-and-conquer LCS algorithm. Hence, the serial cache complexity is a function of n, B, and $\lfloor M/r \rfloor$.

The span can be computed easily

$$T_A(n) \leq (2r-1)T_A\left(\frac{n}{r}\right) + c = (2r-1)\left((2r-1)T_A\left(\frac{n}{r}\right) + c\right) + c$$

$$= (2r-1)^2 T_A\left(\frac{n}{r^2}\right) + c\left((2r-1) + 1\right)$$

$$= (2r-1)^k T_A\left(\frac{n}{r^k}\right) + c\left((2r-1)^{k-1} + \cdots + 1\right) \qquad \left(\text{say } \frac{n}{r^k} = 1\right)$$

$$= (2r-1)^k c + c'(2r-1)^{k-1} = c'(2r-1)^k = c'(2r-1)^{\log_r n} = \mathcal{O}\left(n^{\log_r(2r-1)}\right)$$

The space complexity is $\Theta\left(n\log_r n\right)$ because there are $\Theta\left(\log_r n\right)$ levels in the recursion tree and at every level the input boundaries take $\Theta\left(n\right)$ space. Again, we can reduce the space to $\Theta\left(n\right)$ if at every level we reuse the input boundaries of the parent function call.

> For the edit distance and LCS problems, the r-way divide-and-conquer algorithm achieves $T_1(n) = \Theta\left(n^2\right)$, $Q_A(n) = \mathcal{O}\left(\frac{n^2}{B\lceil M/r \rceil} + \frac{n^2}{\lceil M/r \rceil^2} + \frac{n}{B} + 1\right)$, $T_A(n) = \mathcal{O}\left(n^{\log_r(2r-1)}\right)$, parallelism $= \Theta\left(n^{2-\log_r(2r-1)}\right)$, and $S_\infty(n) = \Theta\left(n\right)$.

A.2 Parenthesis problem

The parenthesis problem [Galil and Park, 1994] is defined by the following recurrence relation:

$$C[i,j] = \begin{cases} x_j & \text{if } 0 \leq i = j-1 < n, \\ \min_{i<k<j}\left\{(C[i,k] + C[k,j]) + w(i,j,k)\right\} & \text{if } 0 \leq i < j-1 < n; \end{cases} \tag{A.2}$$

where x_j's are assumed to be given for $j \in [1,n]$. We also assume that $w(\cdot,\cdot,\cdot)$ is a function that can be computed in-core without additional memory accesses.

The class of problems defined by the recurrence relation above includes optimal chain matrix multiplication, RNA secondary structure prediction, optimal binary search trees, optimal polygon triangulation, string parsing for context-free grammar (CYK algorithm), optimal natural join of database tables (Selinger algorithm), maximum perimeter inscribed

polygon, and offline job scheduling minimizing flow time of jobs. A variant of this recurrence which does not include the $w(i, k, j)$ term and is defined as the simple dynamic program, was considered in [Cherng and Ladner, 2005].

As in [Cherng and Ladner, 2005], instead of recurrence A.2 we will use the following slightly augmented version of A.2 which will considerably simplify the recursive subdivision process in our divide-and-conquer algorithm.

$$C[i, j] = \begin{cases} \infty & \text{if } 0 \leq i = j \leq n, \\ x_j & \text{if } 0 \leq i = j - 1 < n, \\ \min_{i < k \leq j} \left\{ (C[i, k] + C[k, j]) + w(i, j, k) \right\} & \text{if } 0 \leq i < j - 1 < n; \end{cases} \quad \text{(A.3)}$$

where $w(i, j, k)$ is defined to be ∞ when $k = i$ or $k = j$. It is straightforward to see that recurrences A.2 and A.3 are equivalent, i.e., they compute the same values for any given $C[i, j]$, $0 \leq i < j - 1 < n$. In the rest of this section we will assume for simplicity that $n = 2^\ell - 1$ for some integer $\ell \geq 0$.

Please refer Figure A.2. It gives the dependency graph for the parenthesis problem. A cell (i, j) depends on the light green cells. Cell numbered k on the horizontal light green line is paired with cell numbered k on the vertical light green line during the update of cell (i, j). A serial iterative algorithm LOOP-PARENTHESIS and a parallel iterative algorithm PAR-LOOP-PARENTHESIS are also given in the figure. PAR-LOOP-PARENTHESIS completely updates the cells of the diagonals (with slope -1) in the DP table from left to right.

A divide-and-conquer algorithm [Chowdhury and Ramachandran, 2008] (both pseudocode and pictorial representation) are shown in Figure A.2. The algorithm consists of three recursive divide-and-conquer functions named $\mathcal{A}_{par}$, $\mathcal{B}_{par}$ and $\mathcal{C}_{par}$. The three functions differ from each other in their functionalities. Function $\mathcal{A}_{par}(X)$ completely updates a right-triangular region X by reading from itself; $\mathcal{B}_{par}(X, U, V)$ updates a square region X by reading from itself and two other right-triangular regions U and V; and finally the function $\mathcal{C}_{par}(X, U, V)$ updates a square region X by reading from two other square regions U and V.

The work done by each of the base case kernels: $\mathcal{A}_{loop-par}$, $\mathcal{B}_{loop-par}$, and $\mathcal{C}_{loop-par}$ is cubic w.r.t. their input parameter. The number of function calls to $\mathcal{A}_{loop-par}$, $\mathcal{B}_{loop-par}$, and $\mathcal{C}_{loop-par}$ are $\Theta(n)$, $\Theta(n^2)$, and $\Theta(n^3)$, respectively. This means that if we optimize only the base case $\mathcal{C}_{loop-par}$ then we can get large performance gains. Another important point to note is that the kernel $\mathcal{B}_{loop-par}$ is more flexible than $\mathcal{A}_{loop-par}$. Similarly, kernel $\mathcal{C}_{loop-par}$ is more flexible than $\mathcal{B}_{loop-par}$.

Complexity analysis for the PAR-LOOP-PARENTHESIS algorithm

For $f \in \{\text{PAR-LOOP-PARENTHESIS}\}$, let $W_f(n), Q_f(n), T_f(n)$, and $S_f(n)$ denote the work, serial cache complexity, span, and space consumption of f_{par} on a matrix of size $n \times n$. The total work remains same compared to the serial algorithm $W_f(n) = \Theta(n^3)$. $Q_f(n)$ can be computed for the three loops as $Q_f(n) = \Theta(n \cdot n \cdot (n)) = \Theta(n^3)$. The third loop i.e., k-loop accesses data from both horizontal or vertical lines. Therefore, irrespective of whether we use row-major order or column-major order, we cannot get spatial locality for the innermost loop. The span is computed as $T_f(n) = \Theta(n \cdot (\log n + n \cdot (1))) = \Theta(n^2)$. Finally, $S_f(n) = \Theta(n^2)$.

LOOP-PARENTHESIS(C, n)

1. **for** $i \leftarrow n - 1$ **to** 0 **do**
2.   **for** $j \leftarrow i + 2$ **to** n **do**
3.   **for** $k \leftarrow i + 1$ **to** j **do**
4. $C[i, j] \leftarrow \min \left(\begin{array}{c} C[i, k] + C[k, j] \\ + w(i, j, k), C[i, j] \end{array} \right)$

PAR-LOOP-PARENTHESIS(C, n)

1. **for** $t \leftarrow 2$ **to** $n - 1$ **do**
2.   **parallel for** $i \leftarrow 1$ **to** $n - t$ **do**
3. $j \leftarrow t + i$
4.   **for** $k \leftarrow i + 1$ **to** j **do**
5. $C[i, j] \leftarrow \min \left(\begin{array}{c} C[i, k] + C[k, j] \\ + w(i, j, k), C[i, j] \end{array} \right)$

$\mathcal{A}_{par}(X)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}par}(X)$
2. **else**
3.   **parallel:** $\mathcal{A}_{par}(X_{11})$, $\mathcal{A}_{par}(X_{22})$
4. $\mathcal{B}_{par}(X_{12}, X_{11}, X_{22})$

$\mathcal{C}_{par}(X, U, V)$

1. **if** X is a small matrix **then** $\mathcal{C}_{loop\text{-}par}(X, U, V)$
2. **else**
3.   **parallel:** $\mathcal{C}_{par}(X_{11}, U_{11}, V_{11})$, $\mathcal{C}_{par}(X_{12}, U_{11}, V_{12})$,
         $\mathcal{C}_{par}(X_{21}, U_{21}, V_{11})$, $\mathcal{C}_{par}(X_{22}, U_{21}, V_{12})$
4.   **parallel:** $\mathcal{C}_{par}(X_{11}, U_{12}, V_{21})$, $\mathcal{C}_{par}(X_{12}, U_{12}, V_{22})$,
         $\mathcal{C}_{par}(X_{21}, U_{22}, V_{21})$, $\mathcal{C}_{par}(X_{22}, U_{22}, V_{22})$

$\mathcal{B}_{par}(X, U, V)$

1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}par}(X, U, V)$
2. **else**
3. $\mathcal{B}_{par}(X_{21}, U_{22}, V_{11})$
4.   **parallel:** $\mathcal{C}_{par}(X_{11}, U_{12}, X_{21})$, $\mathcal{C}_{par}(X_{22}, X_{21}, V_{12})$
5.   **parallel:** $\mathcal{B}_{par}(X_{11}, U_{11}, V_{11})$, $\mathcal{B}_{par}(X_{22}, U_{22}, V_{22})$
6. $\mathcal{C}_{par}(X_{12}, U_{12}, X_{22})$
7. $\mathcal{C}_{par}(X_{12}, X_{11}, V_{12})$
8. $\mathcal{B}_{par}(X_{12}, U_{11}, V_{22})$

Figure A.2: Top: Serial and parallel iterative algorithms for solving the parenthesis problem. Middle: A divide-and-conquer algorithm. Initial call to the algorithm is $\mathcal{A}_{par}(C)$, where C is the DP table. Bottom: Dependency graph and pictorial representation of the divide-and-conquer algorithm.

```
𝒜_{loop−par}(X, n)

1.  for t ← 2 to n − 1 do
2.      for i ← 0 to n − t − 1 do
3.          j ← t + i
4.          for k ← i + 1 to j do
5.              X[i, j] ← min (X[i, j], X[i, k] + X[k, j] + w(xi + i, xj + j, xj + i + 1))
```

```
ℬ_{loop−par}(X, U, V, n)

1.  for t ← n − 1 to 0 do
2.      for i ← t to n − 1 do
3.          j ← i − t
4.          for k ← i to n − 1 do
5.              X[i, j] ← min (X[i, j], U[i, k] + V[k, j] + w(xi + i, xj + j, uj + i))
6.          for k ← 0 to j do
7.              X[i, j] ← min (X[i, j], U[i, k] + V[k, j] + w(xi + i, xj + j, uj))
```

```
𝒞_{loop−par}(X, U, V, n)

1.  for i ← 0 to n − 1 do
2.      for j ← 0 to n − 1 do
3.          for k ← 0 to n − 1 do
4.              X[i, j] ← min (X[i, j], U[i, k] + V[k, j] + w(xi + i, xj + j, uj))
```

Figure A.3: Base cases (iterative kernels) of the three recursive functions of the divide-and-conquer algorithm for the parenthesis problem.

> For the parenthesis problem, the PAR-LOOP-PARENTHESIS algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \Theta(n^3)$, $T_\infty(n) = \Theta(n^2)$, parallelism $= \Theta(n)$, and $S_\infty(n) = \Theta(n^2)$.

Complexity analysis for the 2-way $\mathcal{A}_{par}$ algorithm

For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, let $W_f(n), Q_f(n), T_f(n)$, and $S_f(n)$ denote the work, serial cache complexity, span, and space consumption of f_{par} on a matrix of size $n \times n$. Then

$$
\begin{aligned}
W_\mathcal{A}(n) = W_\mathcal{B}(n) = W_\mathcal{C}(n) &= \Theta(1) && \text{if } n = 1, \\
W_\mathcal{A}(n) &= 2W_\mathcal{A}\left(\tfrac{n}{2}\right) + W_\mathcal{B}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
W_\mathcal{B}(n) &= 4\left(W_\mathcal{B}\left(\tfrac{n}{2}\right) + W_\mathcal{C}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n > 1; \\
W_\mathcal{C}(n) &= 8W_\mathcal{C}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$

$$
\begin{aligned}
Q_\mathcal{A}(n) = Q_\mathcal{B}(n) = Q_\mathcal{C}(n) &= \mathcal{O}\left(\tfrac{n^2}{B} + n\right) && \text{if } n^2 \leq \gamma M, \\
Q_\mathcal{A}(n) &= 2Q_\mathcal{A}\left(\tfrac{n}{2}\right) + Q_\mathcal{B}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_A M; \\
Q_\mathcal{B}(n) &= 4\left(Q_\mathcal{B}\left(\tfrac{n}{2}\right) + Q_\mathcal{C}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n^2 > \gamma_B M; \\
Q_\mathcal{C}(n) &= 8Q_\mathcal{C}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_C M;
\end{aligned}
$$

$$
\begin{aligned}
T_\mathcal{A}(n) = T_\mathcal{B}(n) = T_\mathcal{C}(n) &= \mathcal{O}(1) && \text{if } n = 1, \\
T_\mathcal{A}(n) &= T_\mathcal{A}\left(\tfrac{n}{2}\right) + T_\mathcal{B}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
T_\mathcal{B}(n) &= 3\left(T_\mathcal{B}\left(\tfrac{n}{2}\right) + T_\mathcal{C}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n > 1; \\
T_\mathcal{C}(n) &= 2T_\mathcal{C}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$

$$\begin{aligned}
S_{\mathcal{A}}(n) = S_{\mathcal{B}}(n) = S_{\mathcal{C}}(n) &= \mathcal{O}(1) && \text{if } n = 1, \\
S_{\mathcal{A}}(n) &= 2S_{\mathcal{A}}\left(\tfrac{n}{2}\right) + S_{\mathcal{B}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
S_{\mathcal{B}}(n) &= S_{\mathcal{B}}\left(\tfrac{n}{2}\right) + 3\max\left(S_{\mathcal{B}}\left(\tfrac{n}{2}\right), S_{\mathcal{C}}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n > 1; \\
S_{\mathcal{C}}(n) &= 4S_{\mathcal{C}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}$$

where, γ, γ_A, γ_B and γ_C are suitable constants. Solving, $Q_{\mathcal{A}}(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$, $W_{\mathcal{A}}(n) = \Theta(n^3)$, $T_{\mathcal{A}}(n) = \mathcal{O}\left(n^{\log 3}\right)$, and $S_{\mathcal{A}}(n) = \Theta(n^2)$.

> For the parenthesis problem, the 2-way $\mathcal{A}_{par}$ algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$, $T_\infty(n) = \Theta\left(n^{\log 3}\right)$, parallelism $= \Theta\left(n^{3-\log 3}\right)$, and $S_\infty(n) = \Theta(n^2)$.

$\mathcal{A}_{par}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{A}_{loop\text{-}par}(X, U, V)$
   else
3.   Let $diagonal$ represent $(j - i)$
4.   **parallel:** $\mathcal{A}_{par}(X_{i,j}, U_{i,j}, V_{i,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal = 0$
5.   **for** $k \leftarrow 1$ **to** $r - 1$ **do**
6.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U_{i,k+i-1}, V_{k+i-1,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal \in [k, \min\{2k-2, r-1\}]$
7.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U_{i,1+i}, V_{1+i,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal \in [k, \min\{2k-3, r-1\}]$
8.   **parallel:** $\mathcal{B}_{par}(X_{i,j}, U_{i,i}, V_{j,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal = k$

$\mathcal{B}_{par}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{B}_{loop\text{-}par}(X, U, V)$
   else
3.   Let $U'_{i,\ell} = \begin{cases} X_{i,\ell} & \text{if } \ell > 0, \\ U_{i,\ell+r} & \text{if } \ell \leq 0. \end{cases}$ and $V'_{\ell,j} = \begin{cases} V_{\ell,j} & \text{if } \ell > 0, \\ X_{\ell+r,j} & \text{if } \ell \leq 0. \end{cases}$
4.   Let $diagonal$ represent $(j - i)$
5.   **for** $k \leftarrow 1$ **to** $2r - 1$ **do**
6.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U'_{i,k-r+i-1}, V'_{k-r+i-1,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal + r \in [k, \min\{2k-2, 2r-1\}]$
7.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U'_{i,1+i-r}, V'_{1+i-r,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal + r \in [k, \min\{2k-3, 2r-1\}]$
8.   **parallel:** $\mathcal{B}_{par}(X_{i,j}, U_{i,i}, V_{j,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal + r = k$

$\mathcal{C}_{par}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{C}_{loop\text{-}par}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U_{i,k}, V_{k,j}, d+1)$ for $i, j \in [1, r]$

Figure A.4: r-way divide-and-conquer algorithm for the parenthesis algorithm.

Complexity analysis for the r-way $\mathcal{A}_{par}$ algorithm

For simplicity of exposition we assume that n is a power of r. For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, let $W_f(n), Q_f(n), T_f(n)$, and $S_f(n)$ denote the total work, serial cache complexity, span, and parallel space consumption of r-way f_{par} for the parameter n. Then

$$W_{\mathcal{A}}(n) = W_{\mathcal{B}}(n) = W_{\mathcal{C}}(n) = \Theta(1) \qquad\qquad\qquad\qquad\qquad\qquad \text{if } n = 1,$$
$$W_{\mathcal{A}}(n) = rW_{\mathcal{A}}\left(\frac{n}{r}\right) + \frac{r(r-1)}{2}W_{\mathcal{B}}\left(\frac{n}{r}\right) + \frac{r(r-1)(r-2)}{6}W_{\mathcal{C}}\left(\frac{n}{r}\right) + \Theta(1) \quad \text{if } n > 1;$$
$$W_{\mathcal{B}}(n) = r^2 W_{\mathcal{B}}\left(\frac{n}{r}\right) + r^2(r-1)W_{\mathcal{C}}\left(\frac{n}{r}\right) + \Theta(1) \qquad\qquad\qquad \text{if } n > 1;$$
$$W_{\mathcal{C}}(n) = r^3 W_{\mathcal{C}}\left(\frac{n}{r}\right) + \Theta(1) \qquad\qquad\qquad\qquad\qquad\qquad \text{if } n > 1;$$

$$Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) = \mathcal{O}\left(\frac{n^2}{B} + 1\right) \qquad\qquad\qquad\qquad \text{if } n^2 \leq \gamma M,$$
$$Q_{\mathcal{A}}(n) = rQ_{\mathcal{A}}\left(\frac{n}{r}\right) + \frac{r(r-1)}{2}Q_{\mathcal{B}}\left(\frac{n}{r}\right) + \frac{r(r-1)(r-2)}{6}Q_{\mathcal{C}}\left(\frac{n}{r}\right) + \Theta(1) \quad \text{if } n^2 > \gamma_A M;$$
$$Q_{\mathcal{B}}(n) = r^2 Q_{\mathcal{B}}\left(\frac{n}{r}\right) + r^2(r-1)Q_{\mathcal{C}}\left(\frac{n}{r}\right) + \Theta(1) \qquad\qquad\qquad \text{if } n^2 > \gamma_B M;$$
$$Q_{\mathcal{C}}(n) = r^3 Q_{\mathcal{C}}\left(\frac{n}{r}\right) + \Theta(1) \qquad\qquad\qquad\qquad\qquad\qquad \text{if } n^2 > \gamma_C M;$$

$$T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) = \mathcal{O}(1) \qquad\qquad\qquad\qquad\qquad \text{if } n = 1,$$
$$T_{\mathcal{A}}(n) = T_{\mathcal{A}}\left(\frac{n}{r}\right) + T_{\mathcal{B}}\left(\frac{n}{r}\right) + \frac{(r-2)(r-1)}{2}T_{\mathcal{C}}\left(\frac{n}{r}\right) + \Theta(1) \quad \text{if } n > 1;$$
$$T_{\mathcal{B}}(n) = (2r-1)\left(T_{\mathcal{B}}\left(\frac{n}{r}\right) + (r-1)T_{\mathcal{C}}\left(\frac{n}{r}\right)\right) + \Theta(1) \qquad \text{if } n > 1;$$
$$T_{\mathcal{C}}(n) = rT_{\mathcal{C}}\left(\frac{n}{r}\right) + \Theta(1) \qquad\qquad\qquad\qquad\qquad\qquad \text{if } n > 1;$$

where, γ, γ_A, γ_B and γ_C are suitable constants.

The serial cache complexity is computed as below.

$$Q_{\mathcal{C}}(n) = r^3 Q_{\mathcal{C}}\left(\frac{n}{r}\right) + c = r^3\left(r^3 Q_{\mathcal{C}}\left(\frac{n}{r^2}\right) + c\right) + c = (r^3)^2 Q_{\mathcal{C}}\left(\frac{n}{r^2}\right) + c\left((r^3)^1 + 1\right)$$
$$= (r^3)^k Q_{\mathcal{C}}\left(\frac{n}{r^k}\right) + c\left((r^3)^{k-1} + \cdots + 1\right) \qquad \left(\textbf{say } \left(\frac{n}{r^k}\right)^2 \leq \gamma M\right)$$
$$= cr^{3k}\left(\left(\frac{n}{r^k}\right)^2 \frac{1}{B} + 1\right) + c\left((r^3)^{k-1} + \cdots + 1\right) = cr^k\frac{n^2}{B} + c\left((r^3)^k + \cdots + 1\right) = cr^k\frac{n^2}{B} + cr^{3k}$$

Let $(n/r^k)^2 \leq \gamma M$. This implies $(n/r^{k-1})^2 > \gamma M$ or in other words $r^{k-1} < n/\sqrt{\gamma M}$. Substituting the value of r^{k-1} in the above equation we have

$$Q_{\mathcal{C}}(n) = cr^{k-1}r\frac{n^2}{B} + cr^{3k-3}r^3 \leq cr\frac{n^3}{B\sqrt{M}} + cr^3\frac{n^3}{M\sqrt{M}} = \mathcal{O}\left(\frac{n^3 r}{B\sqrt{M}} + \frac{n^3 r^3}{M\sqrt{M}}\right)$$
$$= \mathcal{O}\left(\frac{n^3}{B\left\lceil\frac{\sqrt{M}}{r}\right\rceil} + \frac{n^3}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil^3}\right)$$

We now consider the cache complexity of the $\mathcal{B}_{par}$ function using the complexity of $\mathcal{C}_{par}$.

$$
\begin{aligned}
Q_{\mathcal{B}}(n) &\leq r^2 Q_{\mathcal{B}}\left(\frac{n}{r}\right) + r^2(r-1)Q_{\mathcal{C}}\left(\frac{n}{r}\right) + c \\
&= r^2\left(r^2 Q_{\mathcal{B}}\left(\frac{n}{r^2}\right) + r^2(r-1)Q_{\mathcal{C}}\left(\frac{n}{r^2}\right) + c\right) + r^2(r-1)Q_{\mathcal{C}}\left(\frac{n}{r}\right) + c \\
&= (r^2)^2 Q_{\mathcal{B}}\left(\frac{n}{r^2}\right) + (r^2)^2(r-1)Q_{\mathcal{C}}\left(\frac{n}{r^2}\right) + cr^2 + r^2(r-1)Q_{\mathcal{C}}\left(\frac{n}{r}\right) + c \\
&= (r^2)^k Q_{\mathcal{B}}\left(\frac{n}{r^k}\right) + r^2(r-1)\left((r^2)^{k-1}Q_{\mathcal{C}}\left(\frac{n}{r^k}\right) + \cdots + Q_{\mathcal{C}}\left(\frac{n}{r}\right)\right) + c\left((r^2)^{k-1} + \cdots + 1\right) \\
&\leq cr^{2k}\left(\left(\frac{n}{r^k}\right)^2 \frac{1}{B} + 1\right) + r^2(r-1)\sum_{i=1}^{k}\left((r^2)^{i-1}Q_{\mathcal{C}}\left(\frac{n}{r^i}\right)\right) + c\left((r^2)^{k-1} + \cdots + 1\right) \\
&= c\frac{n^2}{B} + cr^2(r-1)\sum_{i=1}^{k}\left((r^2)^{i-1}\left(\left(\frac{n}{r^i}\right)^3\frac{r}{B\sqrt{M}} + \left(\frac{n}{r^i}\right)^3\frac{r^3}{M\sqrt{M}}\right)\right) + cr^{2k} \\
&= c\frac{n^2}{B} + c\frac{n^3}{B\sqrt{M}}r(r-1)\sum_{i=1}^{k}\frac{1}{r^i} + c\frac{n^3}{M\sqrt{M}}r^3(r-1)\sum_{i=1}^{k}\frac{1}{r^i} + cr^{2k} \\
&= c\frac{n^2}{B} + c\frac{n^3}{B\sqrt{M}}r(r-1)\frac{r^k-1}{r^k(r-1)} + c\frac{n^3}{M\sqrt{M}}r^3(r-1)\frac{r^k-1}{r^k(r-1)} + cr^{2k} \\
&= c\frac{n^2}{B} + c\frac{n^3}{B\sqrt{M}}\left(r - \frac{1}{r^{k-1}}\right) + c\frac{n^3}{M\sqrt{M}}\left(r^3 - \frac{r^2}{r^{k-1}}\right) + cr^{2k-2}r^2
\end{aligned}
$$

Substituting the upper bound of r^{k-1}, we have

$$
\begin{aligned}
Q_{\mathcal{B}}(n) &\leq c\frac{n^2}{B} + c\frac{n^3}{B\sqrt{M}}\left(r - \frac{\sqrt{M}}{n}\right) + c\frac{n^3}{M\sqrt{M}}\left(r^3 - \frac{r^2\sqrt{M}}{n}\right) + cr^{2k-2}r^2 \\
&\leq c\frac{n^2}{B} + c\frac{n^3 r}{B\sqrt{M}} + c\frac{n^3 r^3}{M\sqrt{M}} + c\frac{n^2 r^2}{M} = \mathcal{O}\left(\frac{n^3 r}{B\sqrt{M}} + \frac{n^3 r^3}{M\sqrt{M}} + \frac{n^2}{B} + \frac{n^2 r^2}{M}\right) \\
&= \mathcal{O}\left(\frac{n^3}{B\left\lceil\frac{\sqrt{M}}{r}\right\rceil} + \frac{n^3}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil^3} + \frac{n^2}{B} + \frac{n^2}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil^2}\right)
\end{aligned}
$$

With the cache complexity of $\mathcal{B}_{par}$ and $\mathcal{C}_{par}$ functions, we can easily find the cache com-

plexity of the $\mathcal{A}$ function.

$$Q_{\mathcal{A}}(n) \leq rQ_{\mathcal{A}}\left(\frac{n}{r}\right) + \frac{r(r-1)}{2}Q_{\mathcal{B}}\left(\frac{n}{r}\right) + \frac{r(r-1)(r-2)}{6}Q_{\mathcal{C}}\left(\frac{n}{r}\right) + c$$

$$= r\left(rQ_{\mathcal{A}}\left(\frac{n}{r^2}\right) + \frac{r(r-1)}{2}Q_{\mathcal{B}}\left(\frac{n}{r^2}\right) + \frac{r(r-1)(r-2)}{6}Q_{\mathcal{C}}\left(\frac{n}{r^2}\right) + c\right)$$

$$+ \frac{r(r-1)}{2}Q_{\mathcal{B}}\left(\frac{n}{r}\right) + \frac{r(r-1)(r-2)}{6}Q_{\mathcal{C}}\left(\frac{n}{r}\right) + c$$

$$= r^2 Q_{\mathcal{A}}\left(\frac{n}{r^2}\right) + \frac{r(r-1)}{2}\left(rQ_{\mathcal{B}}\left(\frac{n}{r^2}\right) + Q_{\mathcal{B}}\left(\frac{n}{r}\right)\right)$$

$$+ \frac{r(r-1)(r-2)}{6}\left(rQ_{\mathcal{C}}\left(\frac{n}{r^2}\right) + Q_{\mathcal{C}}\left(\frac{n}{r}\right)\right) + c(r+1)$$

$$= r^k Q_{\mathcal{A}}\left(\frac{n}{r^k}\right) + \frac{r(r-1)}{2}\left(r^{k-1}Q_{\mathcal{B}}\left(\frac{n}{r^k}\right) + \cdots + Q_{\mathcal{B}}\left(\frac{n}{r}\right)\right)$$

$$+ \frac{r(r-1)(r-2)}{6}\left(r^{k-1}Q_{\mathcal{C}}\left(\frac{n}{r^k}\right) + \cdots + Q_{\mathcal{C}}\left(\frac{n}{r}\right)\right) + c(r^{k-1} + \cdots + 1)$$

$$= cr^k\left(\left(\frac{n}{r^k}\right)^2\frac{1}{B} + 1\right) + \frac{r(r-1)}{2}\sum_{i=1}^{k}r^{i-1}Q_{\mathcal{B}}\left(\frac{n}{r^i}\right)$$

$$+ \frac{r(r-1)(r-2)}{6}\sum_{i=1}^{k}r^{i-1}Q_{\mathcal{C}}\left(\frac{n}{r^i}\right) + c(r^{k-1} + \cdots + 1)$$

$$= c\frac{1}{r^k}\frac{n^2}{B} + c\frac{r(r-1)}{2}\sum_{i=1}^{k}\left(r^{i-1}\left(\left(\frac{n}{r^i}\right)^3\frac{r}{B\sqrt{M}} + \left(\frac{n}{r^i}\right)^3\frac{r^3}{M\sqrt{M}} + \left(\frac{n}{r^i}\right)^2\frac{1}{B} + \left(\frac{n}{r^i}\right)^2\frac{r^2}{M}\right)\right)$$

$$+ c\frac{r(r-1)(r-2)}{6}\sum_{i=1}^{k}\left(r^{i-1}\left(\left(\frac{n}{r^i}\right)^3\frac{r}{B\sqrt{M}} + \left(\frac{n}{r^i}\right)^3\frac{r^3}{M\sqrt{M}}\right)\right) + cr^k$$

$$= c\frac{1}{r^k}\frac{n^2}{B} + c\frac{r(r-1)}{2}\frac{n^3}{B\sqrt{M}}\sum_{i=1}^{k}\frac{1}{r^{2i}} + c\frac{r^3(r-1)}{2}\frac{n^3}{M\sqrt{M}}\sum_{i=1}^{k}\frac{1}{r^{2i}} + c\frac{(r-1)}{2}\frac{n^2}{B}\sum_{i=1}^{k}\frac{1}{r^i}$$

$$+ c\frac{r^2(r-1)}{2}\frac{n^2}{M}\sum_{i=1}^{k}\frac{1}{r^i} + c\frac{r(r-1)(r-2)}{6}\frac{n^3}{B\sqrt{M}}\sum_{i=1}^{k}\frac{1}{r^{2i}} + c\frac{r^3(r-1)(r-2)}{6}\frac{n^3}{M\sqrt{M}}\sum_{i=1}^{k}\frac{1}{r^{2i}} + cr^k$$

After substituting $\sum_{i=1}^{k}(1/r^i) = (r^k - 1)/(r^k(r-1))$ and $\sum_{i=1}^{k}(1/r^{2i}) = (r^{2k} - 1)/(r^{2k}(r^2 - 1))$ in the equation above and simplifying, we have

$$Q_{\mathcal{A}}(n) \leq c\frac{1}{rr^{k-1}}\frac{n^2}{B} + \frac{c}{2}\frac{r}{r+1}\left(1 - \frac{1}{r^2 r^{2k-2}}\right)\frac{n^3}{B\sqrt{M}} + \frac{c}{2}\frac{r^3}{r+1}\left(1 - \frac{1}{r^2 r^{2k-2}}\right)\frac{n^3}{M\sqrt{M}}$$

$$+ \frac{c}{2}\left(1 - \frac{1}{rr^{k-1}}\right)\frac{n^2}{B} + \frac{c}{2}\left(r^2 - \frac{r}{r^{k-1}}\right)\frac{n^2}{M} + \frac{c}{6}\frac{r(r-2)}{r+1}\left(1 - \frac{1}{r^2 r^{2k-2}}\right)\frac{n^3}{B\sqrt{M}}$$

$$+ \frac{c}{6}\frac{r^3(r-2)}{r+1}\left(1 - \frac{1}{r^2 r^{2k-2}}\right)\frac{n^3}{M\sqrt{M}} + crr^{k-1}$$

Substituting the value of r^{k-1}, we get

$$Q_{\mathcal{A}}(n) \leq c\frac{\sqrt{M}}{nr}\frac{n^2}{B} + \frac{c}{2}\frac{r}{r+1}\left(1 - \frac{M}{n^2r^2}\right)\frac{n^3}{B\sqrt{M}} + \frac{c}{2}\frac{r^3}{r+1}\left(1 - \frac{M}{n^2r^2}\right)\frac{n^3}{M\sqrt{M}}$$

$$+ \frac{c}{2}\left(1 - \frac{\sqrt{M}}{nr}\right)\frac{n^2}{B} + \frac{c}{2}\left(r^2 - \frac{\sqrt{M}r}{n}\right)\frac{n^2}{M} + \frac{c}{6}\frac{r(r-2)}{r+1}\left(1 - \frac{M}{n^2r^2}\right)\frac{n^3}{B\sqrt{M}}$$

$$+ \frac{c}{6}\frac{r^3(r-2)}{r+1}\left(1 - \frac{M}{n^2r^2}\right)\frac{n^3}{M\sqrt{M}} + c\frac{nr}{\sqrt{M}}$$

$$\leq c\frac{n\sqrt{M}}{rB} + \frac{c}{2}\frac{n^3}{B\sqrt{M}} + \frac{c}{2}\frac{n^3r^2}{M\sqrt{M}} + \frac{c}{2}\frac{n^2}{B} + \frac{c}{2}\frac{n^2r^2}{M} + \frac{c}{6}\frac{n^3r}{B\sqrt{M}} + \frac{c}{6}\frac{n^3r^3}{M\sqrt{M}} + c\frac{nr}{\sqrt{M}}$$

$$= \mathcal{O}\left(\frac{n^3r}{B\sqrt{M}} + \frac{n^3r^3}{M\sqrt{M}} + \frac{n^2}{B} + \frac{n^2r^2}{M} + \frac{n\sqrt{M}}{rB} + \frac{nr}{\sqrt{M}}\right)$$

$$= \mathcal{O}\left(\frac{n^3}{B\left\lceil\frac{\sqrt{M}}{r}\right\rceil} + \frac{n^3}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil^3} + \frac{n^2}{B} + \frac{n^2}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil^2} + \frac{n}{B}\left\lceil\frac{\sqrt{M}}{r}\right\rceil + \frac{n}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil}\right)$$

We now show the analysis for the span of the r-way divide-and-conquer algorithm. Similar to the previous analysis we assume $(n/r^k) = 1$, or $r^k = n$ or $k = \log_r n$.

$$T_{\mathbb{C}}(n) = rT_{\mathbb{C}}\left(\frac{n}{r}\right) + c = r\left(rT_{\mathbb{C}}\left(\frac{n}{r^2}\right) + c\right) + c = r^2T_{\mathbb{C}}\left(\frac{n}{r^2}\right) + c(r+1)$$

$$= r^kT_{\mathbb{C}}\left(\frac{n}{r^k}\right) + c\left(r^{k-1} + \cdots + 1\right) = cr^k(1) + c\left(r^{k-1} + \cdots + 1\right) \leq c'r^k = \Theta(n)$$

$$T_{\mathcal{B}}(n) = (2r-1)T_{\mathcal{B}}\left(\frac{n}{r}\right) + (r-1)(2r-1)T_{\mathbb{C}}\left(\frac{n}{r}\right) + c$$

$$= (2r-1)\left((2r-1)T_{\mathcal{B}}\left(\frac{n}{r^2}\right) + (r-1)(2r-1)T_{\mathbb{C}}\left(\frac{n}{r^2}\right) + c\right) + (r-1)(2r-1)T_{\mathbb{C}}\left(\frac{n}{r}\right) + c$$

$$= (2r-1)^2T_{\mathcal{B}}\left(\frac{n}{r^2}\right) + (r-1)(2r-1)\left((2r-1)T_{\mathbb{C}}\left(\frac{n}{r^2}\right) + T_{\mathbb{C}}\left(\frac{n}{r}\right)\right) + c\left((2r-1)+1\right)$$

$$= (2r-1)^kT_{\mathcal{B}}\left(\frac{n}{r^k}\right) + (r-1)(2r-1)\left((2r-1)^{k-1}T_{\mathbb{C}}\left(\frac{n}{r^k}\right) + \cdots + T_{\mathbb{C}}\left(\frac{n}{r}\right)\right) + c\left((2r-1)^{k-1} + \cdots + 1\right)$$

$$\leq (2r-1)^k(c) + (r-1)(2r-1)\sum_{i=1}^{k}(2r-1)^{i-1}T_{\mathbb{C}}\left(\frac{n}{r^i}\right) + c\left((2r-1)^{k-1} + \cdots + 1\right)$$

$$\leq (2r-1)^k(c) + c(r-1)(2r-1)\sum_{i=1}^{k}(2r-1)^{i-1}\frac{n}{r^i} + c\left((2r-1)^{k-1} + \cdots + 1\right)$$

$$= c(r-1)(2r-1)\left(\frac{\left(2-\frac{1}{r}\right)^k - 1}{r-1}\right) + c'(2r-1)^k = c(2r-1)\left(\left(2-\frac{1}{r}\right)^k - 1\right) + c'(2r-1)^k$$

151

Substituting the value of $k = \log_r n$, we have

$$T_\mathcal{B}(n) \le c(2r-1)\left(\left(2-\frac{1}{r}\right)^{\log_r n} - 1\right) + c'(2r-1)^{\log_r n}$$

$$= c(2r-1)\left(n^{\log_r\left(2-\frac{1}{r}\right)} - 1\right) + c'n^{\log_r(2r-1)} = \Theta\left(rn^{\log_r\left(2-\frac{1}{r}\right)} + n^{\log_r(2r-1)}\right)$$

$$T_\mathcal{A}(n) = T_\mathcal{A}\left(\frac{n}{r}\right) + T_\mathcal{B}\left(\frac{n}{r}\right) + \frac{(r-2)(r-1)}{2}T_\mathcal{C}\left(\frac{n}{r}\right) + c$$

$$= \left(T_\mathcal{A}\left(\frac{n}{r^2}\right) + T_\mathcal{B}\left(\frac{n}{r^2}\right) + \frac{(r-2)(r-1)}{2}T_\mathcal{C}\left(\frac{n}{r^2}\right) + c\right) + T_\mathcal{B}\left(\frac{n}{r}\right) + \frac{(r-2)(r-1)}{2}T_\mathcal{C}\left(\frac{n}{r}\right) + c$$

$$= T_\mathcal{A}\left(\frac{n}{r^2}\right) + T_\mathcal{B}\left(\frac{n}{r^2}\right) + T_\mathcal{B}\left(\frac{n}{r}\right) + \frac{(r-2)(r-1)}{2}\left(T_\mathcal{C}\left(\frac{n}{r^2}\right) + T_\mathcal{C}\left(\frac{n}{r}\right)\right) + 2c$$

$$= T_\mathcal{A}\left(\frac{n}{r^k}\right) + \left(T_\mathcal{B}\left(\frac{n}{r^k}\right) + \cdots + T_\mathcal{B}\left(\frac{n}{r}\right)\right) + \frac{(r-2)(r-1)}{2}\left(T_\mathcal{C}\left(\frac{n}{r^k}\right) + \cdots + T_\mathcal{C}\left(\frac{n}{r}\right)\right) + kc$$

$$\le c + \sum_{i=1}^{k}T_\mathcal{B}\left(\frac{n}{r^i}\right) + \frac{(r-2)(r-1)}{2}\sum_{i=1}^{k}T_\mathcal{C}\left(\frac{n}{r^i}\right) + kc$$

$$\le c\sum_{i=1}^{k}\left(r\left(\frac{n}{r^i}\right)^{\log_r\left(2-\frac{1}{r}\right)} + \left(\frac{n}{r^i}\right)^{\log_r(2r-1)}\right) + c\frac{(r-2)(r-1)}{2}\sum_{i=1}^{k}\frac{n}{r^i} + kc$$

$$\le crn^{\log_r\left(2-\frac{1}{r}\right)}\sum_{i=1}^{k}\frac{1}{\left(r^{\log_r\left(2-\frac{1}{r}\right)}\right)^i} + cn^{\log_r(2r-1)}\sum_{i=1}^{k}\frac{1}{\left(r^{\log_r(2r-1)}\right)^i} + cn\frac{(r-2)(r-1)}{2}\sum_{i=1}^{k}\frac{1}{r^i} + kc$$

$$\le crn^{\log_r\left(2-\frac{1}{r}\right)}\sum_{i=1}^{k}\frac{1}{\left(2-\frac{1}{r}\right)^i} + cn^{\log_r(2r-1)}\sum_{i=1}^{k}\frac{1}{(2r-1)^i} + cn\frac{(r-2)(r-1)}{2}\sum_{i=1}^{k}\frac{1}{r^i} + kc$$

$$\le cn^{\log_r\left(2-\frac{1}{r}\right)}\frac{r^2}{r-1}\left(1-\frac{1}{\left(2-\frac{1}{r}\right)^k}\right) + cn^{\log_r(2r-1)}\frac{1}{2(r-1)}\left(1-\frac{1}{(2r-1)^k}\right) + cn\frac{(r-2)}{2}\left(1-\frac{1}{r^k}\right) + kc$$

Substituting the value of $k = \log_r n$ and simplifying we have

$$T_\mathcal{A}(n) = \mathcal{O}\left(rn^{\log_r\left(2-\frac{1}{r}\right)} + \frac{n^{\log_r(2r-1)}}{r} + nr\right)$$

> For the parenthesis problem, the r-way $\mathcal{A}_{par}$ algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\left\lceil\frac{\sqrt{M}}{r}\right\rceil} + \frac{n^3}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil^3} + \frac{n^2}{B} + \frac{n^2}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil^2} + \frac{n}{B}\left\lceil\frac{\sqrt{M}}{r}\right\rceil + \frac{n}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil}\right)$, $T_\infty(n) = \Theta\left(rn^{\log_r\left(2-\frac{1}{r}\right)} + \frac{n^{\log_r(2r-1)}}{r} + nr\right)$, **parallelism** $= \frac{T_1(n)}{T_\infty(n)}$, **and** $S_\infty(n) = \Theta(n^2)$.

A.3   Floyd-Warshall's all-pairs shortest path

An algebraic structure known as a closed semiring [Aho et al., 1974] serves as a general framework for solving path problems in directed graphs. In [Aho et al., 1974], an algorithm

is given for finding the set of all paths between each pair of vertices in a directed graph. Both Floyd-Warshall's algorithm for finding all-pairs shortest paths [Floyd, 1962] and Warshall's algorithm for finding transitive closures [Warshall, 1962] are instantiations of this algorithm.

Consider a directed graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, and each edge (v_i, v_j) is labeled by an element $\ell(v_i, v_j)$ of some closed semiring $(S, \oplus, \odot, 0, 1)$. If $(v_i, v_j) \notin E$, $\ell(v_i, v_j)$ is assumed to have a value 0. The path-cost of a path is defined as the product ($\odot$) of the labels of the edges in the path, taken in order. The path-cost of a zero length path is 1. For each pair $v_i, v_j \in V$, $d[i, j]$ is defined to be the sum of the path-costs of all paths going from v_i to v_j. By convention, the sum over an empty set of paths is 0. Even if there are infinitely many paths between v_i and v_j (due to presence of cycles), $d[i, j]$ will still be well-defined due to the properties of a closed semiring.

For $i, j \in [1, n]$ and $k \in [0, n]$, let $D[i, j, k]$ denote cost of the smallest cost path from v_i to v_j with no intermediate vertex higher than v_k. Then $d[i, j] = D[i, j, n]$. The following recurrence computes all $D[i, j, k]$.

$$D[i, j, k] = \begin{cases} 1 & \text{if } k = 0 \text{ and } i = j, \\ l(v_i, v_j) & \text{if } k = 0 \text{ and } i \neq j, \\ D[i, j, k - 1] \oplus (D[i, k, k - 1] \odot D[k, j, k - 1]) & \text{if } k > 0. \end{cases} \quad \text{(A.4)}$$

The dependency structure of the DP is shown in Figure A.5. Floyd-Warshall's APSP performs computations over a particular closed semiring $(\mathbb{R}, \min, +, +\infty, 0)$.

A serial and a parallel iterative algorithm for computing the path costs are given in Figure A.5. The structure of the three loops seems similar to the matrix multiplication algorithm but there are differences. In matrix multiplication, the i-j-k loops can be ordered in any of the $3! = 6$ ways. However, in the computation of path costs, the k-loop must always be the outermost loop.

Figures A.5 and A.6 show a divide-and-conquer implementation [Chowdhury and Ramachandran, 2010] of the Floyd-Warshall's APSP algorithm. The implementation consists of four recursive divide-and-conquer functions. The number of times $\mathcal{A}_{loop-FW}$, $\mathcal{B}_{loop-FW}$, $\mathcal{C}_{loop-FW}$, and $\mathcal{D}_{loop-FW}$ will be called are $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^2)$, and $\Theta(n^3)$ respectively. Hence, function $\mathcal{D}_{FW}$ is dominating.

Complexity analysis for the PAR-LOOP-PATH-COSTS algorithm

For $f \in \{\text{PAR-LOOP-PATH-COSTS}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{FW} on a matrix of size $n \times n$. $W_f(n)$ is $\Theta(n^3)$. $Q_f(n)$ is computed as $Q_f(n) = \Theta\left(n \cdot n \cdot \left(\frac{n}{B} + 1\right)\right) = \Theta\left(\frac{n^3}{B} + n^2\right)$. The span is $T_f(n) = \Theta(n \cdot (\log n + (\log n + \Theta(1)))) = \Theta(n \log n)$. The properties of the given semiring implies that $d[i, j]$ for all pairs of vertices $v_i, v_j \in V$ uses only $\Theta(n^2)$ space as shown in Figure A.5.

> For the Floyd-Warshall's all-pairs shortest path algorithm, the PAR-LOOP-PATH-COSTS implementation achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \Theta\left(\frac{n^3}{B} + n^2\right)$, $T_\infty(n) = \Theta(n \log n)$, parallelism $= \Theta\left(\frac{n^2}{\log n}\right)$, and $S_\infty(n) = \Theta(n^2)$.

Complexity analysis for the 2-way $\mathcal{A}_{FW}$ algorithm

For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{FW} on a matrix of size $n \times n$. Then
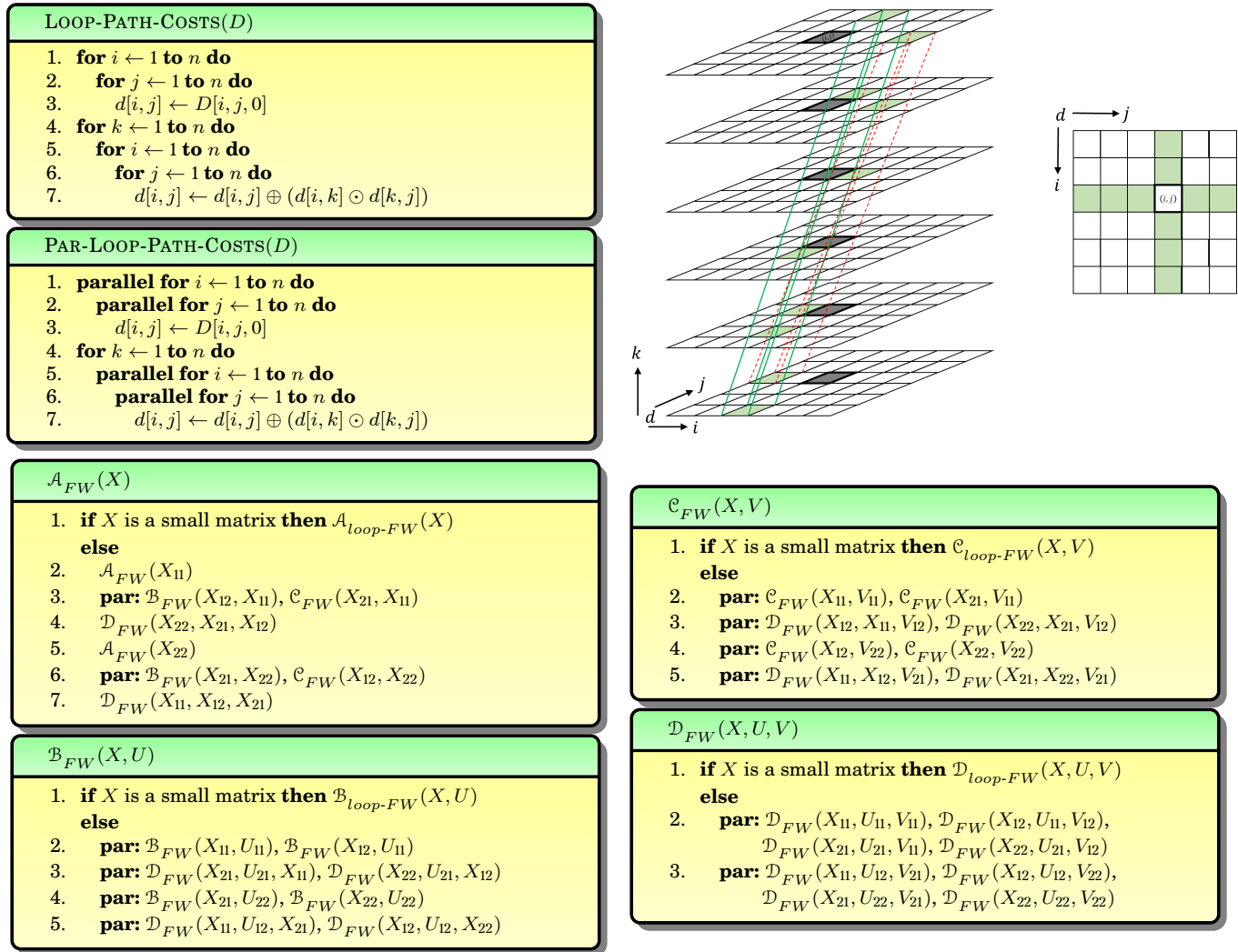
Figure A.5: Top left: Serial and parallel iterative algorithms to compute path costs over a closed semiring $(S, \oplus, \odot, 0, 1)$. Top right: Dependency structure for Floyd-Warshall's APSP algorithm. Bottom: Divide-and-conquer algorithm.

$$
\begin{aligned}
Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) = Q_{\mathcal{D}}(n) &= \mathcal{O}\left(\tfrac{n^2}{B} + n\right) && \text{if } n^2 \leq \gamma M, \\
Q_{\mathcal{A}}(n) &= 2\left(Q_{\mathcal{A}}\left(\tfrac{n}{2}\right) + Q_{\mathcal{B}}\left(\tfrac{n}{2}\right) + Q_{\mathcal{C}}\left(\tfrac{n}{2}\right) + Q_{\mathcal{D}}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n^2 > \gamma_A M; \\
Q_{\mathcal{B}}(n) &= 4\left(Q_{\mathcal{B}}\left(\tfrac{n}{2}\right) + Q_{\mathcal{D}}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n^2 > \gamma_B M; \\
Q_{\mathcal{C}}(n) &= 4\left(Q_{\mathcal{C}}\left(\tfrac{n}{2}\right) + Q_{\mathcal{D}}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n^2 > \gamma_C M; \\
Q_{\mathcal{D}}(n) &= 8Q_{\mathcal{D}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_D M;
\end{aligned}
$$

$$
\begin{aligned}
T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) = T_{\mathcal{D}}(n) &= \mathcal{O}(1) && \text{if } n = 1, \\
T_{\mathcal{A}}(n) &= 2\left(T_{\mathcal{A}}\left(\tfrac{n}{2}\right) + \max\left\{T_{\mathcal{B}}\left(\tfrac{n}{2}\right), T_{\mathcal{C}}\left(\tfrac{n}{2}\right)\right\} + T_{\mathcal{D}}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{B}}(n) &= 2\left(T_{\mathcal{B}}\left(\tfrac{n}{2}\right) + T_{\mathcal{D}}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{C}}(n) &= 2\left(T_{\mathcal{C}}\left(\tfrac{n}{2}\right) + T_{\mathcal{D}}\left(\tfrac{n}{2}\right)\right) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{D}}(n) &= 2T_{\mathcal{D}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$
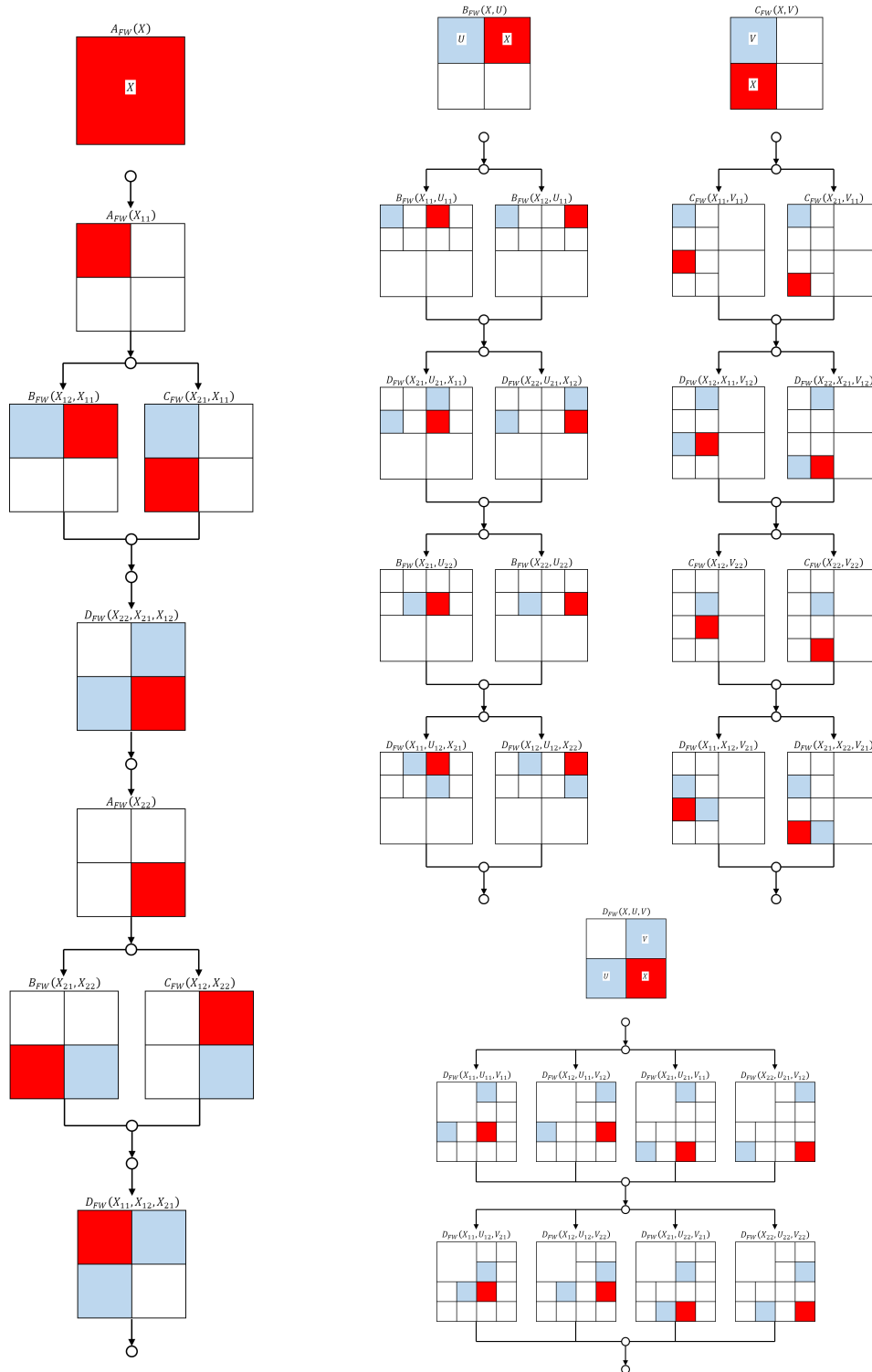
Figure A.6: A divide-and-conquer algorithm $\mathcal{A}_{FW}$ for solving the Floyd-Warshall's all-pairs shortest path problem.

where, γ, γ_A, γ_B, γ_C and γ_D are suitable constants. Solving, $Q_\mathcal{A}(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$ and $T_\mathcal{A}(n) = \mathcal{O}\left(n\log^2 n\right)$.

> For the Floyd-Warshall's all-pairs shortest path algorithm, the $\mathcal{A}_{FW}$ implementation achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$, $T_\infty(n) = \Theta\left(n\log^2 n\right)$, parallelism $= \Theta\left(\frac{n^2}{\log^2 n}\right)$, and $S_\infty(n) = \Theta(n^2)$.

$\mathcal{A}_{FW}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{A}_{loop\text{-}FW}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4. $\mathcal{A}_{FW}(X_{kk}, U_{kk}, V_{kk}, d+1)$
5.   **parallel:**
           for $j \leftarrow 1$ **to** r **do**
             if $j \neq k$ **then** $\mathcal{B}_{FW}(X_{kj}, U_{kk}, V_{kj}, d+1)$
           for $i \leftarrow 1$ **to** r **do**
             if $i \neq k$ **then** $\mathcal{C}_{FW}(X_{ik}, U_{ik}, V_{kk}, d+1)$
6.   **parallel for** $i \leftarrow 1$ **to** r **do**
7.   **parallel for** $j \leftarrow 1$ **to** r **do**
8.   **if** $i \neq k$ **and** $j \neq k$ **then**
9. $\mathcal{D}_{FW}(X_{ij}, U_{ik}, V_{kj}, d+1)$

$\mathcal{B}_{FW}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{B}_{loop\text{-}FW}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel for** $j \leftarrow 1$ **to** r **do**
           $\mathcal{B}_{FW}(X_{kj}, U_{kk}, V_{kj}, d+1)$
5.   **parallel for** $i \leftarrow 1$ **to** r **do**
6.   **parallel for** $j \leftarrow 1$ **to** r **do**
7.   **if** $i \neq k$ **then** $\mathcal{D}_{FW}(X_{ij}, U_{ik}, V_{kj}, d+1)$

$\mathcal{C}_{FW}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{C}_{loop\text{-}FW}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel for** $i \leftarrow 1$ **to** r **do**
           $\mathcal{C}_{FW}(X_{ik}, U_{ik}, V_{kk}, d+1)$
5.   **parallel for** $i \leftarrow 1$ **to** r **do**
6.   **parallel for** $j \leftarrow 1$ **to** r **do**
7.   **if** $j \neq k$ **then** $\mathcal{D}_{FW}(X_{ij}, U_{ik}, V_{kj}, d+1)$

$\mathcal{D}_{FW}(X, U, V)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{D}_{loop\text{-}FW}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel for** $i \leftarrow 1$ **to** r **do**
5.   **parallel for** $j \leftarrow 1$ **to** r **do**
6. $\mathcal{D}_{FW}(X_{ij}, U_{ik}, V_{kj}, d+1)$

Figure A.7: r-way divide-and-conquer algorithm for Floyd-Warshall algorithm.

Complexity analysis for the r-way $\mathcal{A}_{FW}$ algorithm

The r-way divide-and-conquer for Floyd-Warshall algorithm is given in Figure A.7. For simplicity of exposition we assume that n is a power of r. For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, let $W_f(n), Q_f(n), T_f(n)$, and $S_f(n)$ denote the total work, serial cache complexity, span, and parallel space consumption of r-way f_{FW} for the parameter n. Then

$$
\begin{aligned}
&W_\mathcal{A}(n) = W_\mathcal{B}(n) = W_\mathcal{C}(n) = W_\mathcal{D}(n) = \Theta(1) && \text{if } n = 1, \\
&W_\mathcal{A}(n) = rW_\mathcal{A}\left(\tfrac{n}{r}\right) + r(r-1)\left(W_\mathcal{B}\left(\tfrac{n}{r}\right) + W_\mathcal{C}\left(\tfrac{n}{r}\right)\right) + r(r-1)^2 W_\mathcal{D}\left(\tfrac{n}{r}\right) + \Theta(1) && \text{if } n > 1; \\
&W_\mathcal{B}(n) = r^2 W_\mathcal{B}\left(\tfrac{n}{r}\right) + r^2(r-1)W_\mathcal{D}\left(\tfrac{n}{r}\right) + \Theta(1) && \text{if } n > 1; \\
&W_\mathcal{C}(n) = r^2 W_\mathcal{C}\left(\tfrac{n}{r}\right) + r^2(r-1)W_\mathcal{D}\left(\tfrac{n}{r}\right) + \Theta(1) && \text{if } n > 1; \\
&W_\mathcal{D}(n) = r^3 W_\mathcal{D}\left(\tfrac{n}{r}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$

$$Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) = Q_{\mathcal{D}}(n) = \mathcal{O}\left(\frac{n^2}{B} + 1\right) \qquad \text{if } n^2 \leq \gamma M,$$

$$Q_{\mathcal{A}}(n) = rQ_{\mathcal{A}}\left(\frac{n}{r}\right) + r(r-1)\left(Q_{\mathcal{B}}\left(\frac{n}{r}\right) + Q_{\mathcal{C}}\left(\frac{n}{r}\right)\right) + r(r-1)^2 Q_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \quad \text{if } n^2 > \gamma_A M;$$

$$Q_{\mathcal{B}}(n) = r^2 Q_{\mathcal{B}}\left(\frac{n}{r}\right) + r^2(r-1)Q_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_B M;$$

$$Q_{\mathcal{C}}(n) = r^2 Q_{\mathcal{C}}\left(\frac{n}{r}\right) + r^2(r-1)Q_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_C M;$$

$$Q_{\mathcal{D}}(n) = r^3 Q_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_D M;$$

$$T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) = \mathcal{O}(1) \qquad \text{if } n = 1,$$

$$T_{\mathcal{A}}(n) = r\left(T_{\mathcal{A}}\left(\frac{n}{r}\right) + \max\{T_{\mathcal{B}}\left(\frac{n}{r}\right), T_{\mathcal{C}}\left(\frac{n}{r}\right)\} + T_{\mathcal{D}}\left(\frac{n}{r}\right)\right) + \Theta(1) \quad \text{if } n > 1;$$

$$T_{\mathcal{B}}(n) = r\left(T_{\mathcal{B}}\left(\frac{n}{r}\right) + T_{\mathcal{D}}\left(\frac{n}{r}\right)\right) + \Theta(1) \qquad \text{if } n > 1;$$

$$T_{\mathcal{C}}(n) = r\left(T_{\mathcal{C}}\left(\frac{n}{r}\right) + T_{\mathcal{D}}\left(\frac{n}{r}\right)\right) + \Theta(1) \qquad \text{if } n > 1;$$

$$T_{\mathcal{D}}(n) = rT_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n > 1;$$

$$S_{\mathcal{A}}(n) = S_{\mathcal{B}}(n) = S_{\mathcal{C}}(n) = S_{\mathcal{D}}(n) = \mathcal{O}(1) \qquad \text{if } n = 1,$$

$$S_{\mathcal{A}}(n) = \max\{S_{\mathcal{A}}\left(\frac{n}{r}\right), (r-1)\left(S_{\mathcal{B}}\left(\frac{n}{r}\right) + S_{\mathcal{C}}\left(\frac{n}{r}\right)\right), (r-1)^2 S_{\mathcal{D}}\left(\frac{n}{r}\right)\} + \Theta(1) \quad \text{if } n > 1;$$

$$S_{\mathcal{B}}(n) = \max\{rS_{\mathcal{B}}\left(\frac{n}{r}\right), r(r-1)S_{\mathcal{D}}\left(\frac{n}{r}\right)\} + \Theta(1) \qquad \text{if } n > 1;$$

$$S_{\mathcal{C}}(n) = \max\{rS_{\mathcal{C}}\left(\frac{n}{r}\right), r(r-1)S_{\mathcal{D}}\left(\frac{n}{r}\right)\} + \Theta(1) \qquad \text{if } n > 1;$$

$$S_{\mathcal{D}}(n) = r^2 S_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n > 1;$$

where, γ, γ_A, γ_B, γ_C, and γ_D are suitable constants.

> For the all-pairs shortest path problem, the r-way $\mathcal{A}_{FW}$ algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\dfrac{n^3}{B\left\lceil\frac{\sqrt{M}}{r}\right\rceil} + \dfrac{n^3}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil^3}\right)$, and $S_\infty(n) = \Theta(n^2)$.

A.4 Gaussian elimination without pivoting

Gaussian elimination is a method to solve a system of linear equations with n equations in n unknowns. The algorithm uses a series of divisions which leads to errors. Also, the algorithm also does not work if some of the diagonal elements are zeros. A technique called pivoting (either partial or complete) is typically used to make the algorithm more robust and reduce rounding errors. Here, we only consider the version of the algorithm without pivoting. Gaussian elimination without pivoting is the core component of LU factorization without pivoting.

The system of n linear equations with n unknowns can be written in the form $\sum a_{ij}x_i = b_i$ for all $i, j \in [1, n]$, where a_{ij}s and b_is are known real numbers and x_is are unknown real numbers. The system can also be written in the matrix form $AX = B$, where $A_{n\times n}$, $X_{n\times 1}$, and $B_{n\times 1}$ are matrices and vector X is to be computed.

A serial and a parallel iterative algorithm for the Gaussian elimination method (without the backward substitution) are given in Figure A.8. The structure of the three loops seems similar to the matrix multiplication algorithm but there are differences. In matrix multiplication, the i-j-k loops can be ordered in any of the $3! = 6$ ways. However, in the computation of path costs, the k-loop must always be the outermost loop.

LOOP-GAUSSIAN-ELIMINATION(D)
1. **for** $k \leftarrow 1$ **to** $n - 1$ **do**
2.   **for** $i \leftarrow k + 1$ **to** n **do**
3.   **for** $j \leftarrow k$ **to** n **do**
4. $d[i, j] \leftarrow d[i, j] - (d[i, k] \times d[k, j]) / d[k, k]$

PAR-GAUSSIAN-ELIMINATION(D)
1. **for** $k \leftarrow 1$ **to** $n - 1$ **do**
2.   **parallel for** $i \leftarrow k + 1$ **to** n **do**
3.   **parallel for** $j \leftarrow k$ **to** n **do**
4. $d[i, j] \leftarrow d[i, j] - (d[i, k] \times d[k, j]) / d[k, k]$

$\mathcal{A}_{GE}(X)$
1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}GE}(X)$
   else
2. $\mathcal{A}_{GE}(X_{11})$
3.   **par:** $\mathcal{B}_{GE}(X_{12}, X_{11})$, $\mathcal{C}_{GE}(X_{21}, X_{11})$
4. $\mathcal{D}_{GE}(X_{22}, X_{21}, X_{12})$
5. $\mathcal{A}_{GE}(X_{22})$

$\mathcal{B}_{GE}(X, U)$
1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}GE}(X, U)$
   else
2.   **par:** $\mathcal{B}_{GE}(X_{11}, U_{11})$, $\mathcal{B}_{GE}(X_{12}, U_{11})$
3.   **par:** $\mathcal{D}_{GE}(X_{21}, U_{21}, X_{11})$, $\mathcal{D}_{GE}(X_{22}, U_{21}, X_{12})$
4.   **par:** $\mathcal{B}_{GE}(X_{21}, U_{22})$, $\mathcal{B}_{GE}(X_{22}, U_{22})$

$\mathcal{C}_{GE}(X, V)$
1. **if** X is a small matrix **then** $\mathcal{C}_{loop\text{-}GE}(X, V)$
   else
2.   **par:** $\mathcal{C}_{GE}(X_{11}, V_{11})$, $\mathcal{C}_{GE}(X_{21}, V_{11})$
3.   **par:** $\mathcal{D}_{GE}(X_{12}, X_{11}, V_{12})$, $\mathcal{D}_{GE}(X_{22}, X_{21}, V_{12})$
4.   **par:** $\mathcal{C}_{GE}(X_{12}, V_{22})$, $\mathcal{C}_{GE}(X_{22}, V_{22})$

$\mathcal{D}_{GE}(X, U, V)$
1. **if** X is a small matrix **then** $\mathcal{D}_{loop\text{-}GE}(X, U, V)$
   else
2.   **par:** $\mathcal{D}_{GE}(X_{11}, U_{11}, V_{11})$, $\mathcal{D}_{GE}(X_{12}, U_{11}, V_{12})$,
          $\mathcal{D}_{GE}(X_{21}, U_{21}, V_{11})$, $\mathcal{D}_{GE}(X_{22}, U_{21}, V_{12})$
3.   **par:** $\mathcal{D}_{GE}(X_{11}, U_{12}, V_{21})$, $\mathcal{D}_{GE}(X_{12}, U_{12}, V_{22})$,
          $\mathcal{D}_{GE}(X_{21}, U_{22}, V_{21})$, $\mathcal{D}_{GE}(X_{22}, U_{22}, V_{22})$

Figure A.8: Top left: Serial and parallel iterative algorithms for Gaussian elimination. Top right: Dependency structure for Gaussian elimination. Bottom: Divide-and-conquer algorithm.

Figures A.8 and A.9 show a divide-and-conquer implementation of the Gaussian elimination algorithm. The implementation consists of four recursive divide-and-conquer functions. The number of times $\mathcal{A}_{loop-GE}$, $\mathcal{B}_{loop-GE}$, $\mathcal{C}_{loop-GE}$, and $\mathcal{D}_{loop-GE}$ will be called are $\Theta(n), \Theta(n^2), \Theta(n^2)$, and $\Theta(n^3)$ respectively. Hence, function $\mathcal{D}_{GE}$ is dominating.

Complexity analysis for the PAR-GAUSSIAN-ELIMINATION algorithm
For $f \in \{\text{PAR-GAUSSIAN-ELIMINATION}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{FW} on a matrix of size $n \times n$. $W_f(n)$ is $\Theta(n^3)$. $Q_f(n)$ is computed as $Q_f(n) = \Theta\left(n \cdot n \cdot \left(\frac{n}{B} + 1\right)\right) = \Theta\left(\frac{n^3}{B} + n^2\right)$. The span is $T_f(n) = \Theta(n \cdot (\log n + (\log n + \Theta(1)))) = \Theta(n \log n)$. The properties of the given semiring implies that $d[i, j]$ for all pairs of vertices $v_i, v_j \in V$ uses only $\Theta(n^2)$ space as shown in Figure A.5.

For the Gaussian elimination algorithm, the PAR-GAUSSIAN-ELIMINATION implementation achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \Theta\left(\frac{n^3}{B} + n^2\right)$, $T_\infty(n) = \Theta(n \log n)$, parallelism $= \Theta\left(\frac{n^2}{\log n}\right)$, and $S_\infty(n) = \Theta(n^2)$.

Complexity analysis for the 2-way $\mathcal{A}_{GE}$ algorithm
For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{GE} on a matrix of size $n \times n$. Then

Figure A.9: A divide-and-conquer algorithm $\mathcal{A}_{GE}$ for Gaussian elimination.

$$
\begin{aligned}
Q_{\mathcal{A}}(n) &= Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) = Q_{\mathcal{D}}(n) = \mathcal{O}\left(\frac{n^2}{B} + n\right) && \text{if } n^2 \leq \gamma M, \\
Q_{\mathcal{A}}(n) &= 2Q_{\mathcal{A}}\left(\frac{n}{2}\right) + Q_{\mathcal{B}}\left(\frac{n}{2}\right) + Q_{\mathcal{C}}\left(\frac{n}{2}\right) + Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_A M; \\
Q_{\mathcal{B}}(n) &= 4Q_{\mathcal{B}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_B M; \\
Q_{\mathcal{C}}(n) &= 4Q_{\mathcal{C}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_C M; \\
Q_{\mathcal{D}}(n) &= 8Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_D M;
\end{aligned}
$$

159

$$\begin{aligned}
T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) = T_{\mathcal{D}}(n) &= \mathcal{O}(1) && \text{if } n = 1,\\
T_{\mathcal{A}}(n) = 2T_{\mathcal{A}}\left(\tfrac{n}{2}\right) + \max\left\{T_{\mathcal{B}}\left(\tfrac{n}{2}\right), T_{\mathcal{C}}\left(\tfrac{n}{2}\right)\right\} + T_{\mathcal{D}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;\\
T_{\mathcal{B}}(n) = 2T_{\mathcal{B}}\left(\tfrac{n}{2}\right) + T_{\mathcal{D}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;\\
T_{\mathcal{C}}(n) = 2T_{\mathcal{C}}\left(\tfrac{n}{2}\right) + T_{\mathcal{D}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;\\
T_{\mathcal{D}}(n) = 2T_{\mathcal{D}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}$$

where, γ, γ_A, γ_B, γ_C and γ_D are suitable constants. Solving, $Q_{\mathcal{A}}(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$ and $T_{\mathcal{A}}(n) = \mathcal{O}\left(n\log^2 n\right)$.

> For the Gaussian elimination algorithm, the $\mathcal{A}_{GE}$ implementation achieves $T_1(n) = \Theta\left(n^3\right)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$, $T_\infty(n) = \Theta\left(n\log^2 n\right)$, parallelism $= \Theta\left(\frac{n^2}{\log^2 n}\right)$, and $S_\infty(n) = \Theta\left(n^2\right)$.

$\mathcal{A}_{GE}(X, U, V, W, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{A}_{loop\text{-}GE}(X, U, V, W)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4. $\mathcal{A}_{GE}(X_{kk}, U_{kk}, V_{kk}, W_k, d+1)$
5.   **parallel:**
             for $j \leftarrow k+1$ **to** r **do**
              $\mathcal{B}_{GE}(X_{kj}, U_{kk}, V_{kj}, W_k, d+1)$
             for $i \leftarrow k+1$ **to** r **do**
              $\mathcal{C}_{GE}(X_{ik}, U_{ik}, V_{kk}, W_k, d+1)$
6.   **parallel for** $i \leftarrow k+1$ **to** r **do**
7.   **parallel for** $j \leftarrow k+1$ **to** r **do**
8. $\mathcal{D}_{GE}(X_{ij}, U_{ik}, V_{kj}, W_k, d+1)$

$\mathcal{D}_{GE}(X, U, V, W, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{D}_{loop\text{-}GE}(X, U, V, W)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel for** $i \leftarrow 1$ **to** r **do**
5.   **parallel for** $j \leftarrow 1$ **to** r **do**
6. $\mathcal{D}_{GE}(X_{ij}, U_{ik}, V_{kj}, W_k, d+1)$

$\mathcal{B}_{GE}(X, U, V, W, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{B}_{loop\text{-}GE}(X, U, V, W)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel for** $j \leftarrow 1$ **to** r **do**
             $\mathcal{B}_{GE}(X_{kj}, U_{kk}, V_{kj}, W_k, d+1)$
5.   **parallel for** $i \leftarrow k+1$ **to** r **do**
6.   **parallel for** $j \leftarrow 1$ **to** r **do**
7. $\mathcal{D}_{GE}(X_{ij}, U_{ik}, V_{kj}, W_k, d+1)$

$\mathcal{C}_{GE}(X, U, V, W, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{C}_{loop\text{-}GE}(X, U, V, W)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel for** $i \leftarrow 1$ **to** r **do**
             $\mathcal{C}_{GE}(X_{ik}, U_{ik}, V_{kk}, W_k, d+1)$
5.   **parallel for** $i \leftarrow 1$ **to** r **do**
6.   **parallel for** $j \leftarrow k+1$ **to** r **do**
7. $\mathcal{D}_{GE}(X_{ij}, U_{ik}, V_{kj}, W_k, d+1)$

Figure A.10: An r-way divide-and-conquer algorithm for Gaussian elimination.

Complexity analysis for the r-way $\mathcal{A}_{GE}$ algorithm

An r-way divide-and-conquer algorithm for Gaussian elimination is given in Figure D.3. For simplicity of exposition we assume that n is a power of r. For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, let $W_f(n), Q_f(n), T_f(n)$, and $S_f(n)$ denote the total work, serial cache complexity, span, and parallel space consumption of r-way f_{GE} for the parameter n. Then

$$\begin{aligned}
W_{\mathcal{A}}(n) = W_{\mathcal{B}}(n) = W_{\mathcal{C}}(n) = W_{\mathcal{D}}(n) &= \Theta(1) && \text{if } n = 1,\\
W_{\mathcal{A}}(n) = rW_{\mathcal{A}}\left(\tfrac{n}{r}\right) + \tfrac{r(r-1)}{2}\left(W_{\mathcal{B}}\left(\tfrac{n}{r}\right) + W_{\mathcal{C}}\left(\tfrac{n}{r}\right)\right) + \tfrac{r(r-1)(2r-1)}{6}W_{\mathcal{D}}\left(\tfrac{n}{r}\right) + \Theta(1) && \text{if } n > 1;\\
W_{\mathcal{B}}(n) = r^2 W_{\mathcal{B}}\left(\tfrac{n}{r}\right) + \tfrac{r^2(r-1)}{2}W_{\mathcal{D}}\left(\tfrac{n}{r}\right) + \Theta(1) && \text{if } n > 1;\\
W_{\mathcal{C}}(n) = r^2 W_{\mathcal{C}}\left(\tfrac{n}{r}\right) + \tfrac{r^2(r-1)}{2}W_{\mathcal{D}}\left(\tfrac{n}{r}\right) + \Theta(1) && \text{if } n > 1;\\
W_{\mathcal{D}}(n) = r^3 W_{\mathcal{D}}\left(\tfrac{n}{r}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}$$

$$Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) = Q_{\mathcal{D}}(n) = \mathcal{O}\left(\frac{n^2}{B} + 1\right) \qquad \text{if } n^2 \leq \gamma M,$$

$$Q_{\mathcal{A}}(n) = rQ_{\mathcal{A}}\left(\frac{n}{r}\right) + \frac{r(r-1)}{2}\left(Q_{\mathcal{B}}\left(\frac{n}{r}\right) + Q_{\mathcal{C}}\left(\frac{n}{r}\right)\right) + \frac{r(r-1)(2r-1)}{6}Q_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \quad \text{if } n^2 > \gamma_A M;$$

$$Q_{\mathcal{B}}(n) = r^2 Q_{\mathcal{B}}\left(\frac{n}{r}\right) + \frac{r^2(r-1)}{2}Q_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_B M;$$

$$Q_{\mathcal{C}}(n) = r^2 Q_{\mathcal{C}}\left(\frac{n}{r}\right) + \frac{r^2(r-1)}{2}Q_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_C M;$$

$$Q_{\mathcal{D}}(n) = r^3 Q_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_D M;$$

$$T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) = \mathcal{O}(1) \qquad \text{if } n = 1,$$

$$T_{\mathcal{A}}(n) = rT_{\mathcal{A}}\left(\frac{n}{r}\right) + (r-1)\max\left\{T_{\mathcal{B}}\left(\frac{n}{r}\right), T_{\mathcal{C}}\left(\frac{n}{r}\right)\right\} + (r-1)T_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \quad \text{if } n > 1;$$

$$T_{\mathcal{B}}(n) = rT_{\mathcal{B}}\left(\frac{n}{r}\right) + (r-1)T_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n > 1;$$

$$T_{\mathcal{C}}(n) = rT_{\mathcal{C}}\left(\frac{n}{r}\right) + (r-1)T_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n > 1;$$

$$T_{\mathcal{D}}(n) = rT_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n > 1;$$

$$S_{\mathcal{A}}(n) = S_{\mathcal{B}}(n) = S_{\mathcal{C}}(n) = S_{\mathcal{D}}(n) = \mathcal{O}(1) \qquad \text{if } n = 1,$$

$$S_{\mathcal{A}}(n) = \max\left\{S_{\mathcal{A}}\left(\frac{n}{r}\right), (r-1)\left(S_{\mathcal{B}}\left(\frac{n}{r}\right) + S_{\mathcal{C}}\left(\frac{n}{r}\right)\right), (r-1)^2 S_{\mathcal{D}}\left(\frac{n}{r}\right)\right\} + \Theta(1) \quad \text{if } n > 1;$$

$$S_{\mathcal{B}}(n) = \max\left\{rS_{\mathcal{B}}\left(\frac{n}{r}\right), r(r-1)S_{\mathcal{D}}\left(\frac{n}{r}\right)\right\} + \Theta(1) \qquad \text{if } n > 1;$$

$$S_{\mathcal{C}}(n) = \max\left\{rS_{\mathcal{C}}\left(\frac{n}{r}\right), r(r-1)S_{\mathcal{D}}\left(\frac{n}{r}\right)\right\} + \Theta(1) \qquad \text{if } n > 1;$$

$$S_{\mathcal{D}}(n) = r^2 S_{\mathcal{D}}\left(\frac{n}{r}\right) + \Theta(1) \qquad \text{if } n > 1;$$

where, γ, γ_A, γ_B, γ_C, and γ_D are suitable constants.

> For the Gaussian elimination algorithm, the r-way $\mathcal{A}_{GE}$ algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\dfrac{n^3}{B\left\lceil\frac{\sqrt{M}}{r}\right\rceil} + \dfrac{n^3}{\left\lceil\frac{\sqrt{M}}{r}\right\rceil^3}\right)$, and $S_\infty(n) = \Theta(n^2)$.

A.5 Sequence alignment with gap penalty

The sequence alignment problem with gap penalty or often called the gap problem [Galil and Giancarlo, 1989, Galil and Park, 1994, Waterman et al., 1995] is a generalization of the edit distance problem that arises in molecular biology, geology, and speech recognition. When transforming a string $X = x_1 x_2 \ldots x_m$ into another string $Y = y_1 y_2 \ldots y_n$, a sequence of consecutive deletes corresponds to a gap in X, and a sequence of consecutive inserts corresponds to a gap in Y. In many applications the cost of such a gap is not necessarily equal to the sum of the costs of each individual deletion (or insertion) in that gap. In order to handle this general case two new cost functions w and w' are defined, where $w(p, q)$ $(0 \leq p < q \leq m)$ is the cost of deleting $x_{p+1} \ldots x_q$ from X, and $w'(p, q)$ $(0 \leq p < q \leq n)$ is the cost of inserting $y_{p+1} \ldots y_q$ into X. The substitution function $S(x_i, y_j)$ is the same as that of the standard edit distance problem.

Let $G[i, j]$ denote the minimum cost of transforming $X_i = x_1 x_2 \ldots x_i$ into $Y_j = y_1 y_2 \ldots y_j$ (where $0 \leq i \leq m$ and $0 \leq j \leq n$) under this general setting. Then

$$G[i,j] = \begin{cases} 0 & \text{if } i = j = 0, \\ w(0,j) & \text{if } i = 0,\, 1 \leq j \leq n, \\ w'(0,i) & \text{if } j = 0,\, 1 \leq i \leq m, \\ \min \left\{ \begin{array}{l} G[i-1,j-1] + S(x_i,\ y_j), \\ \min_{0 \leq q < j} \left\{ G[i,q] + w(q,j) \right\}, \\ \min_{0 \leq p < i} \left\{ G[p,j] + w'(p,i) \right\} \end{array} \right\} & \text{if } i,j > 0. \end{cases} \qquad \text{(A.5)}$$

Assuming $m = n$, this problem can be solved in $\Theta\left(n^3\right)$ time using $\Theta\left(n^2\right)$ space [Galil and Giancarlo, 1989]. In the rest of this section we will assume for simplicity that $m = n = 2^\ell$ for some integer $\ell \geq 0$.

Figure A.11 shows the dependency structure of the DP. A serial and a parallel iterative algorithm for the gap problem are also given in the figure. The two algorithms work for a generic $m \times n$ DP table.

In Figures A.11 and A.12, a divide-and-conquer algorithm [Chowdhury and Ramachandran, 2006] is given for solving the gap problem. The algorithm consists of three recursive divide-and-conquer functions named $\mathcal{A}_{gap}$, $\mathcal{B}_{gap}$ and $\mathcal{C}_{gap}$. The $\mathcal{C}_{gap}$ function has the structure of recursive matrix multiplication. The initial call to the algorithm is $\mathcal{A}(G)$, where G is the $n \times n$ DP table. For simplicity, we consider square matrices. It is possible to modify the algorithm to make it work for generic $m \times n$ matrices.

LOOP-GAP(G, n)

1.   **for** $i \leftarrow 1$ **to** m **do**
2.   **for** $j \leftarrow 1$ **to** n **do**
3. $G[i,j] \leftarrow G[i-1,j-1] + S(x_i, y_i)$
4.   **for** $k \leftarrow 0$ **to** $j-1$ **do**
5. $G[i,j] \leftarrow \min\left(G[i,j], G[i,k] + w(k,j)\right)$
6.   **for** $k \leftarrow 0$ **to** $i-1$ **do**
7. $G[i,j] \leftarrow \min\left(G[i,j], G[k,j] + w'(k,i)\right)$

PAR-LOOP-GAP(G, n)

1.   **for** $t \leftarrow 2$ **to** $m+n$ **do**
2.   **parallel for** $i \leftarrow \max(1, t-n)$ **to** $\min(t-1, m)$ **do**
3. $j \leftarrow t - i$
4. $G[i,j] \leftarrow G[i-1,j-1] + S(x_i, y_i)$
5.   **for** $k \leftarrow 0$ **to** $j-1$ **do**
6. $G[i,j] \leftarrow \min\left(G[i,j], G[i,k] + w(k,j)\right)$
7.   **for** $k \leftarrow 0$ **to** $i-1$ **do**
8. $G[i,j] \leftarrow \min\left(G[i,j], G[k,j] + w'(k,i)\right)$

$\mathcal{A}_{gap}(X)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}gap}(X)$
   else
2. $\mathcal{A}_{gap}(X_{11})$
3.   **parallel:** $\mathcal{B}_{gap}(X_{12}, X_{11})$, $\mathcal{C}_{gap}(X_{21}, X_{11})$
4.   **parallel:** $\mathcal{A}_{gap}(X_{12})$, $\mathcal{A}_{gap}(X_{21})$
5. $\mathcal{B}_{gap}(X_{22}, X_{21})$
6. $\mathcal{C}_{gap}(X_{22}, X_{12})$
7. $\mathcal{A}_{gap}(X_{22})$

$\mathcal{B}_{gap}(X, U)$

1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}gap}(X, U)$
   else
2.   **parallel:** $\mathcal{B}_{gap}(X_{11}, U_{11})$, $\mathcal{B}_{gap}(X_{12}, U_{11})$, $\mathcal{B}_{gap}(X_{21}, U_{21})$, $\mathcal{B}_{gap}(X_{22}, U_{21})$
3.   **parallel:** $\mathcal{B}_{gap}(X_{11}, U_{12})$, $\mathcal{B}_{gap}(X_{12}, U_{12})$, $\mathcal{B}_{gap}(X_{21}, U_{22})$, $\mathcal{B}_{gap}(X_{22}, U_{22})$

$\mathcal{C}_{gap}(X, V)$

1. **if** X is a small matrix **then** $\mathcal{C}_{loop\text{-}gap}(X, V)$
   else
2.   **parallel:** $\mathcal{C}_{gap}(X_{11}, V_{11})$, $\mathcal{C}_{gap}(X_{12}, V_{12})$, $\mathcal{C}_{gap}(X_{21}, V_{11})$, $\mathcal{C}_{gap}(X_{22}, V_{12})$
3.   **parallel:** $\mathcal{C}_{gap}(X_{11}, V_{21})$, $\mathcal{C}_{gap}(X_{12}, V_{22})$, $\mathcal{C}_{gap}(X_{21}, V_{21})$, $\mathcal{C}_{gap}(X_{22}, V_{22})$

Figure A.11: Top: Serial and parallel iterative algorithms to solve the gap problem. Bottom: Dependency graph and the divide-and-conquer algorithm.

Complexity analysis for the PAR-LOOP-GAP algorithm

For $f \in \{\text{PAR-LOOP-GAP}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{gap} on a matrix of size $m \times n$. The k-loop scans elements of the DP table in both vertical
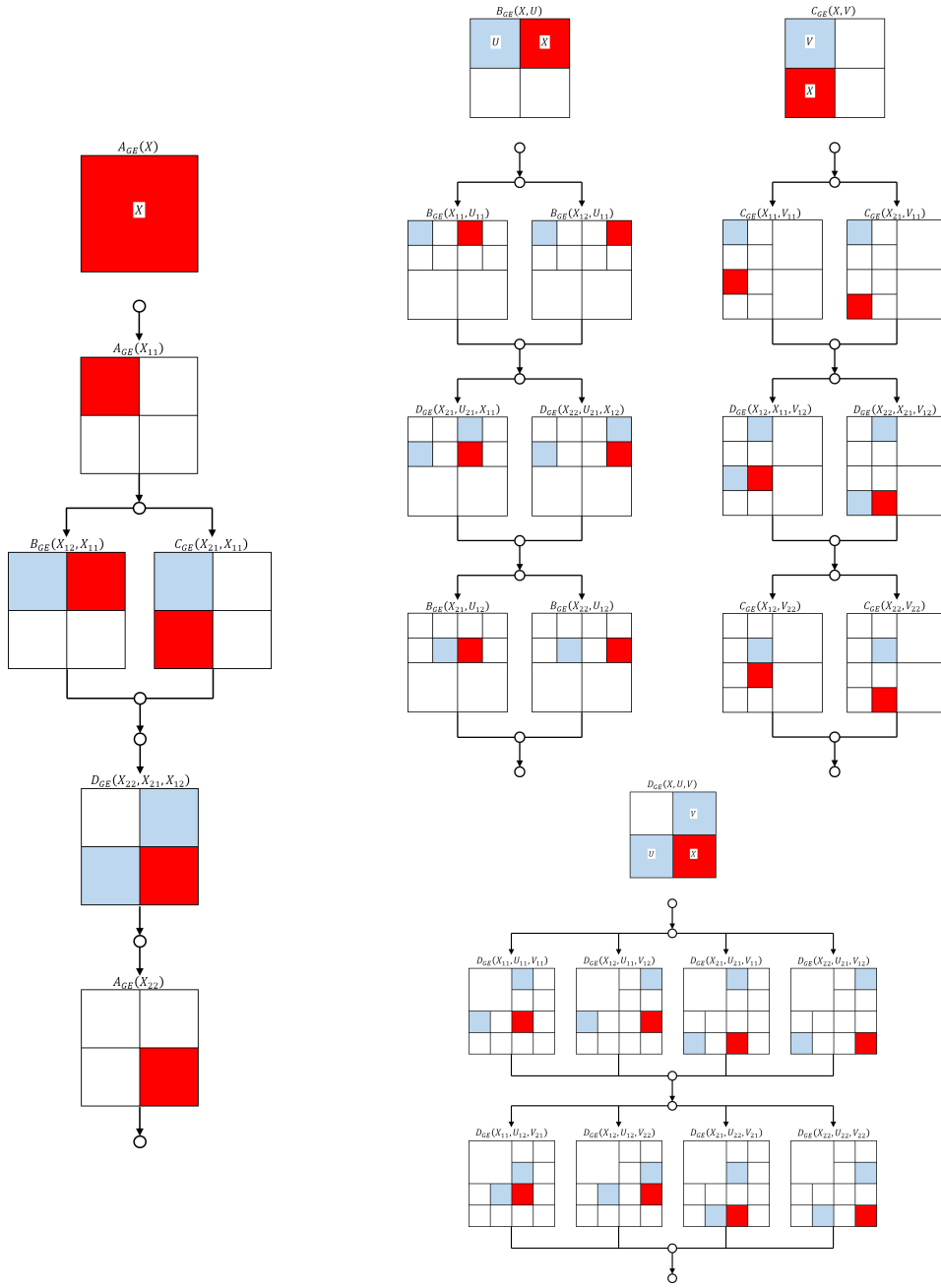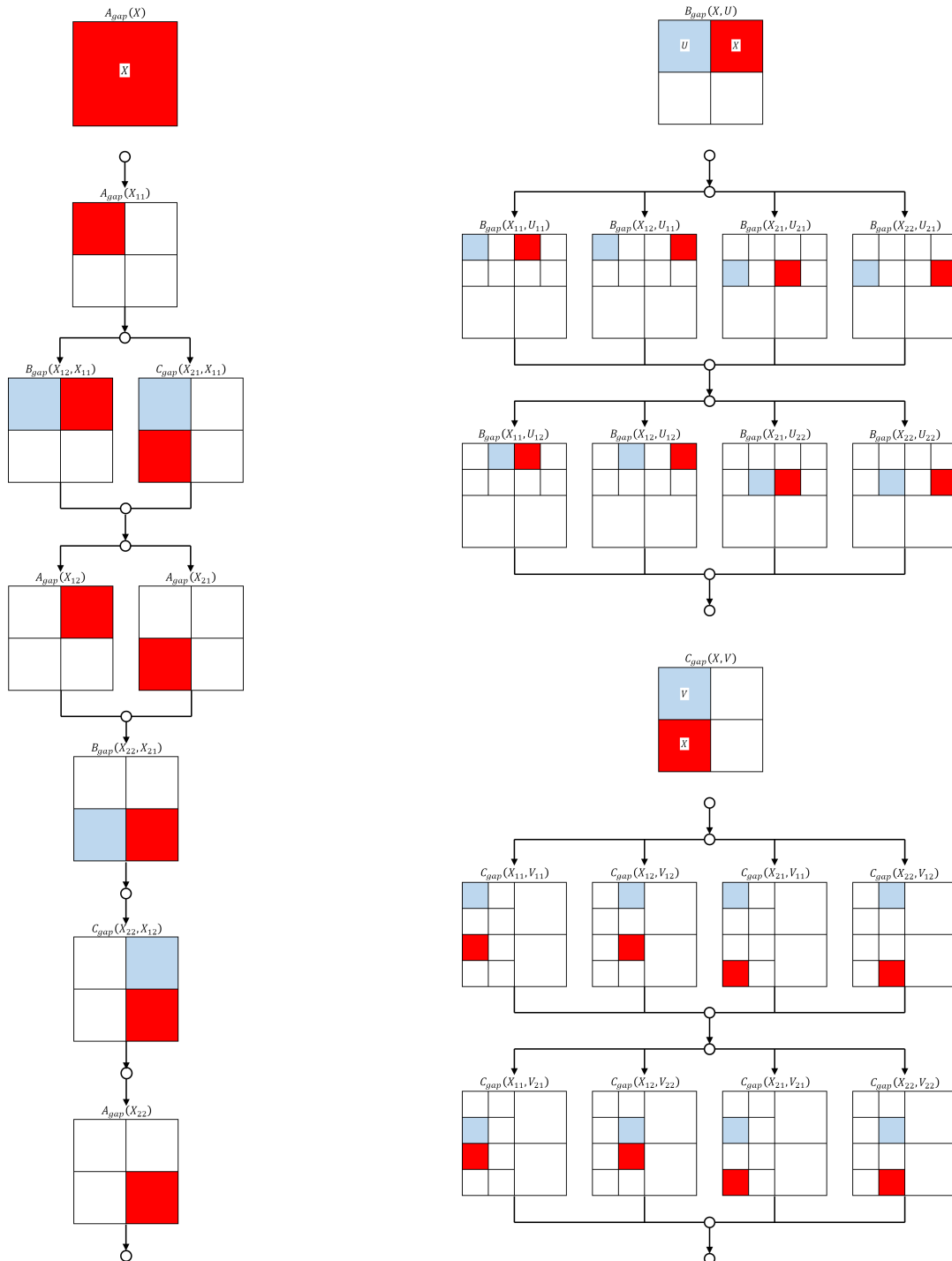
Figure A.12: A divide-and-conquer algorithm for solving the gap problem.

and horizontal direction. Hence, $Q_f(n) = \mathcal{O}\left(m \cdot n \cdot \left(\frac{\max(m,n)}{B} + \min(m,n)\right)\right)$ assuming that we use row-major order if $m > n$ and use column-major order if $n > m$. We find span as follows. The first loop runs for $\Theta(m+n)$ time. The second loop takes $\Theta(\log(m+n))$ to divide the work to processors. The third loop takes $\Theta(\max(m,n))$ time. Hence, $T_f(n) = \Theta((m+n) \cdot (\log(m+n) + \max(m,n))) = \Theta((\max(m,n))^2)$.

> For the sequence alignment with gap penalty problem (or gap problem), the PAR-LOOP-GAP algorithm achieves $T_1(n) = \Theta(mn \cdot \max(m,n))$, $Q_1(n) = \mathcal{O}\left(mn \cdot \left(\frac{\max(m,n)}{B} + \min(m,n)\right)\right)$, $T_\infty(n) = \Theta((\max(m,n))^2)$, parallelism $= \Theta\left(\frac{mn}{\max(m,n)}\right)$, and $S_\infty(n) = \Theta(mn)$.

Complexity analysis for the 2-way $\mathcal{A}_{gap}$ algorithm

For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{gap} on a matrix of size $n \times n$. Then

$$
\begin{aligned}
Q_\mathcal{A}(n) = Q_\mathcal{B}(n) = Q_\mathcal{C}(n) &= \mathcal{O}\left(\frac{n^2}{B} + n\right) && \text{if } n^2 \leq \gamma M, \\
Q_\mathcal{A}(n) &= 4Q_\mathcal{A}\left(\frac{n}{2}\right) + 2Q_\mathcal{B}\left(\frac{n}{2}\right) + 2Q_\mathcal{C}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_A M; \\
Q_\mathcal{B}(n) &= 8Q_\mathcal{B}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_B M; \\
Q_\mathcal{C}(n) &= 8Q_\mathcal{C}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_C M;
\end{aligned}
$$

$$
\begin{aligned}
T_\mathcal{A}(n) = T_\mathcal{B}(n) = T_\mathcal{C}(n) &= \mathcal{O}(1) && \text{if } n = 1, \\
T_\mathcal{A}(n) &= 3T_\mathcal{A}\left(\frac{n}{2}\right) + \max\left\{T_\mathcal{B}\left(\frac{n}{2}\right), T_\mathcal{C}\left(\frac{n}{2}\right)\right\} + T_\mathcal{B}\left(\frac{n}{2}\right) + T_\mathcal{C}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
T_\mathcal{B}(n) &= 2T_\mathcal{B}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
T_\mathcal{C}(n) &= 2T_\mathcal{C}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$

where, γ, γ_A, γ_B, and γ_C are suitable constants. Solving, $Q_\mathcal{A}(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$ and $T_\mathcal{A}(n) = \mathcal{O}\left(n^{\log 3}\right)$.

> For the sequence alignment with gap penalty problem (or gap problem), the 2-way $\mathcal{A}_{gap}$ algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$, $T_\infty(n) = \Theta\left(n^{\log 3}\right)$, parallelism $= \Theta\left(n^{3-\log 3}\right)$, and $S_\infty(n) = \Theta(n^2)$.
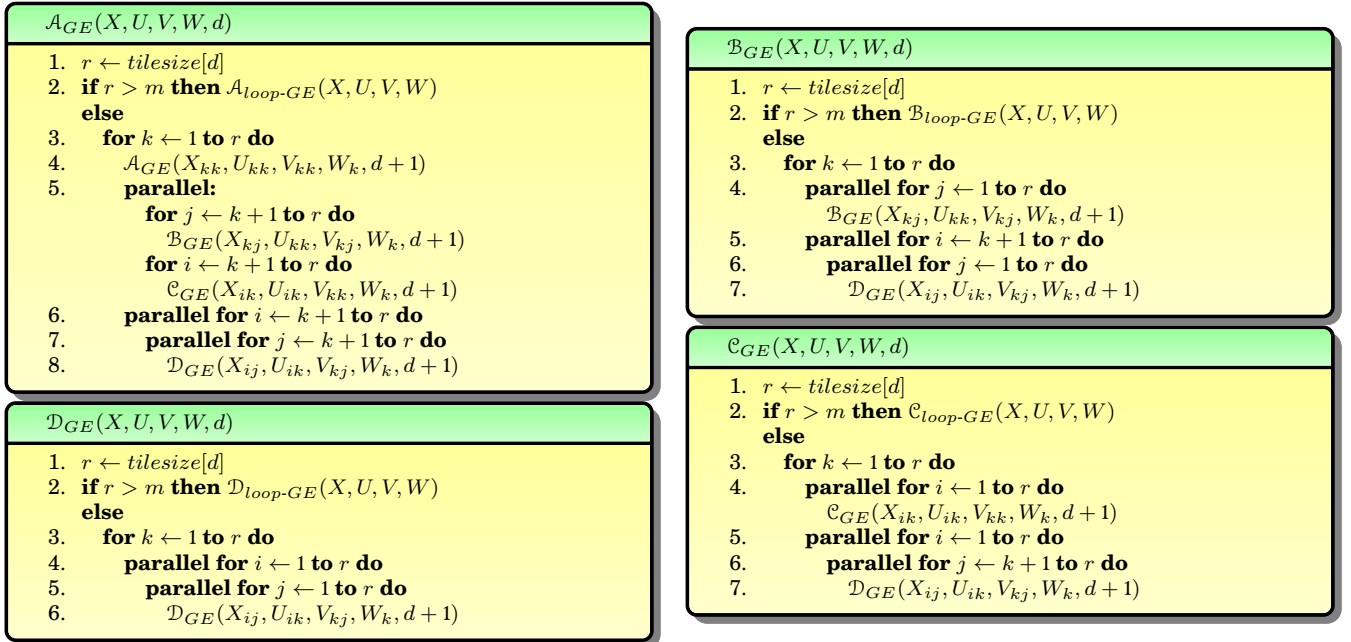
A.6 Protein accordion folding

A protein can be viewed as a string $P[1 : n]$ over the alphabet { A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V } of amino acids[1]. A protein sequence is never straight, and instead it folds itself in a way that minimizes the potential energy. Some of the amino acids (e.g., A, I, L, F, G, P, V)[2] are called hydrophobic as they do not like to be in contact

---

[1]Amino acids: Alanine (A), Arginine (R), Asparagine (N), Aspartic acid (D), Cysteine (C), Glutamic acid (E), Glutamine (Q), Glycine (G), Histidine (H), Isoleucine (I), Leucine (L), Lysine (K), Methionine (M), Phenylalanine (F), Proline (P), Serine (S), Threonine (T), Tryptophan (W), Tyrosine (Y), Valine (V).

[2]Alanine (A), Isoleucine (I), Leucine (L), Phenylalanine (F), Glycine (G), Proline (P), Valine (V).

$\mathcal{A}_{gap}(X,U,d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{A}_{loop\text{-}gap}(X,U)$
   else
3. $\mathcal{A}_{gap}(X_{11},U_{11},d+1)$
4.   **for** $k \leftarrow 1$ **to** $2r-2$ **do**
5.   **parallel for** $i \leftarrow \max(1,k-r+1)$ **to** $\min(k,r)$ **do**
6.   **parallel for** $j \leftarrow k-i+2$ **to** r **do**
7. $\mathcal{B}_{gap}(X_{ij},U_{i(k-i+1)},d+1)$
8.   **parallel for** $j \leftarrow \max(1,k-r+1)$ **to** $\min(k,r)$ **do**
9.   **parallel for** $i \leftarrow k-i+2$ **to** r **do**
10. $\mathcal{C}_{gap}(X_{ij},U_{(k-j+1)j},d+1)$
11.   **parallel for** $i \leftarrow \max(1,k-r+2)$ **to** $\min(k+1,r)$ **do**
12. $j = k-i+2$
13. $\mathcal{A}_{gap}(X_{ij},U_{ij},d+1)$

$\mathcal{B}_{gap}(X,U,d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{B}_{loop\text{-}gap}(X,U)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel for** $i \leftarrow 1$ **to** r **do**
5.   **parallel for** $j \leftarrow 1$ **to** r **do**
6. $\mathcal{B}_{gap}(X_{ij},U_{ik},d+1)$

$\mathcal{C}_{gap}(X,V,d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r > m$ **then** $\mathcal{C}_{loop\text{-}gap}(X,V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel for** $i \leftarrow 1$ **to** r **do**
5.   **parallel for** $j \leftarrow 1$ **to** r **do**
6. $\mathcal{C}_{gap}(X_{ij},V_{kj},d+1)$

Figure A.13: An r-way divide-and-conquer algorithm for sequence alignment with gap penalty problem.

with water (solvent). A desire to minimize the total hydrophobic area exposed to water is a major driving force behind the folding process. In a folded protein hydrophobic amino acids tend to clump together in order to reduce water-exposed hydrophobic area.

We assume for simplicity that a protein is folded into a 2-D square lattice in such a way that the number of pairs of hydrophobic amino acids that are next to each other in the grid (vertically or horizontally) without being next to each other in the protein sequence is maximized. We also assume that the fold is always an *accordion fold* where the sequence first goes straight down, then straight up, then again straight down, and so on (see Figure A.14).
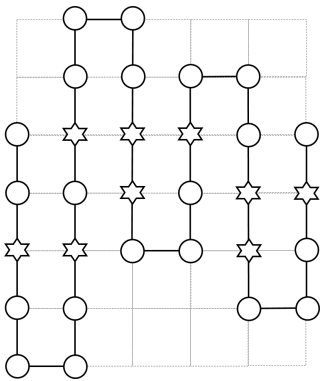
Figure A.14: A protein accordion fold where each star represents a hydrophobic amino acid and each circle a hydrophilic one. The accordion score of this folded sequence is 4 which is not the maximum possible score for this sequence.
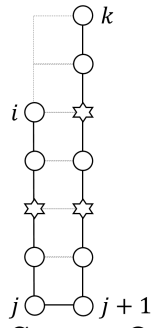
Figure A.15: SCORE-ONE-FOLD(i,j,k) counts the number of aligned hydrophobic amino acids when the protein segment $\mathcal{P}[i:k]$ is folded only once at indices $(j,j+1)$. In this figure, each star represents a hydrophobic amino acid and each circle a hydrophilic one.

The recurrence below shows how to compute the optimal *accordion score* of the protein segment $P[i:j]$. The optimal score for the entire sequence is given by $\max_{1<j\leq n}\{\text{SCORE}[1,j]\}$.

$$\text{SCORE}[i,j] = \begin{cases} 0 & \text{if } j \geq n-1, \\ \max_{j+1 < k \leq n} \{\text{SCORE-ONE-FOLD}(i,j,k) + \text{SCORE}[j+1,k]\} & \text{otherwise.} \end{cases} \quad \text{(A.6)}$$

The function $\text{SCORE-ONE-FOLD}(i,j,k)$ counts the number of aligned hydrophobic amino acids when the protein segment $P[i:k]$ is folded only once at indices $(j, j+1)$. The function is illustrated graphically in Figure A.15. Observe that

$$\text{SCORE-ONE-FOLD}(i,j,k) = \begin{cases} \text{SCORE-ONE-FOLD}(1,j,k) & \text{if } k \leq 2j-i+1, \\ \text{SCORE-ONE-FOLD}(1,j,2j-i+1) & \text{otherwise.} \end{cases} \quad \text{(A.7)}$$

$\text{SCORE-ONE-FOLD}(i,j,k,P)$

1. $c \leftarrow 0$
2. **for** $l \leftarrow 1$ **to** $\min(j-i, k-j-1)$ **do**
3.   **if** $\text{HYDROPHOBIC}(P[j-l])$ **and** $\text{HYDROPHOBIC}(P[j+1+l])$ **then** $c \leftarrow c+1$
4. **return** c

Figure A.16: Count the number of aligned hydrophobic amino acids when a protein sequence is folded once.

Hence, in $\Theta(n^2)$ time one can precompute an array $\text{SOF}[1:n, 1:n]$ such that for all $1 \leq i < j < k-1 < n$, $\text{SCORE-ONE-FOLD}(i,j,k) = \text{SOF}[j+1, \min\{k, 2j-i+1\}]$. Thus Recurrence A.6 reduces to the following.

$$\text{SCORE}[i,j] = \begin{cases} 0 & \text{if } j \geq n-1, \\ \max_{j+1 < k \leq n} \{\text{SOF}[j+1, \min\{k, 2j-i+1\}] + \text{SCORE}[j+1,k]\} & \text{otherwise.} \end{cases} \quad \text{(A.8)}$$

In the rest of this section we will assume for simplicity that $n = 2^\ell$ for some integer $\ell \geq 0$.

Figure A.17 shows the dependency structure of the DP. Figures A.17 and A.18 show a divide-and-conquer algorithm for solving the protein accordion folding problem. The algorithm consists of four recursive functions named $\mathcal{A}_{PF}$, $\mathcal{B}_{PF}$, $\mathcal{C}_{PF}$, and $\mathcal{D}_{PF}$.

Complexity analysis for the PAR-LOOP-PROTEIN-FOLDING algorithm

For $f \in \{\text{PAR-LOOP-PROTEIN-FOLDING}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{PF} on a sequence of length $n \times n$. Then $Q_f(n) = \Theta\left(n \cdot n \cdot \left(\frac{n}{B}+1\right)\right) = \Theta\left(\frac{n^3}{B} + n^2\right)$. The span will be $T_f(n) = \Theta(n \cdot (\log n + n \cdot (1))) = \Theta(n^2)$.

> For the protein accordion folding problem, the PAR-LOOP-PROTEIN-FOLDING algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B} + n^2\right)$, $T_\infty(n) = \Theta(n^2)$, parallelism $= \Theta(n)$, and $S_\infty(n) = \Theta(n^2)$.

Complexity analysis for the $\mathcal{A}_{PF}$ algorithm

For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{PF} on a sequence of length $n \times n$. Then

$$\begin{aligned} Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) = Q_{\mathcal{D}}(n) &= \mathcal{O}\left(\frac{n^2}{B} + n\right) && \text{if } n^2 \leq \gamma M, \\ Q_{\mathcal{A}}(n) &= 2Q_{\mathcal{A}}\left(\frac{n}{2}\right) + Q_{\mathcal{B}}\left(\frac{n}{2}\right) + Q_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_A M; \\ Q_{\mathcal{B}}(n) &= 4Q_{\mathcal{B}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_B M; \\ Q_{\mathcal{C}}(n) &= 4Q_{\mathcal{C}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_C M; \\ Q_{\mathcal{D}}(n) &= 8Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_D M; \end{aligned}$$

LOOP-PROTEIN-FOLDING(S, n)
1. **for** $i \leftarrow n - 1$ **to** 1 **do**
2.   **for** $j \leftarrow n - 1$ **to** $i + 1$ **do**
3.   **for** $k \leftarrow j + 2$ **to** n **do**
4. $S[i,j] \leftarrow \min \left(\begin{array}{l} S[i,j], S[j+1,k] + \\ F[j+1, \min(k, 2j - i + 1)] \end{array} \right)$

PAR-LOOP-PROTEIN-FOLDING(S, n)
1. **for** $i \leftarrow n - 1$ **to** 1 **do**
2.   **parallel for** $j \leftarrow n - 1$ **to** $i + 1$ **do**
3.   **for** $k \leftarrow j + 2$ **to** n **do**
4. $S[i,j] \leftarrow \min \left(\begin{array}{l} S[i,j], S[j+1,k] + \\ F[j+1, \min(k, 2j - i + 1)] \end{array} \right)$

$\mathcal{A}_{PF}(X)$
1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}PF}(X)$
else
2. $\mathcal{A}_{PF}(X_{22})$
3. $\mathcal{B}_{PF}(X_{12}, X_{22})$
4. $\mathcal{C}_{PF}(X_{11}, X_{12})$
5. $\mathcal{A}_{PF}(X_{11})$

$\mathcal{C}_{PF}(X, U)$
1. **if** X is a small matrix **then** $\mathcal{C}_{loop\text{-}PF}(X, U)$
else
2.   **par:** $\mathcal{C}_{PF}(X_{11}, U_{11}), \mathcal{D}_{PF}(X_{12}, U_{21}), \mathcal{C}_{PF}(X_{22}, U_{21})$
3.   **par:** $\mathcal{C}_{PF}(X_{11}, U_{12}), \mathcal{D}_{PF}(X_{12}, U_{22}), \mathcal{C}_{PF}(X_{22}, U_{22})$

$\mathcal{D}_{PF}(X, V)$
1. **if** X is a small matrix **then** $\mathcal{D}_{loop\text{-}PF}(X, V)$
else
2.   **par:** $\mathcal{D}_{PF}(X_{11}, V_{11}), \mathcal{D}_{PF}(X_{12}, V_{21}),$
$\mathcal{D}_{PF}(X_{21}, V_{11}), \mathcal{D}_{PF}(X_{22}, V_{21})$
3.   **par:** $\mathcal{D}_{PF}(X_{11}, V_{12}), \mathcal{D}_{PF}(X_{12}, V_{22}),$
$\mathcal{D}_{PF}(X_{21}, V_{12}), \mathcal{D}_{PF}(X_{22}, V_{22})$

$\mathcal{B}_{PF}(X, V)$
1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}PF}(X, V)$
else
2.   **par:** $\mathcal{B}_{PF}(X_{11}, V_{11}), \mathcal{B}_{PF}(X_{12}, V_{22}),$
$\mathcal{B}_{PF}(X_{21}, V_{11}), \mathcal{B}_{PF}(X_{22}, V_{22})$
3.   **par:** $\mathcal{D}_{PF}(X_{11}, V_{12}), \mathcal{D}_{PF}(X_{21}, V_{12})$

Figure A.17: Top: Serial and parallel iterative algorithm for the protein accordion folding problem. Bottom: Divide-and-conquer algorithm.

$$
\begin{aligned}
T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) = T_{\mathcal{D}}(n) &= \mathcal{O}(1) && \text{if } n = 1, \\
T_{\mathcal{A}}(n) = 2T_{\mathcal{A}}\left(\tfrac{n}{2}\right) + T_{\mathcal{B}}\left(\tfrac{n}{2}\right) + T_{\mathcal{C}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{B}}(n) = T_{\mathcal{B}}\left(\tfrac{n}{2}\right) + T_{\mathcal{D}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{C}}(n) = 2\max\left\{ T_{\mathcal{C}}\left(\tfrac{n}{2}\right), T_{\mathcal{D}}\left(\tfrac{n}{2}\right) \right\} + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{D}}(n) = 2T_{\mathcal{D}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$

where, γ, γ_A, γ_B, γ_C and γ_D are suitable constants. Solving, $Q_{\mathcal{A}}(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n \right)$ and $T_{\mathcal{A}}(n) = \mathcal{O}(n \log n)$.

For the protein accordion folding problem, the $\mathcal{A}_{PF}$ algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n \right)$, $T_\infty(n) = \Theta(n \log n)$, **parallelism** $= \Theta\left(\frac{n^2}{\log n} \right)$, and $S_\infty(n) = \Theta(n^2)$.

A.7   Function approximation

The function approximation problem is defined as follows. Let $f(x)$ be a function and (x_i, y_i), where $i \in [1, n]$, be the n points generated by the function such that $x_1 < x_2 < \cdots < x_n$. Given a positive integer q ($\in [2, n]$), the function $g(x)$ is constructed as follows. Choose q points including the first point (x_1, y_1) and last point (x_n, y_n) and construct $g(x)$ by connecting each point to its next point with straight lines. Let $E[i, j]$ denote the error of the best approximation $g(x)$ of the first i points using at most j line segments and $e(p, q) = \sum_{i=p}^{q} (y_i - g(x_i))^2$. Then,
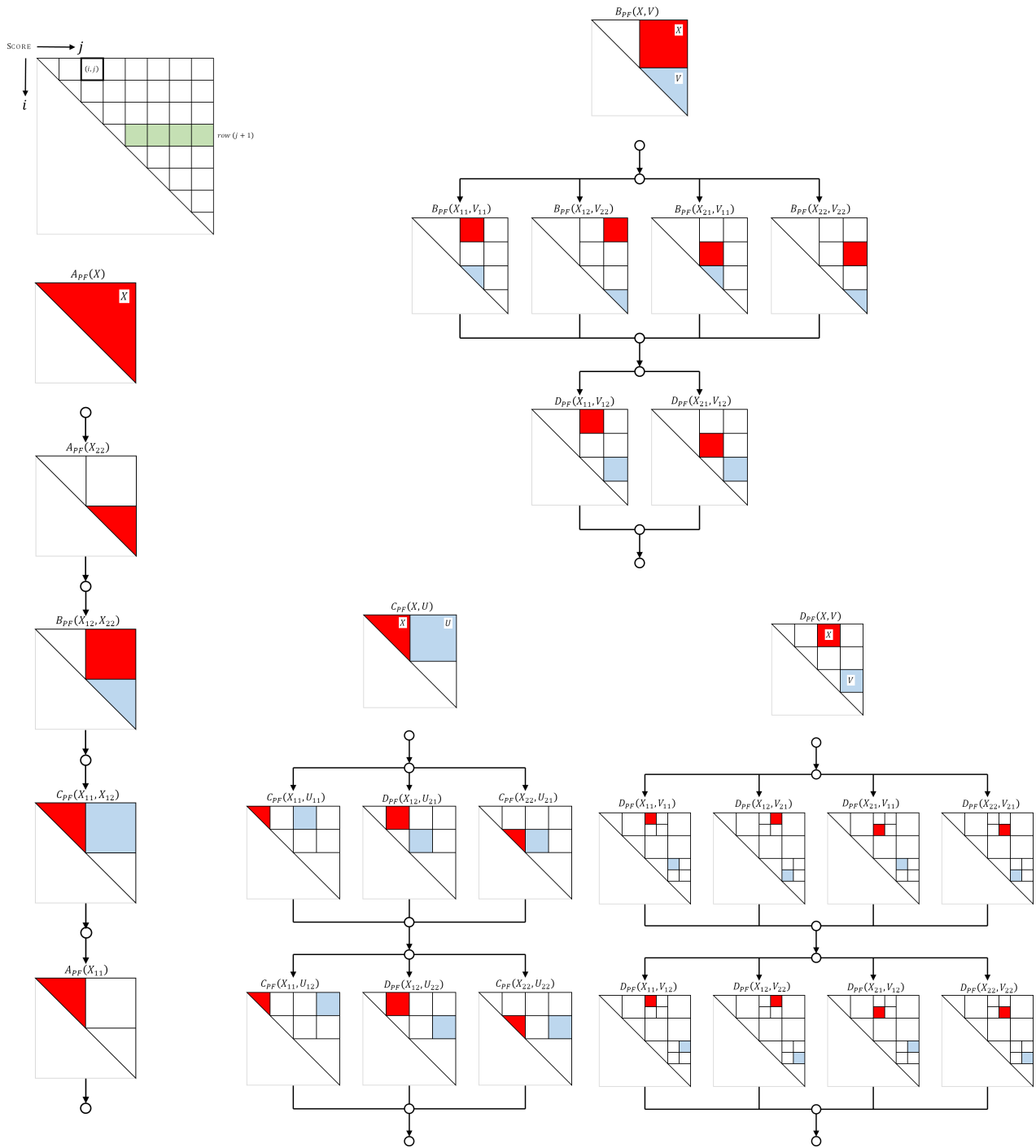
Figure A.18: Top left: Dependency graph for the protein folding problem. Rest: A divide-and-conquer algorithm for solving the protein folding problem. The initial call to the function is $\mathcal{A}_{PF}(\textsc{Score})$.

$$E[i,j] = \begin{cases} 0 & \text{if } i = 1, \\ \infty & \text{if } i > 1 \text{ and } j = 0, \\ \min_{0 < k < i} \{E(k, j-1) + e(k, i)\} & \text{if } i > 1 \text{ and } j > 0. \end{cases} \tag{A.9}$$

Figure A.20 shows the dependency structure for the problem DP. Figure A.19 show serial and parallel iterative algorithms to solve the function approximation problem. A divide-and-conquer algorithm for the problem is given in Figures A.19 and A.19. The algorithm consists of four recursive functions named $\mathcal{A}_{FA}$, $\mathcal{B}_{FA}$, $\mathcal{C}_{FA}$ and $\mathcal{D}_{FA}$.

LOOP-FUNCTION-APPROX(E)

1. **for** $i \leftarrow 1$ **to** n **do**
2.   **for** $j \leftarrow 1$ **to** n **do**
3.   **for** $k \leftarrow 1$ **to** $i - 1$ **do**
4. $E[i,j] \leftarrow \min(E[i,j], E[k, j-1] + e(k,i))$

$\mathcal{A}_{FA}(X)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop-FA}(X)$
2. **else**
3. $\mathcal{A}_{FA}(X_{11})$
4. $\mathcal{B}_{FA}(X_{12})$
5. $\mathcal{C}_{FA}(X_{22}, X_{12})$
6. $\mathcal{A}_{FA}(X_{22})$

$\mathcal{B}_{FA}(X)$

1. **if** X is a small matrix **then** $\mathcal{B}_{loop-FA}(X)$
2. **else**
3. $\mathcal{B}_{FA}(X_{11})$
4. $\mathcal{C}_{FA}(X_{21}, X_{11})$
5.   **parallel:** $\mathcal{B}_{FA}(X_{21}), \mathcal{B}_{FA}(X_{12})$
6. $\mathcal{C}_{FA}(X_{22}, X_{12})$
7. $\mathcal{B}_{FA}(X_{22})$

PAR-LOOP-FUNCTION-APPROX(E)

1. **for** $t \leftarrow 1$ **to** n **do**
2.   **parallel for** $j \leftarrow 1$ **to** t **do**
3. $i \leftarrow t - j - 1$
4.   **for** $k \leftarrow 1$ **to** $i - 1$ **do**
5. $E[i,j] \leftarrow \min(E[i,j], E[k, j-1] + e(k,i))$
6. **for** $t \leftarrow n + 1$ **to** $2n - 1$ **do**
7.   **parallel for** $j \leftarrow t - n + 1$ **to** n **do**
8. $i \leftarrow t - j + 1$
9.   **for** $k \leftarrow 1$ **to** $i - 1$ **do**
10. $E[i,j] \leftarrow \min(E[i,j], E[k, j-1] + e(k,i))$

$\mathcal{C}_{FA}(X, U)$

1. **if** X is a small matrix **then** $\mathcal{C}_{loop-FA}(X, U)$
2. **else**
3.   **parallel:** $\mathcal{C}_{FA}(X_{11}, U_{11}), \mathcal{D}_{FA}(X_{12}, U_{12}), \mathcal{C}_{FA}(X_{22}, U_{12})$
4.   **parallel:** $\mathcal{C}_{FA}(X_{11}, U_{21}), \mathcal{D}_{FA}(X_{12}, U_{22}), \mathcal{C}_{FA}(X_{22}, U_{22})$

$\mathcal{D}_{FA}(X, U)$

1. **if** X is a small matrix **then** $\mathcal{D}_{loop-FA}(X, U)$
2. **else**
3.   **parallel:** $\mathcal{D}_{FA}(X_{11}, U_{11}), \mathcal{D}_{FA}(X_{21}, U_{11}),$
             $\mathcal{D}_{FA}(X_{12}, U_{12}), \mathcal{D}_{FA}(X_{22}, U_{12})$
4.   **parallel:** $\mathcal{D}_{FA}(X_{11}, U_{21}), \mathcal{D}_{FA}(X_{21}, U_{21}),$
             $\mathcal{D}_{FA}(X_{12}, U_{22}), \mathcal{D}_{FA}(X_{22}, U_{22})$

Figure A.19: Serial iterative, parallel iterative, and divide-and-conquer algorithms to solve the function approximation problem.

Complexity analysis for the PAR-LOOP-FUNCTION-APPROX algorithm
For $f \in \{\text{PAR-LOOP-FUNCTION-APPROX}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{FA} on a matrix of size $n \times n$. Then $Q_f(n) = \Theta\left(n \cdot n \cdot \left(\frac{n}{B} + 1\right)\right) = \Theta\left(\frac{n^3}{B} + n^2\right)$. Also, $T_f(n) = \Theta(n(\log n + n(1))) = \Theta(n^2)$.

> For the function approximation problem, the PAR-LOOP-FUNCTION-APPROX algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \Theta\left(\frac{n^3}{B} + n^2\right)$, $T_\infty(n) = \Theta(n^2)$, parallelism $= \Theta(n)$, and $S_\infty(n) = \Theta(n^2)$.

Complexity analysis for the $\mathcal{A}_{FA}$ algorithm
For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{FA} on a matrix of size $n \times n$. Then
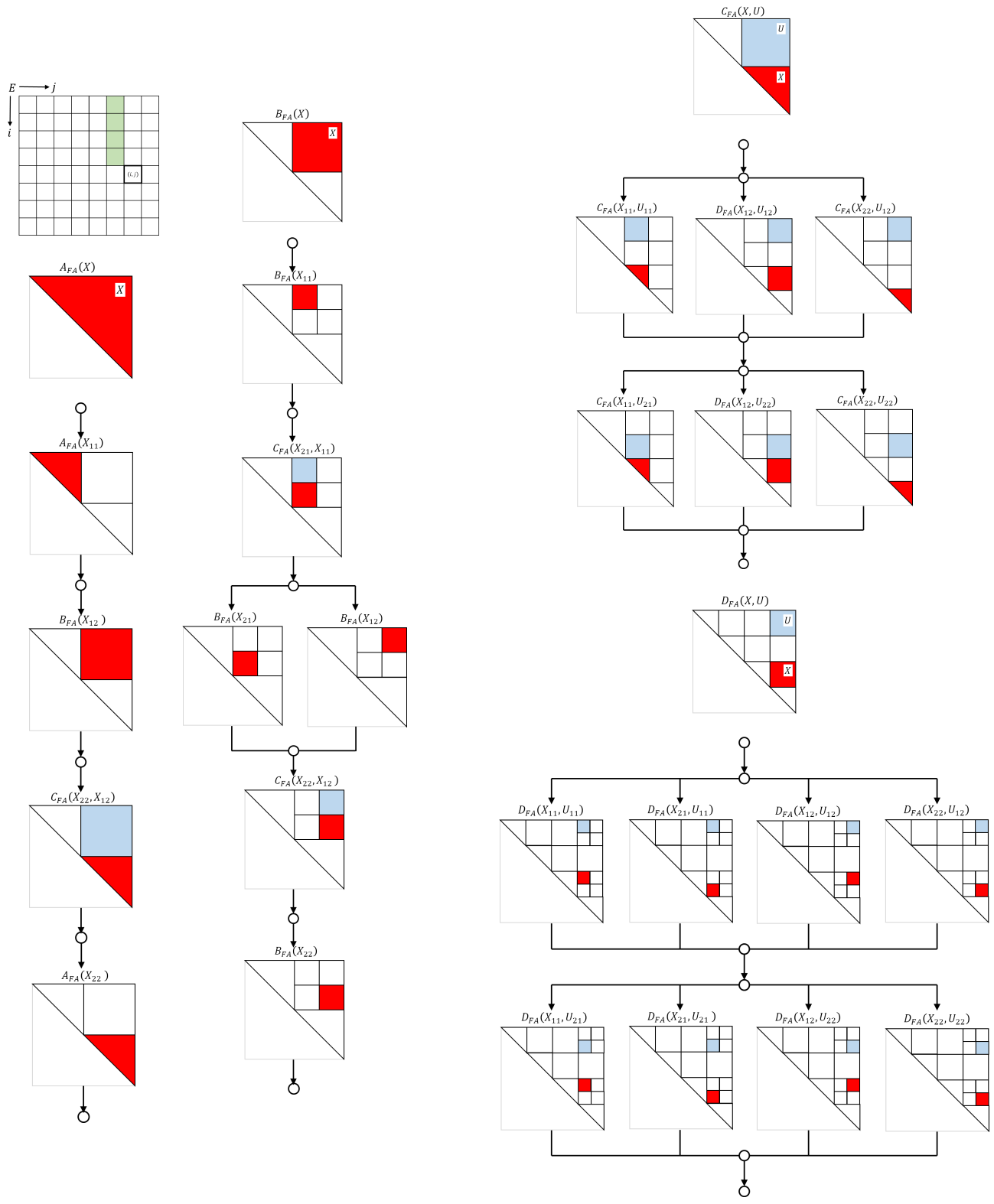
Figure A.20: Top left: Dependency structure of the function approximation DP. Rest: A divide-and-conquer algorithm for solving the problem.

$$Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) = Q_{\mathcal{D}}(n) = \mathcal{O}\left(\frac{n^2}{B} + n\right) \quad \text{if } n^2 \le \gamma M,$$
$$Q_{\mathcal{A}}(n) = 2Q_{\mathcal{A}}\left(\frac{n}{2}\right) + Q_{\mathcal{B}}\left(\frac{n}{2}\right) + Q_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) \quad \text{if } n^2 > \gamma_A M;$$
$$Q_{\mathcal{B}}(n) = 4Q_{\mathcal{B}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) \quad \text{if } n^2 > \gamma_B M;$$
$$Q_{\mathcal{C}}(n) = 4Q_{\mathcal{C}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) \quad \text{if } n^2 > \gamma_C M;$$
$$Q_{\mathcal{D}}(n) = 8Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) \quad \text{if } n^2 > \gamma_D M;$$

$$T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) = T_{\mathcal{D}}(n) = \mathcal{O}(1) \quad \text{if } n = 1,$$
$$T_{\mathcal{A}}(n) = 2T_{\mathcal{A}}\left(\frac{n}{2}\right) + T_{\mathcal{B}}\left(\frac{n}{2}\right) + T_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) \quad \text{if } n > 1;$$
$$T_{\mathcal{B}}(n) = 3T_{\mathcal{B}}\left(\frac{n}{2}\right) + 2T_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) \quad \text{if } n > 1;$$
$$T_{\mathcal{C}}(n) = 2\max\left\{T_{\mathcal{C}}\left(\frac{n}{2}\right), T_{\mathcal{D}}\left(\frac{n}{2}\right)\right\} + \Theta(1) \quad \text{if } n > 1;$$
$$T_{\mathcal{D}}(n) = 2T_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) \quad \text{if } n > 1;$$

where, γ, γ_A, γ_B, γ_C and γ_D are suitable constants. Solving, $Q_{\mathcal{A}}(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$ and $T_{\mathcal{A}}(n) = \mathcal{O}\left(n^{\log 3}\right)$.

> For the function approximation problem, the $\mathcal{A}_{FA}$ algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$, $T_\infty(n) = \Theta\left(n^{\log 3}\right)$, parallelism $= \Theta\left(n^{3-\log 3}\right)$, and $S_\infty(n) = \Theta(n^2)$.

A.8 Spoken word recognition

The spoken word recognition problem [Sakoe and Chiba, 1978] is defined as follows. We give one type of DP recurrence from the Sakoe and Chiba's paper. There are many more similar variants. Given two speech patterns expressed as a sequence of feature vectors $X = \langle x_1, x_2, \dots x_m \rangle$ and $Y = \langle y_1, y_2, \dots y_n \rangle$, and a distance function $d(i,j) = ||x_i - y_j||$, we define $\frac{D[i,j]}{|X_i|+|X_j|}$ ($0 \le i \le m, 0 \le j \le n$) to be the time-normalized distance between the speech patterns $\langle x_1, x_2, \dots x_i \rangle$ and $\langle y_1, y_2, \dots y_j \rangle$. Then $\frac{D[m,n]}{m+n}$ is the time-normalized distance between X and Y, and can be computed using the following recurrence relation:

$$D[i,j] = \begin{cases} d(i,j) & \text{if } i \le 1 \text{ or } j \le 1, \\ \min \begin{Bmatrix} D[i-1,j-2] + 2d(i,j-1) + d(i,j), \\ D[i-1,j-1] + 2d(i,j), \\ D[i-2,j-1] + 2d(i-1,j) + d(i,j) \end{Bmatrix} & \text{otherwise.} \end{cases} \tag{A.10}$$

Figure A.21, a dependency graph, serial iterative and parallel iterative algorithms are given for the spoken word recognition problem. In Figures A.21 and A.22, a divide-and-conquer algorithm is given for solving the spoken word recognition problem. The algorithm consists of three recursive functions named $\mathcal{A}_{SW}$, $\mathcal{B}_{SW}$ and $\mathcal{C}_{SW}$.

Complexity analysis for the PAR-LOOP-SPOKEN-WORD-RECOGNITION algorithm
For $f \in \{$PAR-LOOP-SPOKEN-WORD-RECOGNITION$\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{SW} on a matrix of size $n \times n$. Then $Q_f(n) = \Theta\left(n \cdot \left(\frac{n}{B} + 1\right)\right) = \Theta\left(\frac{n^2}{B} + n\right)$. The span is $T_f(n) = \Theta(n \cdot (\log n + \Theta(1))) = \Theta(n \log n)$.

Loop-Spoken-Word-Recognition

1. **for** $i \leftarrow 2$ **to** n **do**
2.   **for** $j \leftarrow 2$ **to** n **do**
3. $D[i,j] \leftarrow \min \left(D[i-1,j-2] + 2d(i,j-1) + d(i,j), D[i-1,j-1] + 2d(i,j), D[i-2,j-1] + 2d(i-1,j) + d(i,j) \right)$

Par-Loop-Spoken-Word-Recognition

1. **for** $t \leftarrow 3$ **to** n **do**
2.   **parallel for** $j \leftarrow 2$ **to** $t-1$ **do**
3. $i \leftarrow t - j - 1$
4. $D[i,j] \leftarrow \min \left(D[i-1,j-2] + 2d(i,j-1) + d(i,j), D[i-1,j-1] + 2d(i,j), D[i-2,j-1] + 2d(i-1,j) + d(i,j) \right)$
5. **for** $t \leftarrow n+1$ **to** $2n-1$ **do**
6.   **parallel for** $j \leftarrow t-n+1$ **to** n **do**
7. $i \leftarrow t - j + 1$
8. $D[i,j] \leftarrow \min \left(D[i-1,j-2] + 2d(i,j-1) + d(i,j), D[i-1,j-1] + 2d(i,j), D[i-2,j-1] + 2d(i-1,j) + d(i,j) \right)$
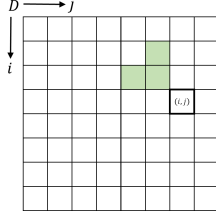
$\mathcal{B}_{SW}(X)$

1. **if** X is a small matrix **then** $\mathcal{B}_{loop-SW}(X)$
2. **else**
3. $\mathcal{B}_{SW}(X_1)$
4.   **parallel:** $\mathcal{B}_{SW}(X_2), \mathcal{B}_{SW}(X_3)$
5. $\mathcal{B}_{SW}(X_4)$

$\mathcal{A}_{SW}(X)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop-SW}(X)$
2. **else**
3. $\mathcal{A}_{SW}(X_1)$
4.   **parallel:** $\mathcal{B}_{SW}(X_2), \mathcal{C}_{SW}(X_3)$
5. $\mathcal{A}_{SW}(X_4)$

$\mathcal{C}_{SW}(X)$

1. **if** X is a small matrix **then** $\mathcal{C}_{loop-SW}(X)$
2. **else**
3. $\mathcal{C}_{SW}(X_1)$
4.   **parallel:** $\mathcal{C}_{SW}(X_2), \mathcal{C}_{SW}(X_3)$
5. $\mathcal{C}_{SW}(X_4)$

Figure A.21: Dependency graph, serial iterative, parallel iterative, and divide-and-conquer algorithms for solving the spoken word recognition problem.

For the spoken word recognition problem, the Par-Loop-Spoken-Word-Recognition algorithm achieves $T_1(n) = \Theta\left(n^2\right)$, $Q_1(n) = \mathcal{O}\left(\frac{n^2}{B} + n\right)$, $T_\infty(n) = \Theta\left(n \log n\right)$, **parallelism** $= \Theta\left(\frac{n}{\log n}\right)$, and $S_\infty(n) = \Theta\left(n\right)$.

Complexity analysis for the $\mathcal{A}_{SW}$ algorithm

For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{SW} on a matrix of size $n \times n$. Then

$$
\begin{aligned}
Q_\mathcal{A}(n) = Q_\mathcal{B}(n) = Q_\mathcal{C}(n) &= \mathcal{O}\left(\frac{n}{B} + 1\right) && \text{if } n \leq \gamma M, \\
Q_\mathcal{A}(n) &= 2Q_\mathcal{A}\left(\frac{n}{2}\right) + Q_\mathcal{B}\left(\frac{n}{2}\right) + Q_\mathcal{C}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > \gamma_A M; \\
Q_\mathcal{B}(n) &= 4Q_\mathcal{B}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > \gamma_B M; \\
Q_\mathcal{C}(n) &= 4Q_\mathcal{C}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > \gamma_C M;
\end{aligned}
$$

$$
\begin{aligned}
T_\mathcal{A}(n) = T_\mathcal{B}(n) = T_\mathcal{C}(n) &= \mathcal{O}(1) && \text{if } n = 1, \\
T_\mathcal{A}(n) &= 2T_\mathcal{A}\left(\frac{n}{2}\right) + \max\left(T_\mathcal{B}\left(\frac{n}{2}\right), T_\mathcal{C}\left(\frac{n}{2}\right)\right) + \Theta(1) && \text{if } n > 1; \\
T_\mathcal{B}(n) &= 2T_\mathcal{B}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
T_\mathcal{C}(n) &= 2T_\mathcal{C}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$

where, γ, γ_A, γ_B, and γ_C are suitable constants. Solving, $Q_\mathcal{A}(n) = \mathcal{O}\left(\frac{n^2}{BM} + \frac{n}{B} + 1\right)$ and $T_\mathcal{A}(n) = \Theta\left(n^{\log 3}\right)$.

Figure A.22: A divide-and-conquer algorithm for solving the spoken word recognition problem.

For the spoken word recognition problem, the $\mathcal{A}_{SW}$ algorithm achieves $T_1(n) = \Theta(n^2)$, $Q_1(n) = \mathcal{O}\left(\frac{n^2}{BM} + \frac{n}{B} + 1\right)$, $T_\infty(n) = \Theta\left(n^{\log 3}\right)$, **parallelism** $= \Theta\left(n^{2-\log 3}\right)$, and $S_\infty(n) = \Theta(n)$.

A.9 Bitonic traveling salesman

We define the bitonic traveling salesman problem [Cormen et al., 2009] as follows. We are given n points $p_1, p_2, \ldots, p_n$ with increasing x-coordinates. The term $B[i,j]$ represents the bitonic path length if we start from point p_i and end at point p_j and $d(i,j)$ denotes the euclidean distance between p_i and p_j. Then

$$B[i,j] = \begin{cases} d(1,2) & \text{if } i = 1 \text{ and } j = 2, \\ B[i, j-1] + d(i, j-1) & \text{if } i < j-1, \\ \min_{k \in [1, i-1]} (B[k, i] + d(k, i)) & \text{if } j > 2 \text{ and } i = j-1. \end{cases} \tag{A.11}$$

Figure A.24 shows the dependency structure of the DP. Serial and parallel iterative algorithms are given in Figure A.23. In Figures A.23 and A.24, a divide-and-conquer algorithm is given for solving the bitonic TSP problem. The algorithm consists of three recursive functions named $\mathcal{A}_{bit}$, $\mathcal{B}_{bit}$ and $\mathcal{C}_{bit}$.

$\mathcal{A}_{bit}(X)$
1. **if** X is a small matrix **then** $\mathcal{A}_{loop-bit}(X)$
2. **else**
3. $\mathcal{A}_{bit}(X_{11})$
4. $\mathcal{B}_{bit}(X_{12})$
5. $\mathcal{C}_{bit}(X_{22}, X_{12})$
6. $\mathcal{A}_{bit}(X_{22})$

LOOP-BITONIC-TSP
1. **for** $j \leftarrow 3$ **to** n **do**
2.   **for** $i \leftarrow 1$ **to** $j - 2$ **do**
3. $B[i,j] \leftarrow B[i, j-1] + d(i, j-1)$
4. $i \leftarrow j - 1$
5.   **for** $k \leftarrow 1$ **to** $i - 1$ **do**
6. $B[i,j] \leftarrow B[k,i] + d(k,i)$

$\mathcal{B}_{bit}(X)$
1. **if** X is a small matrix **then** $\mathcal{B}_{loop-bit}(X)$
2. **else**
3. $\mathcal{B}_{bit}(X_{11})$
4.   **parallel:** $\mathcal{B}_{bit}(X_{21}), \mathcal{B}_{bit}(X_{12})$
5. $\mathcal{B}_{bit}(X_{22})$

PAR-LOOP-BITONIC-TSP
1. **for** $j \leftarrow 3$ **to** n **do**
2.   **parallel for** $i \leftarrow 1$ **to** $j - 2$ **do**
3. $B[i,j] \leftarrow B[i, j-1] + d(i, j-1)$
4. $i \leftarrow j - 1$
5.   **for** $k \leftarrow 1$ **to** $i - 1$ **do**
6. $B[i,j] \leftarrow B[k,i] + d(k,i)$

$\mathcal{C}_{bit}(X, U)$
1. **if** X is a small matrix **then** $\mathcal{C}_{loop-bit}(X, U)$
2. **else**
3.   **parallel:** $\mathcal{C}_{bit}(X_{11}, U_{11}), \mathcal{C}_{bit}(X_{22}, U_{12})$
4.   **parallel:** $\mathcal{C}_{bit}(X_{11}, U_{21}), \mathcal{C}_{bit}(X_{22}, U_{22})$

Figure A.23: Serial iterative, parallel iterative, and divide-and-conquer algorithms for the bitonic TSP problem.

Complexity analysis for the PAR-LOOP-BITONIC-TSP algorithm

For $f \in \{\text{PAR-LOOP-BITONIC-TSP}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{bit} on a matrix of size $n \times n$. Then

> For the bitonic traveling salesman problem, the PAR-LOOP-BITONIC-TSP algorithm achieves $T_1(n) = \Theta(n^2)$, $Q_1(n) = \mathcal{O}\left(\frac{n^2}{BM} + \frac{n^2}{M^2} + \frac{n}{B} + 1\right)$, $T_\infty(n) = \Theta(n^{\log 3})$, parallelism $= \Theta(n^{2-\log 3})$, and $S_\infty(n) = \Theta(n)$.

Complexity analysis for the $\mathcal{A}_{bit}$ algorithm

For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{bit} on a matrix of size $n \times n$. Then

$$
\begin{aligned}
Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) &= \mathcal{O}\left(\frac{n}{B} + 1\right) && \text{if } n \leq \gamma M, \\
Q_{\mathcal{A}}(n) &= 2Q_{\mathcal{A}}\left(\frac{n}{2}\right) + Q_{\mathcal{B}}\left(\frac{n}{2}\right) + Q_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > \gamma_A M; \\
Q_{\mathcal{B}}(n) &= 4Q_{\mathcal{B}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > \gamma_B M; \\
Q_{\mathcal{C}}(n) &= 4Q_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > \gamma_C M; \\[6pt]
T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) &= \mathcal{O}(1) && \text{if } n = 1, \\
T_{\mathcal{A}}(n) &= 2T_{\mathcal{A}}\left(\frac{n}{2}\right) + \max\left(T_{\mathcal{B}}\left(\frac{n}{2}\right), T_{\mathcal{C}}\left(\frac{n}{2}\right)\right) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{B}}(n) &= 3T_{\mathcal{B}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{C}}(n) &= 2T_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$
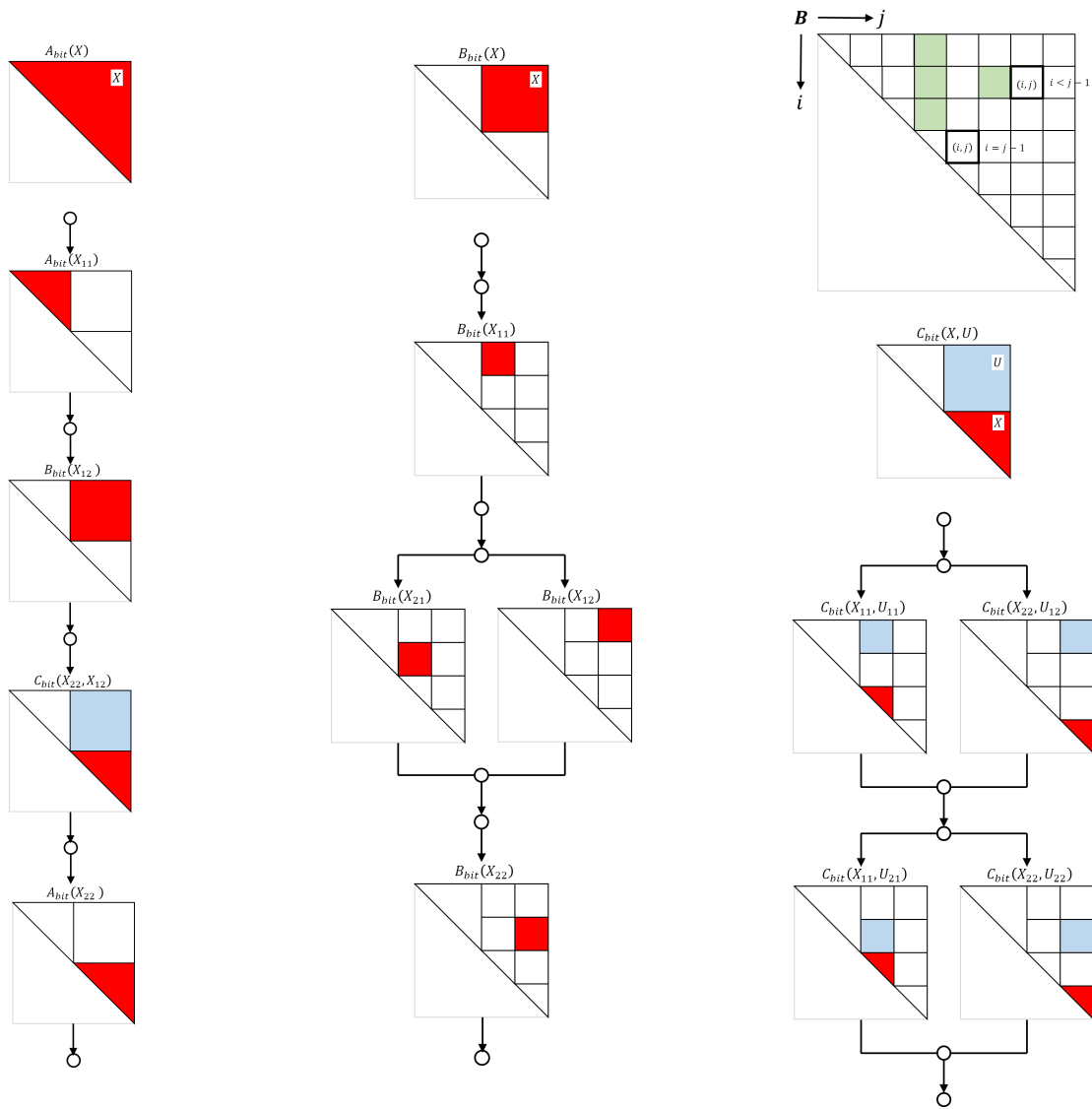
Figure A.24: Top right: Dependency structure of the bitonic TSP DP. Rest: A divide-and-conquer algorithm for solving the problem.

where, γ, γ_A, γ_B, and γ_C are suitable constants. Solving, $Q_{\mathcal{A}}(n) = \mathcal{O}\left(\frac{n^2}{BM} + \frac{n^2}{M^2} + \frac{n}{B} + 1\right)$ and $T_{\mathcal{A}}(n) = \Theta\left(n^{\log 3}\right)$.
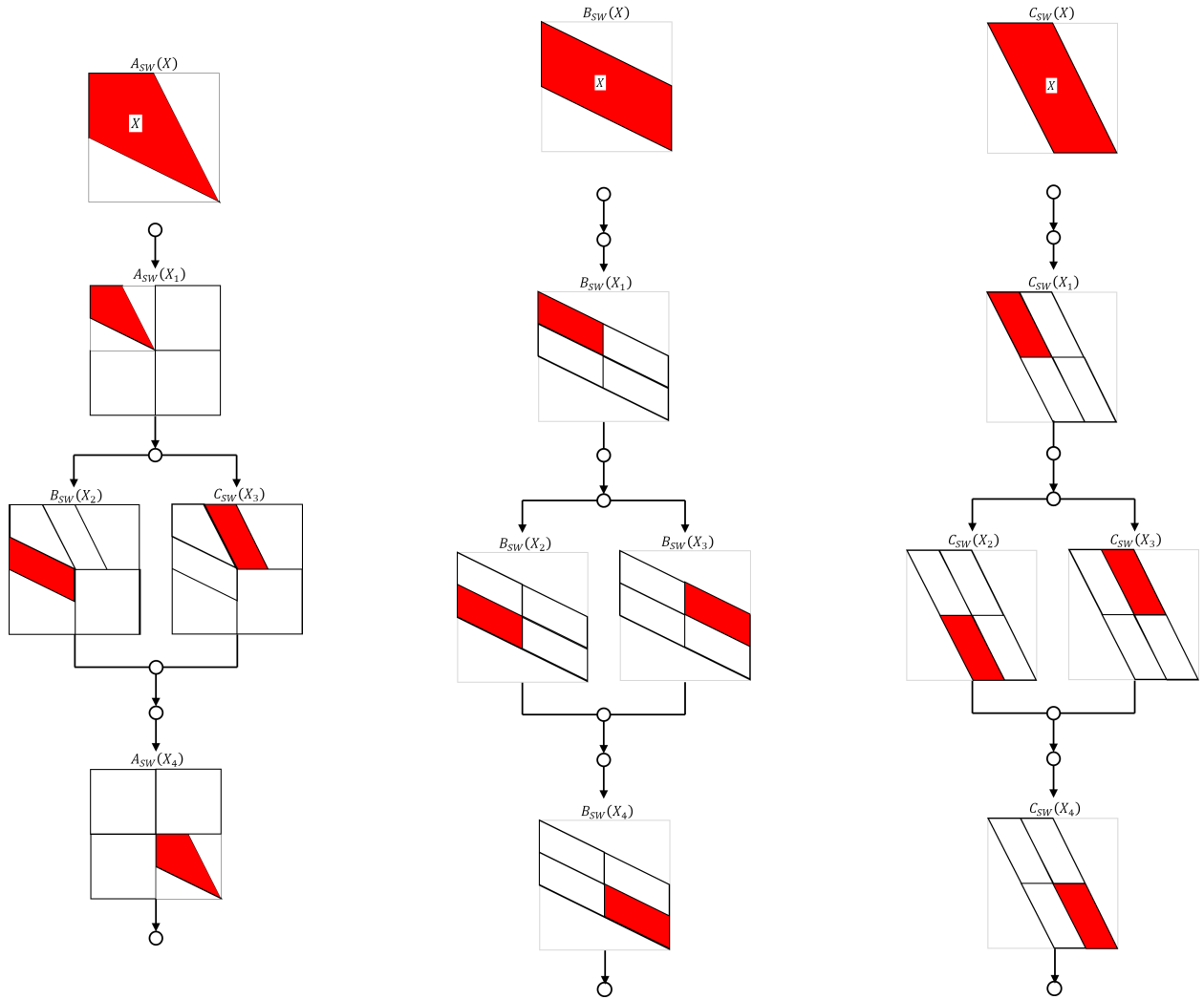
> For the bitonic traveling salesman problem, the $\mathcal{A}_{bit}$ algorithm achieves $T_1(n) = \Theta\left(n^2\right)$, $Q_1(n) = \mathcal{O}\left(\frac{n^2}{BM} + \frac{n^2}{M^2} + \frac{n}{B} + 1\right)$, $T_\infty(n) = \Theta\left(n^{\log 3}\right)$, parallelism $= \Theta\left(n^{2-\log 3}\right)$, and $S_\infty(n) = \Theta\left(n\right)$.

A.10   Cocke-Younger-Kasami algorithm

The CYK algorithm [Hays, 1962, Younger, 1967, Kasami, 1965] was invented by John Cocke, Daniel Younger, and Tadao Kasami. It is a parsing algorithm used to parse context-free

grammars (given in Chomsky normal form (CNF)) and was used in compilers. It is one of the most efficient parsing algorithms.

We are given a string $X = \langle x_1, x_2, \ldots, x_n \rangle$ and a context-free grammar $G = (V, \sum, R, S)$, where V is a set of variables (or non-terminal symbols), $\sum$ is a finite set of terminal symbols, $R = \{R_1, R_2, \ldots, R_t\} : V \to (V \cup \sum)^*$ is a finite set of rules, and S is a start variable chosen from V. We set $P[i, j, c]$ to true provided substring $X_{ij} = x_i x_{i+1} \ldots x_{i+j-1}$ can be generated from rule $R_c \in R$, and to false otherwise. Then

$$
P[i, j, c] = \begin{cases} \text{true} & \text{if } j = 1 \text{ and } R_c \to x_i, \\ \text{true} & \text{if } P[i, k, a] = P[i + k, j - k, b] = \text{true}, k \in [1, i - 1] \text{ and } R_c \to R_a R_b, \\ \text{false} & \text{otherwise.} \end{cases}
$$

(A.12)

The dependency graph of the CYK algorithm is given in Figure A.25. A divide-and-conquer algorithm (developed in collaboration with Jesmin Jahan Tithi) is given in Figure A.25. Note that the divide-and-conquer structure is exactly identical to that of the parenthesis problem (Section A.2) and hence the complexity analysis similar as given in the parenthesis problem. In fact, through a different parameterization of the DP recurrence, we can end up with a DP recurrence that is very similar to that of the parenthesis problem.

> For the CYK DP recurrence, the $\mathcal{A}_{CYK}$ algorithm achieves $T_1(n) = \Theta\left(n^3\right)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$, $T_\infty(n) = \Theta\left(n^{\log 3}\right)$, parallelism $= \Theta\left(n^{3-\log 3}\right)$, and $S_\infty(n) = \Theta\left(n^2\right)$.

A.11 Binomial coefficient

A binomial coefficient, denoted $C(n, k)$ is the number of combinations (or subsets) of k elements taken fro a set of n elements. The term is called so because it comes from the coefficients of the expansion of the binomial formula:

$$(a + b)^n = C(n, 0)a^n + \cdots + C(n, k)a^{n-k}b^k + \cdots + C(n, n)b^n.$$

Binomial coefficient can be computed with the recurrence

$$
C(n, k) = \begin{cases} 0 & \text{if } k = 0 \text{ or } k \geq n, \\ C(n - 1, k - 1) + C(n - 1, k) & \text{if } k \in [1, n - 1]. \end{cases}
$$

Figure A.26 shows the dependency graph for the problem. If we want to compute the binomial coefficient $C[n, k]$, then the part of the DP table that will be computed is shown in the figure. The first part consists of the region $C[0 \ldots k, 0 \ldots k]$ in which the lower right triangular region will be filled. The second part consists of the rectangular region $C[k + 1 \ldots n, 0 \ldots k]$.

Figure A.26 shows serial and parallel iterative algorithms to compute a binomial coefficient. A divide-and-conquer algorithm is also given to solve the problem. It consists of two recursive functions $\mathcal{A}_{BC}$ and $\mathcal{B}_{BC}$. The function $\mathcal{A}_{BC}$ fills a lower right-triangular region and the function $\mathcal{B}_{BC}$ fills a square region. This function can be easily extended / modified
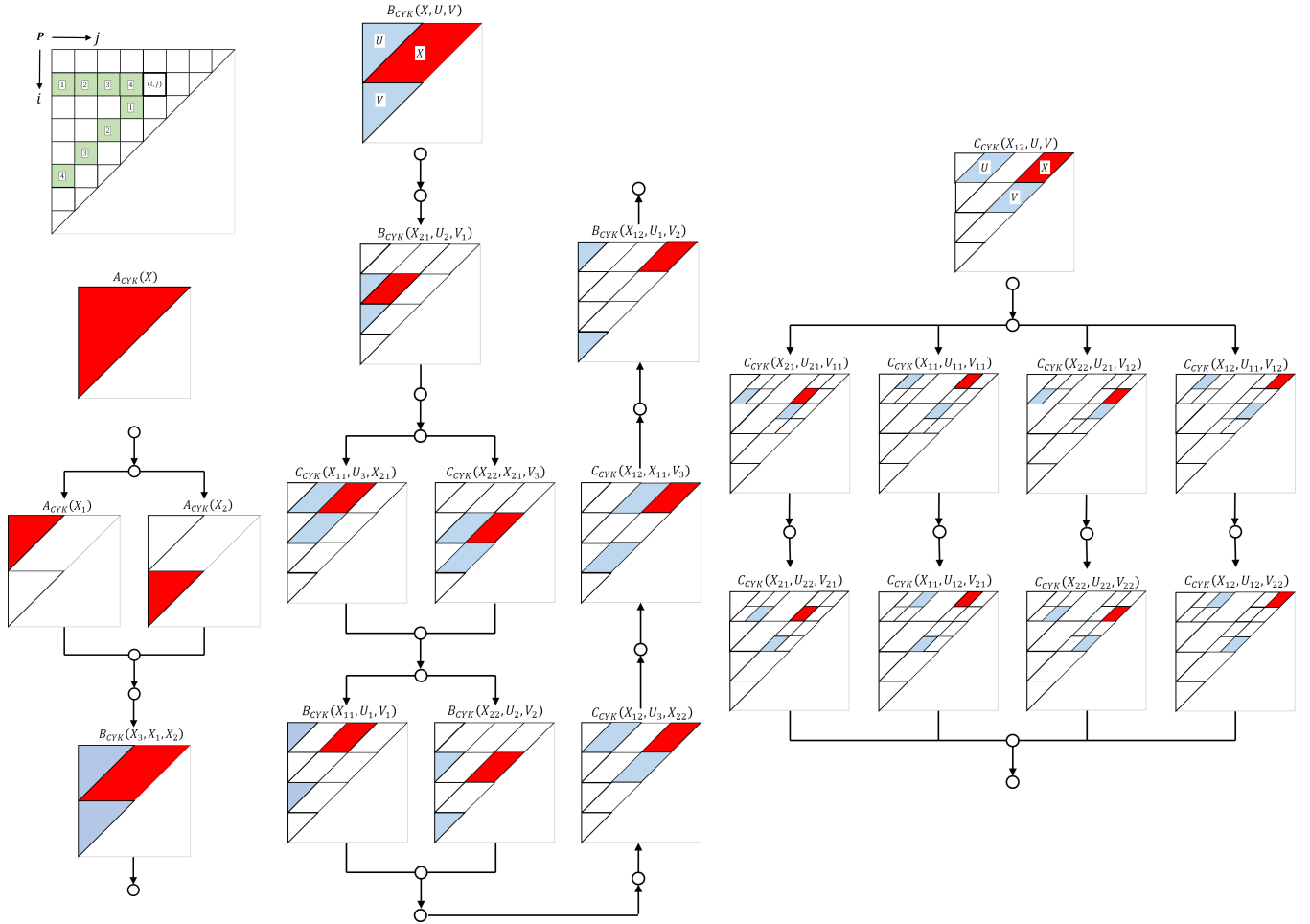
Figure A.25: Top left: Dependency graph for the CYK algorithm Rest: A divide-and-conquer algorithm.

to work for generic rectangular matrices. For simplicity and pedagogical reasons we show the working of the $\mathcal{B}_{BC}$ function for square matrices only.

Complexity analysis for the PAR-LOOP-BINOMIAL-COEFFICIENT algorithm

Let $Q_f(n, k)$ and $T_f(n, k)$ denote the serial cache complexity, and span of PAR-LOOP-BINOMIAL-COEFFICIENT with parameters n and k. Then $Q_f(n, k) = \Theta\left(n \cdot \left(\frac{k}{B} + 1\right)\right) = \Theta\left(\frac{nk}{B} + n\right)$ and $T_f(n, k) = \Theta\left(n \cdot (\log k + \Theta(1))\right) = \Theta(n \log k)$.

> For the binomial coefficient problem, the PAR-LOOP-BINOMIAL-COEFFICIENT algorithm achieves $T_1(n, k) = \Theta(nk)$, $Q_1(n, k) = \Theta\left(\frac{nk}{B} + n\right)$, $T_\infty(n, k) = \Theta(n \log k)$, parallelism $= \Theta\left(\frac{k}{\log k}\right)$, and $S_\infty(n) = \Theta(k)$.

Complexity analysis for the D&C-BINOMIAL-COFFICIENT algorithm

Let $Q_{\mathcal{A}}(k)$ and $Q_{\mathcal{B}}(n, k)$ denote the serial cache complexities and $T_{\mathcal{A}}(k)$ and $T_{\mathcal{B}}(n, k)$ denote the span of the respective functions for parameters n and k. Then

Loop-Binomial-Coefficient(C)

1. **for** $i \leftarrow 0$ **to** n **do**
2.   **for** $j \leftarrow 0$ **to** $\min(i,k)$ **do**
3.   **if** $j = 0$ **or** $j = i$ **then** $C[i,j] \leftarrow 1$
4.   **else** $C[i,j] \leftarrow C[i,j-1] + C[i-1,j]$

Par-Loop-Binomial-Coefficient(C)

1. **for** $i \leftarrow 0$ **to** n **do**
2.   **parallel for** $j \leftarrow 0$ **to** $\min(i,k)$ **do**
3.   **if** $j = 0$ **or** $j = i$ **then** $C[i,j] \leftarrow 1$
4.   **else** $C[i,j] \leftarrow C[i,j-1] + C[i-1,j]$

D&C-Binomial-Coefficient(X)

1. $\mathcal{A}_{BC}(X_1)$ ▷ X_1 is a $k \times k$ right triangle
2. $\mathcal{B}_{BC}(X_2)$ ▷ X_2 is a $(n-k) \times k$ rectangle

$\mathcal{A}_{BC}(X)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}BC}(X)$
2. **else**
3. $\mathcal{A}_{BC}(X_{11})$
4. $\mathcal{B}_{BC}(X_{21})$
5. $\mathcal{A}_{BC}(X_{22})$

$\mathcal{B}_{BC}(X)$

1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}BC}(X)$
2. **else**
3. $\mathcal{B}_{BC}(X_{11})$
4.   **parallel:** $\mathcal{B}_{BC}(X_{12})$, $\mathcal{B}_{BC}(X_{21})$
5. $\mathcal{B}_{BC}(X_{22})$

Figure A.26: Top left: The dependency graph and the part of the DP table that will be filled while computing the binomial coefficient $C(n,k)$. Rest: Serial iterative algorithm, parallel iterative algorithm, and the divide-and-conquer algorithm for the binomial coefficient.

$$Q_{\mathcal{A}}(k) = \mathcal{O}\left(\tfrac{k}{B} + 1\right) \qquad \text{if } k \leq \gamma M,$$
$$Q_{\mathcal{A}}(k) = 2Q_{\mathcal{A}}\left(\tfrac{k}{2}\right) + Q_{\mathcal{B}}\left(\tfrac{k}{2}, \tfrac{k}{2}\right) + \Theta(1) \quad \text{if } k > \gamma_A M;$$

$$T_{\mathcal{A}}(k) = \mathcal{O}(1) \qquad \text{if } k = 1,$$
$$T_{\mathcal{A}}(k) = 2T_{\mathcal{A}}\left(\tfrac{k}{2}\right) + T_{\mathcal{B}}\left(\tfrac{k}{2}, \tfrac{k}{2}\right) + \Theta(1) \quad \text{if } k > 1;$$

where, γ, γ_A are suitable constants. The recursive structure of $\mathcal{B}_{BC}$ is similar to that of the LCS algorithm (see Section A.1). Hence the complexity of the $\mathcal{B}_{BC}$ function derives from that of the LCS algorithm. Therefore, $Q_{\mathcal{B}}(n,k) = \Theta\left(\frac{nk}{BM} + \frac{nk}{M^2} + \frac{n+k}{B} + 1\right)$ and $T_{\mathcal{B}}(n,k) = \Theta\left(\max(n,k) \cdot (\min(n,k))^{(\log 3)-1}\right)$. Using this result in the recurrence, we have, $Q_{\mathcal{A}}(k) = \mathcal{O}\left(\frac{k^2}{BM} + \frac{k^2}{M^2} + \frac{k}{B} + 1\right)$ and $T_{\mathcal{A}}(n) = \Theta\left(k^{\log 3}\right)$.

178

For the final complexities of the divide-and-conquer algorithm, we must compute $Q_A(k) +$ $Q_B(n - k, k)$ and $T_A(k) + T_B(n - k, k)$. Thus we have

> For the binomial coefficient problem, the D&C-BINOMIAL-COEFFICIENT algorithm achieves $T_1(n, k) = \Theta(nk)$, $Q_1(n, k) = \mathcal{O}\left(\frac{nk}{BM} + \frac{nk}{M^2} + \frac{n+k}{B} + 1\right)$, $T_\infty(n, k) = \Theta\left(k^{\log 3} + \max(n - k, k) \cdot (\min(n - k, k))^{(\log 3) - 1}\right)$, and $S_\infty(n) = \Theta(n + k)$.

A.12 Egg dropping

The egg dropping problem is defined as follows. There is an n-floored building and we are given k identical eggs. We define *threshold floor* of a building as the highest floor in the building from and below which when the egg is dropped, the egg does not break, and above which when the egg is dropped, the egg breaks.

We want to find the threshold floor of the k-floored building. The threshold floor of the building can be anything in the range $[0, n]$. A threshold floor 0 means that when the egg is dropped from floor 1, the egg breaks, which in turn means that there is no threshold floor in the building.

What strategy finds the threshold floor minimizing the worst-case number of drops?

There are multiple ways to solve the problem. But, every single way of solving the problem seems to use dynamic programming. We can formulate the dynamic programming recurrence in majorly three different ways as follows to solve our original problem:

✶ [*Drops.*] What is the minimum number of drops to find the threshold floor when there are n floors and k eggs?
✶ [*Floors.*] What is the maximum number of floors in which we can find the threshold floor if we have k eggs and maximum number of drops allowed is d?
✶ [*Eggs.*] What is the minimum number of eggs required to find the threshold floor if we have n floors and maximum number of drops allowed is d?

Drops DP recurrence
We denote the minimum number of drops required to find the threshold floor in first i floors using j eggs as $drops[i][j]$. So, in this generalized problem, we are interested in calculating $drops[n][k]$, where n is the number of floors in the building and k is the number of eggs we can use.

$$\text{Minimum number of drops} = drops[n][k]$$

We use a magical wand to solve this problem. That wand can turn stones into gold. The magical wand we are talking about is an algorithm design technique called *dynamic programming*. Before proceeding further, the reader is recommended to understand the technique from a good algorithms textbook.

We compute $drops[n][k]$ using the following recurrence.

$$drops[i][j] = \begin{cases} i & \text{if } j = 1, \\ 1 & \text{if } i = 1, \\ 1 + \min_{x \in [1, i]} \left(\max\left(drops[x - 1][j - 1], drops[i - x][j]\right)\right) & \text{otherwise.} \end{cases}$$

179

The base cases when $j = 1$ (1 egg) and $i = 1$ (1 floor) is straightforward. The recursion case for i floors and j eggs goes like this. If we drop an egg from floor $x \in [1, i]$, there can be two cases: (i) the egg breaks, in which case we are left with $j - 1$ eggs and we need to test floors in the range $[1, i - 1]$. Here, the total number of drops is $1 + drops[x - 1][j - 1]$; and (ii) the egg does not break, in which case we are left with j eggs and we need to test floors in the range $[x + 1, i]$. Here, the total number of drops is $1 + drops[i - x][j]$. We consider the maximum of the two cases (which represents the worst-case) and take the minimum of all maximums by varying x in the range $[1, i]$.

Figure A.28 gives the dependency structure of the $drops[i][j]$ recurrence. Figure A.27 gives a serial iterative DP algorithm to compute $drops[n][k]$. In Figure A.28, we present a divide-and-conquer algorithm to solve the $drops[i][j]$ DP recurrence. For simplicity of exposition, we consider $n = k$. In [Frigo et al., 1999], a divide-and-conquer algorithm was presented to solve the matrix multiplication problem for generic matrices. Using the ideas presented in that paper, it is possible to extend our divide-and-conquer algorithm to compute $drops[n][k]$ for generic values of n and k.

The divide-and-conquer algorithm consists of three functions $\mathcal{A}_{ED}, \mathcal{B}_{ED}$, and $\mathcal{C}_{ED}$. The $\mathcal{B}_{ED}$ function is special. It reads from the region it writes to. Hence, several possible dependencies of cells are combined into that function. In the directed acyclic graph (DAG) of the $\mathcal{B}_{ED}$ function, only two parameters X and V are shown in the diagram. The function $\mathcal{B}_{ED}(\langle X, V, X \rangle, \langle X, X, V \rangle, \langle X, X, X \rangle)$ is represented as $\mathcal{B}_{ED}(X, V)$ for simplicity.

Improved algorithm. The iterative DP algorithm from Figure A.27 computes only the minimum number of drops required to find the threshold floor and not the information about the threshold floor and which egg has to be dropped from which floor. The algorithm can be modified suitably to find this extra information as well. Though the algorithm is correct and computes the optimum number of drops, it executes slow and its time complexity is $\Theta(n^2 k)$.

The algorithm's runtime can be reduced to $\Theta(nk \log n)$. We can find $\min_{x \in [1,i]} (\max(drops[x - 1][j - 1], drops[i - x][j]))$ in $\Theta(\log n)$ time using binary search. The reason is as follows. When x is a variable, $drops[x - 1][j - 1]$ is an increasing function and $drops[i - x][j]$ is a decreasing function and hence $\max(drops[x - 1][j - 1], drops[i - x][j])$ will have a global minimum. This global minimum can be found using a variant of binary search.

We can further reduce the runtime to $\Theta(nk)$. We store the optimal x for every cell of the $drops$ DP table. When we want to find the optimal x of a new cell, we make use of the optimal x of its previous cells.

The plot of $drops[n][k]$ for varying n is given in Figure A.29.

Complexity analysis for the $\mathcal{A}_{ED}$ algorithm

For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity, and span of f_{ED} on a matrix of size $n \times n$. Then

$$
\begin{aligned}
Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) &= \mathcal{O}\left(\frac{n^2}{B} + 1\right) && \text{if } n^2 \leq \gamma M, \\
Q_{\mathcal{A}}(n) &= 2\left(Q_{\mathcal{A}}\left(\frac{n}{2}\right) + Q_{\mathcal{B}}\left(\frac{n}{2}\right) + Q_{\mathcal{C}}\left(\frac{n}{2}\right)\right) + \Theta(1) && \text{if } n^2 > \gamma_A M; \\
Q_{\mathcal{B}}(n) &= 4\left(Q_{\mathcal{B}}\left(\frac{n}{2}\right) + Q_{\mathcal{C}}\left(\frac{n}{2}\right)\right) + \Theta(1) && \text{if } n^2 > \gamma_B M; \\
Q_{\mathcal{C}}(n) &= 8Q_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_C M;
\end{aligned}
$$

Figure A.27: Left: An algorithm to solve the k eggs problem for n floors. Right: The core idea for the algorithm.

$$
\begin{aligned}
T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) &= \mathcal{O}(1) && \text{if } n = 1, \\
T_{\mathcal{A}}(n) &= 2\left(T_{\mathcal{A}}\left(\tfrac{n}{2}\right) + T_{\mathcal{B}}\left(\tfrac{n}{2}\right)\right) + T_{\mathcal{C}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{B}}(n) &= 3T_{\mathcal{B}}\left(\tfrac{n}{2}\right) + 4T_{\mathcal{C}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{C}}(n) &= 2T_{\mathcal{C}}\left(\tfrac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$

where, γ, γ_A, γ_B, and γ_C are suitable constants. Solving, $Q_{\mathcal{A}}(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$ and $T_{\mathcal{A}}(n) = \Theta\left(n^{\log 3}\right)$.

> For the egg dropping problem, the $\mathcal{A}_{ED}$ algorithm achieves $T_1(n) = \Theta(n^3)$, $Q_1(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$, $T_\infty(n) = \Theta\left(n^{\log 3}\right)$, **parallelism** $= \Theta\left(n^{3-\log 3}\right)$, **and** $S_\infty(n) = \Theta(n^2)$.

A.13 Sorting algorithms

Sorting is a computational process of rearranging a multiset of items in ascending or descending order [Knuth, 1998]. It is one of the most fundamental problems in computer science. Sorting is used in real-life scenarios. Smartphone contacts are sorted based on names; students' (resp. employees' and patients') profiles are sorted based on student ID (resp. employee ID and patient ID); knock-out wrestling tournaments are like priority queues to sort wrestlers based on muscle strength; passengers waiting in a queue to board a bus sort themselves based on time-of-arrival; flight (or bus or train) information is sorted based on time-of-departure; people posing for a group photo often sort themselves based on height; importance given to different jobs and people are sorted based on priorities; to-be-worn dresses are usually sorted based on least-recently-used; and, research papers to top theory conferences are sorted based on incomprehensibility.

Sorting is used as an intermediate step to solve several computer science problems [Skiena, 1998, Knuth, 1998] such as: bringing all items with the same identification together, matching items in two or more files, searching for a specific value, testing if all elements are unique, deleting duplicates, finding the kth largest element (selection problem),
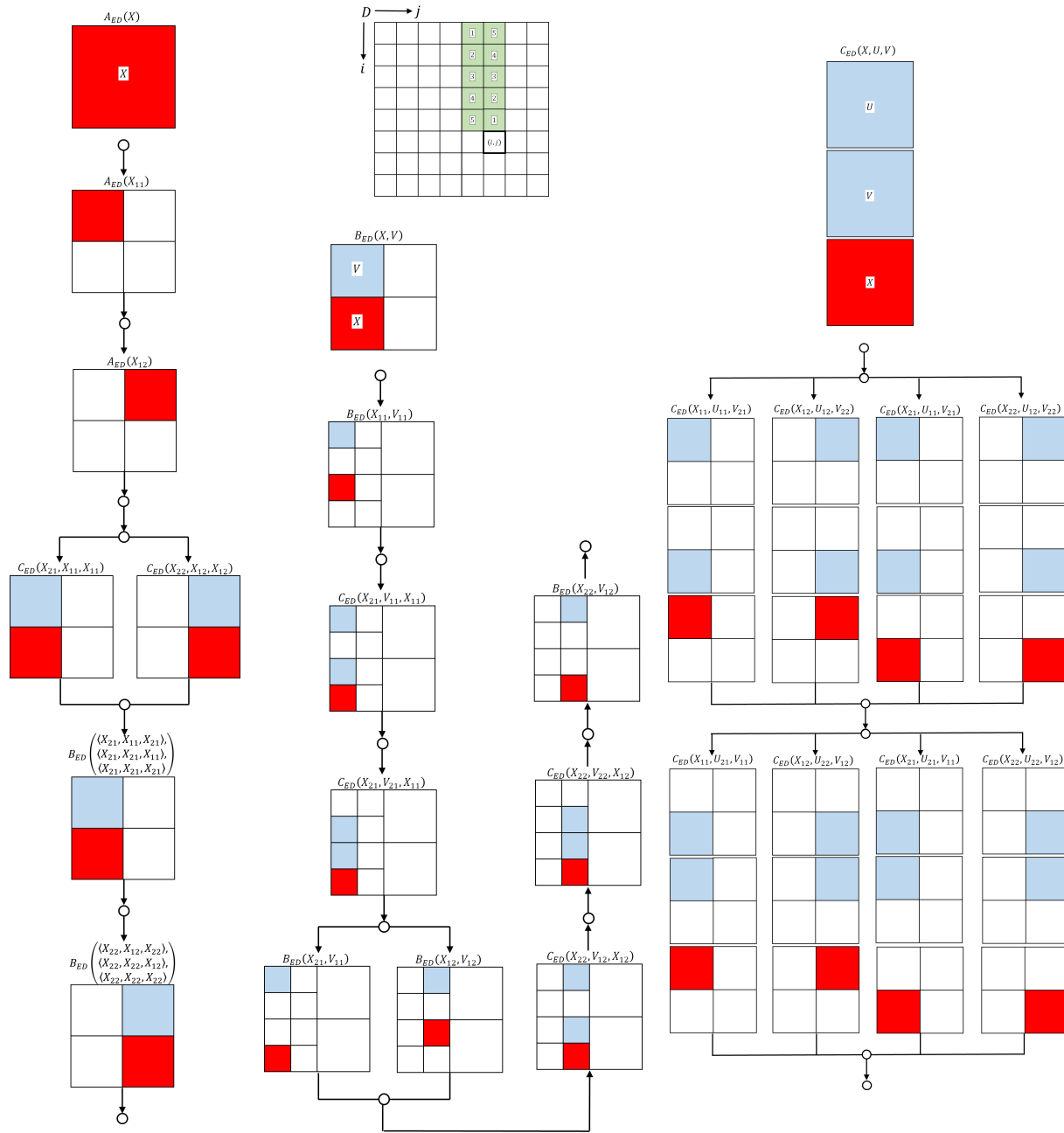
Figure A.28: Top center: Dependency graph for the egg dropping problem. Rest: A divide-and-conquer algorithm for solving the problem by computing $drops[n][n]$.

finding the kth most frequently occurring element, finding set union/intersection, finding the closest pair of points, finding a convex hull, and so on. Even if sorting was totally useless, it is an exceptionally interesting problem that leads to several stunning algorithms and mind-blowing analyses. We remember a quote often credited to Richard Feynman, "Science is like sex, sometimes something useful comes out, but that is not the reason why we do it." Due to the reasons mentioned above, sorting is a well studied problem and has a large literature.

Several algorithms have been discovered to solve the sorting problem. More than six decades of research has yielded over a hundred algorithms to solve the sorting problem,
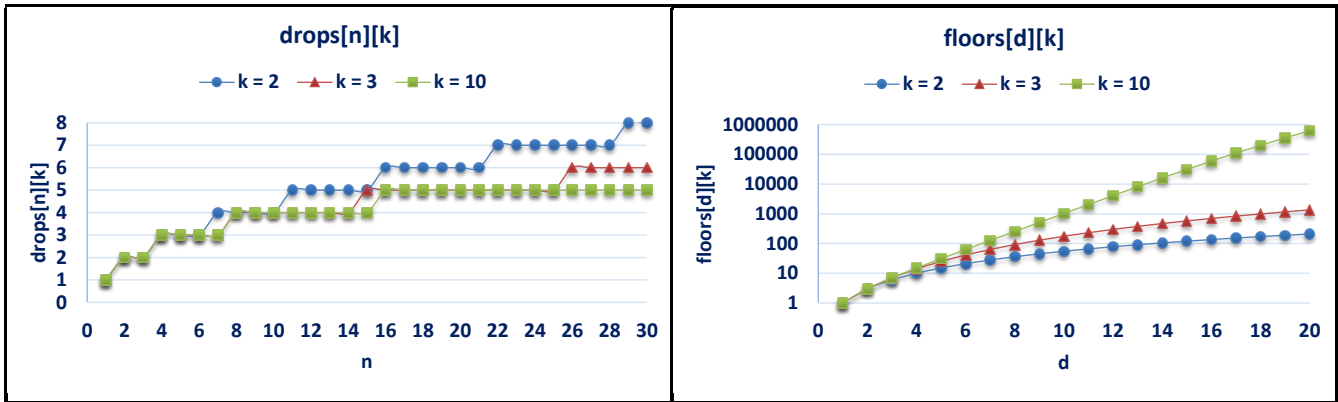
Figure A.29: Left: Plot for $drops[n][k]$ when n is varying from 1 to 30 and k is fixed at 2, 3, or 10. Right: Plot for $floors[d][k]$ when d is varying from 1 to 20 and k is fixed at 2, 3, or 10.

many of which are either minor or major variations of tens of standard algorithms. The sorting algorithms can be classified as in-place or not-in-place, stable or unstable, iterative or recursive, serial or parallel [Akl, 2014, Cole, 1988], internal memory or external memory, deterministic or randomized, adaptive [Estivill-Castro and Wood, 1992] or non-adaptive, and self-improving [Ailon et al., 2011] or non-self-improving.

In this section, we present recursive divide-and-conquer variants of bubble, selection, and insertion sorting algorithms.

Why care for divide-and-conquer algorithms?. Divide-and-conquer is an algorithm design strategy used to solve problems by breaking them into one or more subproblems, solving them, and combining their solutions to solve the original problem. Often, the subproblems are independent and the divide-and-conquer algorithms are implemented recursively.

Divide-and-conquer algorithms have the following advantages:
* ✭ They can be represented succinctly and can be analyzed for its complexities using *recurrence relations* [Bentley, 1980].
* ✭ They are usually *efficient* [Levitin, 2011] in the sense that they reduce the total number of computations.
* ✭ They can be *parallelized easily* [Mou and Hudak, 1988].
* ✭ They often are (or can be made) *cache-efficient* and *cache-oblivious* [Frigo et al., 1999, Chatterjee et al., 2002, Frens and Wise, 1997].

For example, some of the fastest sorting algorithms such as merge sort and quicksort are based on recursive divide-and-conquer.[3].

Why care for bubble, selection, & insertion sorts?. Among all sorting algorithms, bubble, selection, and insertion sorts [Levitin, 2011] are three of the most elementary algorithms that are widely taught to computer science students. The algorithms are popular majorly because they are arguably simple, easy-to-remember, and easy-to-program.

From unreliable sources, it is known that there is a tribe in the Amazon forests called *Zozo*. Every night, a volunteer from the tribe makes sure that the people are sorted based on age and they get their food share in that order. Interestingly, the tribe uses bubble sort to sort the people.

---

[3]The fastest sorting algorithm runs in $\mathcal{O}(1)$ time – simply call a `sort()` function from a standard library. :)

We too use bubble, selection, and insertion sorts depending on our moods and occasions. For stress management, we often recommend students to mentally sort the numbers from 100 to 1 using insertion sort, unlike counting numbers from 100 to 1.

Model. Analogous to cache-oblivious model (or ideal-cache model) [Frigo et al., 1999], we develop a *memory-oblivious model*. In this model, a human mind is assumed to consist of a constant number of processing elements (or processors) and a hierarchy of memories (short-term and long-term), as shown in Figure A.30. The amount of information that can be stored at memory levels near the processors are lesser than that those away from the processors. Similarly, time to access information from memory levels near the processors is smaller than that of higher memory levels. Also, the number of memory levels, memory sizes, etc varies from person to person. It is important to note that the traditional cache-aware or external-memory (or analogously memory-aware) model is not suitable because people do not know their memory sizes or the number of levels of memory they have.

When a processing element looks for some information in a memory-level and that information is not found, that leads to a memory miss. The total number of memory misses is measured as memory-complexity. The better the memory complexity for an algorithm, the fewer the number of thoughts required to check different memory levels for information and faster it is for the mind to execute the algorithm.

Divide-and-conquer variants of bubble, selection, & insertion sorts. Humans (i.e., minds) heavily use bubble, selection, and insertion sorts because these algorithms are simple and intuitive. On the other hand, computers (i.e., machines) heavily use merge sort and quicksort because these algorithms are very fast. Our aim is to improve the performance of the elementary sorting algorithms and save billions of minutes of time for millions of people in their sorting business.

In this section, we present divide-and-conquer variants of the elementary sorting algorithms. By the virtue of divide-and-conquer, such algorithms are memory-efficient and memory-oblivious.
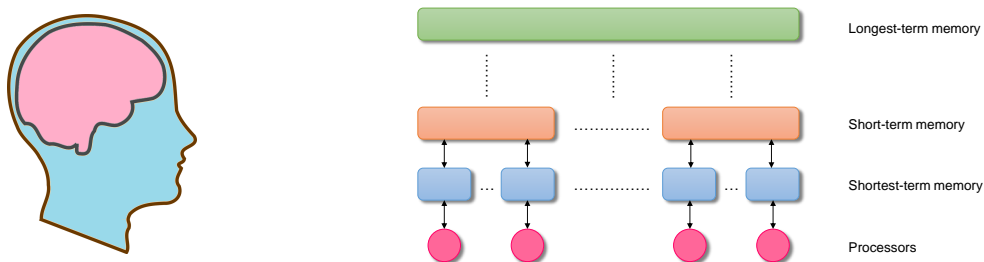
Figure A.30: Left: Place-holder of the human mind. Right: The memory-hierarchy of the human mind.

Our divide-and-conquer algorithms are memory-oblivious. Any person can use it irrespective of her/his memory hierarchy. The algorithms are provably memory-efficient too. This implies, the number of thoughts that accesses different memory levels to execute an algorithm in mind is minimized asymptotically and hence sorting can be done asymptotically much faster.

Students complain about non-remembrance and older adults complain about forgetfulness. Clinical observation [Kral, 1962] shows that memory loss occurs significantly in aging individuals. In algorithmic terms, it can be called the memory-shrinkage problem.

Using the results from [Bender et al., 2014], we can prove that our divide-and-conquer variants of sorting algorithms are memory-adaptive in the sense that they work optimally even if the memory size changes dynamically. This implies, the young and the old can use the proposed algorithms and minimize the time they take to sort out things.

In this section, we present divide-and-conquer variants of bubble, selection, and insertion sorts. Both recursive bubble and selection sorts use partition algorithms. On the other hand, the recursive insertion sort uses a merge algorithm. We explain the logic of recursive bubble sort in more details. The explanation to other two algorithms can be extended in a similar way.

Notations & terminologies. For all algorithms, we sort the n-sized array $A[0..n-1]$. For simplicity, we assume that n is a power of 2. In all recursive function calls, we use notations such as $\ell, h, m, \ell\ell, \ell h, r\ell, rh$, etc, all of which represent the indices in the array A. The notations ℓ, m, h mean low, mid, and high, respectively. Terms $\ell\ell$ and rm mean low in the left array and mid in the right array, respectively. Other terms can be defined in a similar way. When a subproblem size $h - \ell + 1$ or $\ell h - \ell\ell + 1$ becomes less than or equal to the base case size b, then we execute an iterative base case kernel having an algorithm-dependent logic. The terms used in the section are summarized in Table A.1.

Symbol	Meaning
A	array to be sorted
n	Input parameter
ℓ, h, m	Low, high, and mid
$\ell\ell$	Left subarray's low index
rh	Right subarray's high index
I-BS	Iterative bubble sort
R-BS	Recursive bubble sort
P-BS	Partition in bubble sort
M-IS	Merge in insertion sort

Table A.1: Standard notations used for the sorting algorithms. Other notations are such as I-SS, R-SS, etc are similarly defined.

Related work. Bubble sort [Friend, 1956, Gotlieb, 1963] is also called sinking sort, exchange selection, shuttle sort, propagation, or push-down sort. Variations of bubble sort exists such as cocktail sort [Knuth, 1998] (a.k.a shaker sort, bidirectional bubble sort, shaker sort, ripple sort, or shuffle sort), where the direction of bubbling alternates between left-to-right and right-to-left, and odd-even sort [Habermann, 1972] (a.k.a odd-even transposition sort or brick sort).

Selection sort has several variants such as cocktail sort (a.k.a shaker sort or dual selection sort), where both the minimum and maximum elements are found in every pass, and bingo sort [Bin,], which scans the remaining elements to find the greatest value and shifts all elements with that value to their final locations.

Insertion sort variants include Shell sort [Shell, 1959], where elements separated by a distance are compared; binary insertion sort, which uses binary search to find the exact location of the new elements to be inserted; binary merge sort, which uses binary insertion sort and merge sort; heap sort, where insertions and searches are performed with a sophisticated data structure called a heap; and library sort [Bender et al., 2006b] (a.k.a

gapped insertion sort), where small number of spaces are left unused to make gaps for the elements to be inserted. Insertion sort is generally faster than selection sort which almost always is faster than bubble sort.

The cache performance effects on sorting algorithms have been studied by [LaMarca and Ladner, 1999]. The lower-bounds on sorting in external-memory model is given by Aggarwal and Vitter [Aggarwal et al., 1988]. The serial cache-oblivious model was proposed in [Frigo et al., 1999].

I-BS(A, n)

1. **for** $i \leftarrow 0$ **to** $n - 1$ **do**
2.   **for** $j \leftarrow 0$ **to** $n - i - 1$ **do**
3.   **if** $A[j] > A[j + 1]$ **then**
4.   Swap$(A[j], A[j + 1])$

I-SS(A, n)

1. **for** $i \leftarrow 0$ **to** $n - 1$ **do**
2. $min \leftarrow i$
3.   **for** $j \leftarrow i$ **to** $n - 1$ **do**
4.   **if** $A[j] < A[min]$ **then**
5. $min \leftarrow j$
6.   Swap$(A[i], A[min])$

I-IS(A, n)

1. **for** $i \leftarrow 1$ **to** $n - 1$ **do**
2. $key \leftarrow a[i]$
3.   **for** $j \leftarrow i - 1$ **to** 0 **do**
4.   **if** $A[j] > key$ **then**
5. $A[j + 1] \leftarrow A[j]$
6. $A[j + 1] \leftarrow key$
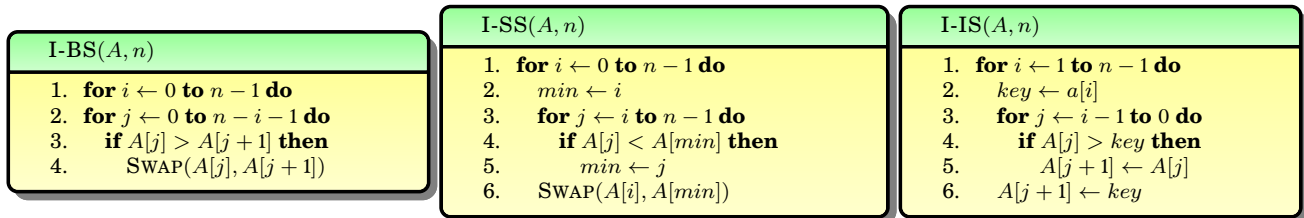
Figure A.31: Standard iterative algorithms for bubble sort (I-BS), selection sort (I-SS), and insertion sort (I-IS). Optimizations are not shown for pedagogical reasons.

A.13.1 Bubble sort

Bubble sort is one of the simplest sorting algorithms that is taught in an undergraduate algorithms course. Though it is a very slow algorithm taking $\mathcal{O}(n^2)$ time to sort n elements, it is simple to understand and easy to program.

A simple iterative algorithm called I-BS is given in Figure A.31. It has n iterations. In each iteration $i \ (\in [0, n - 1))$, every two adjacent elements j and $j + 1$, where $j \in [0, n - i - 1]$, are compared and sorted if they are not already in their sorted order. The number of comparisons at iteration i is $n - i$ and at the end of the iteration, the array $(i + 1)$th largest element will be in its correct position.

A recursive divide-and-conquer variant of bubble sort called R-BS is shown in Figure A.32. The aim is to sort the entire array $A[0..n - 1]$. The function R-BS(A, ℓ, h, n) sorts the subarray $A[\ell..h]$. The initial invocation to the algorithm is by calling R-BS$(A, 0, n - 1, n)$. The function R-BS in turn calls the P-BS function. The P-BS function is the partition function for bubble sort that brings the smallest $n/2$ elements to the left half and the largest $n/2$ elements to the right half of array A. Once the array A is partitioned, then the R-BS is recursively called onto the left and right halves in parallel to sort the two halves. After the two halves are sorted recursively, the entire array $A[0..n-1]$ will be sorted. When the subproblem reaches the base case, it is sorted using the standard iterative bubble sort logic.

The partition function P-BS$(A, \ell\ell, \ell h, r\ell, rh, n)$ partitions the elements such that after the partition, the largest element in $A[\ell\ell..\ell h]$ will be less than or equal to the smallest element in $A[r\ell..rh]$. The function works as follows.

In the base case, we use two loops: the outer-loop ranging over the right subarray and the inner-loop ranging over the left subarray. Using a logic similar to that of iterative bubble sort, the largest elements in the left subarray are pushed to the right subarray after every iteration.

```
┌─────────────────────────────┐   ┌──────────────────────────────────────────────┐
│  R-BS(A, ℓ, h, n)           │   │  P-BS(A, ℓℓ, ℓh, rℓ, rh, n)                    │
├─────────────────────────────┤   ├──────────────────────────────────────────────┤
│ 1.  if (h − ℓ + 1) ≤ b then │   │ 1.  if (ℓh − ℓℓ + 1) ≤ b then                  │
│ 2.    for i ← ℓ to h do     │   │ 2.    for i ← rℓ to rh do                      │
│ 3.      for j ← ℓ to ℓ+h−i−1 do │ │ 3.      for j ← ℓℓ to ℓh − 1 do             │
│ 4.        if A[j] > A[j+1] then │ │ 4.        if A[j] > A[j+1] then             │
│ 5.          SWAP(A[j], A[j+1]) │ │ 5.          SWAP(A[j], A[j+1])               │
│ 6.  else                    │   │ 6.        if A[ℓh] > A[i] then SWAP(A[ℓh], A[i]) │
│ 7.    m ← (ℓ + h)/2         │   │ 7.  else                                       │
│ 8.    P-BS(A, ℓ, m, m+1, h, n) │ │ 8.    ℓm ← (ℓℓ + ℓh)/2; rm ← (rℓ + rh)/2     │
│ 9.    parallel: R-BS(A, ℓ, m, n), │ │ 9.    parallel: P-BS(A, ℓℓ, ℓm, rℓ, rm, n), │
│                 R-BS(A, m+1, h, n) │ │                  P-BS(A, ℓm+1, ℓh, rm+1, rh, n) │
│                             │   │ 10.   parallel: P-BS(A, ℓℓ, ℓm, rm+1, rh, n), │
│                             │   │                  P-BS(A, ℓm+1, ℓh, rℓ, rm, n) │
└─────────────────────────────┘   └──────────────────────────────────────────────┘
```
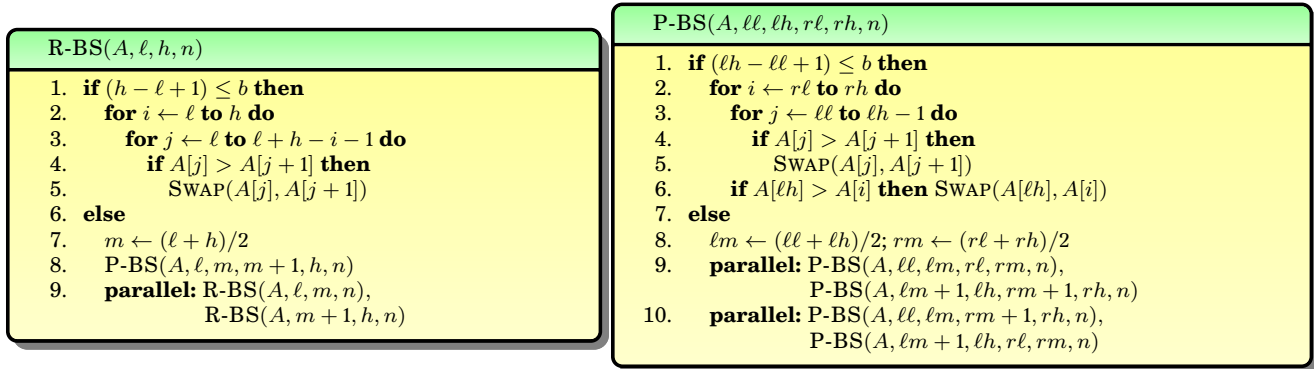
Figure A.32: A recursive divide-and-conquer variant of bubble sort. Initial call to the recursive algorithm is R-BS$(A, 0, n − 1, n)$, where $A[0..n − 1]$ is the array to be sorted.

In the recursion case, the P-BS calls itself four times. The reason for requiring four function calls is simple. Let ℓm and rm be the midpoints of the left and right subarray, respectively. The left subarray $A[\ell\ell..\ell h]$ can be divided into two subarrays $A[\ell\ell..\ell m]$ and $A[(\ell m + 1)..\ell h]$ and the right subarray $A[r\ell..rh]$ can be divided into two subarrays $A[r\ell..rm]$ and $A[(rm + 1)..rh]$. This means there are a total of four possible combinations of left and right subsubarrays. The P-BS function invokes two functions that work on different regions of the array, in parallel. In the next step, two more functions are invoked in parallel that work on disjoint regions. After the four invocations the larger elements would have moved to the right subarray leaving the smaller elements in the left subarray.

> **Theorem 14 (Bubble sort correctness).** *The divide-and-conquer variant of bubble sort algorithm i.e.,* R-BS *correctly sorts the input array.*

Proof. We use mathematical induction to prove the theorem. First we prove the correctness of P-BS function. Then we prove R-BS correct. For simplicity, we assume that n and b are powers of 2 such that $n \geq b$. We term $A[\ell\ell..\ell\ell h]$ and $A[r\ell..rh]$ as left and right input subarrays, respectively.

(1) [*Correctness of* P-BS.]
Basis. The logic of the base case when the input subarray is of size b is straightforward. The external loop runs b times and in each iteration, one of the larger elements sifts to the right subarray.
Induction. We assume that P-BS works correctly when the input subarrays are of size 2^k for some k, such that $2^k \geq b$. We need to prove that P-BS works for input subarrays of size 2^{k+1}.

Let $Q_1, Q_2, Q_3,$ and Q_4, where Q stands for "quarter", represent the locations $A[\ell\ell..\ell m], A[(\ell m + 1)..\ell h], A[r\ell..rm],$ and $A[(rm + 1)..rh]$, respectively, where each subarray is of size 2^k. Let $W, X, Y,$ and Z be the initial sets of numbers present at $Q_1, Q_2, Q_3,$ and Q_4, respectively. Let SMALL(S_1, S_2) (resp. LARGE(S_1, S_2)) of two sets S_1 and S_2 of numbers represent a set consisting of the smallest half (resp. largest half) of the numbers from sets S_1 and S_2. Also, let $S_1 \leq S_2$ denote that all elements of S_1 is less than or equal to all elements of S_2.

After execution of line 8, the states of the four quarters of the array A are $Q_1 = W, Q_2 = X, Q_3 = Y,$ and $Q_4 = Z$. After execution of line 9, the states of the four quarters of the array

A are: $Q_1 = \text{SMALL}(W, Y)$, $Q_2 = \text{SMALL}(X, Z)$, $Q_3 = \text{LARGE}(W, Y)$, and $Q_4 = \text{LARGE}(X, Z)$. After execution of line 10, the states of the four quarters of the array A are:

* $Q_1 = \text{SMALL}(\text{SMALL}(W, Y), \text{LARGE}(X, Z))$
* $Q_2 = \text{SMALL}(\text{SMALL}(X, Z), \text{LARGE}(W, Y))$
* $Q_3 = \text{LARGE}(\text{SMALL}(X, Z), \text{LARGE}(W, Y))$
* $Q_4 = \text{LARGE}(\text{SMALL}(W, Y), \text{LARGE}(X, Z))$

It is easy to see that

* $Q_1 \le Q_4$
* $Q_2 \le Q_3$
* $Q_1 \le \text{SMALL}(W, Y) \le \text{LARGE}(W, Y) \le Q_3$
* $Q_2 \le \text{SMALL}(X, Z) \le \text{LARGE}(X, Z) \le Q_4$

As $Q_1 \le Q_3$, $Q_1 \le Q_4$, $Q_2 \le Q_3$, and $Q_2 \le Q_4$, the input subarrays of size 2^{k+1} have been partitioned.

(2) [*Correctness of* R-BS.]
Basis. The base case when the input subarray is of size b is exactly same as the standard iterative bubble sort and hence is correct.
Induction. We assume that R-BS works correctly when the input subarrays are of size 2^k for some k, such that $2^k \ge b$. We need to prove that R-BS works for input subarrays of size 2^{k+1}. We know that the P-BS function is correct and hence after line 8, the left subarray ($A[\ell..m]$) and the right subarray ($A[(m+1)..h]$) would be partitioned such that the largest element in the left subarray will not be greater than the smallest element in the right subarray. Then after line 9, we recursively sort the subarrays without affecting the partition constraint and hence the total subarray will be sorted. $\qquad\square$

Complexity analysis

For function $f \in \{\text{I-BS}, \text{R-BS}, \text{P-BS}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity and span of f, respectively, to sort an n-sized array. Then

$$Q_{\text{I-BS}}(n) = \sum_{i=0}^{n-1} \Theta\left(((n-i)/B) + 1\right) = \Theta\left(n^2/B + n\right).$$

$$
\begin{aligned}
Q_{\text{R-BS}}(n) &= Q_{\text{P-BS}}(n) = \mathcal{O}\left(n/B + 1\right) & \text{if } n \le \gamma M, \\
Q_{\text{R-BS}}(n) &= 2Q_{\text{R-BS}}\left(n/2\right) + Q_{\text{P-BS}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > \gamma M, \\
Q_{\text{P-BS}}(n) &= 4Q_{\text{P-BS}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > \gamma M.
\end{aligned}
$$

$$
\begin{aligned}
T_{\text{R-BS}}(n) &= T_{\text{P-BS}}(n) = \mathcal{O}\left(1\right) & \text{if } n = 1, \\
T_{\text{R-BS}}(n) &= T_{\text{R-BS}}\left(n/2\right) + T_{\text{P-BS}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > 1, \\
T_{\text{P-BS}}(n) &= 2T_{\text{P-BS}}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{if } n > 1.
\end{aligned}
$$

where, γ is a suitable constant. The cache complexity of a subproblem of size n when it fits cache i.e., $n \le \gamma M$, is $\Theta\left(n/B + 1\right) = \mathcal{O}\left(M/B + 1\right)$. The cache complexity of a subproblem when it does not fit into cache is recursively computed using its subproblems. Solving the recurrences we get $Q_{\text{R-BS}}(n) = \mathcal{O}\left(n^2/(BM) + n^2/M^2 + n/B + n/M + 1\right)$ and $T_{\text{R-BS}}(n) = \Theta\left(n\right)$.

> The R-BS algorithm achieves $T_1(n) = \Theta\left(n^2\right)$, $Q_1(n) = \mathcal{O}\left(\frac{n^2}{BM} + \frac{n^2}{M^2} + \frac{n}{B} + 1\right)$, $T_\infty(n) = \Theta\left(n\right)$, parallelism $= \Theta\left(n\right)$, and $S_\infty(n) = \Theta\left(n\right)$.

A.13.2   Selection sort

Selection sort is another slow running sorting algorithm that sorts n numbers in $\mathcal{O}\left(n^2\right)$ time.

An iterative algorithm called I-SS is given in Figure A.31. The algorithm has n iterations. In each iteration $i \in [0, n-1]$, the position of the minimum element, denoted by min is found in the range $A[i..n-1]$. Then the elements $A[min]$ and $A[i]$ are swapped, which makes sure that after iteration i, the ith smallest element is in its correct position.

R-SS(A, ℓ, h, n)
1. **if** $(h - \ell + 1) \leq b$ **then**
2.   **for** $i \leftarrow \ell$ **to** $h - 1$ **do**
3. $min \leftarrow i$
4.   **for** $j \leftarrow i + 1$ **to** h **do**
5.   **if** $A[j] < A[min]$ **then**
6. $min \leftarrow j$
7.   **if** $min \neq i$ **then**
8.   SWAP$(A[i], A[min])$
9. **else**
10. $m \leftarrow (\ell + h)/2$
11.   P-SS$(A, \ell, m, m+1, h, n)$
12.   **parallel:** R-SS(A, ℓ, m, n),
R-SS$(A, m+1, h, n)$

P-SS$(A, \ell\ell, \ell h, r\ell, rh, n)$
1. **if** $(\ell h - \ell\ell + 1) \leq b$ **then**
2.   **for** $i \leftarrow \ell\ell$ **to** ℓh **do**
3. $min \leftarrow i$
4.   **for** $j \leftarrow r\ell$ **to** rh **do**
5.   **if** $A[j] < A[min]$ **then** $min \leftarrow j$
6.   **if** $min \neq i$ **then**
7.   SWAP$(A[i], A[min])$
8. **else**
9. $\ell m \leftarrow (\ell\ell + \ell h)/2; rm \leftarrow (r\ell + rh)/2$
10.   **parallel:** P-SS$(A, \ell\ell, \ell m, r\ell, rm, n)$,
P-BS$(A, \ell m + 1, \ell h, rm + 1, rh, n)$
11.   **parallel:** P-SS$(A, \ell\ell, \ell m, rm + 1, rh, n)$,
P-SS$(A, \ell m + 1, \ell h, r\ell, rm, n)$

Figure A.33: A recursive divide-and-conquer variant of selection sort. Initial call to the recursive algorithm is R-SS$(A, 0, n-1, n)$, where $A[0..n-1]$ is the array to be sorted.

A recursive divide-and-conquer variant of selection sort R-SS is shown in Figure A.33. The initial invocation to the algorithm is by calling R-SS$(A, 0, n-1, n)$. The recursive structure of the algorithm is exactly the same as that of bubble sort. The R-SS function invokes P-SS function to partition the array A into two halves where the largest element in the first half is lesser than or equal to the smallest element in the second half. After the partition, the R-SS functions are invoked on the two halves to sort them recursively. The partition function P-SS calls itself four times: two P-SS functions in two parallel steps. The only difference between the bubble sort and selection sort divide-and-conquer algorithms are the base cases of R-SS and P-SS functions.

The base case kernel of R-SS function is equivalent to that of the standard iterative I-SS algorithm. The P-SS$(A, \ell\ell, \ell h, r\ell, rh, n)$ function base case kernel in each iteration finds an element that belongs to the left subarray and pushes it into the left subarray. After several iterations, the elements in the two subarrays would be partitioned in such a way that the largest element in the left subarray $A[\ell\ell..\ell h]$ would be lesser than or equal to the smallest element in the right subarray $A[r\ell..rh]$.

> **Theorem 15 (Selection sort correctness).** *The divide-and-conquer variant of selection sort algorithm i.e.,* R-SS *correctly sorts the input array.*

Proof. We use mathematical induction to prove the theorem. First we prove the correctness of P-SS function. Then we prove R-SS correct. For simplicity, we assume that n and b are powers of 2 such that $n \geq b$. We term $A[\ell\ell..\ell\ell h]$ and $A[r\ell..rh]$ as left and right input subarrays, respectively.

(1) [*Correctness of* P-SS.]
Basis. The logic of the base case when the input subarray is of size b is straightforward.

The external loop runs b times. In each iteration, we find the index of one the smallest elements in the right subarray and if the element on the right subarray is less than an element in the left subarray, then we swap it. In this way, the smallest b elements will sift to the left subarray.

Induction. The argument is similar to that given in Theorem 14.

(2) [*Correctness of* R-SS.]

Basis. The base case when the input subarray is of size b is exactly same as the standard iterative selection sort and hence is correct.

Induction. The argument is similar to that given in Theorem 14. □

Complexity analysis

For function $f \in \{\text{I-SS}, \text{R-SS}, \text{P-SS}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity and span of f, respectively, to sort an n-sized array. Then

$$Q_{\text{I-SS}}(n) = \sum_{i=0}^{n-1} \Theta\left(((n-i)/B) + 1\right) = \Theta\left(n^2/B + n\right).$$

$$\begin{aligned}
Q_{\text{R-SS}}(n) &= Q_{\text{P-SS}}(n) = \mathcal{O}\left(n/B + 1\right) && \text{if } n \leq \gamma M, \\
Q_{\text{R-SS}}(n) &= 2Q_{\text{R-SS}}(n/2) + Q_{\text{P-SS}}(n/2) + \mathcal{O}(1) && \text{if } n > \gamma M, \\
Q_{\text{P-SS}}(n) &= 4Q_{\text{P-SS}}(n/2) + \mathcal{O}(1) && \text{if } n > \gamma M.
\end{aligned}$$

$$\begin{aligned}
T_{\text{R-SS}}(n) &= T_{\text{P-SS}}(n) = \mathcal{O}(1) && \text{if } n = 1, \\
T_{\text{R-SS}}(n) &= T_{\text{R-SS}}(n/2) + T_{\text{P-SS}}(n/2) + \mathcal{O}(1) && \text{if } n > 1, \\
T_{\text{P-SS}}(n) &= 2T_{\text{P-SS}}(n/2) + \mathcal{O}(1) && \text{if } n > 1.
\end{aligned}$$

where, γ is a suitable constant. Solving the recurrences we get $T_{\text{R-SS}}(n) = \Theta(n)$ and $Q_{\text{R-SS}}(n) = \mathcal{O}\left(n^2/(BM) + n^2/M^2 + n/B + n/M + 1\right)$.

> The R-SS algorithm achieves $T_1(n) = \Theta(n^2)$, $Q_1(n) = \mathcal{O}\left(\frac{n^2}{BM} + \frac{n^2}{M^2} + \frac{n}{B} + 1\right)$, $T_\infty(n) = \Theta(n)$, parallelism $= \Theta(n)$, and $S_\infty(n) = \Theta(n)$.

A.13.3 Insertion sort

Insertion sort is a pretty fast algorithm that sorts a set of n elements in $\mathcal{O}(n^2)$ worst case time and $\mathcal{O}(n \log n)$ average case time.

An iterative insertion sort algorithm is given in Figure A.31. The algorithm has $n - 1$ iterations. In iteration i ($\in [1, n-1)$, the array element $A[i]$ will be inserted in a sorted position in the range $A[0 \ldots i-1]$. After each iteration i, the subarray $A[0 \ldots i]$ will be sorted. The runtime complexity is data-sensitive.

A recursive divide-and-conquer variant of the insertion sort is given in Figure A.34. The initial invocation to the algorithm is by calling R-IS$(A, 0, n-1, n)$. The recursive structure of the algorithm is similar to bubble and selection sorts but order of function calls are different. The R-IS function calls itself twice to sort the left and right halves separately and simultaneously. Then it invokes M-IS function to merge the elements from the two halves using the logic of the iterative insertion sort. After the merge, the entire array would have been sorted.

The merge function M-IS calls itself a total of three times: the first two calls in parallel and then a third serial call. The first call M-IS$(A, \ell\ell, \ell m, r\ell, rm, n)$ brings the smallest elements to $A[\ell\ell..\ell m]$ in sorted order. The second call M-IS$(A, \ell m+1, \ell h, rm+1, rh, n)$ brings the

```
┌─ R-IS(A, ℓ, h, n) ──────────────────────┐
│  1.  if (h − ℓ + 1) ≤ b then            │
│  2.     for i ← ℓ + 1 to h do           │
│  3.        key ← A[i]                    │
│  4.        j ← i − 1                     │
│  5.        while j ≥ ℓ & A[j] > key do   │
│  6.           A[j + 1] ← A[j]            │
│  7.           j ← j − 1                   │
│  8.        A[j + 1] ← key                │
│  9.  else                                │
│ 10.     m ← (ℓ + h)/2                     │
│ 11.     parallel: R-IS(A, ℓ, m, n),      │
│                   R-IS(A, m + 1, h, n)    │
│ 12.     M-IS(A, ℓ, m, m + 1, h, n)        │
└──────────────────────────────────────────┘
```

```
┌─ M-IS(A, ℓℓ, ℓh, rℓ, rh, n) ─────────────────────────────┐
│  1.  if (ℓh − ℓℓ + 1) ≤ b then                          │
│  2.     if A[ℓh] > A[rℓ] then                            │
│  3.        for i ← rℓ to rh do                           │
│  4.           key ← A[i]; j ← i − 1                      │
│  5.           while j ≥ rℓ & A[j] > key do               │
│  6.              A[j + 1] ← A[j]; j ← j − 1              │
│  7.           if A[ℓh] > key then                        │
│  8.              A[rℓ] ← A[ℓh]; j ← ℓh − 1               │
│  9.              while j ≥ ℓℓ & A[j] > key do            │
│ 10.                 A[j + 1] ← A[j]; j ← j − 1           │
│ 11.              A[j + 1] ← key                          │
│ 12.     else                                             │
│ 13.        ℓm ← (ℓℓ + ℓh)/2; rm ← (rℓ + rh)/2            │
│ 14.        parallel: M-IS(A, ℓℓ, ℓm, rℓ, rm, n),         │
│                      M-IS(A, ℓm + 1, ℓh, rm + 1, rh, n)   │
│ 15.        M-IS(A, ℓm + 1, ℓh, rℓ, rm, n)                 │
└───────────────────────────────────────────────────────────┘
```
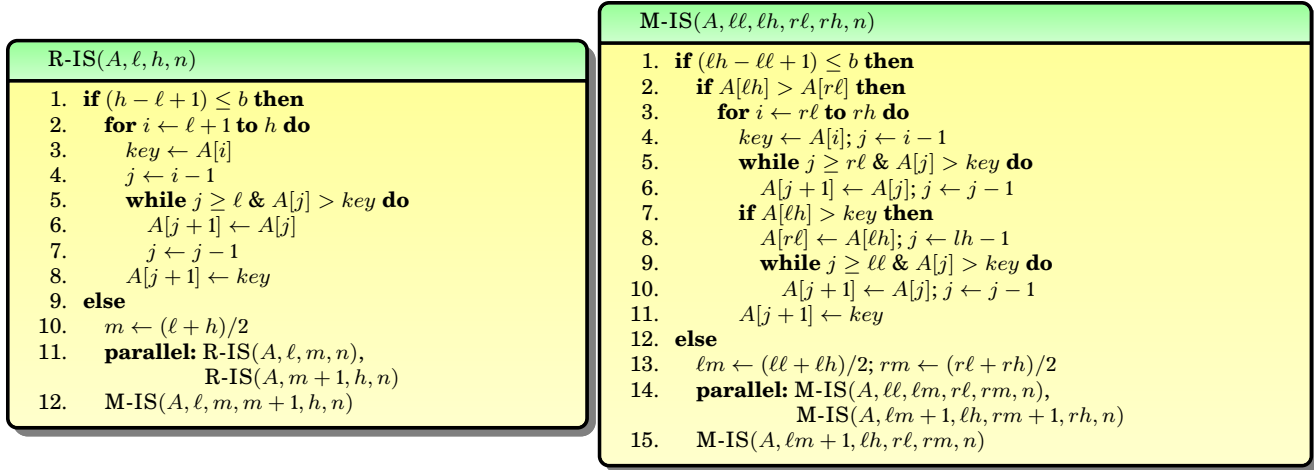
Figure A.34: A recursive divide-and-conquer variant of insertion sort. Initial call to the recursive algorithm is R-IS($A, 0, n − 1, n$), where $A[0..n − 1]$ is the array to be sorted.

largest elements to $A[rm + 1..rh]$ in sorted order. The third call M-IS($A, \ell m + 1, \ell h, r\ell, rm, n$) brings the remaining elements to $A[\ell m + 1..rm]$ in the sorted order.

The base case kernel of R-IS function is equivalent to that of the I-IS algorithm. The M-IS($A, \ell\ell, \ell h, r\ell, rh, n$) function base case kernel merges two sorted subarrays into sorted order (retaining the sorted elements in those two input subarrays). In iteration k, the kth element of the right subarray gets merged with its previous elements (in the right subarray) and with the elements of left subarray. After $rh − r\ell + 1$ iterations, the elements in the two subarrays would be merged into a sorted order – the left subarray will be sorted, the right subarray will be sorted, and the last element of the left subarray will be less than or equal to the first element of the right subarray.

> **Theorem 16 (Insertion sort correctness).** *The divide-and-conquer variant of insertion sort algorithm i.e.,* R-IS *correctly sorts the input array.*

Proof. We use mathematical induction to prove the theorem. First we prove the correctness of M-IS function. Then we prove R-IS correct. For simplicity, we assume that n and b are powers of 2 such that $n \geq b$. We term $A[\ell\ell..\ell h]$ and $A[r\ell..rh]$ as left and right input subarrays, respectively.

(1) [*Correctness of* M-IS.]
Basis. The logic of the base case when the input subarray is of size b is straightforward. The external loop runs b times for all elements in the right subarray. In each iteration, the element from the right subarray is inserted into its correct position towards it left by shifting elements.
Induction. We assume that M-IS works correctly when the input subarrays are of size 2^k for some k, such that $2^k \geq b$. We need to prove that M-IS works for input subarrays of size 2^{k+1}.

Let $Q_1, Q_2, Q_3,$ and Q_4, where Q stands for "quarter", represent the locations $A[\ell\ell..\ell m], A[(\ell m + 1)..\ell h], A[r\ell..rm],$ and $A[(rm + 1)..rh]$, respectively, where each subarray is of size 2^k. Let $W, X, Y,$ and Z be the initial sets of numbers present at $Q_1, Q_2, Q_3,$ and Q_4, respectively. Let SMALL(S_1, S_2) (resp. LARGE(S_1, S_2)) of two sets S_1 and S_2 of numbers represent a set consisting of the smallest half (resp. largest half) of the numbers from sets S_1 and S_2. Also,

191

let $S_1 \leq S_2$ denote that all elements of S_1 is less than or equal to all elements of S_2. As the input subarrays are sorted we have $W \leq X$ and hence we can write $W = \text{SMALL}(W, X)$ and $X = \text{LARGE}(W, X)$. Similarly, $Y \leq Z$ and therefore we can write $Y = \text{SMALL}(Y, Z)$ and $Z = \text{LARGE}(Y, Z)$.

After execution of line 13, the states of the four quarters of the array A are $Q_1 = W$, $Q_2 = X$, $Q_3 = Y$, and $Q_4 = Z$. After execution of line 14, the states of the four quarters of the array A are: $Q_1 = \text{SMALL}(W, Y)$, $Q_2 = \text{SMALL}(X, Z)$, $Q_3 = \text{LARGE}(W, Y)$, and $Q_4 = \text{LARGE}(X, Z)$. After execution of line 15, the states of the four quarters of the array A are:

✶ $Q_1 = \text{SMALL}(W, Y)$
✶ $Q_2 = \text{SMALL}(\text{SMALL}(X, Z), \text{LARGE}(W, Y))$
✶ $Q_3 = \text{LARGE}(\text{SMALL}(X, Z), \text{LARGE}(W, Y))$
✶ $Q_4 = \text{LARGE}(X, Z)$

It is easy to see that

✶ $Q_1 = \text{SMALL}(\text{SMALL}(W, X), \text{SMALL}(Y, Z)) = \text{SMALL}(\text{SMALL}(W, Y), \text{SMALL}(X, Z)) \leq Q_2$
✶ $Q_2 \leq Q_3$
✶ $Q_3 \leq \text{LARGE}(\text{LARGE}(X, Z), \text{LARGE}(W, Y)) = \text{LARGE}(\text{LARGE}(W, X), \text{LARGE}(Y, Z)) = Q_4$

As $Q_1 \leq Q_2 \leq Q_3 \leq Q_4$, the input subarrays of size 2^{k+1} have been merged.

(2) [*Correctness of* R-IS.]

Basis. The base case when the input subarray is of size b is exactly same as the standard iterative insertion sort and hence is correct.

Induction. We assume that R-IS works correctly when the input subarrays are of size 2^k for some k, such that $2^k \geq b$. We need to prove that R-IS works for input subarrays of size 2^{k+1}. We know that the R-IS function is correct and hence after line 11, the left subarray ($A[\ell..m]$) would be sorted and the right subarray ($A[(m+1)..h]$) would be sorted. Then after line 12, we merge the two subarrays. As we have shown that the merge function M-IS is correct, the entire subarray of size 2^{k+1} would be merged and sorted. □

Complexity analysis

For function $f \in \{\text{I-IS}, \text{R-IS}, \text{M-IS}\}$, let $W_f(n)$, $Q_f(n)$, and $T_f(n)$ denote the work, serial cache complexity, and span of f, respectively, to sort an n-sized array. We use the term W instead of T_1 for simplicity. Then

$$Q_{\text{I-IS}}(n) = \sum_{i=0}^{n-1} \mathcal{O}\left(((n-i)/B) + 1\right) = \mathcal{O}\left(n^2/B + n\right).$$

$$
\begin{array}{ll}
W_{\text{R-IS}}(n) = W_{\text{M-IS}}(n) = \mathcal{O}(1) & \text{if } n = 1, \\
W_{\text{R-IS}}(n) = 2W_{\text{R-IS}}(n/2) + W_{\text{M-IS}}(n/2) + \Theta(1) & \text{if } n > 1, \\
W_{\text{M-IS}}(n) = 3W_{\text{M-IS}}(n/2) + \Theta(1) & \text{if } n > 1.
\end{array}
$$

$$
\begin{array}{ll}
Q_{\text{R-IS}}(n) = Q_{\text{M-IS}}(n) = \mathcal{O}(n/B + 1) & \text{if } n \leq \gamma M, \\
Q_{\text{R-IS}}(n) = 2Q_{\text{R-IS}}(n/2) + Q_{\text{M-IS}}(n/2) + \mathcal{O}(1) & \text{if } n > \gamma M, \\
Q_{\text{M-IS}}(n) = 3Q_{\text{M-IS}}(n/2) + \mathcal{O}(1) & \text{if } n > \gamma M.
\end{array}
$$

$$
\begin{array}{ll}
T_{\text{R-IS}}(n) = T_{\text{M-IS}}(n) = \mathcal{O}(1) & \text{if } n = 1, \\
T_{\text{R-IS}}(n) = T_{\text{R-IS}}(n/2) + T_{\text{M-IS}}(n/2) + \mathcal{O}(1) & \text{if } n > 1, \\
T_{\text{M-IS}}(n) = 2T_{\text{M-IS}}(n/2) + \mathcal{O}(1) & \text{if } n > 1.
\end{array}
$$

where, γ is a suitable constant. Solving the recurrences we get $W_{\text{R-IS}}(n) = \mathcal{O}\left(n^{\log 3}\right)$, $T_{\text{R-IS}}(n) = \mathcal{O}(n)$, and $Q_{\text{R-IS}}(n) = \mathcal{O}\left(n^{\log 3}/(BM^{(\log 3)-1}) + n^{\log 3}/M^{\log 3} + n/B + n/M + 1\right)$. All logarithms are taken to the base 2.

We compute the serial cache complexity of R-IS. We initially find $Q_{\text{M-IS}}$ and then use it to compute $Q_{\text{R-IS}}$. We assume that $n/2^k = \gamma M$ for some γ.

$$
\begin{aligned}
Q_{\text{M-IS}}(n) &\leq 3Q_{\text{M-IS}}\left(\frac{n}{2}\right) + c = 3\left(3Q_{\text{M-IS}}\left(\frac{n}{2^2}\right) + c\right) + c = 3^2 Q_{\text{M-IS}}\left(\frac{n}{2^2}\right) + 3c + c \\
&= 3^k Q_{\text{M-IS}}\left(\frac{n}{2^k}\right) + c\left(3^{k-1} + 3^{k-2} + \cdots + 1\right) \\
&\leq 3^k\left(Q_{\text{M-IS}}\left(\frac{n}{2^k}\right) + c\right) \leq c\left(\frac{n}{M}\right)^{\log 3}\left(\frac{M}{B} + 1\right) \\
&= \mathcal{O}\left(\frac{n^{\log 3}}{BM^{\log 3 - 1}} + \frac{n^{\log 3}}{M^{\log 3}}\right)
\end{aligned}
$$

Plugging in $Q_{\text{M-IS}}$ value into the recurrence of $Q_{\text{R-IS}}$, we get

$$
\begin{aligned}
Q_{\text{R-IS}}(n) &\leq 2Q_{\text{R-IS}}\left(\frac{n}{2}\right) + Q_{\text{M-IS}}\left(\frac{n}{2}\right) + c \leq 2Q_{\text{R-IS}}\left(\frac{n}{2}\right) + c\frac{n^{\log 3}}{BM^{\log 3 - 1}} + c\frac{n^{\log 3}}{M^{\log 3}} + c \\
&\leq 2\left(2Q_{\text{R-IS}}\left(\frac{n}{2^2}\right) + c\frac{\left(\frac{n}{2}\right)^{\log 3}}{BM^{\log 3 - 1}} + c\frac{\left(\frac{n}{2}\right)^{\log 3}}{M^{\log 3}} + c\right) + c\frac{n^{\log 3}}{BM^{\log 3 - 1}} + c\frac{n^{\log 3}}{M^{\log 3}} + c \\
&= 2^2 Q_{\text{R-IS}}\left(\frac{n}{2^2}\right) + 2c + c + 2c\frac{\left(\frac{n}{2}\right)^{\log 3}}{BM^{\log 3 - 1}} + 2c\frac{\left(\frac{n}{2}\right)^{\log 3}}{M^{\log 3}} + c\frac{n^{\log 3}}{BM^{\log 3 - 1}} + c\frac{n^{\log 3}}{M^{\log 3}} \\
&= 2^2 Q_{\text{R-IS}}\left(\frac{n}{2^2}\right) + c\left(\frac{n^{\log 3}}{BM^{\log 3 - 1}} + \frac{n^{\log 3}}{M^{\log 3}}\right)\left(1 + \frac{1}{2^{\log 3 - 1}}\right) + c\left(2 + 1\right) \\
&\leq 2^k Q_{\text{R-IS}}\left(\frac{n}{2^k}\right) + c\left(\frac{n^{\log 3}}{BM^{\log 3 - 1}} + \frac{n^{\log 3}}{M^{\log 3}}\right)\left(1 + \cdots + \frac{1}{\left(2^{\log 3 - 1}\right)^{k-1}}\right) + c\left(2^{k-1} + \cdots + 1\right) \\
&\leq c\frac{n}{M}\left(\frac{M}{B} + 1\right) + c\frac{n^{\log 3}}{BM^{\log 3 - 1}} + c\frac{n^{\log 3}}{M^{\log 3}} \\
&= \mathcal{O}\left(\frac{n^{\log 3}}{BM^{\log 3 - 1}} + \frac{n^{\log 3}}{M^{\log 3}} + \frac{n}{B} + \frac{n}{M} + 1\right)
\end{aligned}
$$

> The R-IS algorithm achieves $T_1(n) = \mathcal{O}\left(n^{\log 3}\right)$, $Q_1(n) = \mathcal{O}\left(\frac{n^{\log 3}}{BM^{(\log 3)-1}} + \frac{n^{\log 3}}{M^{\log 3}} + \frac{n}{B} + \frac{n}{M} + 1\right)$, $T_\infty(n) = \mathcal{O}(n)$, parallelism $= \Theta\left(n^{(\log 3)-1}\right)$, and $S_\infty(n) = \Theta(n)$.

A.13.4 Experimental results

Table A.2 summarizes the theoretical complexities of our algorithms. This section presents empirical results showing the performance improvements from high parallelism, better cache complexity, and less work.

Problem	Iterative algorithm				Divide-and-conquer algorithm			
	T_1	Q_1	T_∞	T_1/T_∞	T_1	Q_1	T_∞	T_1/T_∞
Bubble sort [Levitin, 2011]	$\Theta\left(n^2\right)$	$\Theta\left(\frac{n^2}{B}\right)$	$\Theta\left(n^2\right)$	$\Theta\left(1\right)$	$\Theta\left(n^2\right)$	$\mathcal{O}\left(\frac{n^2}{BM}\right)$	$\Theta\left(n\right)$	$\Theta\left(n\right)$
Selection sort [Levitin, 2011]	$\Theta\left(n^2\right)$	$\Theta\left(\frac{n^2}{B}\right)$	$\Theta\left(n^2\right)$	$\Theta\left(1\right)$	$\Theta\left(n^2\right)$	$\mathcal{O}\left(\frac{n^2}{BM}\right)$	$\Theta\left(n\right)$	$\Theta\left(n\right)$
Insertion sort [Levitin, 2011]	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^2}{B}\right)$	$\mathcal{O}\left(n^2\right)$	$\Theta\left(1\right)$	$\mathcal{O}\left(n^{\log 3}\right)$	$\mathcal{O}\left(\frac{n^{\log 3}}{BM^{\log 3 - 1}}\right)$	$\mathcal{O}\left(n\right)$	$\Omega\left(n\right)$

Table A.2: Work (T_1), serial cache complexity (Q_1), span (T_∞), and parallelism (T_1/T_∞) of iterative and recursive divide-and-conquer algorithms for the bubble, selection, and insertion sorts. For the cache complexity, only the most significant terms are shown.

Setup. Our experiments were performed on a multicore machine with dual-socket 8-core 2.7 GHz Intel Sandy Bridge processors ($2 \times 8 = 16$ cores in total) and 32 GB RAM. Each core was linked to a 32 KB private L1 cache and a 256 KB private L2 cache. All cores in a processor shared a 20 MB L3 cache. With hyper-threading, we can simulate a total of 32 threads from 16 cores. The algorithms were implemented in C++. Intel Cilk Plus extension was used to parallelize the programs. Intel C++ Compiler v13.0 (`icc`) was used to compile the implementations with parameters `-O3 -ipo -parallel -AVX -xhost`. Apart from these parameters no optimizations were used for the programs.

Implementations. The three standard iterative sorting algorithms: I-BS, I-SS, and I-IS as shown in Figure A.31, were implemented without any optimization and the implementations were inherently serial.

The divide-and-conquer variants of the three algorithms i.e., R-BS, P-BS, R-SS, P-SS, R-IS, and M-IS were also implemented without optimizations. When the subproblem size ($h-\ell+1$ or $\ell h-\ell\ell+1$) became less than or equal to a base case size $b = 2^8 = 256$, we switched to an iterative kernel having an algorithm-dependent logic. The recursive algorithms were run with 32 threads. We define speedup as follows:

$$\text{Speedup} = \frac{\text{Runtime of iterative algorithm}}{\text{Runtime of recursive algorithm with 32 threads}} \qquad \text{(A.13)}$$

In our experiments, we used two types of input: (i) random input (using `rand` function), and (ii) descending order input. The input size (n) was varied from $2^8 = 256$ to $2^{19} = 524288$ and the speedup was found for different algorithms based on Equation A.13.

Results. Table A.35 shows the speedup graphs for the three sorting algorithms.

The speedup of the R-BS program increased from $0.7\times$ to $76\times$ for the random input when n increased. For the descending order input, the speedup increased from $0.6\times$ to $30.6\times$. The good speedup is majorly due to parallelism and to some extent by the cache performance.

The R-SS program speedup increased from $1\times$ to $23.9\times$ for the random input when n increased. When the input was in decreasing order, the speedup increased from $1\times$ to $19.8\times$. Compared to bubble sort, the speedup is less for selection sort. The reason is that R-SS does more number of comparisons than I-SS.

For the random input, the speedup of the R-IS program increased from $1.1\times$ to $280\times$. For the decreasing order input, the speedup increased from $1.1\times$ to $1314.7\times$. Note that such a large speedup is not possible from parallelism and cache performance alone. The factor that is increasing the speedup is the asymptotic less work that the R-IS performs compared to I-IS as shown in the complexity analysis subsection of A.13.3.
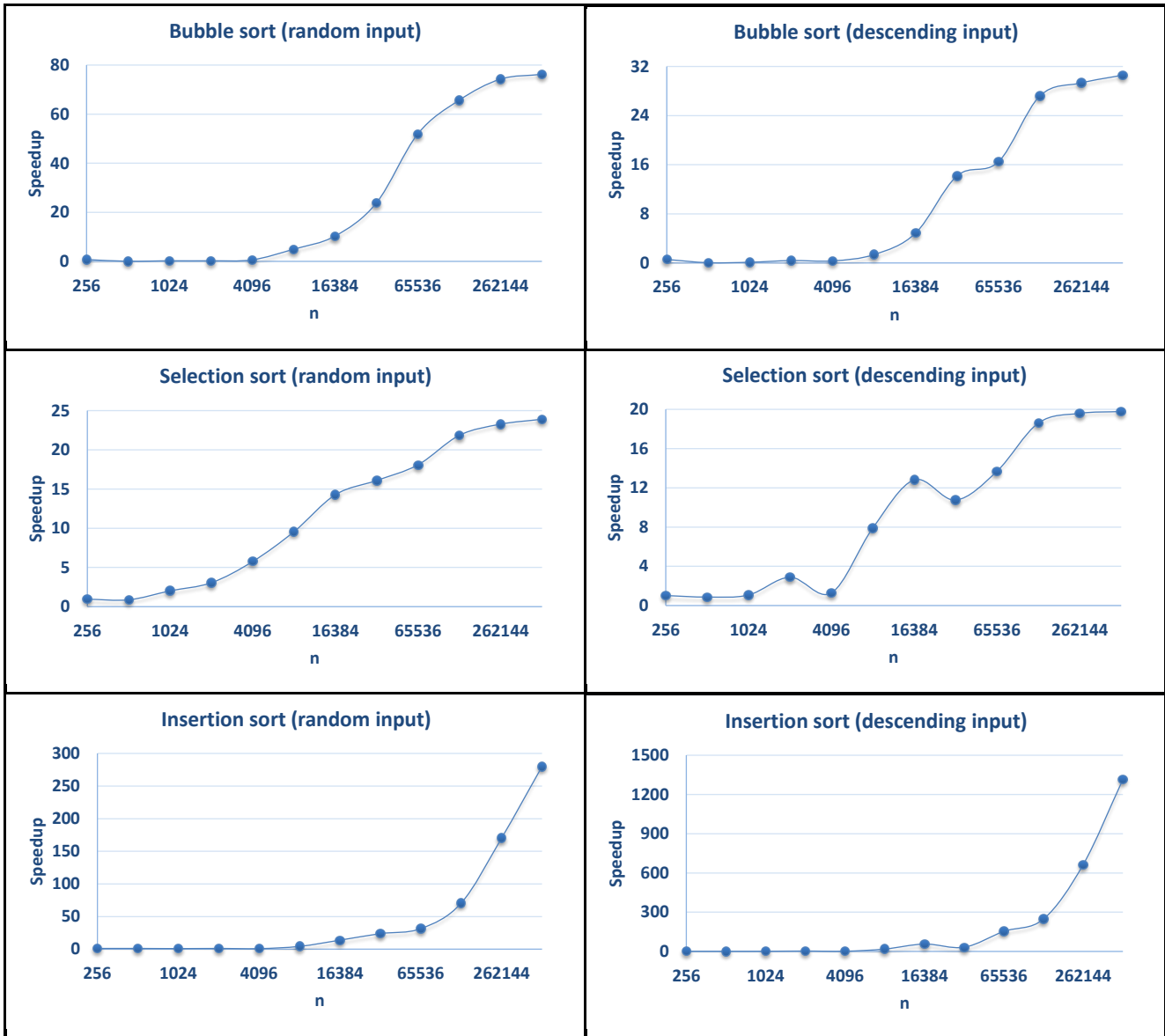
Figure A.35: Speedup of the divide-and-conquer bubble, selection, and insertion sorts for (i) random input (left column) and (ii) descending order input (right column). The definition of speedup is given in Equation A.13.

A.14 Conclusion and open problems

In this chapter we saw several divide-and-conquer DP algorithms. Each such algorithm is distinct and significant in its own way. For example:

* ✫ Parenthesis and egg dropping problems require combining nodes.
* ✫ Floyd-Warshall's APSP require handling one-way sweep property violation.
* ✫ Spoken-word recognition and CYK algorithms have non-orthogonal regions.
* ✫ Bitonic TSP has two types of dependencies in the DP table.
* ✫ Binomial coefficient has two types of divide-and-conquer algorithms for solving different regions of the DP table.
* ✫ Egg dropping problem can be solved using different divide-and-conquer algorithms.
* ✫ Irregular DP problems such as Sieve of Eratosthenes and Knapsack problems are
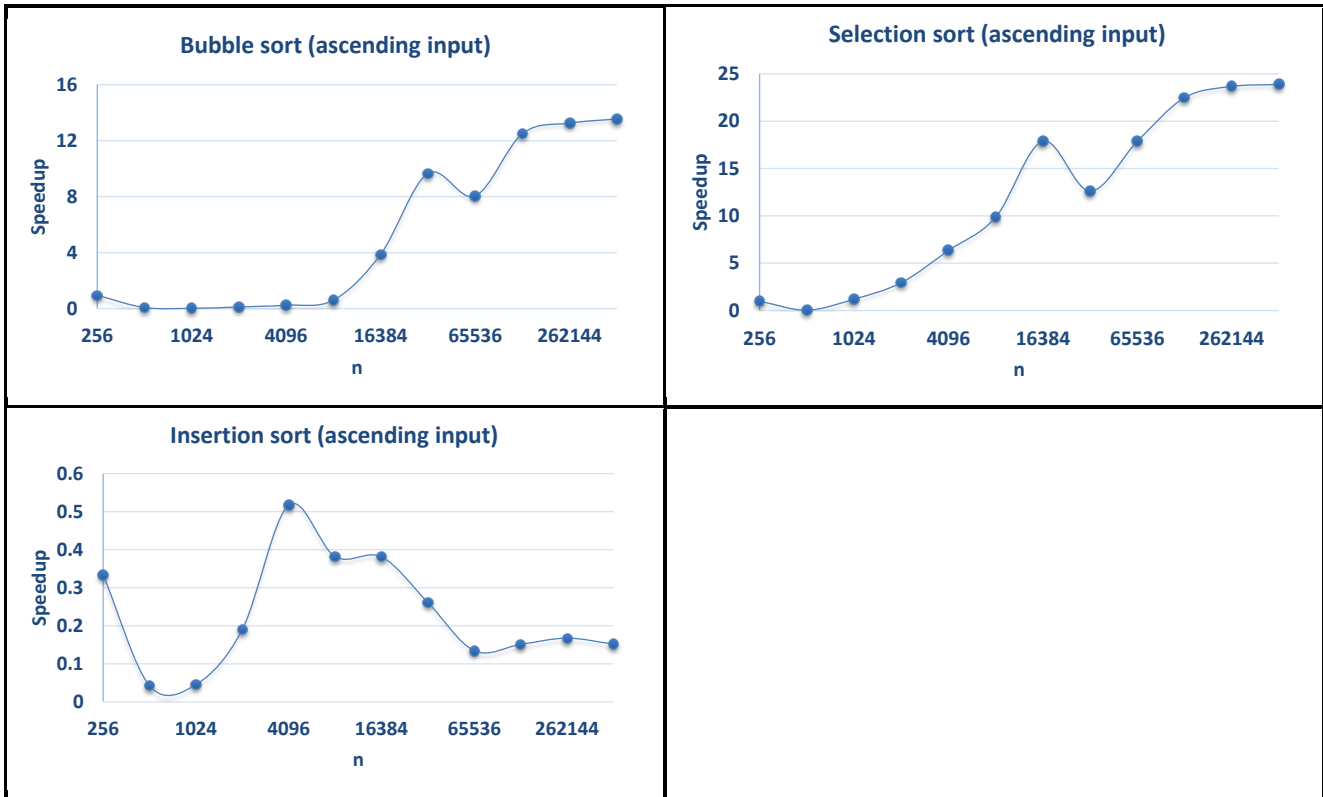
195

Figure A.36: Speedup of the divide-and-conquer bubble, selection, and insertion sorts for ascending order input. The definition of speedup is given in Equation A.13.

difficult to solve efficiently. A good way to solve such problems is by creating a data structure and collecting all to-be-accessed DP table cells into the data structure.

⋆ Elementary sorting algorithms are non-DP problems and we can solve them using the core idea of AUTOGEN.

Also, we showed that these auto-discovered divide-and-conquer algorithms have excellent and often optimal serial cache complexity and good span (for typically non-local dependencies in high dimensions).

Some open problems are as follows:

⋆ [*Closed-form formula for recurrences*.] Given any recurrence relation, is it possible to check if the recurrence relation can be denoted using a closed-form formula?

⋆ [*Tight lower bounds for solving recurrences*.] Given any recurrence relation, is it possible to develop tight lower bounds to compute the recurrence?

⋆ [*Irregular dynamic programming problems*.] Develop cache-efficient divide-and-conquer algorithms for irregular (e.g.: data-dependent or data-sensitive) DP problems such as knapsack problem.

⋆ [*Non-DP problems*.] Develop divide-and-conquer algorithms to many more non-DP problems.

Appendix B

Efficient Divide-&-Conquer Wavefront DP Algorithms

In this section, we present the recursive divide-and-conquer wavefront algorithms for matrix multiplication, LCS, Floyd-Warshall's APSP, and gap problem. The standard 2-way recursive divide-and-conquer algorithms are presented in Chapter A. Here we give only the timing functions and not the entire divide-and-conquer wavefront algorithm.

B.1 Matrix multiplication

The timing functions are computed as follows. The completion-time is found as:

$$\mathfrak{C}(i, j, k) = k$$

Similarly, the start- and end-time functions are as follows.

$$
\begin{aligned}
\mathcal{S}_\mathcal{A}\left(X, U, V\right) = \mathcal{E}_\mathcal{A}\left(X, U, V\right) = \mathfrak{C}(x_r, x_c, u_c) && \text{if } X \text{ is a } n' \times n' \text{ chunk,} \\
\mathcal{S}_\mathcal{A}\left(X, U, V\right) = \mathcal{S}_\mathcal{A}\left(X_{11}, U_{11}, V_{11}\right) && \text{if } X \text{ is not a } n' \times n' \text{ chunk,} \\
\mathcal{E}_\mathcal{A}\left(X, U, V\right) = \mathcal{E}_\mathcal{A}\left(X_{22}, U_{22}, V_{22}\right) && \text{if } X \text{ is not a } n' \times n' \text{ chunk.}
\end{aligned}
$$

Solving, we have

$$
\begin{aligned}
\mathfrak{C}(i, j, k) = k; && \mathcal{S}_\mathcal{A}\left(X, U, V\right) = \mathfrak{C}(x_r, x_c, u_c) \\
\mathcal{E}_\mathcal{A}\left(X, U, V\right) = \mathfrak{C}(x_r + n - n', x_c + n - n', u_c + n - n');
\end{aligned}
$$

where, (x_r, x_c) is the top-left corner of X. Figure B.1 gives a $\mathcal{WR}\text{-}\mathcal{DP}$ algorithm for matrix multiplication. Figure B.3 shows the timestamps of the $\mathcal{WR}\text{-}\mathcal{DP}$ on a 4×4 DP table.

B.2 Longest common subsequence & edit distance

The LCS problem is described in Section A.1. The timing functions are computed as follows.
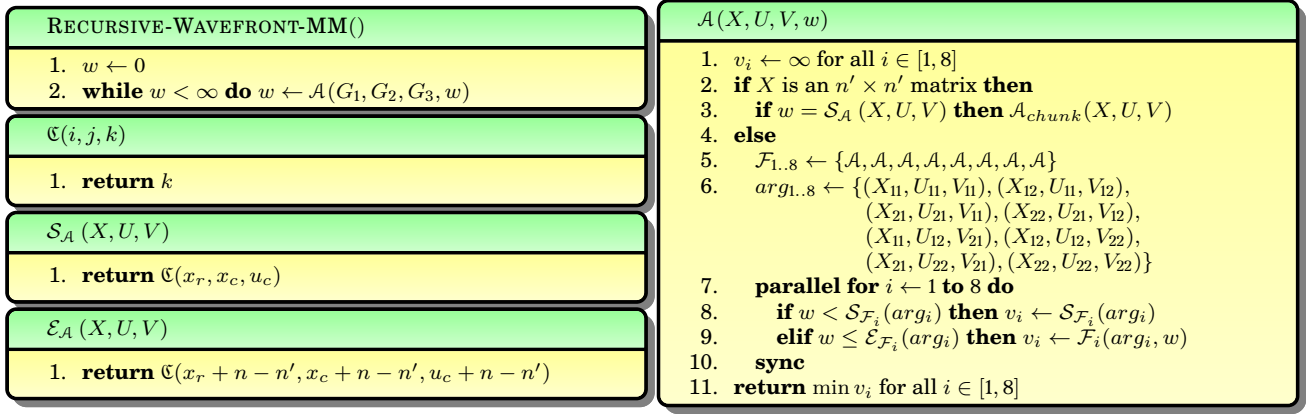
Figure B.1: A recursive divide-and-conquer wavefront DP algorithm for matrix multiplication.

Completion-time is found from the LCS DP recurrence:

$$\mathfrak{C}(i,j) = \begin{cases} 0 & \text{if } i < 0 \,||\, j < 0 \,||\, i = j = 0, \\ \max\left(\mathfrak{C}(i-1,j), \mathfrak{C}(i,j-1), \mathfrak{C}(i-1,j-1)\right) + 1 & \text{otherwise.} \end{cases}$$

From Definition 22, we have $smax(i,j) = \max(\mathfrak{C}(i-1,j), \mathfrak{C}(i,j-1), \mathfrak{C}(i-1,j-1))$ and $su(i,j) = 0$ as there is no tuple writing on (i,j) and reading from itself. Similarly, the start- and end-time functions are as follows.

$$\begin{aligned} \mathcal{S}_{\mathcal{A}}(X) = \mathcal{E}_{\mathcal{A}}(X) = (\mathfrak{C}(x_r, x_c)).0 && \text{if } X \text{ is a } n' \times n' \text{ chunk,} \\ \mathcal{S}_{\mathcal{A}}(X) = \mathcal{S}_{\mathcal{A}}(X_{11}) && \text{if } X \text{ is not a } n' \times n' \text{ chunk,} \\ \mathcal{E}_{\mathcal{A}}(X) = \mathcal{E}_{\mathcal{A}}(X_{22}) && \text{if } X \text{ is not a } n' \times n' \text{ chunk.} \end{aligned}$$

Solving, we have

$$\mathfrak{C}(i,j) = i + j; \quad \mathcal{S}_{\mathcal{A}}(X) = \mathfrak{C}(x_r, x_c); \quad \mathcal{E}_{\mathcal{A}}(X) = \mathfrak{C}(x_r + n - n', x_c + n - n');$$

where, (x_r, x_c) is the top-left corner of X. Figure B.2 gives a recursive wavefront algorithm for the LCS / edit distance problem. Figure B.3 shows the timestamps of the $\mathcal{WR}\text{-}\mathcal{DP}$ on a 4×4 DP table.
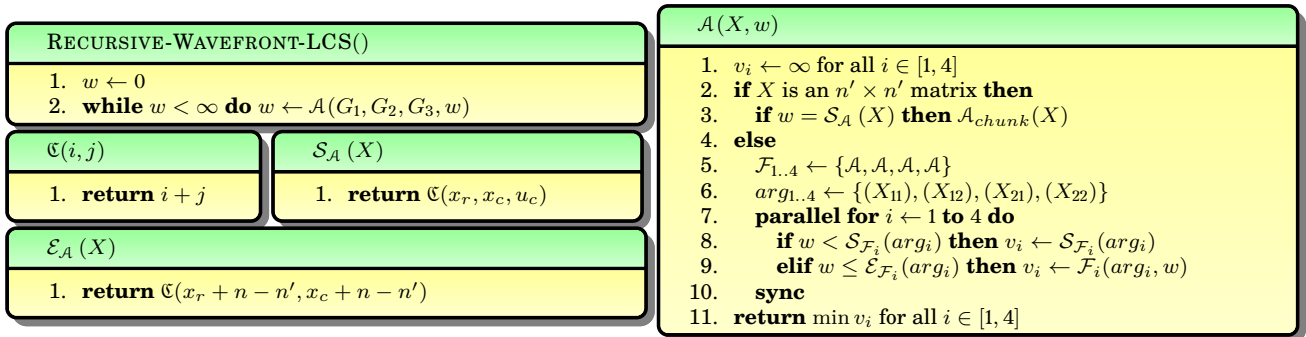

Figure B.2: A recursive divide-and-conquer wavefront DP algorithm for LCS / edit distance.

$A_0A_1A_2A_3$	$A_0A_1A_2A_3$	$A_0A_1A_2A_3$	$A_0A_1A_2A_3$
$A_0A_1A_2A_3$	$A_0A_1A_2A_3$	$A_0A_1A_2A_3$	$A_0A_1A_2A_3$
$A_0A_1A_2A_3$	$A_0A_1A_2A_3$	$A_0A_1A_2A_3$	$A_0A_1A_2A_3$
$A_0A_1A_2A_3$	$A_0A_1A_2A_3$	$A_0A_1A_2A_3$	$A_0A_1A_2A_3$

Matrix multiplication

A_0	A_1	A_2	A_3
A_1	A_2	A_3	A_4
A_2	A_3	A_4	A_5
A_3	A_4	A_5	A_6

LCS

Figure B.3: Timestamps of recursive wavefront algorithms for a 4×4 table. Left: matrix multiplication problem. Right: LCS problem.

B.3   Floyd-Warshall's all-pairs shortest path

A 2-way $\mathcal{R}\text{-}\mathcal{DP}$ algorithm is given in Section A.3.

Let (x_r, x_c, x_h) be the cell with the smallest coordinates in X. Then for each $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, we have

$$\mathfrak{C}(i, j, k) = 3k + [i \neq k] + [j \neq k],$$
$$\mathcal{S}_{\mathcal{F}}(X, \ldots) = \mathfrak{C}(x_r, x_c, x_h), \text{ and}$$
$$\mathcal{E}_{\mathcal{F}}(X, \ldots) = \max \left\{ \begin{array}{c} \mathfrak{C}(x_r, x_c, x_h + n - n'), \\ \mathfrak{C}(x_r, x_c + n - n', x_h + n - n'), \\ \mathfrak{C}(x_r + n - n', x_c, x_h + n - n'), \\ \mathfrak{C}(x_r + n - n', x_c + n - n', x_h + n - n') \end{array} \right\}.$$

where, $[\,]$ is the Iversion bracket.

Figure B.4 shows the timestamps for the four planes (k represents the plane number). The upper diagram shows the timestamps for a $4 \times 4 \times 4$ table and the lower diagram shows the timestamps for a 4×4 table. Figure B.5 shows a $\mathcal{WR}\text{-}\mathcal{DP}$ algorithm for the Floyd-Warshall's APSP.

B.4   Sequence alignment with gap penalty

The gap problem and the $\mathcal{R}\text{-}\mathcal{DP}$ algorithm to solve it is described in Section A.5.

Completion-time for the $\mathcal{WR}\text{-}\mathcal{DP}$ algorithm is as follows.

$$\mathfrak{C}(i, j) = \begin{cases} 0 & \text{if } i = -1 \,||\, j = -1 \,||\, i = j = 0, \\ \max\left(\mathfrak{C}(i-1, j), \mathfrak{C}(i, j-1)\right) + 2 & \text{otherwise.} \end{cases}$$

From Definition 22, we have $smax(i, j) = \max(\mathfrak{C}(i-1, j), \mathfrak{C}(i, j-1))$ and $su(i, j) = 1$ due to function $\mathcal{A}_{gap}$.

Figure B.4: Timestamps of recursive wavefront algorithm for FW problem for a 4×4 table.

Figure B.5: A recursive divide-and-conquer wavefront DP algorithm for Floyd-Warshall's APSP.

Start- and end-times are given by

$$
\mathcal{S}_{\mathcal{A}}(X, X) = \begin{cases} (\mathfrak{C}(x_r, x_c)).0 & \text{if } X \text{ is a } n' \times n' \text{ chunk,} \\ \mathcal{S}_{\mathcal{A}}(X_{11}, X_{11}) & \text{otherwise.} \end{cases}
$$

$$
\mathcal{S}_{\mathcal{B}}(X, U) = \begin{cases} (\mathfrak{C}(u_r, u_c) + 1).0 & \text{if } X \text{ is a } n' \times n' \text{ chunk,} \\ \mathcal{S}_{\mathcal{B}}(X_{11}, U_{11}) & \text{otherwise.} \end{cases}
$$

$$
\mathcal{S}_{\mathcal{C}}(X, V) = \begin{cases} (\mathfrak{C}(v_r, v_c) + 1).[x_c \geq n'] & \text{if } X \text{ is a } n' \times n' \text{ chunk,} \\ \mathcal{S}_{\mathcal{C}}(X_{11}, V_{11}) & \text{otherwise.} \end{cases}
$$

Function $\mathcal{B}$ does not have races and hence its $ra(X) = 0$. But when we add function $\mathcal{C}$, it will have race condition with $\mathcal{B}$. Therefore, to avoid clashes with $\mathcal{B}$, we use $ra(X) = [x_c \geq n']$. The way we compute $ra(X)$ is that $ra(X) = 1$ if $\mathfrak{C}(v_r, v_c) \leq \mathfrak{C}(x_r, x_c - n')$ and $x_c \geq n'$. We know that if the second condition is true the first condition is always true.

Hence $ra(X) = [x_c \geq n']$. We can write similar recurrence for the end-times. Solving, we have

$$
\begin{aligned}
&\mathfrak{C}(i, j) = 2(i + j), \\
&\mathcal{S}_{\mathcal{A}}(X, X) = \mathfrak{C}(x_r, x_c); \quad \mathcal{E}_{\mathcal{A}}(X, X) = \mathfrak{C}(x_r + n - n', x_c + n - n'), \\
&\mathcal{S}_{\mathcal{B}}(X, U) = \mathfrak{C}(u_r, u_c) + 1; \quad \mathcal{E}_{\mathcal{B}}(X, U) = \mathfrak{C}(u_r + n - n', u_c + n - n') + 1, \\
&\mathcal{S}_{\mathcal{C}}(X, V) = (\mathfrak{C}(v_r, v_c) + 1).[x_c \geq n']; \quad \mathcal{E}_{\mathcal{C}}(X, V) = (\mathfrak{C}(v_r + n - n', v_c + n - n') + 1).[x_c \geq n'];
\end{aligned}
$$

RECURSIVE-WAVEFRONT-GAP()

1. $w \leftarrow 0$
2. **while** $w < \infty$ **do** $w \leftarrow \mathcal{A}(G, G, w)$

$\mathcal{A}(X, X, w)$

1. $v_i \leftarrow \infty$ for all $i \in [1, 8]$
2. **if** X is an $n' \times n'$ matrix **then**
3.   **if** $w = \mathcal{S}_{\mathcal{A}}(X, X)$ **then** $\mathcal{A}_{chunk}(X, X)$
4. **else**
5. $\mathcal{F}_{1..8} \leftarrow \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{A}, \mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{A}\}$
6. $arg_{1..8} \leftarrow \{(X_{11}, X_{11}), (X_{12}, X_{11}), (X_{21}, X_{11}),$
            $(X_{12}, X_{12}), X_{21}, X_{21}), (X_{22}, X_{21}),$
            $(X_{22}, X_{12}), (X_{22}, X_{22})\}$
7.   **parallel for** $i \leftarrow 1$ **to** 8 **do**
8.   **if** $w < \mathcal{S}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{S}_{\mathcal{F}_i}(arg_i)$
9.   **elif** $w \leq \mathcal{E}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{F}_i(arg_i, w)$
10.   **sync**
11. **return** $\min v_i$ for all $i \in [1, 8]$

$\mathcal{B}(X, U, w)$

1. $v_i \leftarrow \infty$ for all $i \in [1, 8]$
2. **if** X is an $n' \times n'$ matrix **then**
3.   **if** $w = \mathcal{S}_{\mathcal{B}}(X, U)$ **then** $\mathcal{B}_{chunk}(X, U)$
4. **else**
5. $\mathcal{F}_{1..8} \leftarrow \{\mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}\}$
6. $arg_{1..8} \leftarrow \{(X_{11}, U_{11}), (X_{12}, U_{11}), (X_{21}, U_{21}),$
            $(X_{22}, U_{21}), X_{11}, U_{12}), (X_{12}, U_{12}),$
            $(X_{21}, U_{22}), (X_{22}, U_{22})\}$
7.   **parallel for** $i \leftarrow 1$ **to** 8 **do**
8.   **if** $w < \mathcal{S}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{S}_{\mathcal{F}_i}(arg_i)$
9.   **elif** $w \leq \mathcal{E}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{F}_i(arg_i, w)$
10.   **sync**
11. **return** $\min v_i$ for all $i \in [1, 8]$

$\mathfrak{C}(i, j)$

1. **return** $2(i + j)$

$\mathcal{C}(X, V, w)$

1. $v_i \leftarrow \infty$ for all $i \in [1, 8]$
2. **if** X is an $n' \times n'$ matrix **then**
3.   **if** $w = \mathcal{S}_{\mathcal{C}}(X, V)$ **then** $\mathcal{C}_{chunk}(X, V)$
4. **else**
5. $\mathcal{F}_{1..8} \leftarrow \{\mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C}\}$
6. $arg_{1..8} \leftarrow \{(X_{11}, V_{11}), (X_{12}, V_{12}), (X_{21}, V_{11}),$
            $(X_{22}, V_{12}), X_{11}, V_{21}),$
            $(X_{12}, V_{22}),$
            $(X_{21}, V_{21}), (X_{22}, V_{22})\}$
7.   **parallel for** $i \leftarrow 1$ **to** 8 **do**
8.   **if** $w < \mathcal{S}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{S}_{\mathcal{F}_i}(arg_i)$
9.   **elif** $w \leq \mathcal{E}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{F}_i(arg_i, w)$
10.   **sync**
11. **return** $\min v_i$ for all $i \in [1, 8]$

$\mathcal{S}_{\mathcal{A}}(X, X)$

1. **return** $\mathfrak{C}(x_r, x_c)$

$\mathcal{E}_{\mathcal{A}}(X, X)$

1. **return** $\mathfrak{C}(x_r + n - n', x_c + n - n')$

$\mathcal{S}_{\mathcal{B}}(X, U)$

1. **return** $\mathfrak{C}(u_r, u_c) + 1$

$\mathcal{E}_{\mathcal{B}}(X, U)$

1. **return** $\mathfrak{C}(u_r + n - n', u_c + n - n') + 1$

$\mathcal{S}_{\mathcal{C}}(X, V)$

1. **return** $(\mathfrak{C}(v_r, v_c) + 1).[x_c \geq n']$

$\mathcal{E}_{\mathcal{C}}(X, V)$

1. **return** $(\mathfrak{C}(v_r + n - n', v_c + n - n') + 1).[x_c \geq n']$

Figure B.6: A recursive divide-and-conquer wavefront DP algorithm for sequence alignment with gap penalty.

Figure B.7 gives both the integral and decimal timestamps for the gap problem for 4×4 DP table. Figure B.6 gives a $\mathcal{WR}$-$\mathcal{DP}$ algorithm for the gap problem.

A_0	$B_1 A_2$	$B_1 B_3 A_4$	$B_1 B_3 B_5 A_6$
$C_1 A_2$	$B_3 C_4 A_5$	$B_3 C_5 B_6 A_7$	$B_3 B_6 C_7 B_8 A_9$
$C_1 C_3 A_4$	$C_3 B_5 C_6 A_7$	$B_5 C_6 B_8 C_9 A_{10}$	$B_5 C_7 B_8 C_{10} B_{11} A_{12}$
$C_1 C_3 C_5 A_6$	$C_3 C_6 B_7 C_8 A_9$	$C_5 B_7 C_8 B_{10} C_{11} A_{12}$	$B_7 C_8 B_{10} C_{11} B_{13} C_{14} A_{15}$

A_0	$B_1 A_2$	$B_1 B_3 A_4$	$B_1 B_3 B_5 A_6$
$C_1 A_2$	$B_3 C_{3.1} A_4$	$B_3 B_5 C_{5.1} A_6$	$B_3 B_5 B_7 C_{7.1} A_8$
$C_1 C_3 A_4$	$C_{3.1} B_5 C_{5.1} A_6$	$B_5 C_{5.1} B_7 C_{7.1} A_8$	$B_5 B_7 C_{7.1} B_9 C_{9.1} A_{10}$
$C_1 C_3 C_5 A_6$	$C_{3.1} C_{5.1} B_7 C_{7.1} A_8$	$C_{5.1} B_7 C_{7.1} B_9 C_{9.1} A_{10}$	$B_7 C_{7.1} B_9 C_{9.1} B_{11} C_{11.1} A_{12}$

Figure B.7: Timestamps of recursive wavefront algorithm for gap problem for a 4×4 table. Top: integral timestamps. Bottom: decimal timestamps.

Appendix C

Efficient Divide-&-Conquer DP Algorithms for Irregular Problems

In this section, we present divide-and-conquer algorithms for irregular DP problems. Here, irregular means any kind of DP dependency that does not follow fractal property. For example, data-sensitive DP problems such as Viterbi algorithm and Knapsack problem are irregular, and DP problems which have weird DP dependencies such as Sieve of Eratosthenes are irregular.

C.1   Sieve of Eratosthenes

Let $P[i]$ represent whether the integer i ($i \geq 2$) is prime or not. Then $P[i]$ is computed using the sieve of Eratosthenes algorithm.

Figure C.1 shows the dependency structure for the sieve of Eratosthenes DP algorithm. The figure shows the serial and parallel iterative sieve of Eratosthenes algorithms. Note that the parallel algorithm has race conditions. A composite number is marked as composite by multiple threads which leads to race condition but does not hamper correctness of the algorithm.

Figure C.1 shows a cache-oblivious divide-and-conquer algorithm to solve the DP recurrence. The algorithm consists of three recursive functions: $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. The initial invocation to the function is $\mathcal{A}_{SE}(P[1 \ldots n])$. The $\mathcal{A}_{SE}(X[1 \ldots n])$ function has two steps. In the first step, it computes the first $\sqrt{n}$ elements in the prime table recursively using the $\mathcal{A}$ function. In the second step, using the first $\sqrt{n}$ numbers of the prime table the rest of the numbers in the prime table is computed.

In the $\mathcal{B}$ function for parameter n, we compute $(n - \sqrt{n})$ numbers in the prime table using $\sqrt{n}$ numbers. We do this in $((n - \sqrt{n})/\sqrt{n})$ steps and in each step we invoke function $\mathcal{C}$ to compute $\sqrt{n}$ numbers using $\sqrt{n}$ numbers. The function $\mathcal{C}$ calls itself recursively four times.

Complexity analysis for the PAR-LOOP-SE algorithm

Let $W_f(n)$, $Q_f(n)$, $T_f(n)$, and $S_f(n)$ denote the work, serial cache complexity, span, and space consumption of PAR-LOOP-SE with parameter n. Then $W_f(n) = \sum_{i=2}^{\lfloor \sqrt{n} \rfloor} \lfloor \frac{n}{i} \rfloor = \Theta\left(n \log \log n\right)$, $Q_f(n) = W_f(n) = \Theta\left(n \log \log n\right)$, $T_f(n) = \Theta\left(\log n + \log n\right) = \Theta\left(\log n\right)$, and $S_f(n) = \Theta\left(n\right)$.

Loop-SE(n)

1. $P[1] \leftarrow$ false
2. **for** $i \leftarrow 2$ **to** n **do** $P[i] \leftarrow$ true
3. **for** $i \leftarrow 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**
4.   **if** $P[i] =$ true **then**
5.   **for** $j \leftarrow 2$ **to** $\lfloor n/i \rfloor$ **do** $P[i \times j] \leftarrow$ false

Par-Loop-SE(n)

1. $P[1] \leftarrow$ false
2. **parallel for** $i \leftarrow 2$ **to** n **do** $P[i] \leftarrow$ true
3. **parallel for** $i \leftarrow 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**
4.   **if** $P[i] =$ true **then**
5.   **parallel for** $j \leftarrow 2$ **to** $\lfloor n/i \rfloor$ **do**
6. $P[i \times j] \leftarrow$ false

$\mathcal{A}_{SE}(X)$

1. **if** X is a small array **then** $\mathcal{A}_{loop-SE}(X)$
2. **else**
3. $\mathcal{A}_{SE}(\lfloor \sqrt{X} \rfloor)$
4. $\mathcal{B}_{SE}(X - \lfloor \sqrt{X} \rfloor, \lfloor \sqrt{X} \rfloor)$

$\mathcal{B}_{SE}(X[1..m], U[1..\ell])$

1. **for** $i \leftarrow 0$ **to** $\lceil m/\ell \rceil - 1$ **do**
2. $\mathcal{C}_{SE}(X[i \times \ell + 1.. \min((i+1)\ell, m)], U[1..\ell])$

$\mathcal{C}_{SE}(X, U)$

1. **if** X is small array **then** $\mathcal{C}_{loop-SE}(X, U)$
2. **else**
3.   **parallel:** $\mathcal{C}_{SE}(X_1, U_1), \mathcal{C}_{SE}(X_2, U_1)$
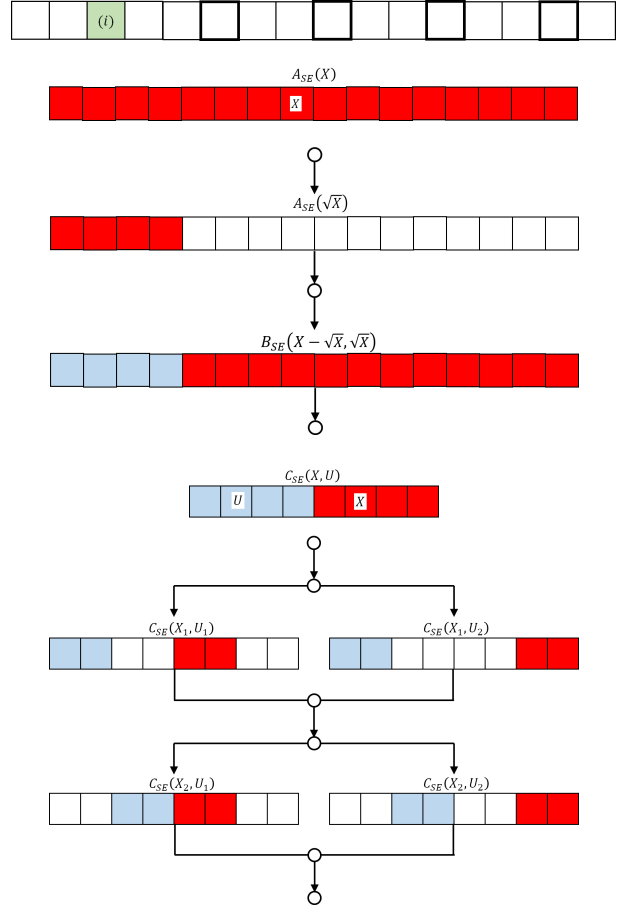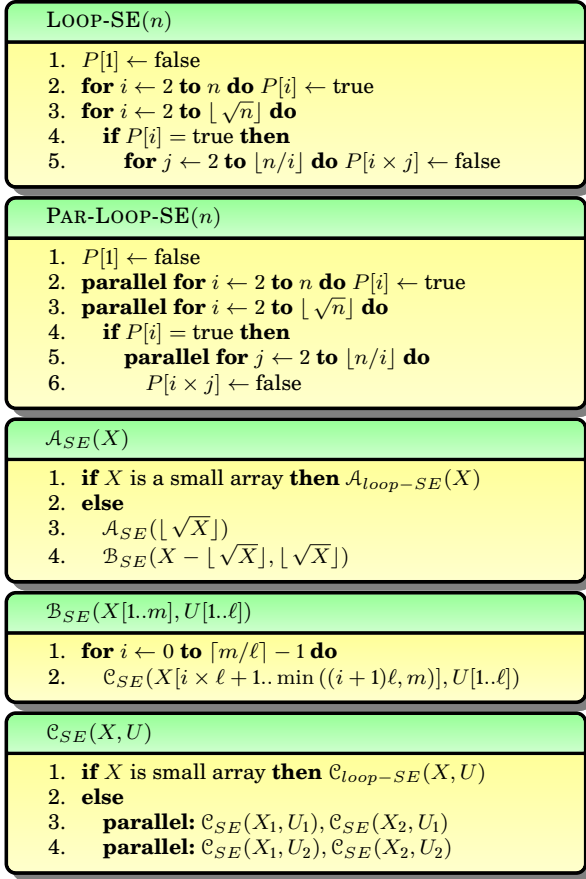4.   **parallel:** $\mathcal{C}_{SE}(X_1, U_2), \mathcal{C}_{SE}(X_2, U_2)$

Figure C.1: Top right: Dependency structure of Sieve of Eratosthenes DP: cell (i) affects the thick bordered cells. Rest: Parallel iterative and recursive divide-and-conquer algorithm for solving the sieve of Eratosthenes DP recurrence.

For the sieve of Eratosthenes DP recurrence, the PAR-LOOP-SE algorithm achieves $T_1(n) = \Theta(n \log \log n)$, $Q_1(n) = \Theta(n \log \log n)$, $T_\infty(n) = \Theta(\log n)$, **parallelism** $= \Theta\left(\frac{n \log \log n}{\log n}\right)$, and $S_\infty(n) = \Theta(n)$.

Complexity analysis for the cache-inefficient cache-oblivious $\mathcal{A}_{SE}$ algorithm

For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity and span of f_{SE} on an array of size n. For $\mathcal{B}$, function there are two parameters m and n. Then

$$
\begin{aligned}
Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) &= \mathcal{O}\left(\frac{n}{B} + 1\right) && \text{if } n \leq \gamma M, \\
Q_{\mathcal{A}}(n) = Q_{\mathcal{A}}(\sqrt{n}) + Q_{\mathcal{B}}(n - \sqrt{n}, \sqrt{n}) + \Theta(1) && \text{if } n > \gamma_A M; \\
Q_{\mathcal{B}}(m, n) = \frac{m}{n} Q_{\mathcal{C}}(n) + \Theta(1) && \text{if } n > \gamma_B M; \\
Q_{\mathcal{C}}(n) = 4 Q_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > \gamma_C M;
\end{aligned}
$$

$$
\begin{aligned}
T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) &= \mathcal{O}(1) && \text{if } n = 1, \\
T_{\mathcal{A}}(n) = T_{\mathcal{A}}(\sqrt{n}) + T_{\mathcal{B}}(n - \sqrt{n}, \sqrt{n}) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{B}}(m, n) = \frac{m}{n} T_{\mathcal{C}}(n) + \Theta(1) && \text{if } n > 1; \\
T_{\mathcal{C}}(n) = 2 T_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n > 1;
\end{aligned}
$$

where, γ, γ_A, γ_B, and γ_C are suitable constants. Solving, we get $W_{\mathcal{A}}(n) = \Theta(n \log \log n)$,

$Q_{\mathcal{A}}(n) = \mathcal{O}\left(\frac{n^{1.5}}{BM} + \frac{n^{1.5}}{M} + \frac{n}{B} + 1\right)$, $T_{\mathcal{A}}(n) = \Theta(n)$, and $S_{\mathcal{A}}(n) = \Theta(n)$.

> The cache-oblivious divide-and-conquer $\mathcal{A}_{SE}$ variant of the sieve of Eratosthenes algorithm achieves $T_1(n) = \Theta(n \log \log n)$, $Q_1(n) = \Theta\left(\frac{n^{1.5}}{BM} + \frac{n^{1.5}}{M} + \frac{n}{B} + 1\right)$, $T_\infty(n) = \Theta(n)$, parallelism $= \Theta(\log \log n)$, and $S_\infty(n) = \Theta(n)$.

The serial cache complexity of the algorithm is horrible. We can modify the algorithm to improve the cache complexity but the algorithm will become cache-aware. Here is the cache-efficient cache-aware algorithm. We remove the $\mathcal{C}$ function and modify the $\mathcal{B}$ function. To implement the function B we store $P[1 \ldots M]$ in memory. For simplicity we can assume that the cache consists of $(M/B+1)$ blocks. By simply scanning all blocks in the range $P[\sqrt{n} \ldots n]$ we set to false all indices that are multiples of the prime numbers in the range $1 \ldots M$. The scan requires $\mathcal{O}\left(\frac{n}{B}\right)$ cache misses. Setting to false all indices in the range $P[\sqrt{n} \ldots n]$ that are multiples of the primes in the range $1 \ldots M$ requires $\mathcal{O}(n \log \log_M n)$ cache misses.

Complexity analysis for the cache-efficient cache-aware $\mathcal{A}_{SE}$ algorithm

The terms used for the complexities are the same as explained in the previous paragraphs. Let $Q_{\mathcal{B}}(P[\ell+1 \ldots h], P[1 \ldots \ell])$ denote the serial cache complexity for the $\mathcal{B}$ function. We use h and ℓ as the parameters for the $\mathcal{B}$ function. Then we have

$$Q_{\mathcal{B}}(\ell, h) = \mathcal{O}\left(\max\left(1, \frac{h - M}{B}\right) + \sum_{p \in [M, \ell]} \frac{h - \ell}{p}\right).$$

$$T_{\mathcal{B}}(\ell, h) = \Theta\left(\sum_{p \in [1, \ell]} \frac{h - \ell}{i}\right) = \Theta\left((h - \ell) \log \log \ell\right).$$

where p is a prime. Solving, $Q_{\mathcal{A}}(n) = \Theta\left(\frac{n}{B} + n \log \log_M n\right)$ and $T_{\mathcal{A}}(n) = \Theta(n \log \log n)$.

> The cache-aware divide-and-conquer $\mathcal{A}_{SE}$ variant of the sieve of Eratosthenes algorithm achieves $T_1(n) = \Theta(n \log \log n)$, $Q_1(n) = \Theta\left(\frac{n}{B} + n \log \log_M n\right)$, $T_\infty(n) = \Theta(n \log \log n)$, parallelism $= \Theta(1)$, and $S_\infty(n) = \Theta(n)$.

Cache-efficient cache-oblivious sieve of Eratosthenes algorithm

In this section, we give algorithms to generate primes from the sieve of Eratosthenes (having $\Theta(n \log \log n)$ computations) using cache-efficient priority queues (PQ).

Due to the lack of locality of reference, the naive implementation of the sieve of Eratosthenes has a cache complexity of $\Theta(n \log \log n)$ and a work complexity of $\Theta(n \log \log n)$. A more sophisticated approach – creating lists of the multiples of each prime, and then sorting them together – improves the locality at the cost of additional computation, leading to a cache complexity of $\mathcal{O}\left(\frac{n \log \log n}{B} \log_{\frac{M}{B}} \frac{n \log \log n}{B}\right)$ and total work of $\mathcal{O}(n \log n \log \log n)$. We can sharpen this approach by using a (general) efficient data structure instead of the sorting step, and then further by introducing a data structure designed specifically for this problem.

The sieve of Eratosthenes can be implemented easily to get an amortized near-optimal sorting I/O-complexity using a *black-box priority queue*. The priority queue can be any of

the cache-aware structures such as M/B-way merger [Brodal and Katajainen, 1998a] and heap with buffers [Fadel et al., 1999], or any of the cache-oblivious structures such as up-down buffered priority queue [Arge et al., 2002b] and funnel heap [Brodal and Fagerberg, 2002], which all have the optimal amortized insert and deletemin cost. A priority queue data structure stores a set of primes and their multiples – one multiple per prime. Simply plugging in a theoretically good PQ (i.e., priority queues that have optimal amortized I/O complexity for both INSERTs and DELETEMINs), the cache complexity of implementing the sieve of Eratosthenes will be $\mathcal{O}\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\log\log n\right)$.

The sieve of Eratosthenes problem has a special property that can be exploited to design better priority queues that can achieve optimal sorting I/O bounds. The important observation is that *there are more multiples of smaller primes than that of larger ones*. This means that the accesses to smaller primes are more frequent than larger ones. Using an idea similar to Huffman coding (where, more common symbols are encoded with fewer bits), we like to have separate priority queues for different primes such that the amortized cost for a more frequently accessed smaller prime is relatively lesser than that of the rarely accessed larger prime. Then, the total cost of access to many multiples of smaller primes match the total cost of access to fewer multiples of larger primes. Thus, the sieve of Eratosthenes can be implemented using the standard priority queues having a complexity $\mathcal{O}\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\log\log n\right)$ to achieve the optimal lower bound for sorting n numbers.

Using a standard priority queue

The priority queue Q used to sieve primes, as shown in Figure C.2, consists of $\langle k, v\rangle$ pairs, where v is a prime and k is a multiple of v. That is, the prime multiples are the values and the prime factors are the keys. It is important to note that the priority queue operations such as INSERTs and DELETEMINs are based on the keys. E.g.: The minimum of the priority queue is a pair with the minimum key (or prime multiple).

We start off by inserting the first pair $\langle 4, 2\rangle$ into Q, and at each step, we extract (and delete) the minimum composite $\langle k, v\rangle$ pair in Q. Any number less than k which has never been inserted into Q must be prime. We keep track of the last deleted composite k', and check if $k > k' + 1$. If so, we declare $p = k' + 1$ as prime, and insert $\langle p^2, p\rangle$ into Q. In each of these iterations, we always insert the next multiple $\langle k + v, v\rangle$ into Q.

The black-box priority can be any cache- and/or work-efficient priority queue. A short list of priority queues in the literature in the RAM, cache-aware, and cache-oblivious models are shown in Table C.1. We can use any of the cache-aware PQs such as M/B-way

```
SE-USING-PQ(n)

Input: n
Output: Primes ≤ n
1.  p ← 2; prevk ← 2; print p
2.  Q.INSERT(⟨4, 2⟩)
3.  while k ≤ n do
4.      ⟨k, v⟩ ← Q.DELETEMIN()
5.      if k = prevk + 2 then
6.          p ← k − 1; print p
7.          Q.INSERT(⟨p², p⟩)
8.      prevk ← k
9.      Q.INSERT(⟨k + v, v⟩)
```
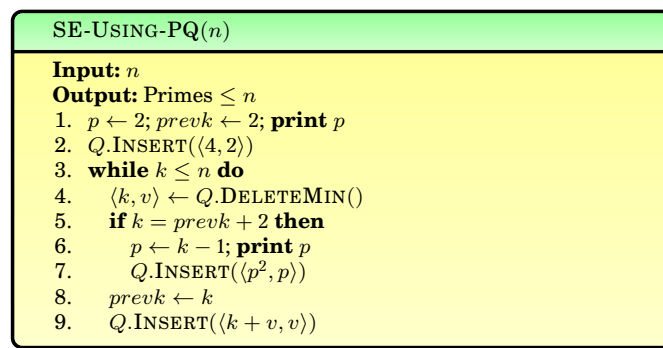
Figure C.2: Sieve of Eratosthenes using a priority queue Q.

merger PQ [Brodal and Katajainen, 1998a] and heap with buffers [Fadel et al., 1999] but then the algorithm becomes cache-aware. As we are interested in cache-oblivious algorithms, we use buffered PQ [Arge et al., 2002b] and funnel heap [Brodal and Fagerberg, 2002] in Table C.1.

Data structure	INSERT	DELETEMIN	Reference
RAM (or internal-memory) model			
Unsorted linked list	$\mathcal{O}(1)$	$\mathcal{O}(n)$	
Sorted linked list	$\mathcal{O}(n)$	$\mathcal{O}(1)$	
Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	[Williams, 1964]
Balanced search tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	[Knuth, 1998]
Binomial heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	[Vuillemin, 1978]
Pairing heap	$\mathcal{O}(1)$	$\mathcal{O}(\log n)^*$	[Fredman et al., 1986]
Fibonacci heap	$\mathcal{O}(1)$	$\mathcal{O}(\log n)^*$	[Fredman and Tarjan, 1987]
Relaxed heap	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	[Driscoll et al., 1988]
Meldable priority queue	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	[Brodal, 1995]
Brodal queue	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	[Brodal, 1996]
Cache-aware (or I/O or external-memory or cache-conscious or DAM or cache-sensitive) model			
Modified heap	$\mathcal{O}(\log_B n)$	$\mathcal{O}(\log_B n)$	[Williams, 1964]
B-tree	$\mathcal{O}(\log_B n)$	$\mathcal{O}(\log_B n)$	[Bayer and McCreight, 1972]
Tournament tree	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{B}\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{B}\right)^*$	[Kumar and Schwabe, 1996]
M/B-way merger	$\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{n}{B}\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{n}{B}\right)^*$	[Brodal and Katajainen, 1998b]
Heap with buffers	$\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{n}{B}\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{n}{B}\right)^*$	[Fadel et al., 1999]
Buffer tree	$\mathcal{O}\left(\frac{1}{B}\log_M n\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log_M n\right)^*$	[Arge, 2003]
Buffer heap	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{M}\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{M}\right)^*$	[Chowdhury and Ramachandran, 2004]
Cache-oblivious model			
B-tree	$\mathcal{O}(\log_B n)^*$	$\mathcal{O}(\log_B n)^*$	[Bayer and McCreight, 1972]
Buffered priority queue	$\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{n}{B}\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{n}{B}\right)^*$	[Arge et al., 2007]
Funnel heap	$\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{n}{B}\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{n}{B}\right)^*$	[Brodal and Fagerberg, 2002]
Buffer heap	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{B}\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{B}\right)^*$	[Chowdhury and Ramachandran, 2004]
Bucket heap	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{B}\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{B}\right)^*$	[Brodal et al., 2004]
Quick heap	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{M}\right)^*$	$\mathcal{O}\left(\frac{1}{B}\log\frac{n}{M}\right)^*$	[Navarro and Paredes, 2010]

Table C.1: Priority queues in different models. Here, * represents amortized values.

Complexity analysis of the SE-USING-PQ algorithm

By the prime number theorem [Hardy and Wright, 1979], the number of primes in the range $[1,\ldots,n]$ is $\mathcal{O}\left(\frac{n}{\log n}\right)$.

The total number of INSERTs and DELETEMINs in the standard sieve of Eratosthenes is $O(n\log\log n)$. The amortized I/O cost for one insert or deletemin to a PQ that can have a total of $\frac{\sqrt{n}}{\log\sqrt{n}}$ primes using any of the PQs mentioned above is $\frac{1}{B}\log_{\frac{M}{B}}\frac{n}{B}$. Hence, the total I/O complexity is $\mathcal{O}\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\log\log n\right)$.

> The sieve of Eratosthenes implemented using a cache-efficient cache-oblivious priority queue such as up-down buffered PQ [Arge et al., 2002b] or funnel heap [Brodal and Fagerberg, 2002] achieves $T_1(n) = \Theta(n \log \log n \log n)$, $Q_1(n) = \mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B} \log \log n\right)$, $T_\infty(n) = \Theta(n \log \log n \log n)$, and $S_\infty(n) = \Theta\left(\frac{\sqrt{n}}{\log n}\right)$.

Using a value-sensitive priority queue.

In the above algorithm, the key-value pairs corresponding to smaller values are accessed more frequently because smaller primes have more multiples in a given range. Therefore, a structure that prioritizes the efficiency of operations on smaller primes (values) outperforms a generic priority queue. We introduce a *value-sensitive priority queue*, in which the amortized access cost of an operation with value v depends on v instead of the size of the data structure.
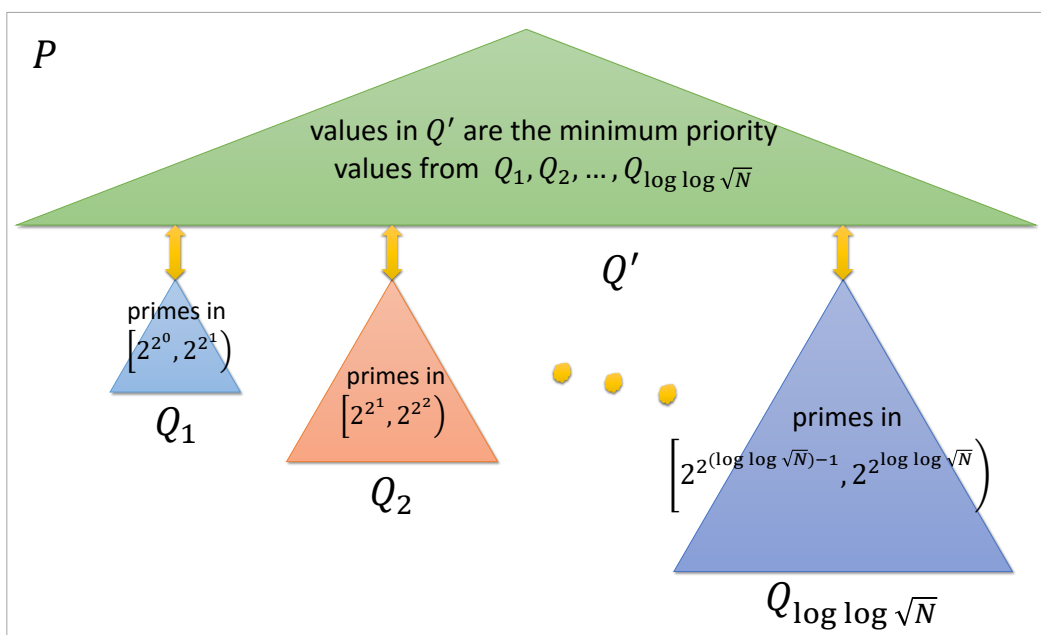

Figure C.3: Value-sensitive priority queue

A value-sensitive priority queue Q (see Figure C.3) has two parts – the *top part* consisting of a single internal-memory priority queue Q' and the *bottom part* consisting of $\lceil \log \log n \rceil$ external-memory priority queues $Q_1, Q_2, \ldots, Q_{\lceil \log \log n \rceil}$.

Each Q_i in the bottom-part of Q is a cache-efficient priority queue that stores $\langle k, v \rangle$ pairs, for $v \in [2^{2^i}, 2^{2^{i+1}})$. Hence, each Q_i contains fewer than $n_i = 2^{2^{i+1}}$ items. With a cache of size M, Q_i supports INSERT and DELETEMIN operations in $\mathcal{O}\left((\log_{M/B} n_i)/B\right)$ cache misses (amortized). Moreover, in each Q_i we have $\log v = \Theta(\log n_i)$. Thus, the cost reduces to $\mathcal{O}\left((\log_{M/B} v)/B\right)$ cache misses for an item with value v. Though we divide the cache equally among all Q_i's, the asymptotic cost per operation remains unchanged assuming $M > \Omega\left(B(\log \log n)^{1+\epsilon}\right)$ for some constant $\epsilon > 0$.

The queue Q' in the top part only contains the minimum composite (key) item from each Q_i, and so the size of Q' will be $\Theta(\log \log n)$. We use the dynamic integer set data structure [Patrascu and Thorup, 2014] to implement Q' which supports INSERT and DELETEMIN

operations on Q' in $\mathcal{O}(1)$ time using only $\mathcal{O}(\log n)$ space. We also maintain an array $A[1 : \lceil \log \log n \rceil]$ such that $A[i]$ stores Q_i's contributed item to Q'; thus we can access it in constant time.

To perform a DELETEMIN, we extract the minimum key item from Q', check its value to find the Q_i it came from, extract the minimum key item from that Q_i and insert it into Q'. To INSERT an item, we first check its value to determine the destination Q_i, compare it with the item in $A[i]$, and depending on the result of the comparison we either insert the new item directly into Q_i or move Q_i's current item in Q' to Q_i and insert the new item into Q'. The following lemma summarizes the performance of these operations.

Complexity analysis of the SE-USING-PQ algorithm

> Using a value-sensitive priority queue Q as defined above, an item INSERT with value v takes $\mathcal{O}\left(\frac{1}{B} \log_{M/B} v\right)$ cache misses, and a DELETEMIN that returns an item with value v takes $\mathcal{O}\left(\frac{1}{B} \log_{M/B} v\right)$ cache misses, assuming $M = \Omega\left(B^{1+\epsilon}\right)$ for some constant $\epsilon > 0$.

We now use this value-sensitive priority queue to efficiently implement the sieve of Eratosthenes. Each prime p is involved in $\Theta(n/p)$ priority queue operations, and by the prime number theorem [Hardy and Wright, 1979], there are $\mathcal{O}\left(\frac{\sqrt{n}}{\log n}\right)$ prime numbers in $[1, \sqrt{n}]$, and the ith prime number is approximately $i \ln i$.

We use the symbol p to denote a prime number. We use the terms $sort_E(n)$ (resp. $sort_I(n)$) to denote the per-element I/O cost (resp. internal memory cost) required to sort a set of n numbers. That is $sort_E(n) = \Theta\left(\frac{1}{B} \log_{M/B} \frac{n}{B}\right)$ and from [Han, 2002] $sort_I(n) = \log \log n$. The serial cache complexity of the algorithm is computed as

$$Q_1(n) = \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot \text{I/Os for one INSERT or DELETEMIN in } Q$$

$$= \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot (\text{I/Os for } Q_i + \text{I/Os for } Q') = \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot \text{I/Os for } Q_i = \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot sort_E(p)$$

$$= \mathcal{O}\left(\frac{n}{B \log \frac{M}{B \log \log n}} \sum_{p \leq \sqrt{n}} \frac{\log p}{p}\right) = \mathcal{O}\left(\frac{n}{B \log \frac{M}{B}} \cdot \ln n\right) = \mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} n\right)$$

The total work of the algorithm is computed as

$$T_1(n) = \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot \text{computations for one INSERT or DELETEMIN}$$

$$= \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot (\text{comp. for } Q_i + \text{comp. for } Q') = \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot \text{comp. for } Q_i + \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot \text{comp. for } Q'$$

$$= \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot \mathcal{O}(\log n) + \sum_{p \leq \sqrt{n}} \frac{n}{p} \cdot sort_I(\log \log n) = \mathcal{O}(n \log n \log \log n)$$

The space complexity of the algorithm is $\mathcal{O}\left(\frac{\sqrt{n}}{\log n}\right)$ as we store only the prime numbers not greater than $\lceil \sqrt{n} \rceil$.

> The sieve of Eratosthenes implemented using a cache-efficient cache-oblivious value-sensitive priority queue where the Q_is are cache-efficient cache-oblivious priority queues such as up-down buffered PQ [Arge et al., 2002b] or funnel heap [Brodal and Fagerberg, 2002], where $M = \Omega\left(\log n + B^{1+\epsilon}\right)$ for some constant $\epsilon > 0$, achieves $T_1(n) = \Theta\left(n \log \log n \log n\right)$, $Q_1(n) = \mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} n\right)$, $T_\infty(n) = \Theta\left(n \log \log n \log n\right)$, and $S_\infty(n) = \Theta\left(\frac{\sqrt{n}}{\log n}\right)$.

Though we reduced the cache complexity the total work increased. We can decrease both the cache complexity and the total work using the priority queue of Arge and Thorup [Arge and Thorup, 2013], which is simultaneously efficient in RAM and in external memory. However, the data structure is cache-aware and is no longer cache-oblivious. In particular, their priority queue can handle INSERTs with $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{n}{B}\right)$ amortized I/Os and $\mathcal{O}\left(\log_{M/B} \frac{n}{B}\right)$ amortized RAM operations. DELETEMIN requires $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{n}{B}\right)$ amortized I/Os and $\mathcal{O}\left(\log_{M/B} \frac{n}{B} + \log \log M\right)$ amortized RAM operations. They assume that each element fits in a machine word and use integer sorting techniques to achieve this low RAM cost while retaining optimal I/O complexity. The total work is computed as

$$
T_1(n) = \sum_{p \le \sqrt{n}} \frac{n}{p} \cdot \text{computations for one INSERT or DELETEMIN}
$$

$$
= \sum_{p \le \sqrt{n}} \frac{n}{p} \cdot (\text{comp. for } Q_i + \text{comp. for } Q') = \sum_{p \le \sqrt{n}} \frac{n}{p} \cdot \text{comp. for } Q_i + \sum_{p \le \sqrt{n}} \frac{n}{p} \cdot \text{comp. for } Q'
$$

$$
= \sum_{p \le \sqrt{n}} \frac{n}{p} \cdot \left(sort_E(p) + sort_I\left(\frac{M}{\log \log n}\right) \right) + \sum_{p \le \sqrt{n}} \frac{n}{p} \cdot sort_I(\log \log n)
$$

$$
= \mathcal{O}\left(\frac{n}{B \log \frac{M}{B \log \log n}} \sum_{p \le \sqrt{n}} \frac{\log p}{p} + n \cdot \log \log M \sum_{p \le \sqrt{n}} \frac{1}{p} + n \cdot \log \log \log \log n \sum_{p \le \sqrt{n}} \frac{1}{p} \right)
$$

$$
= \mathcal{O}\left(\frac{n}{B \log \frac{M}{B}} \cdot \ln n + n \cdot (\log \log \log \log n + \log \log M) \cdot \log \log n \right)
$$

$$
= \mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B} + n \log \log n \cdot \log \log M \right)
$$

> The sieve of Eratosthenes implemented using a cache-efficient cache-aware value-sensitive priority queue where the Q_is are cache-efficient cache-aware priority queues of Arge and Thorup [Arge and Thorup, 2013], where $M = \Omega\left(\log n + B(\log \log n)^{1+\epsilon}\right)$ for some constant $\epsilon > 0$, achieves $T_1(n) = \mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B} + n \log \log n \cdot \log \log M\right)$, $Q_1(n) = \mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} n\right)$, $T_\infty(n) = \mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B} + n \log \log n \cdot \log \log M\right)$, and $S_\infty(n) = \Theta\left(\frac{\sqrt{n}}{\log n}\right)$.

C.2 Knapsack problem

We are given a knapsack of capacity W, a set of n unique items, where item i has a value v_i and weight w_i and the item can be used x_i number of times. Then, the knapsack problem

is defined as

$$\text{maximize } \sum_{i=1}^{n} v_i x_i \text{ such that } \sum_{i=1}^{n} w_i x_i \leq W$$

There are several variations of the problem:
* [*Fractional knapsack*.] When $x_i \in \mathbb{R}^+$ and $x_i \in [0,1]$ and, the problem is called fractional knapsack.
* [*0-1 knapsack*.] When $x_i \in \{0,1\}$, the problem is called 0-1 knapsack.
* [*Bounded knapsack*.] When $x_i \in [0, k_i]$, the problem is called bounded / limited knapsack.
* [*Unbounded knapsack*.] When $x_i \in [0, \infty)$, the problem is called unbounded / unlimited knapsack.

The knapsack problem is a problem in the domain of combinatorial optimization. The book [Martello and Toth, 1990] contains several different algorithms and computer implementations for the knapsack problems. The fractional knapsack can be solved in polynomial time. However, 0-1 knapsack, bounded knapsack, and unbounded knapsack are examples of integer programming problems. The decision versions of the problems are NP-complete and the optimization versions of the problems are NP-hard. No known polynomial time solutions exist for these problems. But, there are pseudo polynomial time algorithms for solving integer knapsack problems, using our magical wand called dynamic programming. In this section, we focus on 0-1 knapsack problem.

A simple dynamic programming recurrence for the 0-1 knapsack problem is as follows. Let $K[i,j]$ denote the maximum value of a subset of items $\{1, 2, \ldots, i\}$ whose total weight is less than or equal to j. Then,

$$K[i,j] = \begin{cases} 0 & \text{if } j = 0, \\ K[i-1, j] & \text{if } w_i > j, \\ \text{MAX}(K[i-1, j], K[i-1, j-w_i] + v_i) & \text{otherwise.} \end{cases}$$

The dependency structure for the problem is given in Figure C.4.
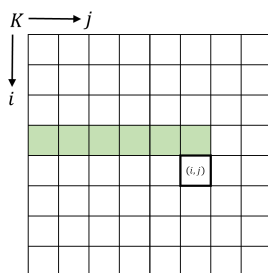
Figure C.4: The dependency structure for the knapsack problem. The (i,j) call depends on one of the cells in green color.

Several algorithms have been proposed to solve the knapsack problem. To the best of our knowledge, there is no cache-efficient (exploiting temporal locality)

Standard looping algorithm. Algorithm KNAPSACK-01 in Figure C.5 shows a standard looping algorithm to solve the knapsack problem. The algorithm fills a DP table of size nW (space can be reduced to $\Theta(W)$). The total work is $\Theta(nW)$, serial cache complexity is $\Theta(nW/B)$, **span** is $\Theta(n)$, and **parallelism** is $\Theta(W)$.

KNAPSACK-01(n W, $w[1\ldots n]$, $v[1\ldots n]$)
1. **for** $j \leftarrow 0$ **to** W **do**
2. $K[0,j] \leftarrow 0$
3. **for** $i \leftarrow 1$ **to** n **do**
4.   **for** $j \leftarrow 0$ **to** W **do**
5. $K[i,j] \leftarrow K[i-1,j]$
6.   **if** $w[i] \leq j$ **and**
$K[i-1,j-w[i]]+v[i] > K[i-1,j]$ **then**
7. $K[i,j] \leftarrow K[i-1,j-w[i]]+v[i]$

KNAPSACK-01-QUEUES(n, W, $w[1\ldots n]$, $v[1\ldots n]$)
1. SORT($w[1], w[2], \ldots, w[n]$)
2. Q_{prev}.INSERT($0, 0$)
3. **for** $i \leftarrow 1$ **to** n **do**
4.   **for** increasing $index$ in Q_{prev} **do**
5. $\langle key, val \rangle \leftarrow Q_{prev}$.GET($index$)
6.   **if** $val + v[i] > W$
7.   **break**
8. Q_{cur}.INSERT($key + w[i], val + v[i]$)
9. $Q_{merge} \leftarrow$ MERGE(Q_{prev}, Q_{cur}) w.r.t. key
10.   SWAP(Q_{prev}, Q_{merge})
11. **return** Q_{prev}

KNAPSACK-01-BBST(n, W, $w[1\ldots n]$, $v[1\ldots n]$)
1. T_{prev}.INSERT($0, 0$)
2. **for** $i \leftarrow 1$ **to** n **do**
3.   **while** !T_{prev}.EMPTY() **do**
{ val is the max value, key is the weight }
4. $\langle key, val \rangle \leftarrow T_{prev}$.DELETEMIN()
{ Insert key into T_{cur} }
5. $\langle key, val' \rangle \leftarrow T_{cur}$.SEARCH($key$)
6.   **if** $\langle key, val' \rangle$ exists **then**
7. T_{cur}.DELETE(key)
8. T_{cur}.INSERT(key, MAX(val, val'))
{ Insert k into T_{cur} }
9. $k \leftarrow key + w[i]$
10.   **if** $k \leq W$ **then**
11. $\langle k, v \rangle \leftarrow T_{cur}$.SEARCH($k$)
12.   **if** $\langle k, v \rangle$ does not exist **then**
13. T_{cur}.INSERT($k, val + v[i]$)
14.   **else**
15. T_{cur}.DELETE(k)
16. T_{cur}.INSERT(k, MAX($v, val + v[i]$))
17.   SWAP(T_{prev}, T_{cur})
18. **return** T_{prev}

Figure C.5: Three algorithms to solve the knapsack problem: one using standard loops, another using queues, and third using balanced binary search trees.

Algorithm using queues. Algorithm KNAPSACK-01-QUEUES in Figure C.5 shows an algorithm to solve the knapsack problem. There are n iterations in the algorithm and in each iteration we find the maximized values possible for different weights not greater than W. Before the ith iteration, Q_{prev} contains all $\langle key, val \rangle$ pairs, where, val is the maximum value possible for weight key from a subset of items from $\{x_1, x_2, \ldots, x_{i-1}\}$. Each element in Q_{prev} assumes a subset that does not contain the element x_i. Hence, we create another queue Q_{cur} from Q_{prev} that considers subsets considering the element x_i. We merge both Q_{prev} and Q_{cur} into Q_{merge}. This Q_{merge} queue contains all $\langle key, val \rangle$ pairs, where, val is the maximum value possible for weight key from a subset of items from $\{x_1, x_2, \ldots, x_{i-1}, x_i\}$.

There are a total of 2^n subsets of $\{x_1, \ldots, x_n\}$. For each subset, take the sum of all elements inside the it. Let the number of subsets whose subset-sums are less than or equal to W be M. We know that $M \in [0, W]$. This algorithm has better efficiency when M is relatively much lesser than W. The work done by the algorithm is $\mathcal{O}(nM)$ or $\mathcal{O}(nW)$. The serial cache complexity is $\mathcal{O}(nM/B)$ or $\mathcal{O}(nW/B)$.

Algorithm using balanced binary search tree. The balanced binary search trees (BBST) T_{prev} and T_{cur}, consists of $\langle k, v \rangle$ pairs, where v represents maximum value to for k weight of items. It is important to note that the tree operations such as INSERT, DELETE, DELETEMIN and SEARCH are based on the keys. The working of this algorithm is similar to that using queues. Using BBST can be useful when the number of subset-sums are relatively much smaller than W.

Let the number of subset-sums of elements in $\{x_1, \ldots, x_n\}$ not greater than W be M. The total work done by the algorithm is $\mathcal{O}(nM \log M)$ and the serial cache complexity is $\mathcal{O}\left((nM/B) \log_{M/B}(M/B)\right)$, assuming we use BBSTs that have I/O optimality in all its basic operations. Note that the complexity of this algorithm is worse than that using queues and the algorithm is more complicated.

Tiled iterative algorithm

We assume that the weights are sorted. The entire DP table (of size nW) is divided into $(n/n') \times (W/n')$ chunks each of size $(n' \times n')$, as shown in Figure C.6. We define two functions: $\mathcal{A}(X, X)$ that both reads and writes to the same chunk X, and $\mathcal{B}(X, U)$ that reads from chunk U and writes to a different region X, where $X \neq U$. The order of function calls (timestamps) for a grid (of chunks) of size 4×6 is shown in Figure C.7. The subscripts for the functions represent the timesteps at which a function can update a particular region.
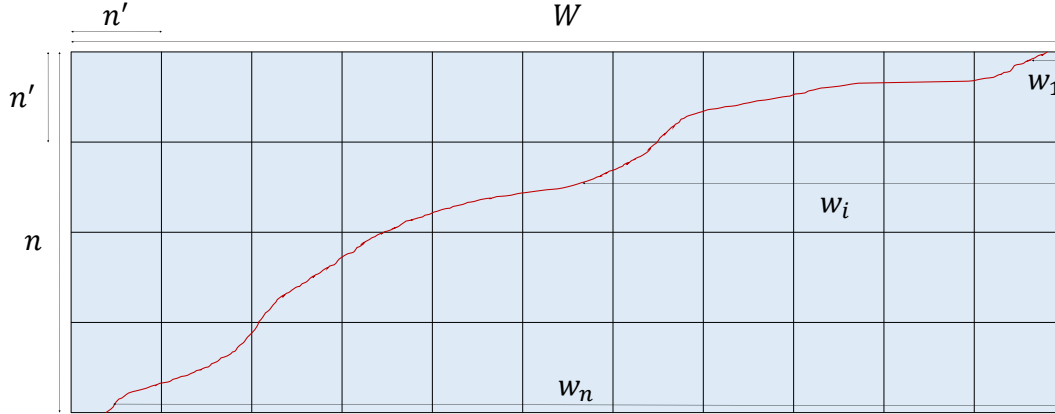
Figure C.6: Division of the DP table into chunks.

TILED-KNAPSACK()

1. **for** $i \leftarrow 1$ **to** n/n' **do**
2.   **for** $j \leftarrow 1$ **to** W/n' **do**
3. $U \leftarrow chunk[i,j]; \mathcal{A}(U, U)$
4.   **for** $k \leftarrow 1$ **to** n' **do**
5. $pos \leftarrow (w_{(j-1) \times n' + k})/n'$
6. $X \leftarrow chunk[i, j + pos]; \mathcal{B}(X, U)$
7. $X \leftarrow chunk[i, j + pos + 1]; \mathcal{B}(X, U)$

A_0	B_1A_2	$B_1B_3A_4$	$B_1B_3B_5A_6$	$B_1B_3B_5B_7A_8$	$B_1B_3B_5B_7B_9A_{10}$
A_2	B_3A_4	$B_3B_5A_6$	$B_3B_5B_7A_8$	$B_3B_5B_7B_9A_{10}$	$B_3B_5B_7B_9B_{11}A_{12}$
A_4	B_5A_6	$B_5B_7A_8$	$B_5B_7B_9A_{10}$	$B_5B_7B_9B_{11}A_{12}$	$B_5B_7B_9B_{11}B_{13}A_{14}$
A_6	B_7A_8	$B_7B_9A_{10}$	$B_7B_9B_{11}A_{12}$	$B_7B_9B_{11}B_{13}A_{14}$	$B_7B_9B_{11}B_{13}B_{15}A_{16}$
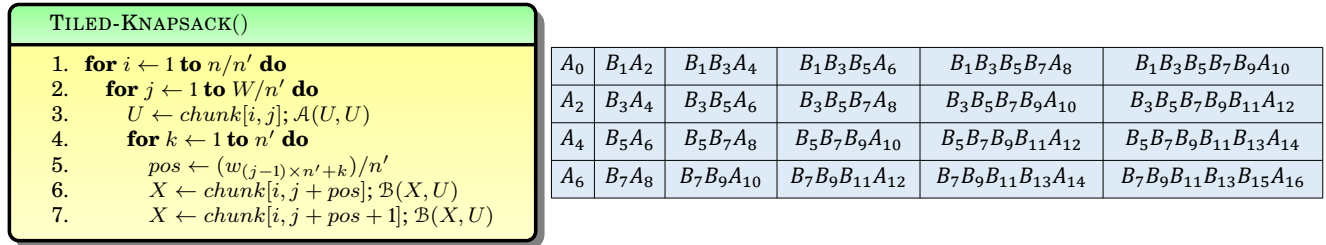
Figure C.7: Tiled knapsack algorithm.

The complexity of the tiled algorithm is given in Theorem 17.

Theorem 17 (Complexity of TILED-KNAPSACK). *The cache-aware processor-aware algorithm achieves* $T_1(n) = \Theta(nW)$, $Q_1(n) = \mathcal{O}\left(\frac{nW}{B} + n\right)$, *and* $S_p(n) = \Theta(pWn')$, *where p is the number of processors.*

Proof. We prove the theorem in several parts.

Work. We will not write recurrence relations to compute the work. This is because the number of function calls invoked at every level varies as a function call $\mathcal{F}(X, U)$, where $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}\}$ is invoked only if there is a dependency of any cell from X region to U region. Hence, we use a different technique to compute the total work.

Consider Figure C.6. We define something called a dependency curve. It is the cells on the path of the red curve shown in the figure on which the cells in the last column of the DP table depends on. It is simply the graph of the increasing order of weights shown from the rightmost column. Every column of blocks will have a dependency curve too. In total, there can be at most $\mathcal{O}(nW/(n')^2)$ number of function calls.

213

Every cell (i, j) in the DP table depends on cell $(i - 1, j)$. Therefore, every chunk (I, J) depends on itself, which requires an $\mathcal{A}$ function invocation. Hence, the total number of $\mathcal{A}$ function calls will be $(n/n') \times (W/n') = nW/(n')^2$.

A cell (i, j) depends on the cell $(i - 1, j - w_i)$ if $j \geq w_i$. Let the cell (i, j) be present in chunk (I, J) and the cell $(i - 1, j - w_i)$ be present in chunk (I, J'), where $J' \leq J - 1$. Then, a $\mathcal{B}$ function will be executed that writes to chunk (I, J) reading from chunk (I, J'), where $J' \leq J - 1$. As we will never call a $\mathcal{B}$ function twice, that writes to a chunk X and reads from a chunk U, the total number of $\mathcal{B}$ calls will be $\Theta\left((n/n') \times (W/n')\right) = \Theta\left(nW/(n')^2\right)$.

The total work is the product of the number of function calls to $\mathcal{A}$ and $\mathcal{B}$ and the work done by those functions. That is, the total work is $\Theta\left(nW/(n')^2\right) \times \Theta\left((n')^2\right) = \Theta\left(nW\right)$.

Space complexity. In the serial case, we process the $n' \times n'$ chunks present in single row of chunks. Hence, the serial space complexity is $\Theta\left(nn'\right)$. In the parallel version with p processors, we run at most p rows of chunks. Hence, the parallel space complexity will be $\Theta\left(pWn'\right)$.

Serial cache complexity. Temporal locality can be exploited as long as the work is asymptotically greater than the space. The $\mathcal{B}$ function takes $\mathcal{O}\left((n')^2\right)$ space and does $\mathcal{O}\left((n')^2\right)$. As $\mathcal{B}$ function is invoked a total of $\mathcal{O}\left(nW/(n')^2\right)$ times it is a dominating function. Hence, there cannot be any temporal locality. So, the serial cache complexity is $\mathcal{O}\left(nW/B\right)$. □

Divide-and-conquer wavefront ($\mathcal{WR}$-$\mathcal{DP}$) algorithm

Please refer Chapter 3 for a comprehensive introduction and analysis to divide-and-conquer wavefront algorithms. Algorithm RECURSIVE-WAVEFRONT-KNAPSACK in Figure C.8 gives a $\mathcal{WR}$-$\mathcal{DP}$ algorithm for the knapsack problem. A diagrammatic representation of the divide-and-conquer algorithm is given in Figure C.9. It is assumed that before calling our divide-and-conquer algorithm, we would have a sorted list of item weights.

Suppose $f \in \{\mathcal{A}, \mathcal{B}\}$ and $f(X, U)$ be a function call. It is important to note that f will be invoked if and only if DEPENDS(X, U) is true i.e., there is some cell in X that depends on some cell in U.

Completion-time is found from the DP recurrence:

$$
\mathfrak{C}(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0, \\ \mathfrak{C}(i, j - 1) + 2 & \text{if } j > 0, \\ \mathfrak{C}(i - 1, j) + 2 & \text{if } i > 0. \end{cases}
$$

The start- and end-time functions are computed as follows.

$$
\begin{array}{ll}
\mathcal{S}_{\mathcal{A}}(X, X) = \mathcal{E}_{\mathcal{A}}(X, X) = \mathfrak{C}(x_r, x_c) & \text{if } X \text{ is a } n' \times n' \text{ chunk,} \\
\mathcal{S}_{\mathcal{A}}(X, X) = \mathcal{S}_{\mathcal{A}}(X_{11}, X_{11}) & \text{if } X \text{ is not a } n' \times n' \text{ chunk,} \\
\mathcal{E}_{\mathcal{A}}(X, X) = \mathcal{E}_{\mathcal{A}}(X_{22}, X_{22}) & \text{if } X \text{ is not a } n' \times n' \text{ chunk,} \\
\mathcal{S}_{\mathcal{B}}(X, U) = \mathcal{E}_{\mathcal{B}}(X, U) = \mathfrak{C}(u_r, u_c) + 1 & \text{if } X \text{ is a } n' \times n' \text{ chunk,} \\
\mathcal{S}_{\mathcal{B}}(X, U) = \mathcal{S}_{\mathcal{B}}(X_{11}, U_{11}) & \text{if } X \text{ is not a } n' \times n' \text{ chunk,} \\
\mathcal{E}_{\mathcal{B}}(X, U) = \mathcal{E}_{\mathcal{B}}(X_{22}, U_{22}) & \text{if } X \text{ is not a } n' \times n' \text{ chunk.}
\end{array}
$$

Solving the recurrences, we have

RECURSIVE-WAVEFRONT-KNAPSACK()
1. $w \leftarrow 0$
2. **while** $w < \infty$ **do** $w \leftarrow \mathcal{A}(G, G, w)$

$\mathcal{A}(X, X, w)$
1. $v_i \leftarrow \infty$ for all $i \in [1, 6]$
2. **if** X is an $n' \times n'$ matrix **then**
3.   **if** $w = \mathcal{S}_{\mathcal{A}}(X, X)$ **then** $\mathcal{A}_{chunk}(X, X)$
4. **else**
5. $\mathcal{F}_{1..6} \leftarrow \{\mathcal{A}, \mathcal{B}, \mathcal{A}, \mathcal{A}, \mathcal{B}, \mathcal{A}\}$
6. $arg_{1..6} \leftarrow \{(X_{11}, X_{11}), (X_{12}, X_{11}), (X_{12}, X_{12}),$
          $(X_{21}, X_{21}), (X_{22}, X_{21}), (X_{22}, X_{22})\}$
7.   **parallel for** $i \leftarrow 1$ **to** 6 **do**
8.   **if** DEPENDS(arg_i) **then**
9.   **if** $w < \mathcal{S}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{S}_{\mathcal{F}_i}(arg_i)$
10.   **elif** $w \leq \mathcal{E}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{F}_i(arg_i, w)$
11.   **sync**
12. **return** $\min v_i$ for all $i \in [1, 6]$

$\mathfrak{C}(i, j)$
1. **return** $2(i + j)$

$\mathcal{S}_{\mathcal{A}}(X, X)$
1. **return** $\mathfrak{C}(x_r, x_c)$

$\mathcal{S}_{\mathcal{B}}(X, U)$
1. **return** $\mathfrak{C}(u_r, u_c) + 1$

$\mathcal{B}(X, U, w)$
1. $v_i \leftarrow \infty$ for all $i \in [1, 8]$
2. **if** X is an $n' \times n'$ matrix **then**
3.   **if** $w = \mathcal{S}_{\mathcal{B}}(X, U)$ **then** $\mathcal{B}_{chunk}(X, U)$
4. **else**
5. $\mathcal{F}_{1..8} \leftarrow \{\mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}\}$
6. $arg_{1..8} \leftarrow \{(X_{11}, U_{11}), (X_{12}, U_{11}), (X_{21}, U_{21}),$
          $(X_{22}, U_{21}), (X_{11}, U_{12}), (X_{12}, U_{12}),$
          $(X_{21}, U_{22}), (X_{22}, U_{22})\}$
7.   **parallel for** $i \leftarrow 1$ **to** 8 **do**
8.   **if** DEPENDS(arg_i) **then**
9.   **if** $w < \mathcal{S}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{S}_{\mathcal{F}_i}(arg_i)$
10.   **elif** $w \leq \mathcal{E}_{\mathcal{F}_i}(arg_i)$ **then** $v_i \leftarrow \mathcal{F}_i(arg_i, w)$
11.   **sync**
12. **return** $\min v_i$ for all $i \in [1, 8]$

DEPENDS(X, U)
1. **if** some cell(s) in X depends on some cell(s) in U **then**
2.   **return** true
3. **else return** false

$\mathcal{E}_{\mathcal{A}}(X, X)$
1. **return** $\mathfrak{C}(x_r + n - n', x_c + n - n')$

$\mathcal{E}_{\mathcal{B}}(X, U)$
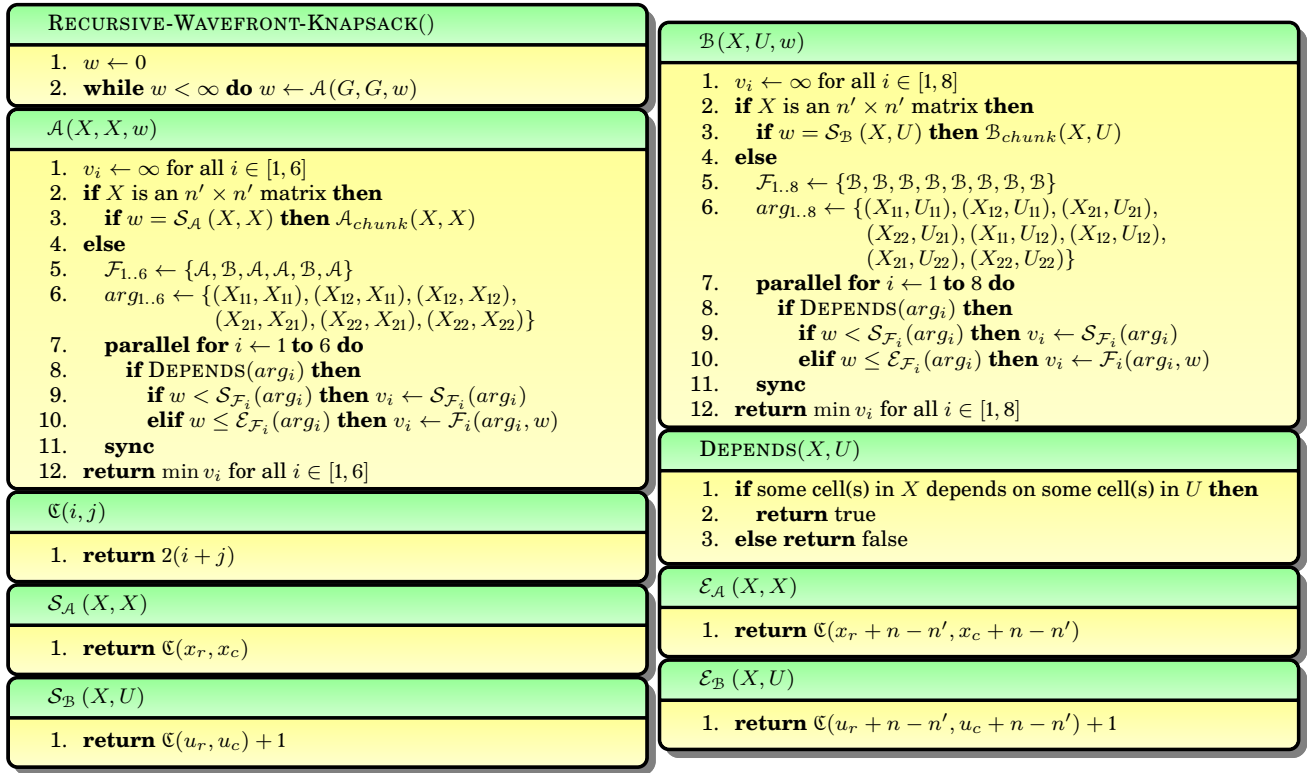1. **return** $\mathfrak{C}(u_r + n - n', u_c + n - n') + 1$

Figure C.8: A recursive divide-and-conquer wavefront DP algorithm for the knapsack problem. A region Z has its top-left corner at (z_r, z_c) and is of size $n \times n$.

$$\mathfrak{C}(i, j) = 2(i + j)$$
$$\mathcal{S}_{\mathcal{A}}(X, X) = \mathfrak{C}(x_r, x_c); \quad \mathcal{E}_{\mathcal{A}}(X, X) = \mathfrak{C}(x_r + n - n', x_c + n - n')$$
$$\mathcal{S}_{\mathcal{B}}(X, U) = \mathfrak{C}(u_r, u_c) + 1; \quad \mathcal{E}_{\mathcal{B}}(X, U) = \mathfrak{C}(u_r + n - n', u_c + n - n') + 1$$

Theorem 18 (Complexity of RECURSIVE-WAVEFRONT-KNAPSACK). *The cache-aware processor-aware algorithm achieves $T_1(n) = \Theta(nW)$, $Q_1(n) = \mathcal{O}\left(\frac{nW}{B} + n\right)$, and $S_p(n) = \Theta(pWn')$, where p is the number of processors.*

Proof. The proof is similar to that of Theorem 17. $\square$

Sliding window algorithm

Let $w_1, \ldots, w_n$ be sorted in increasing order and let $w_n = f(M)$, where $f(M) = o(M)$. The sliding window algorithm is given in Figure C.10. Let $\ell = 1$. Let array $output[0 \ldots W]$ represent the 0th row of the knapsack DP table. We find the maximum value of h such that $w_\ell + w_{\ell+1} + \cdots + w_h \leq \gamma M$, for some constant γ. Then using $w_\ell + w_{\ell+1} + \cdots + w_h$ space inside RAM, we can compute the rows $\ell \ldots h$ of the DP table and output the hth row of the knapsack DP table to the external-memory. We repeat the process by setting $\ell = h + 1$ and reading again from the array $output[0 \ldots W]$.
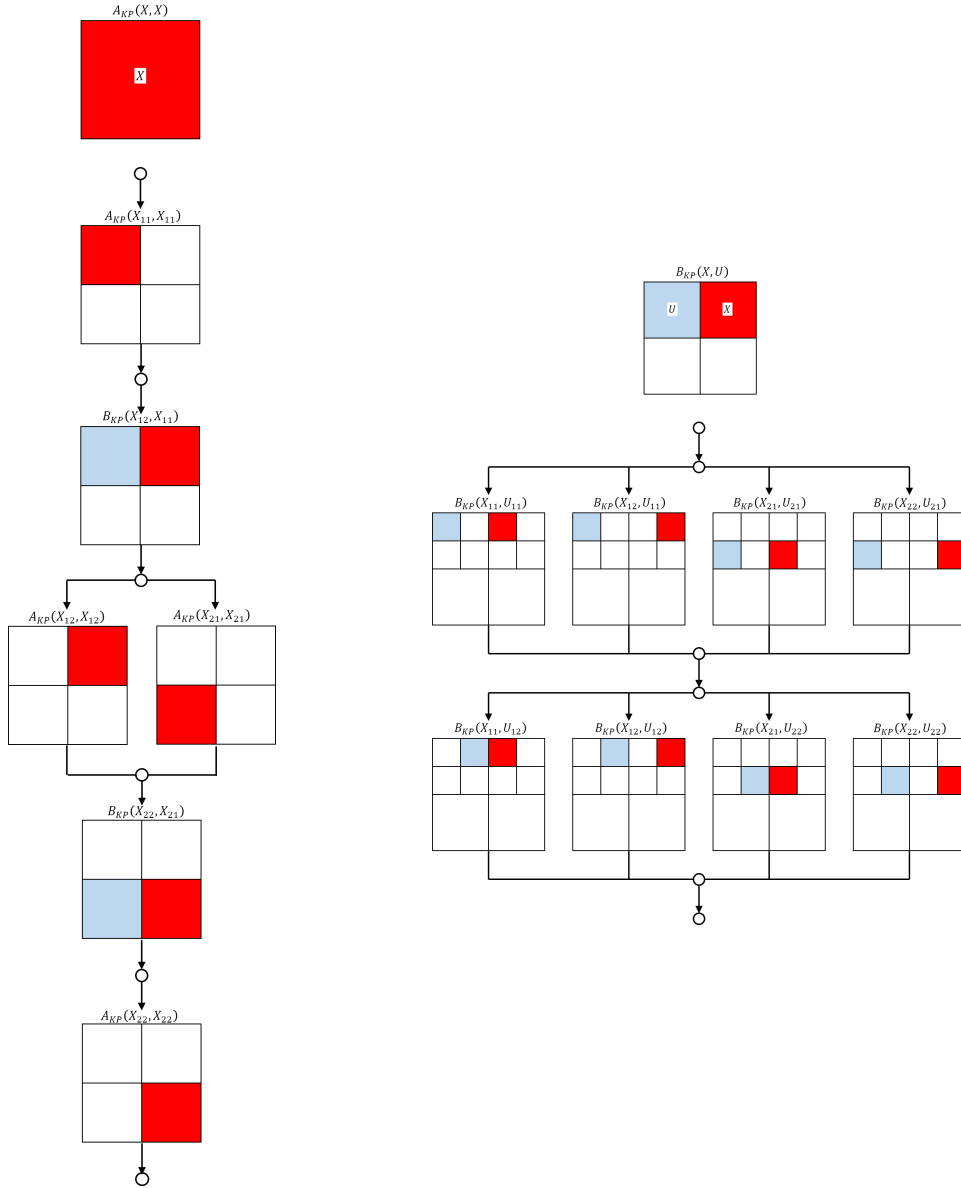
Figure C.9: A recursive divide-and-conquer DP algorithm for the knapsack problem.

Theorem 19 (Complexity of SLIDING-WINDOW-KNAPSACK). *The cache-aware algorithm achieves* $T_1(n) = \Theta(nW)$, $Q_1(n) = \Theta\left(wcount \cdot \left(\frac{W}{B} + 1\right)\right)$, *where* $wcount = \mathcal{O}\left(\frac{n}{\left(\frac{M}{f(M)}\right)}\right)$, *and* $S_1(n) = \mathcal{O}\left(\frac{W}{B}\right)$.

Proof. Only the required computations are performed by the algorithm and no extra computations are done. Hence, the work remains $\Theta(nW)$. At any time, the we use only constant number of rows of the knapsack DP table. Hence, the space used is $\Theta(W/B)$. We see that at a time (or any particular iteration), a bunch of $\omega(1)$ rows are computed in RAM and the array is written to the external memory for that bunch only once. Every time we write / overwrite a row to the disk, we increment the $wcount$ (meaning write counter) by 1. Hence, the total number of times we write a row is $wcount$. Hence, the total serial cache complexity is $\Theta(wcount \cdot ((W/B) + 1))$. We know that $wcount = o(n/\omega(1))$ and hence we exploit temporal locality and the algorithm is cache-efficient.
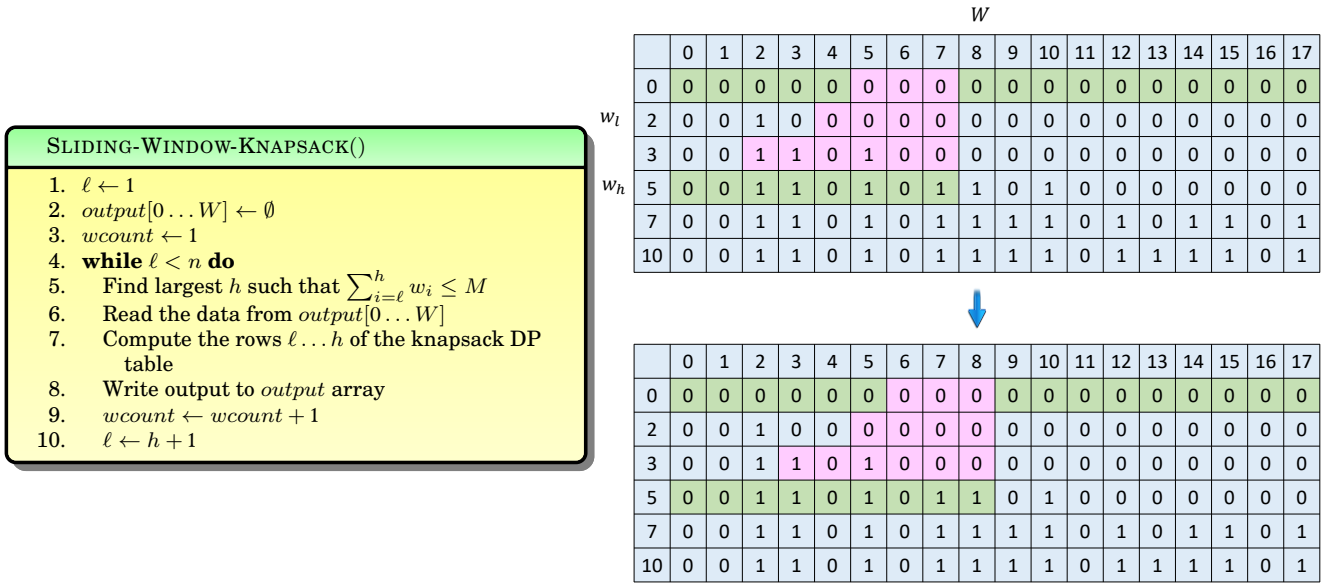
216

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	1	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0
7	0	0	1	1	0	1	0	1	1	1	1	0	1	0	1	1	0	1
10	0	0	1	1	0	1	0	1	1	1	1	0	1	1	1	1	0	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	1	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0
7	0	0	1	1	0	1	0	1	1	1	1	0	1	0	1	1	0	1
10	0	0	1	1	0	1	0	1	1	1	1	0	1	1	1	1	0	1

SLIDING-WINDOW-KNAPSACK()

1. $\ell \leftarrow 1$
2. $output[0 \ldots W] \leftarrow \emptyset$
3. $wcount \leftarrow 1$
4. **while** $\ell < n$ **do**
5.   Find largest h such that $\sum_{i=\ell}^{h} w_i \leq M$
6.   Read the data from $output[0 \ldots W]$
7.   Compute the rows $\ell \ldots h$ of the knapsack DP table
8.   Write output to $output$ array
9. $wcount \leftarrow wcount + 1$
10. $\ell \leftarrow h + 1$

Figure C.10: Sliding window knapsack algorithm. Left: The pseudocode. Right: An example of subset sum problem.

Sliding merging window algorithm

This algorithm has similarities to the sliding window knapsack algorithm. Let $w_1, \ldots, w_n$ be sorted in increasing order. Let s_i represent the maximum number of subset sums (using weights $\{w_1, \ldots, w_i\}$) in the range $[a, a + w_i]$, for all $a \in [0, W - w_i]$. Let $s_n = f(M)$, where $f(M) = o(M)$.

Let $\ell = 1$. Let array $output[0 \ldots W]$ represent the 0th row of the knapsack DP table. We find the maximum value of h such that $m_\ell + m_{\ell+1} + \cdots + m_h \leq \gamma M$, for some constant γ. Then using $m_\ell + m_{\ell+1} + \cdots + m_h$ space inside RAM, we can compute the rows $\ell \ldots h$ of the DP table and output the hth row of the knapsack DP table to the external-memory. We repeat the process by setting $\ell = h + 1$ and reading again from the array $output[0 \ldots W]$.

Computing s_i. How can we find s_i? We can compute s_i using the formula $s_i = (w_i/d_i)$, where d_i can be found by solving the integer programming problem. Let $\sum_{j=1}^{i} x_j w_j = d_i$, where $d_i \in [1, w_1]$ and d_i is minimum possible value, and $x_j \in \{-1, 0, 1\}$. The integer programming problem might be hard to solve in polynomial time. But, probably there exist constant factor approximation algorithms.

Variants of the problem

The Knapsack problem is similar to the change making problem and the subset-sum problem. All these problems have data-sensitive dependency. That is, the DP dependency structure for the problem is dependent on each run / execution of the algorithm.

Change making problem. We have n coins of denominations $c_1, c_2, \ldots, c_n$. We need to make a change for m amounts of money in minimum number of coins. We need to find the number of coins of each denomination that makes a change for m. There are four versions of the problem.

* Greedy algorithm when there are unlimited number of coins of each denomination. See Figure C.11.
* DP algorithm when there are unlimited number of coins of each denomination. See Figure C.11.
* DP algorithm when there is exactly 1 coin of each denomination. The problem is also called as 01 change making problem. See Figure C.11.
* DP algorithm when there are limited number of coins of each denomination, say $k_1, k_2, \ldots, k_n$. See Figure C.11.

Subset-sum problem. In this problem, we are given a multiset $S = \{x_1, x_2, \ldots, x_n\}$ of n integers. We set $P[i, j]$ to true if a subset of $\{x_1, x_2, \ldots, x_i\}$ sums to j, and to false otherwise. Then,

$$
P[i, j] = \begin{cases} \text{true} & \text{if } j = 0, \\ \text{true} & \text{if } P[i-1, j] = \text{true or } P[i-1, j-x_i] = \text{true}, \\ \text{false} & \text{otherwise.} \end{cases}
$$

CM-UNLIMITED-GREEDY($n\ m, c[1\dots n]$)

Require: $c[1] > c[2] > \cdots > c[n]$
1. $money \leftarrow m$
2. **for** $i \leftarrow 1$ **to** n **do**
3. $a[i] \leftarrow \left\lfloor \frac{money}{c[i]} \right\rfloor$
4. $mc \leftarrow mc + a[i]$
5. $money \leftarrow money - a[i]c[i]$
6. **if** $money > 0$ **then**
7.   **print** Cannot make change
8. **else return** $a[1\dots n], mc$

CM-UNLIMITED-DP($n\ m, c[1\dots n]$)

{ Compute the minimum number of coins }
1. $mc[0] \leftarrow 0$
2. **for** $i \leftarrow 1$ **to** m **do**
3. $mc[i] \leftarrow \infty$
4.   **for** $j \leftarrow 1$ **to** n **do**
5.   **if** $c[j] \le i$ **and** $mc[i - c[j]] + 1 < mc[i]$ **then**
6. $mc[i] \leftarrow mc[i - c[j]] + 1; parent[i] \leftarrow j$

{ Compute #coins of each denomination }
7. **if** $mc[m] = \infty$ **then print** Cannot make change
8. **else**
9. $a[1\dots n] \leftarrow \emptyset; i \leftarrow m$
10.   **while** $i \ge 1$ **do**
11. $a[parent[i]] \leftarrow a[parent[i]] + 1; i \leftarrow i - c[parent[i]]$
12.   **return** $a[1\dots n], mc[m]$

CM-LIMITED-01-DP($n\ m, c[1\dots n]$)

{ Compute the minimum number of coins }
1. $mc[0, 0\dots m] \leftarrow \infty; mc[0\dots n, 0] \leftarrow \emptyset$
2. **for** $i \leftarrow 1$ **to** n **do**
3.   **for** $j \leftarrow 1$ **to** m **do**
4. $mc[i, j] \leftarrow mc[i - 1, j]; parent[i, j] \leftarrow 0$
5.   **if** $c[i] \le j$ **and** $mc[i - 1, j - c[i]] + 1 < mc[i, j]$ **then**
6. $mc[i, j] \leftarrow mc[i, j - c[i]] + 1; parent[i, j] \leftarrow 1$

{ Compute #coins of each denomination }
7. **if** $mc[n, m] = \infty$ **then print** Cannot make change
8. **else**
9. $a[1\dots n] \leftarrow \emptyset; j \leftarrow m$
10.   **for** $i \leftarrow n$ **to** 1 **do**
11.   **if** $parent[i, j] = 1$ **then**
12. $j = j - c[i]; a[i] \leftarrow a[i] + 1$
13.   **return** $a[1\dots n], mc[n, m]$

CM-LIMITED-DP($n, m, c[1\dots n], k[1\dots n]$)

Input: n number of denominations, m money to be changed for, $c[1\dots n]$ coin denominations, $k[1\dots n]$ number of coins available
Output: $a[1\dots n]$ number of coins used, $mincoins[nn][mm]$ minimum number of coins
Require: $a[1] + a[2] + \cdots + a[n]$ is minimized

{ Convert the limited change making problem to 0-1 change making problem }
1. $mm \leftarrow m; nn \leftarrow 0$
2. **for** $i \leftarrow 1$ **to** n **do**
3.   **if** $k[i] > 0$ **then**
4. $count \leftarrow 0; power \leftarrow 1$
5.   **repeat**
6.   **if** $count + power > k[i]$ **then**
7. $power \leftarrow k[i] - count$
8. $nn \leftarrow nn + 1$
9. $cc[nn] \leftarrow power \times c[i]$
10. $d[nn] \leftarrow i;$
11. $v[nn] \leftarrow power$
12. $count \leftarrow count + power$
13. $power \leftarrow 2 \times power$
14.   **until** $count = k[i]$

{ Compute the minimum number of coins }
15. $mincoins[0\dots nn][0] \leftarrow \emptyset$
16. $mincoins[0][0\dots nn] \leftarrow \emptyset$
17. **for** $i \leftarrow 1$ **to** nn **do**
18.   **for** $j \leftarrow 1$ **to** mm **do**
19. $mincoins[i][j] \leftarrow mincoins[i - 1][j]$
20. $parent[i][j] \leftarrow 0$
21.   **if** $cc[i] \le j$ **and** $mincoins[i - 1][j - cc[i]] + v[i] < mincoins[i][j]$ **then**
22. $mincoins[i][j] \leftarrow mincoins[i - 1][j - cc[i]] + v[i]$
23. $parent[i][j] \leftarrow 1$

{ Compute #coins of each denomination }
24. **if** $mincoins[nn][mm] = \infty$ **then**
25.   **print** Cannot make change
26. **else**
27. $a[1\dots nn] \leftarrow \emptyset; j \leftarrow mm$
28.   **for** $i \leftarrow nn$ **to** 1 **do**
29.   **if** $parent[i][j] = 1$ **then**
30. $j \leftarrow j - cc[i]$
31. $a[d[i]] \leftarrow a[d[i]] + v[i]$
32.   **return** $a[1\dots nn], mincoins[nn][mm]$

Figure C.11: Change making algorithms.

Appendix D

Efficient Divide-&-Conquer Tiled DP Algorithms

In this section, we present new r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms for several DP problems. Such algorithms can be easily converted to cache-aware tiled algorithms by following the process as described in Section 3.2.

D.1 Matrix multiplication

Though MM is not a DP problem, the approach followed to get a 2-way divide-and-conquer algorithm is similar as for a DP problem. A 2-way divide-and-conquer algorithm for MM was presented in [Frigo et al., 1999]. It is extremely easy to generalize it to an r-way divide-and-conquer algorithm and it is presented in Figure D.1.

$\mathcal{A}_{MM}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{A}_{loop\text{-}MM}(X, U, V)$
    else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel:** $\mathcal{A}_{MM}(X_{ij}, U_{ik}, V_{kj}, d+1)$ for $i, j \in [1, r]$

Figure D.1: An r-way $\mathcal{R}$-$\mathcal{DP}$ for matrix multiplication.

D.2 Longest common subsequence

The problem is described in detail in Section A.1. An r-way $\mathcal{R}$-$\mathcal{DP}$ to solve the problem is given in Figure D.2.

D.3 Floyd-Warshall's all-pairs shortest path

The problem is described in detail in Section A.3. An r-way $\mathcal{R}$-$\mathcal{DP}$ algorithm for Floyd-Warshall's APSP is described in Figure A.7.

```
𝒜_LCS(X, d)
  1.  r ← tilesize[d]
  2.  if r ≥ m then 𝒜_loop-LCS(X)
      else
  3.      for k ← 0 to 2r − 2 do
  4.          parallel: 𝒜_LCS(X_ij, d + 1) for i ∈ [max(1, k − r + 2), min(k + 1, r)] and j = k − i + 2
```

Figure D.2: An r-way $\mathcal{R}\text{-}\mathcal{DP}$ for LCS.

D.4 Gaussian elimination without pivoting

The problem is described in detail in Section A.4. An r-way $\mathcal{R}\text{-}\mathcal{DP}$ algorithm for Gaussian elimination without pivoting is given in Figure D.3.

```
𝒜_GE(X, U, V, W, d)
  1.  r ← tilesize[d]
  2.  if r ≥ m then 𝒜_loop-GE(X, U, V, W)
      else
  3.      for k ← 1 to r do
  4.          𝒜_GE(X_kk, U_kk, V_kk, W_k, d + 1)
  5.          parallel: ℬ_GE(X_kj, U_kk, V_kj, W_k, d + 1), 𝒞_GE(X_ik, U_ik, V_kk, W_k, d + 1) for i, j ∈ [k + 1, r]
  6.          parallel: 𝒟_GE(X_ij, U_ik, V_kj, W_k, d + 1) for i, j ∈ [k + 1, r]
```

```
ℬ_GE(X, U, V, W, d)
  1.  r ← tilesize[d]
  2.  if r ≥ m then ℬ_loop-GE(X, U, V, W)
      else
  3.      for k ← 1 to r do
  4.          parallel: ℬ_GE(X_kj, U_kk, V_kj, W_k, d + 1) for j ∈ [1, r]
  5.          parallel: 𝒟_GE(X_ij, U_ik, V_kj, W_k, d + 1) for i ∈ [k + 1, r], j ∈ [1, r]
```

```
𝒞_GE(X, U, V, W, d)
  1.  r ← tilesize[d]
  2.  if r ≥ m then 𝒞_loop-GE(X, U, V, W)
      else
  3.      for k ← 1 to r do
  4.          parallel: 𝒞_GE(X_ik, U_ik, V_kk, W_k, d + 1) for i ∈ [1, r]
  5.          parallel: 𝒟_GE(X_ij, U_ik, V_kj, W_k, d + 1) for i ∈ [1, r], j ∈ [k + 1, r]
```

```
𝒟_GE(X, U, V, W, d)
  1.  r ← tilesize[d]
  2.  if r ≥ m then 𝒟_loop-GE(X, U, V, W)
      else
  3.      for k ← 1 to r do
  4.          parallel: 𝒟_GE(X_ij, U_ik, V_kj, W_k, d + 1) for i, j ∈ [1, r]
```

Figure D.3: An r-way $\mathcal{R}\text{-}\mathcal{DP}$ for Gaussian elimination without pivoting algorithm.

D.5 Parenthesis problem

The problem is described in detail in Section A.2. An r-way $\mathcal{R}\text{-}\mathcal{DP}$ was presented in [Chowdhury and Ramachandran, 2008] and its complete pseudocode is given in Figure D.4.

$\mathcal{A}_{par}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{A}_{loop\text{-}par}(X, U, V)$
   else
3.   Let $diagonal$ represent $(j - i)$
4.   **parallel:** $\mathcal{A}_{par}(X_{i,j}, U_{i,j}, V_{i,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal = 0$
5.   **for** $k \leftarrow 1$ **to** $r - 1$ **do**
6.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U_{i,k+i-1}, V_{k+i-1,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal \in [k, \min\{2k - 2, r - 1\}]$
7.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U_{i,1+i}, V_{1+i,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal \in [k, \min\{2k - 3, r - 1\}]$
8.   **parallel:** $\mathcal{B}_{par}(X_{i,j}, U_{i,i}, V_{j,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal = k$

$\mathcal{B}_{par}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{B}_{loop\text{-}par}(X, U, V)$
   else
3.   Let $U'_{i,\ell} = \begin{cases} X_{i,\ell} & \text{if } \ell > 0, \\ U_{i,\ell+r} & \text{if } \ell \leq 0. \end{cases}$ and $V'_{\ell,j} = \begin{cases} V_{\ell,j} & \text{if } \ell > 0, \\ X_{\ell+r,j} & \text{if } \ell \leq 0. \end{cases}$
4.   Let $diagonal$ represent $(j - i)$
5.   **for** $k \leftarrow 1$ **to** $2r - 1$ **do**
6.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U'_{i,k-r+i-1}, V'_{k-r+i-1,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal + r \in [k, \min\{2k - 2, 2r - 1\}]$
7.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U'_{i,1+i-r}, V'_{1+i-r,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal + r \in [k, \min\{2k - 3, 2r - 1\}]$
8.   **parallel:** $\mathcal{B}_{par}(X_{i,j}, U_{i,i}, V_{j,j}, d+1)$ for $i, j \in [1, r]$ and $diagonal + r = k$

$\mathcal{C}_{par}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{C}_{loop\text{-}par}(X, U, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel:** $\mathcal{C}_{par}(X_{i,j}, U_{i,k}, V_{k,j}, d+1)$ for $i, j \in [1, r]$

Figure D.4: An r-way $\mathcal{R}$-$\mathcal{DP}$ for parenthesis problem.

D.6 Sequence alignment with gap penalty

The problem is described in detail in Section A.5. An r-way $\mathcal{R}$-$\mathcal{DP}$ to solve the problem is given in Figure D.5.

$\mathcal{A}_{gap}(X, U, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{A}_{loop\text{-}gap}(X, U)$
   else
3. $\mathcal{A}_{gap}(X_{11}, U_{11}, d+1)$
4.   **for** $k \leftarrow 1$ **to** $2r-2$ **do**
5.   **parallel:** $\mathcal{B}_{gap}(X_{ij}, U_{i(k-i+1)}, d+1)$ for $i \in [\max(1, k-r+1), \min(k, r)]$ and $j \in [k-i+2, r]$
6.   **parallel:** $\mathcal{C}_{gap}(X_{ij}, U_{(k-j+1)j}, d+1)$ for $j \in [\max(1, k-r+1), \min(k, r)]$ and $i \in [k-j+2, r]$
7.   **parallel:** $\mathcal{A}_{gap}(X_{ij}, U_{ij}, d+1)$ for $i \in [\max(1, k-r+2), \min(k+1, r)]$ and $j = k-i+2$

$\mathcal{B}_{gap}(X, U, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{B}_{loop\text{-}gap}(X, U)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
4.   **parallel:** $\mathcal{B}_{gap}(X_{ij}, U_{ik}, d+1)$ for $i, j \in [1, r]$

$\mathcal{C}_{gap}(X, V, d)$

1. $r \leftarrow tilesize[d]$
2. **if** $r \geq m$ **then** $\mathcal{C}_{loop\text{-}gap}(X, V)$
   else
3.   **for** $k \leftarrow 1$ **to** r **do**
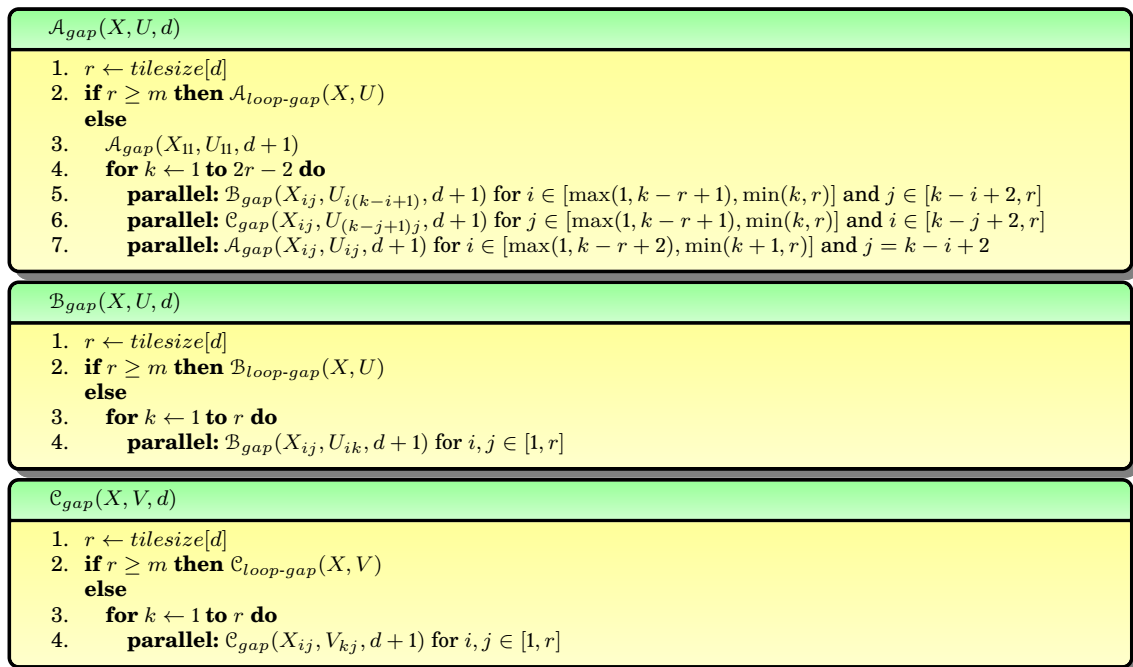4.   **parallel:** $\mathcal{C}_{gap}(X_{ij}, V_{kj}, d+1)$ for $i, j \in [1, r]$

Figure D.5: An r-way $\mathcal{R}$-$\mathcal{DP}$ for sequence alignment with gap penalty problem.

Appendix E

Efficient Divide-&-Conquer Hybrid DP Algorithms

In this section, we develop hybrid divide-and-conquer algorithms for three problems: multi-instance Viterbi algorithm, Floyd-Warshall's all-pairs shortest path, and protein accordion folding.

E.1   Matrix multiplication

Complexities for Lemma 3 is shown below.

$$Q_1(n) = 8Q_1\left(\frac{n}{2}\right) + c = 8\left(Q_1\left(\frac{n}{2^2}\right) + c\right) + c = 8^2 Q_1\left(\frac{n}{2^2}\right) + c\left(8^1 + 8^0\right)$$

$$= 8^k Q_1\left(\frac{n}{2^k}\right) + c\left(8^{k-1} + \cdots + 8^0\right) \leq 8^k c'\left(\frac{M}{B} + \sqrt{M}\right) + c8^k$$

$$\leq 8^{\log_4 \frac{n^2}{M}} c'\left(\frac{M}{B} + \sqrt{M}\right) + c8^{\log_4 \frac{n^2}{M}} \qquad \left(\because (n/2^k)^2 \leq \alpha M\right)$$

$$= \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$$

$$= \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + n\right) \qquad \left(\because M = \Omega\left(B^2\right)\right)$$

$$S_p(m) = 8S_p\left(\frac{m}{2}\right) + cm^2 = 8^2 S_p\left(\frac{m}{2^2}\right) + cm^2\left(\frac{8}{2^2} + 1\right) = 8^k S_p\left(\frac{m}{2^k}\right) + cm^2\left(\frac{8^{k-1}}{(2^{k-1})^2} + \cdots + 1\right)$$

$$\leq 8^k S_p\left(\frac{m}{2^k}\right) + cm^2 2^k \qquad \left(\mathbf{Say}\ (2^k)^3 \leq p\right)$$

$$\leq 8^{\log_8 p} \cdot \Theta\left(\left(\frac{n}{2^{\log_8 p}}\right)^2\right) + cn^2 2^{\log_8 p} = \Theta\left(n^2 p^{1/3}\right) \qquad \left(\mathbf{Set}\ m = n\right)$$

The detailed derivation of the complexities of Theorem 13 is given here.

$$T_1(n) = 8T_1\left(\frac{n}{2}\right) + c = 8\left(8T_1\left(\frac{n}{2^2}\right) + c\right) + c = 8^2 T_1\left(\frac{n}{2^2}\right) + c\left(8^1 + 1\right)$$

$$= 8^k T_1\left(\frac{n}{2^k}\right) + c\left(8^{k-1} + \cdots + 1\right) \leq 8^k c'\left(\frac{n}{2^k}\right)^3 + 8^k c = \mathcal{O}\left(n^3\right)$$

$$Q_1(n) = 8Q_1\left(\frac{n}{2}\right) + c = 8\left(Q_1\left(\frac{n}{2^2}\right) + c\right) + c = 8^2 Q_1\left(\frac{n}{2^2}\right) + c\left(8^1 + 8^0\right)$$

$$= 8^k Q_1\left(\frac{n}{2^k}\right) + c\left(8^{k-1} + \cdots + 8^0\right) \leq 8^k c'\left(\left(\frac{n}{2^k}\right)^3 \frac{1}{B\sqrt{M}}\right) + c8^k$$

$$\leq c'\frac{n^3}{B\sqrt{M}} + c8^{\log_4 \frac{n^2}{M}} = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + n\right)$$

$$T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + c\log n = T_\infty\left(\frac{n}{2^2}\right) + c\left(\log n + \log\frac{n}{2}\right) = T_\infty\left(\frac{n}{2^k}\right) + c\left(\log n + \cdots + \log\frac{n}{2^{k-1}}\right)$$

$$= c'\frac{n}{2^k} + c\left((k-1)\log n - \frac{(k-1)k}{2}\right) = \mathcal{O}\left(\frac{n}{2^k} + k(2\log n - k)\right)$$

E.2 Multi-instance Viterbi algorithm

A multi-instance Viterbi algorithm is described in Section 4.3. An in-place divide-and-conquer algorithm is given in . The $\mathcal{A}_{VA}$ function in the algorithm will be called $\Theta(t)$ times and the function is similar in recursive structure to $\mathcal{A}_{MM}$ of Figure 6.3. In this Viterbi problem, there is only one recursive function $\mathcal{A}_{VA}$ and it is MM-like. Hence, it satisfies the span property (Theorem 12) and we can replace it with its not-in-place / hybrid version of the function.

> **Lemma 4 (In-place VA).** *The in-place VA Multi-Instance-Viterbi, assuming a tall cache, has a complexity of* $T_1(n) = \Theta(n^3 t)$*,* $Q_1(n) = \Theta\left(n^3 t/(B\sqrt{M}) + n^2 t/B + nt\right)$*,* $T_\infty(n) = \Theta(nt)$*, and* $S_p(n) = \Theta(n^2)$*.*

Proof. The recurrences for the complexities of $\mathcal{A}_{VA}$ is exactly the same as that in Lemma 2. Also, the function $\mathcal{A}_{VA}$ is invoked t times as there are t timesteps. Hence, the lemma follows. □

Following the argument of Lemma 4 and recurrences from Lemma 3, we can prove Lemma 5. Similarly, following the argument of Lemma 4 and recurrences from Theorem 13, we can prove Theorem 20.

> **Lemma 5 (Not-in-place VA).** *The not-in-place multi-instance Viterbi algorithm, assuming a tall cache, has a complexity of* $T_1(n) = \Theta(n^3 t)$*,* $Q_1(n) = \Theta\left(n^3 t/(B\sqrt{M}) + n^2 t/B + nt\right)$*,* $T_\infty(n) = \Theta\left(t\log^2 n\right)$*, and* $S_p(n) = \Theta\left(p^{1/3} n^2\right)$*.*

> **Theorem 20 (Hybrid VA).** *The hybrid multi-instance Viterbi algorithm, assuming a tall cache, has a complexity of* $T_1(n) = \Theta(n^3 t)$*,* $Q_1(n) = \mathcal{O}\left(n^3 t/(B\sqrt{M}) + n^2 t/B + nt\right)$*,* $T_\infty(n) = \mathcal{O}\left(kt(2\log n - k) + nt/2^k\right)$*, and* $S_p(n) = \mathcal{O}\left(\min\{p^{1/3}, 2^k\}n^2\right)$*.*

E.3 Floyd-Warshall's all-pairs shortest path

Floyd-Warshall's all-pairs shortest path algorithm is described in Section A.3. Figures A.5 and A.6 show an in-place 2-way divide-and-conquer algorithm. The $\mathcal{D}$ function in the algorithm is matrix-multiplication-like and dominates the total work complexity. Hence, by simply modifying the $\mathcal{D}$ function to use extra space, we end up with not-in-place FW algorithm.

> **Lemma 6 (In-place FW).** *The in-place FW, assuming a tall cache, has a complexity of $T_1(n) = \Theta(n^3)$, $Q_1(n) = \Theta\left(n^3/(B\sqrt{M}) + n^2/B + n\right)$, $T_\infty(n) = \Theta\left(n\log^2 n\right)$, and $S_p(n) = \Theta(n^2)$.*

Proof. The recurrences are as follows. For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity and span of f_{FW} on a matrix of size $n \times n$. Then

$$Q_\mathcal{A}(n) = Q_\mathcal{B}(n) = Q_\mathcal{C}(n) = Q_\mathcal{D}(n) = \mathcal{O}\left(\frac{n^2}{B} + n\right) \qquad \text{if } n^2 \leq \gamma M,$$

$$Q_\mathcal{A}(n) = 2\left(Q_\mathcal{A}\left(\frac{n}{2}\right) + Q_\mathcal{B}\left(\frac{n}{2}\right) + Q_\mathcal{C}\left(\frac{n}{2}\right) + Q_\mathcal{D}\left(\frac{n}{2}\right)\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_A M.$$

$$Q_\mathcal{B}(n) = 4\left(Q_\mathcal{B}\left(\frac{n}{2}\right) + Q_\mathcal{D}\left(\frac{n}{2}\right)\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_B M.$$

$$Q_\mathcal{C}(n) = 4\left(Q_\mathcal{C}\left(\frac{n}{2}\right) + Q_\mathcal{D}\left(\frac{n}{2}\right)\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_C M.$$

$$Q_\mathcal{D}(n) = 8Q_\mathcal{D}\left(\frac{n}{2}\right) + \Theta(1) \qquad \text{if } n^2 > \gamma_D M.$$

$$T_\mathcal{A}(n) = T_\mathcal{B}(n) = T_\mathcal{C}(n) = T_\mathcal{D}(n) = \mathcal{O}(1) \qquad \text{if } n = 1,$$

$$T_\mathcal{A}(n) = 2\left(T_\mathcal{A}\left(\frac{n}{2}\right) + + \max\left\{T_\mathcal{B}\left(\frac{n}{2}\right), T_\mathcal{C}\left(\frac{n}{2}\right)\right\} + T_\mathcal{D}\left(\frac{n}{2}\right)\right) + \Theta(1) \qquad \text{if } n > 1.$$

$$T_\mathcal{B}(n) = 2\left(T_\mathcal{B}\left(\frac{n}{2}\right) + T_\mathcal{D}\left(\frac{n}{2}\right)\right) + \Theta(1) \qquad \text{if } n > 1.$$

$$T_\mathcal{C}(n) = 2\left(T_\mathcal{C}\left(\frac{n}{2}\right) + T_\mathcal{D}\left(\frac{n}{2}\right)\right) + \Theta(1) \qquad \text{if } n > 1.$$

$$T_\mathcal{D}(n) = 2T_\mathcal{D}\left(\frac{n}{2}\right) + \Theta(1) \qquad \text{if } n > 1.$$

where, γ, γ_A, γ_B, γ_C and γ_D are suitable constants, $Q_1(n) = Q_\mathcal{A}(n)$, and $T_\infty(n) = T_\mathcal{A}(n)$. Solving, we have the lemma. $\qquad \square$

> **Lemma 7 (Not-in-place FW).** *The not-in-place FW, assuming a tall cache, has a complexity of $T_1(n) = \Theta(n^3)$, $Q_1(n) = \Theta\left(n^3/(B\sqrt{M}) + n^2/B + n\right)$, $T_\infty(n) = \Theta(n\log n)$, and $S_p(n) = \Theta\left(p^{1/3}n^2\right)$.*

Proof. The recurrences from Lemma 6 will be the same except the following.

$$T_\mathcal{D}(n) = T_\mathcal{D}\left(\frac{n}{2}\right) + \Theta(\log n) \qquad \text{if } n > 1.$$

Solving the recurrences, the lemma follows. $\qquad \square$

The recurrences for the hybrid algorithm are:

$$Q_{\mathcal{A}}(m) = Q_{\mathcal{B}}(m) = Q_{\mathcal{C}}(m) = Q_{\mathcal{D}}(m) = \mathcal{O}\left(\frac{(n/2^k)^3}{B\sqrt{M}} + \frac{(n/2^k)^2}{B} + (n/2^k)\right) \qquad \text{if } m = \frac{n}{2^k},$$

$$Q_{\mathcal{A}}(m) = 2\left(Q_{\mathcal{A}}\left(\frac{m}{2}\right) + Q_{\mathcal{B}}\left(\frac{m}{2}\right) + Q_{\mathcal{C}}\left(\frac{m}{2}\right) + Q_{\mathcal{D}}\left(\frac{m}{2}\right)\right) + \Theta(1) \qquad \text{if } m > \frac{n}{2^k}.$$

$$Q_{\mathcal{B}}(m) = 4\left(Q_{\mathcal{B}}\left(\frac{m}{2}\right) + Q_{\mathcal{D}}\left(\frac{m}{2}\right)\right) + \Theta(1) \qquad \text{if } m > \frac{n}{2^k}.$$

$$Q_{\mathcal{C}}(m) = 4\left(Q_{\mathcal{C}}\left(\frac{m}{2}\right) + Q_{\mathcal{D}}\left(\frac{m}{2}\right)\right) + \Theta(1) \qquad \text{if } m > \frac{n}{2^k}.$$

$$Q_{\mathcal{D}}(m) = 8Q_{\mathcal{D}}\left(\frac{m}{2}\right) + \Theta(1) \qquad \text{if } m > \frac{n}{2^k}.$$

$$T_{\mathcal{A}}(m) = \Theta\left((n/2^k)\log^2(n/2^k)\right) \qquad \text{if } m = n/2^k,$$

$$T_{\mathcal{B}}(m) = T_{\mathcal{C}}(m) = \Theta\left((n/2^k)\log(n/2^k)\right) \qquad \text{if } m = n/2^k,$$

$$T_{\mathcal{D}}(m) = \Theta\left(n/2^k\right) \qquad \text{if } m = n/2^k,$$

$$T_{\mathcal{A}}(m) = 2\left(T_{\mathcal{A}}\left(\frac{m}{2}\right) + \max\{T_{\mathcal{B}}\left(\frac{m}{2}\right), T_{\mathcal{C}}\left(\frac{m}{2}\right)\} + T_{\mathcal{D}}\left(\frac{m}{2}\right)\right) + \Theta(\log m) \qquad \text{if } m > n/2^k.$$

$$T_{\mathcal{B}}(m) = 2\left(T_{\mathcal{B}}\left(\frac{m}{2}\right) + T_{\mathcal{D}}\left(\frac{m}{2}\right)\right) + \Theta(\log m) \qquad \text{if } m > n/2^k.$$

$$T_{\mathcal{C}}(m) = 2\left(T_{\mathcal{C}}\left(\frac{m}{2}\right) + T_{\mathcal{D}}\left(\frac{m}{2}\right)\right) + \Theta(\log m) \qquad \text{if } m > n/2^k.$$

$$T_{\mathcal{D}}(m) = T_{\mathcal{D}}\left(\frac{m}{2}\right) + \Theta(\log m) \qquad \text{if } m > n/2^k.$$

$$S_p(n) = \min\{p^{1/3}, n^2\sum_{i=0}^{k-1} 8^i\left(\frac{n}{2^i} \times \frac{n}{2^i}\right)\} = \Theta\left(\min\{p^{1/3}, 2^k\}n^2\right)$$

Derivation of the cache complexity is given here.

$$Q_{\mathcal{D}}(n) = 8Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c = 8\left(8Q_{\mathcal{D}}\left(\frac{n}{2^2}\right) + c\right) + c$$

$$= 8^2 Q_{\mathcal{D}}\left(\frac{n}{2^2}\right) + c(8^1 + 8^0) = 8^k Q_{\mathcal{D}}\left(\frac{n}{2^k}\right) + c(8^{k-1} + 8^{k-2} + \cdots + 8^0)$$

$$\leq 8^k c'\left(\frac{(n/2^k)^3}{B\sqrt{M}}\right) + c8^k = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$$

It is important to note that

$$Q_{\mathcal{D}}\left(\frac{n}{2^i}\right) = \mathcal{O}\left(\frac{(n/2^i)^3}{B\sqrt{M}}\right)$$

$$Q_{\mathcal{C}}(n) = 4Q_{\mathcal{C}}\left(\frac{n}{2}\right) + 4Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c = 4\left(4Q_{\mathcal{C}}\left(\frac{n}{2^2}\right) + 4Q_{\mathcal{D}}\left(\frac{n}{2^2}\right) + c\right) + 4Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c$$

$$= 4^2 Q_{\mathcal{C}}\left(\frac{n}{2^2}\right) + 4^2 Q_{\mathcal{D}}\left(\frac{n}{2^2}\right) + 4Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c\left(4^1 + 4^0\right)$$

$$= 4^k Q_{\mathcal{C}}\left(\frac{n}{2^k}\right) + 4^k Q_{\mathcal{D}}\left(\frac{n}{2^k}\right) + \cdots + 4Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c\left(4^{k-1} + 4^{k-2} + \cdots + 4^0\right)$$

$$\leq 4^k c'\left(\frac{(n/2^k)^3}{B\sqrt{M}}\right) + \sum_{i=1}^{k}\left(4^i Q_{\mathcal{D}}\left(\frac{n}{2^i}\right)\right) + c4^k \leq 4^k c'\left(\frac{(n/2^k)^3}{B\sqrt{M}}\right) + c'\sum_{i=1}^{k}\left(4^i\left(\frac{(n/2^i)^3}{B\sqrt{M}}\right)\right) + c4^k$$

$$\leq c'\left(\frac{1}{2^k}\frac{n^3}{B\sqrt{M}}\right) + (c'+1)\left(\frac{n^3}{B\sqrt{M}}\right) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$$

The complexity $Q_{\mathcal{B}}(n)$ can be found in a similar method as described above.

$$Q_{\mathcal{B}}(n) = 4Q_{\mathcal{B}}\left(\frac{n}{2}\right) + 4Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$$

$$
\begin{aligned}
Q_{\mathcal{A}}(n) &= 2Q_{\mathcal{A}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{B}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{C}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c = 2Q_{\mathcal{A}}\left(\frac{n}{2}\right) + 6Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c \\
&= 2\left(2Q_{\mathcal{A}}\left(\frac{n}{2^2}\right) + 6Q_{\mathcal{D}}\left(\frac{n}{2^2}\right) + c\right) + 6Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c \\
&= 2^2 Q_{\mathcal{A}}\left(\frac{n}{2^2}\right) + 2\cdot 6Q_{\mathcal{D}}\left(\frac{n}{2^2}\right) + 6Q_{\mathcal{D}}\left(\frac{n}{2}\right) + c\left(2^1 + 2^0\right) \\
&= 2^k Q_{\mathcal{A}}\left(\frac{n}{2^k}\right) + 6\left(2^{k-1}Q_{\mathcal{D}}\left(\frac{n}{2^k}\right) + \cdots + Q_{\mathcal{D}}\left(\frac{n}{2}\right)\right) + c\left(2^{k-1} + 2^{k-2} + \cdots + 2^0\right) \\
&\leq 2^k c'\left(\frac{(n/2^k)^3}{B\sqrt{M}}\right) + 6\left(2^{k-1}Q_{\mathcal{D}}\left(\frac{n}{2^k}\right) + \cdots + Q_{\mathcal{D}}\left(\frac{n}{2}\right)\right) + c2^k \\
&\leq c'\left(\frac{1}{4^k}\frac{n^3}{B\sqrt{M}}\right) + 6\sum_{i=1}^{k}\left(2^{i-1}Q_{\mathcal{D}}\left(\frac{n}{2^i}\right)\right) + c2^k \leq c'\left(\frac{1}{4^k}\frac{n^3}{B\sqrt{M}}\right) + 6c'\sum_{i=1}^{k}\left(2^{i-1}\left(\frac{(n/2^i)^3}{B\sqrt{M}}\right)\right) + c2^k \\
&= c'\left(\frac{1}{4^k}\frac{n^3}{B\sqrt{M}}\right) + 3c'\left(\frac{n^3}{B\sqrt{M}}\sum_{i=1}^{k}\frac{1}{4^i}\right) + c2^k \\
&= \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + n\right)
\end{aligned}
$$

E.4 Protein accordion folding

The protein accordion folding (PF) is described in Section A.6. Figures A.17 and A.18 show an in-place divide-and-conquer algorithm. The $\mathcal{D}$ function in the algorithm is matrix-multiplication-like and dominates the total work complexity. Hence, by simply modifying the $\mathcal{D}$ function to use extra space, we end up with not-in-place PF algorithm.

> **Lemma 8 (In-place PF).** *The in-place PF, assuming a tall cache, has a complexity of $T_1(n) = \Theta\left(n^3\right)$, $Q_1(n) = \Theta\left(n^3/(B\sqrt{M}) + n^2/B + n\right)$, $T_\infty(n) = \Theta\left(n\log n\right)$, and $S_p(n) = \Theta\left(n^2\right)$.*

Proof. The recurrences are as follows. For $f \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, let $Q_f(n)$ and $T_f(n)$ denote the serial cache complexity and span of f_{PF} on a matrix of size $n \times n$. Then

$$
\begin{aligned}
Q_{\mathcal{A}}(n) = Q_{\mathcal{B}}(n) = Q_{\mathcal{C}}(n) = Q_{\mathcal{D}}(n) &= \mathcal{O}\left(\frac{n^2}{B} + n\right) && \text{if } n^2 \leq \gamma M, \\
Q_{\mathcal{A}}(n) &= 2Q_{\mathcal{A}}\left(\frac{n}{2}\right) + Q_{\mathcal{B}}\left(\frac{n}{2}\right) + Q_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_A M. \\
Q_{\mathcal{B}}(n) &= 4Q_{\mathcal{B}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_B M. \\
Q_{\mathcal{C}}(n) &= 4Q_{\mathcal{C}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_C M. \\
Q_{\mathcal{D}}(n) &= 8Q_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta(1) && \text{if } n^2 > \gamma_D M.
\end{aligned}
$$

$$T_{\mathcal{A}}(n) = T_{\mathcal{B}}(n) = T_{\mathcal{C}}(n) = T_{\mathcal{D}}(n) = \mathcal{O}\,(1) \qquad\qquad \text{if } n = 1,$$

$$T_{\mathcal{A}}(n) = 2T_{\mathcal{A}}\left(\frac{n}{2}\right) + T_{\mathcal{B}}\left(\frac{n}{2}\right) + T_{\mathcal{C}}\left(\frac{n}{2}\right) + \Theta\,(1) \qquad\qquad \text{if } n > 1.$$

$$T_{\mathcal{B}}(n) = T_{\mathcal{B}}\left(\frac{n}{2}\right) + T_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta\,(1) \qquad\qquad \text{if } n > 1.$$

$$T_{\mathcal{C}}(n) = 2\max\{T_{\mathcal{C}}\left(\frac{n}{2}\right), T_{\mathcal{D}}\left(\frac{n}{2}\right)\} + \Theta\,(1) \qquad\qquad \text{if } n > 1.$$

$$T_{\mathcal{D}}(n) = 2T_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta\,(1) \qquad\qquad \text{if } n > 1.$$

where, γ, γ_A, γ_B, γ_C and γ_D are suitable constants, $Q_1(n) = Q_{\mathcal{A}}(n)$, and $T_\infty(n) = T_{\mathcal{A}}(n)$. Solving, we have the lemma. □

Lemma 9 (Not-in-place PF). *The not-in-place PF, assuming a tall cache, has a complexity of $T_1(n) = \Theta\,(n^3)$, $Q_1(n) = \Theta\left(n^3/(B\sqrt{M}) + n^2/B + n\right)$, $T_\infty(n) = \Theta\,(n)$, and $S_p(n) = \Theta\left(\min\{p^{1/3}, 2^k\}n^2\right)$.*

Proof. The recurrences from Lemma 8 will be the same except the following.

$$T_{\mathcal{D}}(n) = T_{\mathcal{D}}\left(\frac{n}{2}\right) + \Theta\,(\log n) \qquad\qquad \text{if } n > 1.$$

Solving the recurrences, the lemma follows. □

The recurrences for the hybrid algorithm are:

$$Q_{\mathcal{A}}(m) = Q_{\mathcal{B}}(m) = Q_{\mathcal{C}}(m) = Q_{\mathcal{D}}(m) = \mathcal{O}\left(\frac{(n/2^k)^3}{B\sqrt{M}} + \frac{(n/2^k)^2}{B} + (n/2^k)\right) \qquad \text{if } m = \frac{n}{2^k},$$

$$Q_{\mathcal{A}}(m) = 2Q_{\mathcal{A}}\left(\frac{m}{2}\right) + Q_{\mathcal{B}}\left(\frac{m}{2}\right) + Q_{\mathcal{C}}\left(\frac{m}{2}\right) + \Theta\,(1) \qquad\qquad \text{if } m > \frac{n}{2^k}.$$

$$Q_{\mathcal{B}}(m) = 4Q_{\mathcal{B}}\left(\frac{m}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{m}{2}\right) + \Theta\,(1) \qquad\qquad \text{if } m > \frac{n}{2^k}.$$

$$Q_{\mathcal{C}}(m) = 4Q_{\mathcal{C}}\left(\frac{m}{2}\right) + 2Q_{\mathcal{D}}\left(\frac{m}{2}\right) + \Theta\,(1) \qquad\qquad \text{if } m > \frac{n}{2^k}.$$

$$Q_{\mathcal{D}}(m) = 8Q_{\mathcal{D}}\left(\frac{m}{2}\right) + \Theta\,(1) \qquad\qquad \text{if } m > \frac{n}{2^k}.$$

$$T_{\mathcal{A}}(m) = \Theta\left((n/2^k)\log^2(n/2^k)\right) \qquad\qquad \text{if } m = n/2^k,$$

$$T_{\mathcal{B}}(m) = T_{\mathcal{C}}(m) = \Theta\left((n/2^k)\log(n/2^k)\right) \qquad\qquad \text{if } m = n/2^k,$$

$$T_{\mathcal{D}}(m) = \Theta\left(n/2^k\right) \qquad\qquad \text{if } m = n/2^k,$$

$$T_{\mathcal{A}}(m) = 2\left(T_{\mathcal{A}}\left(\frac{m}{2}\right) + \max\{T_{\mathcal{B}}\left(\frac{m}{2}\right), T_{\mathcal{C}}\left(\frac{m}{2}\right)\} + T_{\mathcal{D}}\left(\frac{m}{2}\right)\right) + \Theta\,(1) \qquad \text{if } m > n/2^k.$$

$$T_{\mathcal{B}}(m) = 2\left(T_{\mathcal{B}}\left(\frac{m}{2}\right) + T_{\mathcal{D}}\left(\frac{m}{2}\right)\right) + \Theta\,(\log m) \qquad\qquad \text{if } m > n/2^k.$$

$$T_{\mathcal{C}}(m) = 2\left(T_{\mathcal{C}}\left(\frac{m}{2}\right) + T_{\mathcal{D}}\left(\frac{m}{2}\right)\right) + \Theta\,(\log m) \qquad\qquad \text{if } m > n/2^k.$$

$$T_{\mathcal{D}}(m) = T_{\mathcal{D}}\left(\frac{m}{2}\right) + \Theta\,(\log m) \qquad\qquad \text{if } m > n/2^k.$$

Bibliography

[Bin,] Bingo Sort. `http://xlinux.nist.gov/dads//HTML/bingosort.html`.

[Com,] Comet supercomputing cluster. `http://www.sdsc.edu/support/user_guides/comet.html`.

[Int,] Intel Cilk™ Plus. `http://www.cilkplus.org/`.

[PAP,] Performance Application Programming Interface (PAPI). `http://icl.cs.utk.edu/papi/`.

[Sta,] Stampede supercomputing cluster. `https://www.tacc.utexas.edu/stampede/`.

[STX,] Standard Template Library for Extra Large Data Sets (STXXL). `http://stxxl.sourceforge.net/`.

[Acar et al., 2000] Acar, U. A., Blelloch, G. E., and Blumofe, R. D. (2000). The data locality of work stealing. In Symposium on Parallelism in Algorithms and Architectures, pages 1–12.

[Aga et al., 2015] Aga, S., Krishnamoorthy, S., and Narayanasamy, S. (2015). Cilkspec: optimistic concurrency for cilk. In SC, page 83.

[Agarwal et al., 2003] Agarwal, P. K., Arge, L., Danner, A., and Holland-Minkley, B. (2003). Cache-oblivious data structures for orthogonal range searching. In Symposium on Computational geometry, pages 237–245.

[Aggarwal et al., 1988] Aggarwal, A., Vitter, J., et al. (1988). The input/output complexity of sorting and related problems. Communications of the ACM, pages 31(9):1116–1127.

[Agrawal et al., 2010] Agrawal, K., Leiserson, C. E., and Sukha, J. (2010). Executing task graphs using work-stealing. In Parallel and Distributed Processing Symposium, pages 1–12.

[Ahmed and Pingali, 2000] Ahmed, N. and Pingali, K. (2000). Automatic generation of block-recursive codes. In Euro-Par, pages 368–378.

[Aho et al., 1974] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). The design andanalysis of computer algorithms. MA: Addison-Welsey.

[Ailon et al., 2011] Ailon, N., Chazelle, B., Clarkson, K. L., Liu, D., Mulzer, W., and Seshadhri, C. (2011). Self-improving algorithms. SIAM Journal on Computing, pages 40(2):350–375.

[Akl, 2014] Akl, S. G. (12:2014). Parallel sorting algorithms. Academic press.

[Anderson and Miller, 1990] Anderson, R. J. and Miller, G. L. (1990). A simple randomized parallel algorithm for list-ranking. Information Processing Letters, pages 33(5):269–273.

[Anderson and Miller, 1991] Anderson, R. J. and Miller, G. L. (1991). Deterministic parallel list ranking. Algorithmica, pages 6(1–6):859–868.

[Arge, 2003] Arge, L. (2003). The buffer tree: A technique for designing batched external data structures. Algorithmica, pages 37(1):1–24.

[Arge et al., 2007] Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., and Ian Munro, J. (2007). An optimal cache-oblivious priority queue and its application to graph algorithms. SIAM Journal on Computing, pages 36(6):1672–1695.

[Arge et al., 2002a] Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., and Munro, J. I. (2002a). Cache-oblivious priority queue and graph algorithm applications. In ACM symposium on Theory of computing, pages 268–276.

[Arge et al., 2002b] Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., and Munro, J. I. (2002b). Cache-oblivious priority queue and graph algorithm applications. In ACM Symposium on Theory of Computing (STOC), pages 268–276.

[Arge et al., 2005] Arge, L., de Berg, M., and Haverkort, H. (2005). Cache-oblivious r-trees. In Symposium on Computational geometry, pages 170–179.

[Arge and Thorup, 2013] Arge, L. and Thorup, M. (2013). Ram-efficient external memory sorting. In ISAAC, pages 491–501.

[Armando et al., 1999] Armando, A., Smaill, A., and Green, I. (1999). Automatic synthesis of recursive programs: The proof-planning paradigm. Automated Software Engineering, pages 6(4):329–356.

[Arora et al., 2001] Arora, N. S., Blumofe, R. D., and Plaxton, C. G. (2001). Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems, pages 34(2):115–144.

[Bae et al., 2013] Bae, H., Mustafa, D., Lee, J.-W., Lin, H., Dave, C., Eigenmann, R., and Midkiff, S. P. (2013). The cetus source-to-source compiler infrastructure: overview and evaluation. International Journal of Parallel Programming, pages 41(6):753–767.

[Bafna and Edwards, 2003] Bafna, V. and Edwards, N. (2003). On de novo interpretation of tandem mass spectra for peptide identification. In International conference on Research in computational molecular biology, pages 9–18.

[Banerjee et al., 1993] Banerjee, U., Eigenmann, R., Nicolau, A., and Padua, D. A. (1993). Automatic program parallelization. Proceedings of the IEEE, pages 81(2):211–243.

[Bayer and McCreight, 1972] Bayer, R. and McCreight, E. (1972). Organization and maintenance of large ordered indexes. Acta Informatica, pages 1:173–189.

[Bellman, 1957] Bellman, R. (1957). Dynamic Programming. Princeton University Press.

[Bender et al., 2002a] Bender, M. A., Cole, R., and Raman, R. (2002a). Exponential structures for efficient cache-oblivious algorithms. In Automata, Languages and Programming, pages 195–207.

[Bender et al., 2000] Bender, M. A., Demaine, E. D., and Farach-Colton, M. (2000). Cache-oblivious b-trees. In Foundations of Computer Science, pages 399–409.

[Bender et al., 2005a] Bender, M. A., Demaine, E. D., and Farach-Colton, M. (2005a). Cache-oblivious b-trees. SIAM Journal on Computing, pages 35(2):341–358.

[Bender et al., 2002b] Bender, M. A., Duan, Z., Iacono, J., and Wu, J. (2002b). A locality-preserving cache-oblivious dynamic dictionary. In ACM-SIAM symposium on Discrete algorithms, pages 29–38.

[Bender et al., 2014] Bender, M. A., Ebrahimi, R., Fineman, J. T., Ghasemiesfeh, G., Johnson, R., and McCauley, S. (2014). Cache-adaptive algorithms. In ACM-SIAM Symposium on Discrete Algorithms, pages 958–971.

[Bender et al., 2006a] Bender, M. A., Farach-Colton, M., and Kuszmaul, B. C. (2006a). Cache-oblivious string b-trees. In ACM symposium on Principles of database systems, pages 233–242.

[Bender et al., 2006b] Bender, M. A., Farach-Colton, M., and Mosteiro, M. A. (2006b). Insertion sort is o(n log n). Theory of Computing Systems, pages 39(3):391–397.

[Bender et al., 2005b] Bender, M. A., Fineman, J. T., Gilbert, S., and Kuszmaul, B. C. (2005b). Concurrent cache-oblivious b-trees. In ACM symposium on Parallelism in algorithms and architectures, pages 228–237.

[Bentley, 1980] Bentley, J. L. (1980). Multidimensional divide-and-conquer. CACM, pages 23(4):214–229.

[Bertsekas and Tsitsiklis, 1989] Bertsekas, D. P. and Tsitsiklis, J. N. (1989). Parallel and distributed computation: numerical methods, volume 23. Prentice hall Englewood Cliffs, NJ.

[Bille and Stckel, 2012] Bille, P. and Stckel, M. (2012). Fast and cache-oblivious dynamic programming with local dependencies. In Language and Automata Theory and Applications, pages 131–142.

[Bischof et al., 2008] Bischof, C., Bucker, M., Gibbon, P., Joubert, G., Lippert, T., Mohr, B., and Peters, F. (2008). Parallel Computing: Architectures, Algorithms, and Applications, volume 15. IOS Press.

[Blelloch et al., 2008] Blelloch, G. E., Chowdhury, R. A., Gibbons, P. B., Ramachandran, V., Chen, S., and Kozuch, M. (2008). Provably good multicore cache performance for divide-and-conquer algorithms. In ACM-SIAM symposium on Discrete algorithms, pages 501–510.

[Blelloch et al., 2011] Blelloch, G. E., Fineman, J. T., Gibbons, P. B., and Simhadri, H. V. (2011). Scheduling irregular parallel computations on hierarchical caches. In Symposium on Parallelism in Algorithms and Architectures, pages 355–366.

[Blelloch and Gibbons, 2004] Blelloch, G. E. and Gibbons, P. B. (2004). Effectively sharing a cache among threads. In ACM symposium on Parallelism in algorithms and architectures, pages 235–244.

[Blelloch et al., 1995] Blelloch, G. E., Gibbons, P. B., and Matias, Y. (1995). Provably efficient scheduling for languages with fine-grained parallelism. In ACM symposium on Parallel algorithms and architectures, pages 1–12.

[Blelloch et al., 1999] Blelloch, G. E., Gibbons, P. B., and Matias, Y. (1999). Provably efficient scheduling for languages with fine-grained parallelism. Journal of the ACM (JACM), pages 46(2):281–321.

[Blelloch and Maggs, 2010] Blelloch, G. E. and Maggs, B. M. (2010). Parallel algorithms. In Algorithms and theory of computation handbook, pages 25–25.

[Blumofe, 1995] Blumofe, R. D. (1995). Executing multithreaded programs efficiently. PhD thesis, Massachusetts Institute of Technology.

[Blumofe et al., 1996a] Blumofe, R. D., Frigo, M., Joerg, C. F., Leiserson, C. E., and Randall, K. H. (1996a). An analysis of dag-consistent distributed shared-memory algorithms. In Symposium on Parallelism in Algorithms and Architectures, pages 297–308.

[Blumofe et al., 1996b] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1996b). Cilk: An efficient multithreaded runtime system. Journal of parallel and distributed computing, pages 37(1):55–69.

[Blumofe and Leiserson, 1998] Blumofe, R. D. and Leiserson, C. E. (1998). Space-efficient scheduling of multithreaded computations. SIAM Journal on Computing, pages 27(1):202–229.

[Blumofe and Leiserson, 1999] Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. Journal of the ACM, pages 46(5):720–748.

[Bondhugula et al., 2008] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. ACM SIGPLAN Notices, pages 43(6):101–113.

[Brent, 1974] Brent, R. (1974). The parallel evaluation of general arithmetic expressions. JACM, pages 21:201–206.

[Brodal, 1995] Brodal, G. S. (1995). Fast meldable priority queues. Springer.

[Brodal, 1996] Brodal, G. S. (1996). Worst-case efficient priority queues. In ACM-SIAM symposium on Discrete algorithms, pages 52–58.

[Brodal, 2004] Brodal, G. S. (2004). Cache-oblivious algorithms and data structures. In Algorithm Theory-SWAT 2004, pages 3–13.

[Brodal and Fagerberg, 2002] Brodal, G. S. and Fagerberg, R. (2002). Funnel heap-a cache oblivious priority queue. In Algorithms and Computation, pages 219–228.

[Brodal et al., 2002] Brodal, G. S., Fagerberg, R., and Jacob, R. (2002). Cache oblivious search trees via binary trees of small height. In ACM-SIAM symposium on Discrete algorithms, pages 39–48.

[Brodal et al., 2004] Brodal, G. S., Fagerberg, R., Meyer, U., and Zeh, N. (2004). Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In Algorithm Theory-SWAT 2004, pages 480–492.

[Brodal et al., 2008] Brodal, G. S., Fagerberg, R., and Vinther, K. (2008). Engineering a cache-oblivious sorting algorithm. Journal of Experimental Algorithmics (JEA), 12:2–2.

[Brodal and Katajainen, 1998a] Brodal, G. S. and Katajainen, J. (1998a). Worst-case efficient external-memory priority queues. Springer.

[Brodal and Katajainen, 1998b] Brodal, G. S. and Katajainen, J. (1998b). Worst-case efficient external-memory priority queues. Springer.

[Buluc et al., 2010] Buluc, A., Gilbert, J. R., and Budak, C. (2010). Solving path problems on the GPU. Parallel Computing, 36(5):241–253.

[Campanoni et al., 2012] Campanoni, S., Jones, T., Holloway, G., Reddi, V. J., Wei, G.-Y., and Brooks, D. (2012). Helix: automatic parallelization of irregular programs

for chip multiprocessing. In Proc. International Symposium on Code Generation and Optimization, pages 84–93.

[Chan and Chen, 2010] Chan, T. M. and Chen, E. Y. (2010). Optimal in-place and cache-oblivious algorithms for 3-d convex hulls and 2-d segment intersection. Computational Geometry, pages 43(8):636–646.

[Chapman et al., 2008] Chapman, B., Jost, G., and Van Der Pas, R. (2008). Using OpenMP: portable shared memory parallel programming, volume 10. MIT press.

[Chatterjee et al., 2002] Chatterjee, S., Lebeck, A. R., Patnala, P. K., and Thottethodi, M. (2002). Recursive array layouts and fast matrix multiplication. IEEE Transactions on Parallel and Distributed Systems, pages 13(11):1105–1123.

[Chen et al., 2003] Chen, F.-S., Fu, C.-M., and Huang, C.-L. (2003). Hand gesture recognition using a real-time tracking method and hidden markov models. Image and Vision Computing, pages 21(8):745–758.

[Chen et al., 2014] Chen, M., Mao, S., Zhang, Y., and Leung, V. C. (2014). Big data applications. In Big Data, pages 59–79.

[Cheng et al., 2014] Cheng, J., Grossman, M., and McKercher, T. (2014). Professional CUDA C Programming. John Wiley & Sons.

[Cherng and Ladner, 2005] Cherng, C. and Ladner, R. E. (2005). Cache efficient simple dynamic programming. In International Conference on Analysis of Algorithms DMTCS, page 49:58.

[Chowdhury et al., 2016a] Chowdhury, R., Ganapathi, P., Pradhan, V., Tithi, J. J., and Xiao, Y. (2016a). An efficient cache-oblivious parallel viterbi algorithm. In European Conference on Parallel Processing, pages 574–587.

[Chowdhury et al., 2016b] Chowdhury, R., Ganapathi, P., Tithi, J. J., Bachmeier, C., Kuszmaul, B. C., Leiserson, C. E., Solar-Lezama, A., and Tang, Y. (2016b). Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, page 10.

[Chowdhury, 2007] Chowdhury, R. A. (2007). Algorithms and data structures for cache-efficient computation: theory and experimental evaluation. PhD thesis, University of Texas, Austin.

[Chowdhury and Ramachandran, 2004] Chowdhury, R. A. and Ramachandran, V. (2004). Cache-oblivious shortest paths in graphs using buffer heap. In ACM symposium on Parallelism in algorithms and architectures, pages 245–254.

[Chowdhury and Ramachandran, 2006] Chowdhury, R. A. and Ramachandran, V. (2006). Cache-oblivious dynamic programming. In ACM-SIAM Symposium on Discrete Algorithms, pages 591–600.

[Chowdhury and Ramachandran, 2007] Chowdhury, R. A. and Ramachandran, V. (2007). The cache-oblivious Gaussian elimination paradigm: theoretical framework and experimental evaluation. In Symposium on Parallelism in algorithms and architectures, pages 236–236.

[Chowdhury and Ramachandran, 2008] Chowdhury, R. A. and Ramachandran, V. (2008). Cache-efficient dynamic programming algorithms for multicores. In Symposium on Parallelism in Algorithms and Architectures, pages 207–216.

[Chowdhury and Ramachandran, 2010] Chowdhury, R. A. and Ramachandran, V. (2010). The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. Theory of Computing Systems, pages 47(4):878–919.

[Chowdhury et al., 2013] Chowdhury, R. A., Ramachandran, V., Silvestri, F., and Blakeley, B. (2013). Oblivious algorithms for multicores and networks of processors. Journal of Parallel and Distributed Computing, pages 73(7):911–925.

[Chowdhury et al., 2010] Chowdhury, R. A., Silvestri, F., Blakeley, B., and Ramachandran, V. (2010). Oblivious algorithms for multicores and networks of processors. In Parallel and Distributed Processing Symposium.

[Cole, 1988] Cole, R. (1988). Parallel merge sort. SIAM Journal on Computing, pages 17(4):770–785.

[Collins, 1992] Collins, O. M. (1992). The subtleties and intricacies of building a constraint length 15 convolutional decoder. IEEE Transactions on Communications, pages 1810–1819.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). Introduction to algorithms. MIT press.

[Costello et al., 1998] Costello, D. J., Hagenauer, J., Imai, H., and Wicker, S. B. (1998). Applications of error-control coding. IEEE Transactions on Information Theory, pages 44(6):2531–2560.

[Cutting et al., 1992] Cutting, D., Kupiec, J., Pedersen, J., and Sibun, P. (1992). A practical part-of-speech tagger. In Conference on Applied natural language processing, pages 133–140. Association for Computational Linguistics.

[D'Alberto and Nicolau, 2007] D'Alberto, P. and Nicolau, A. (2007). R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. Algorithmica, 47(2):203–213.

[Darte et al., 1997] Darte, A., Silber, G.-A., and Vivien, F. (1997). Combining retiming and scheduling techniques for loop parallelization and loop tiling. Parallel Processing Letters, pages 7(4):379–392.

[Demaine, 2002] Demaine, E. D. (2002). Cache-oblivious algorithms and data structures. Lecture Notes from the EEF Summer School on Massive Data Sets, pages 8(4):1–249.

[Diament and Ferencz, 1999] Diament, B. and Ferencz, A. (1999). Comparison of parallel APSP algorithms.

[Djidjev et al., 2014] Djidjev, H., Thulasidasan, S., Chapuis, G., Andonov, R., and Lavenier, D. (2014). Efficient multi-gpu computation of all-pairs shortest paths. In Parallel and Distributed Processing Symposium, pages 360–369.

[Driscoll et al., 1988] Driscoll, J. R., Gabow, H. N., Shrairman, R., and Tarjan, R. E. (1988). Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. Communications of the ACM, pages 31(11):1343–1354.

[Du et al., 2013] Du, J., Yu, C., Sun, J., Sun, C., Tang, S., and Yin, Y. (2013). Easyhps: A multilevel hybrid parallel system for dynamic programming. In Parallel and Distributed Processing Symposium Workshop, pages 630–639.

[Duckworth and Lewis, 1998] Duckworth, F. C. and Lewis, A. J. (1998). A fair method for resetting the target in interrupted one-day cricket matches. The Journal of the Operational Research Society, pages 49(3):220–227.

[Durbin et al., 1998] Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. (1998). Biological sequence analysis: probabilistic models of proteins and nucleic acids. Cambridge university press.

[Eager et al., 1989] Eager, D. L., Zahorjan, J., and Lazowska, E. D. (1989). Speedup versus efficiency in parallel systems. IEEE Transactions on Computers, 38(3):408–423.

[Eklov et al., 2011] Eklov, D., Nikoleris, N., Black-Schaffer, D., and Hagersten, E. (2011). Cache pirating: Measuring the curse of the shared cache. In Proc. ICPP, pages 165–175.

[Elmroth et al., 2004] Elmroth, E., Gustavson, F., Jonsson, I., and Kgstrm, B. (2004). Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM review, 46(1):3–45.

[Elmroth and Gustavson, 2000] Elmroth, E. and Gustavson, F. G. (2000). Applying recursion to serial and parallel qr factorization leads to better performance. IBM Journal of Research and Development, 44(4):605–624.

[Estivill-Castro and Wood, 1992] Estivill-Castro, V. and Wood, D. (1992). A survey of adaptive sorting algorithms. ACM Computing Surveys (CSUR), pages 24(4):441–476.

[Fadel et al., 1999] Fadel, R., Jakobsen, K. V., Katajainen, J., and Teuhola, J. (1999). Heaps and heapsort on secondary storage. Theoretical Computer Science, pages 220(2):345–362.

[Feautrier, 1996] Feautrier, P. (1996). Automatic parallelization in the polytope model. In The Data Parallel Programming Model, pages 79–103.

[Feautrier and Lengauer, 2011] Feautrier, P. and Lengauer, C. (2011). The polyhedron model. Encyclopedia of parallel computing, pages 1581–1592.

[Feldman et al., 2002] Feldman, J., Abou-Faycal, I., and Frigo, M. (2002). A fast maximum-likelihood decoder for convolutional codes. IEEE Transactions on Vehicular Technology, pages 371–375.

[Floyd, 1962] Floyd, R. W. (1962). Algorithm 97: shortest path. CACM, page 5(6):345.

[Forney Jr, 1973] Forney Jr, G. D. (1973). The viterbi algorithm. Proc. of the IEEE, pages 61(3):268–278.

[Forney Jr, 2005] Forney Jr, G. D. (2005). The viterbi algorithm: A personal history. arXiv preprint cs/0504020.

[Franzini et al., 1990] Franzini, M., Lee, K.-F., and Waibel, A. (1990). Connectionist viterbi training: A new hybrid method for continuous speech recognition. In International Conference on Acoustics, Speech, and Signal Processing, pages 425–428.

[Fredman et al., 1986] Fredman, M. L., Sedgewick, R., Sleator, D. D., and Tarjan, R. E. (1986). The pairing heap: A new form of self-adjusting heap. Algorithmica, pages 1(1–4):111–129.

[Fredman and Tarjan, 1987] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM (JACM), pages 34(3):596–615.

[Frens and Wise, 1997] Frens, J. D. and Wise, D. S. (1997). Auto-blocking matrix-multiplication or tracking blas3 performance from source code. In <u>ACM SIGPLAN Notices</u>, pages 32(7):206–216.

[Friend, 1956] Friend, E. H. (1956). Sorting on electronic computer systems. <u>JACM</u>, pages 3(3):134–168.

[Frigo and Johnson, 2005] Frigo, M. and Johnson, S. G. (2005). The design and implementation of fftw3. <u>Proceedings of the IEEE</u>, pages 93(2):216–231.

[Frigo et al., 1999] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999). Cache-oblivious algorithms. In <u>Foundations of Computer Science</u>, pages 285–297.

[Frigo et al., 2012] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (2012). Cache-oblivious algorithms. <u>ACM Transactions on Algorithms (TALG)</u>, page 8(1):4.

[Frigo et al., 1998] Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The implementation of the cilk-5 multithreaded language. In <u>ACM Sigplan Notices</u>, pages 33(5):212–223.

[Frigo and Strumpen, 2005] Frigo, M. and Strumpen, V. (2005). Cache oblivious stencil computations. In <u>ICS</u>, pages 361–366.

[Frigo and Strumpen, 2007] Frigo, M. and Strumpen, V. (2007). The memory behavior of cache oblivious stencil computations. <u>The Journal of Supercomputing</u>, pages 39(2):93–112.

[Frigo and Strumpen, 2009] Frigo, M. and Strumpen, V. (2009). The cache complexity of multithreaded cache oblivious algorithms. <u>Theory of Computing Systems</u>, pages 45(2):203–233.

[Galil and Giancarlo, 1989] Galil, Z. and Giancarlo, R. (1989). Speeding up dynamic programming with applications to molecular biology. <u>TCS</u>, pages 64(1):107–118.

[Galil and Park, 1994] Galil, Z. and Park, K. (1994). Parallel algorithms for dynamic programming recurrences with more than o(1) dependency. <u>Journal of Parallel and Distributed Computing</u>, pages 21(2):213–222.

[Garey and Johnson, 2002] Garey, M. R. and Johnson, D. S. (2002). <u>Computers and intractability</u>.

[Gensler, 2010] Gensler, H. J. (2010). <u>Introduction to logic</u>. Routledge.

[Gibbons and Rytter, 1989] Gibbons, A. and Rytter, W. (1989). <u>Efficient parallel algorithms</u>. Cambridge University Press.

[Giegerich et al., 2004] Giegerich, R., Meyer, C., and Steffen, P. (2004). A discipline of dynamic programming over sequence data. <u>Science of Computer Programming</u>, pages 51(3):215–263.

[Giegerich and Sauthoff, 2011] Giegerich, R. and Sauthoff, G. (2011). Yield grammar analysis in the bellman's gap compiler. In <u>Workshop on language descriptions, tools and applications</u>, page 7.

[Gilhousen et al., 1991] Gilhousen, K. S., Padovani, R., Viterbi, A. J., Weaver Jr., L. A., and Wheatley III, C. E. (1991). On the capacity of a cellular cdma system. <u>IEEE Transactions on Vehicular Technology</u>, pages 303–312.

[Golub and Van Loan, 2012] Golub, G. H. and Van Loan, C. F. (2012). <u>Matrix computations</u>, volume 3. JHU Press.

[Gotlieb, 1963] Gotlieb, C. (1963). Sorting on computers. CACM, pages 6(5):194–201.

[Goumas et al., 2001] Goumas, G., Sotiropoulos, A., and Koziris, N. (2001). Minimizing completion time for loop tiling with computation and communication overlapping. In Parallel and Distributed Processing Symposium, pages 10–pp.

[Graham, 1969] Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. SIAM journal on Applied Mathematics, pages 17(2):416–429.

[Grosser et al., 2012] Grosser, T., Groesslinger, A., and Lengauer, C. (2012). Pollyper-forming polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters, 22(04).

[Gusfield, 1997] Gusfield, D. (1997). Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press.

[Gustavson, 1997] Gustavson, F. G. (1997). Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM Journal of Research and Development, 41(6):737–755.

[Habermann, 1972] Habermann, N. (1972). Parallel neighbor-sort (or the glory of the induction principle). CMU Technical Report.

[Han, 2002] Han, Y. (2002). Deterministic sorting in o(n log log n) time and linear space. In Annual ACM symposium on Theory of computing, pages 602–608.

[Hardy and Wright, 1979] Hardy, G. H. and Wright, E. M. (1979). An introduction to the theory of numbers. Oxford University Press.

[Harish and Narayanan, 2007] Harish, P. and Narayanan, P. (2007). Accelerating large graph algorithms on the gpu using cuda. In International Conference on High-Performance Computing, pages 197–208.

[Hasan et al., 2010] Hasan, M., Moosa, T. M., and Rahman, M. S. (2010). Cache oblivious algorithms for the rmq and the rmsq problems. Mathematics in Computer Science, pages 3(4):433–442.

[Hays, 1962] Hays, D. G. (1962). Automatic language-data processing.

[He and Luo, 2008] He, B. and Luo, Q. (2008). Cache-oblivious databases: Limitations and opportunities. ACM Transactions on Database Systems (TODS), page 33(2):8.

[Heller and Jacobs, 1971] Heller, J. and Jacobs, I. (1971). Viterbi decoding for satellite and space communication. IEEE Transactions on Communication Technology, pages 19(5):835–848.

[Henderson et al., 1997] Henderson, J., Salzberg, S., and Fasman, K. H. (1997). Finding genes in dna with a hidden markov model. Journal of Computational Biology, pages 4(2):127–141.

[Hendren and Nicolau, 1990] Hendren, L. J. and Nicolau, A. (1990). Parallelizing programs with recursive data structures. Parallel and Distributed Systems, IEEE Transactions on, pages 1(1):35–47.

[Hirschberg, 1975] Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. CACM, pages 18(6):341–343.

[Hurley, 2014] Hurley, P. (2014). A concise introduction to logic. Cengage Learning.

[Iverson, 1962] Iverson, K. E. (1962). A Programming Language. Wiley.

[Jaja, 1992] Jaja, J. (1992). <u>An introduction to parallel algorithms</u>. addison Wesley.

[Jia-Wei and Kung, 1981] Jia-Wei, H. and Kung, H.-T. (1981). I/o complexity: The red-blue pebble game. In <u>ACM Symposium on Theory of Computing (STOC)</u>, pages 326–333.

[Kand and Willson, 1998] Kand, I. and Willson, A. N. (1998). Low-power viterbi decoder for cdma mobile terminals. <u>IEEE Journal of Solid-State Circuits</u>, pages 473–482.

[Kant, 1985] Kant, E. (1985). Understanding and automating algorithm design. <u>IEEE Transactions on Software Engineering</u>, (11):1361–1374.

[Karlin and Upfal, 1988] Karlin, A. R. and Upfal, E. (1988). Parallel hashing: An efficient implementation of shared memory. <u>Journal of the ACM (JACM)</u>, pages 35(4):876–892.

[Karp and Ramachandran, 1988] Karp, R. M. and Ramachandran, V. (1988). A survey of parallel algorithms for shared-memory machines.

[Kasami, 1965] Kasami, T. (1965). An efficient recognition and syntaxanalysis algorithm for context-free languages. Technical report, DTIC Document.

[Katz and Kider Jr, 2008] Katz, G. J. and Kider Jr, J. T. (2008). All-pairs shortest-paths for large graphs on the gpu. In <u>ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware</u>, pages 47–55.

[Kennedy, 1981] Kennedy, J. O. S. (1981). Applications of dynamic programming to agriculture, forestry and fisheries: Review and prognosis. <u>Review of Marketing and Agricultural Economics</u>, 49(03).

[Kim et al., 2014] Kim, G.-H., Trimi, S., and Chung, J.-H. (2014). Big-data applications in the government sector. <u>Communications of the ACM</u>, pages 57(3):78–85.

[Klein and Manning, 2003] Klein, D. and Manning, C. D. (2003). A parsing: fast exact viterbi parse selection. In <u>Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology</u>, pages 40–47.

[Klonatos et al., 2013] Klonatos, Y., Ntzli, A., Spielmann, A., Koch, C., and Kuncak, V. (2013). Automatic synthesis of out-of-core algorithms. In <u>ACM SIGMOD International Conference on Management of Data</u>, pages 133–144.

[Knuth, 1998] Knuth, D. E. (1998). <u>The art of computer programming: sorting and searching</u>, volume 3. Pearson Education.

[Kobayashi, 1971a] Kobayashi, H. (1971a). Application of probabilistic decoding to digital magnetic recording systems. <u>IBM Journal of Research and Development</u>, pages 15(1):64–74.

[Kobayashi, 1971b] Kobayashi, H. (1971b). Correlative level coding and maximum-likelihood decoding. <u>IEEE Transactions on Information Theory</u>, pages 17(5):586–594.

[Kral, 1962] Kral, V. A. (1962). Senescent forgetfulness: benign and malignant. <u>Canadian Medical Association Journal</u>, page 86(6):257.

[Kumar, 2003] Kumar, P. (2003). Cache oblivious algorithms. In <u>Algorithms for Memory Hierarchies</u>, pages 193–212.

[Kumar and Schwabe, 1996] Kumar, V. and Schwabe, E. J. (1996). Improved algorithms and data structures for solving graph problems in external memory. In <u>Symposium on Parallel and Distributed Processing</u>, pages 169–176.

[Kundu et al., 1988] Kundu, A., He, Y., and Bahl, P. (1988). Recognition of handwritten word: first and second order hidden markov model based approach. In CVPR, pages 457–462.

[Kwok and Ahmad, 1999] Kwok, Y.-K. and Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys (CSUR), pages 31(4):406–471.

[Ladner et al., 2002] Ladner, R. E., Fortna, R., and Nguyen, B.-H. (2002). A comparison of cache aware and cache oblivious static search trees using program instrumentation. In Experimental Algorithmics, pages 78–92.

[LaMarca and Ladner, 1999] LaMarca, A. and Ladner, R. E. (1999). The influence of caches on the performance of sorting. Journal of Algorithms, pages 31(1):66–104.

[Leiserson, 2010] Leiserson, C. E. (2010). The cilk++ concurrency platform. The Journal of Supercomputing, pages 51(3):244–257.

[Levitin, 2011] Levitin, A. (2011). Introduction to the Design & Analysis of Algorithms. Pearson.

[Lew and Mauch, 2006] Lew, A. and Mauch, H. (2006). Dynamic programming: A computational tool, volume 38. Springer.

[Liu and Schmidt, 2004] Liu, W. and Schmidt, B. (2004). A generic parallel pattern-based system for bioinformatics. In Euro-Par, pages 989–996.

[Liu et al., 2007] Liu, W., Schmidt, B., Voss, G., and Muller-Wittig, W. (2007). Streaming algorithms for biological sequence alignment on gpus. IEEE transactions on parallel and distributed systems, 18(9):1270–1281.

[Liu et al., 2006] Liu, W., Schmidt, B., Voss, G., Schroder, A., and Muller-Wittig, W. (2006). Bio-sequence database scanning on a gpu. In IEEE International Parallel & Distributed Processing Symposium, pages 8–pp.

[Lou, 1995] Lou, H.-L. (1995). Implementing the viterbi algorithm. IEEE Signal Processing Magazine, pages 12(5):42–52.

[Luby, 1986] Luby, M. (1986). A simple parallel algorithm for the maximal independent set problem. SIAM journal on computing, pages 15(4):1036–1053.

[Lund and Smith, 2010] Lund, B. and Smith, J. W. (2010). A multi-stage cuda kernel for floyd-warshall. arXiv preprint arXiv:1001.4108.

[Maleki et al., 2014] Maleki, S., Musuvathi, M., and Mytkowicz, T. (2014). Parallelizing dynamic programming through rank convergence. In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 219–232.

[Manavski and Valle, 2008] Manavski, S. A. and Valle, G. (2008). Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. BMC bioinformatics, 9(2):1.

[Maon et al., 1986] Maon, Y., Schieber, B., and Vishkin, U. (1986). Parallel ear decomposition search (eds) and st-numbering in graphs. In VLSI Algorithms and Architectures, pages 34–45.

[Martello and Toth, 1990] Martello, S. and Toth, P. (1990). Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc.

[Masek and Paterson, 1980] Masek, W. J. and Paterson, M. S. (1980). A faster algorithm computing string edit distances. Journal of Computer and System sciences, pages 20(1):18–31.

[Matsumoto et al., 2011] Matsumoto, K., Nakasato, N., and Sedukhin, S. G. (2011). Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system. In High Performance Computing and Communications (HPCC), pages 145–152.

[Miller and Ramachandran, 1992] Miller, G. L. and Ramachandran, V. (1992). A new graph triconnectivity algorithm and its parallelization. Combinatorica, pages 12(1):53–76.

[Miller and Reif, 1989] Miller, G. L. and Reif, J. H. (1989). Parallel tree contraction–part i: Fundamentals.

[Miller and Reif, 1991] Miller, G. L. and Reif, J. H. (1991). Parallel tree contraction part 2: further applications. SIAM Journal on Computing, pages 20(6):1128–1147.

[Mou and Hudak, 1988] Mou, Z. G. and Hudak, P. (1988). An algebraic model for divide-and-conquer and its parallelism. The Journal of Supercomputing, pages 2(3):257–278.

[Nam and Kwak, 1998] Nam, H. and Kwak, H. (1998). Viterbi decoder for a high definition television. US Patent 5,844,945.

[Navarro and Paredes, 2010] Navarro, G. and Paredes, R. (2010). On sorting, heaps, and minimum spanning trees. Algorithmica, 57(4):57(4):585–620.

[Nishida et al., 2011] Nishida, K., Ito, Y., and Nakano, K. (2011). Accelerating the dynamic programming for the matrix chain product on the gpu. In International Conference on Networking and Computing (ICNC), pages 320–326.

[Nishida et al., 2012] Nishida, K., Nakano, K., and Ito, Y. (2012). Accelerating the dynamic programming for the optimal polygon triangulation on the gpu. In International Conference on Algorithms and Architectures for Parallel Processing, pages 1–15.

[Olsen and Skov, 2002] Olsen, J. H. and Skov, S. C. (2002). Cache-oblivious algorithms in practice. Master's thesis, University of Copenhagen, Copenhagen, Denmark.

[Omura, 1969] Omura, J. K. (1969). On the viterbi decoding algorithm. IEEE Transactions on Information Theory, pages 177–179.

[Panda et al., 1999] Panda, P. R., Nakamura, H., Dutt, N. D., and Nicolau, A. (1999). Augmenting loop tiling with data alignment for improved cache performance. Computers, IEEE Transactions on, pages 48(2):142–149.

[Park et al., 2004] Park, J.-S., Penner, M., and Prasanna, V. K. (2004). Optimizing graph algorithms for improved cache performance. IEEE Transactions on Parallel and Distributed Systems, 15(9):769–782.

[Patrascu and Thorup, 2014] Patrascu, M. and Thorup, M. (2014). Dynamic integer sets with optimal rank, select, and predecessor search. In FOCS, pages 166–175.

[Plus, 2016] Plus, I. C. (2016). Intel Cilk Plus. https://software.intel.com/en-us/intel-cilk-plus.

[Pouchet et al., 2010] Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., and Sadayappan, P. (2010). Combined iterative and model-driven optimization in an automatic parallelization framework. In SC, pages 1–11.

[Prokop, 1999] Prokop, H. (1999). Cache-oblivious algorithms. Masters Thesis: Massachusetts Institute of Technology.

[Pu et al., 2011] Pu, Y., Bodik, R., and Srivastava, S. (2011). Synthesis of first-order dynamic programming algorithms. In ACM SIGPLAN Notices, pages 46(10):83–98.

[Rabiner, 1989] Rabiner, L. (1989). A tutorial on hidden markov models and selected applications in speech recognition. Proc. of the IEEE, pages 77(2):257–286.

[Rahman et al., 2001] Rahman, N., Cole, R., and Raman, R. (2001). Optimised predecessor data structures for internal memory. In International Workshop on Algorithm Engineering, pages 67–78.

[Rajasekaran and Sen, 1993] Rajasekaran, S. and Sen, S. (1993). Random sampling techniques and parallel algorithms design. Synthesis of Parallel Algorithms, pages 411–451.

[Reid-Miller, 1994] Reid-Miller, M. (1994). List ranking and list scan on the cray c-90. In ACM symposium on Parallel algorithms and architectures, pages 104–113.

[Reif, 1993] Reif, J. H. (1993). Synthesis of parallel algorithms. Morgan Kaufmann Publishers Inc.

[Reitzig, 2012] Reitzig, R. (2012). Automated parallelisation of dynamic programming recursions.

[Renganarayanan et al., 2012] Renganarayanan, L., Kim, D., Strout, M. M., and Rajopadhye, S. (2012). Parameterized loop tiling. TOPLAS, page 34(1):3.

[Rizk and Lavenier, 2009] Rizk, G. and Lavenier, D. (2009). Gpu accelerated rna folding algorithm. In International Conference on Computational Science, pages 1004–1013.

[Robichek et al., 1971] Robichek, A. A., Elton, E. J., and Gruber, M. J. (1971). Dynamic programming applications in finance. The Journal of Finance, pages 26(2):473–506.

[Romer, 2002] Romer, D. (2002). It's fourth down and what does the bellman equation say? a dynamic programming analysis of football strategy. Technical report, National Bureau of Economic Research.

[Rust, 1996] Rust, J. (1996). Numerical dynamic programming in economics. Handbook of computational economics, pages 1:619–729.

[Sakoe and Chiba, 1978] Sakoe, H. and Chiba, S. (1978). Dynamic programming algorithm optimization for spoken word recognition. Transactions on Acoustics, Speech and Signal Processing, pages 26(1):43–49.

[Sanders, 2003] Sanders, P. (2003). Memory hierarchiesmodels and lower bounds. In Algorithms for memory hierarchies, pages 1–13.

[Sarkar and Megiddo, 2000] Sarkar, V. and Megiddo, N. (2000). An analytical model for loop tiling and its solution. In ISPASS, pages 146–153.

[Schmid, 2004] Schmid, H. (2004). Efficient parsing of highly ambiguous context-free grammars with bit vectors. In International conference on Computational Linguistics, page 162. Association for Computational Linguistics.

[Shell, 1959] Shell, D. L. (1959). A high-speed sorting procedure. CACM, pages 2(7):30–32.

[Sibeyn, 2004] Sibeyn, J. F. (2004). External matrix multiplication and all-pairs shortest path. Information Processing Letters, 91(2):99–106.

[Simhadri et al., 2014] Simhadri, H. V., Blelloch, G. E., Fineman, J. T., Gibbons, P. B., and Kyrola, A. (2014). Experimental analysis of space-bounded schedulers. In <u>ACM symposium on Parallelism in algorithms and architectures</u>, pages 30–41.

[Skiena, 1998] Skiena, S. S. (1998). <u>The algorithm design manual</u>, volume 1. Springer Science & Business Media.

[Sleator and Tarjan, 1985] Sleator, D. D. and Tarjan, R. E. (1985). Amortized efficiency of list update and paging rules. <u>Communications of the ACM</u>, pages 28(2):202–208.

[Smith, 2007] Smith, D. K. (2007). Dynamic programming and board games: A survey. <u>European Journal of Operational Research</u>, pages 176(3):1299–1318.

[Sniedovich, 2010] Sniedovich, M. (2010). <u>Dynamic Programming: Foundations and Principles</u>. CRC press.

[Solomon and Thulasiraman, 2010] Solomon, S. and Thulasiraman, P. (2010). Performance study of mapping irregular computations on gpus. In <u>International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum</u>, pages 1–8.

[Solomonik et al., 2013] Solomonik, E., Buluc, A., and Demmel, J. (2013). Minimizing communication in all-pairs shortest paths. In <u>Symposium on Parallel & Distributed Processing (IPDPS)</u>, pages 548–559.

[Soong and Huang, 1991] Soong, F. K. and Huang, E.-F. (1991). A tree-trellis based fast search for finding the n-best sentence hypotheses in continuous speech recognition. In <u>International Conference on Acoustics, Speech, and Signal Processing</u>, pages 705–708.

[Steffen et al., 2009] Steffen, P., Giegerich, R., and Giraud, M. (2009). Gpu parallelization of algebraic dynamic programming. In <u>International Conference on Parallel Processing and Applied Mathematics</u>, pages 290–299.

[Steier and Anderson, 2012] Steier, D. M. and Anderson, A. P. (2012). <u>Algorithm Synthesis: A comparative study</u>. Springer Science & Business Media.

[Stone, 1975] Stone, H. S. (1975). Parallel tridiagonal equation solvers. <u>ACM Transactions on Mathematical Software (TOMS)</u>, pages 1(4):289–307.

[Striemer and Akoglu, 2009] Striemer, G. M. and Akoglu, A. (2009). Sequence alignment with gpu: Performance and design challenges. In <u>International Symposium on Parallel & Distributed Processing</u>, pages 1–10.

[Tan et al., 2006] Tan, G., Feng, S., and Sun, N. (2006). Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In <u>SC</u>, page 78.

[Tang et al., 2012] Tang, S., Yu, C., Sun, J., Lee, B.-S., Zhang, T., Xu, Z., and Wu, H. (2012). Easypdp: An efficient parallel dynamic programming runtime system for computational biology. <u>TPDS</u>, pages 23(5):862–872.

[Tang et al., 2011] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. (2011). The pochoir stencil compiler. In <u>Symposium on Parallelism in Algorithms and Architectures</u>, pages 117–128.

[Tang et al., 2014] Tang, Y., You, R., Kan, H., Tithi, J. J., Ganapathi, P., and Chowdhury, R. A. (2014). Improving parallelism of recursive stencil computations without sacrificing cache performance. In <u>Proceedings of the Second Workshop on Optimizing Stencil Computations</u>, pages 1–7.

[Tang et al., 2015] Tang, Y., You, R., Kan, H., Tithi, J. J., Ganapathi, P., and Chowdhury, R. A. (2015). Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In Principles and Practice of Parallel Programming, pages 50(8):205–214.

[Tarjan and Vishkin, 1985] Tarjan, R. E. and Vishkin, U. (1985). An efficient parallel biconnectivity algorithm. SIAM Journal on Computing, pages 14(4):862–874.

[Taylor and Black, 1998] Taylor, P. and Black, A. W. (1998). Assigning phrase breaks from part-of-speech sequences. Computer Speech & Language, pages 12(2):99–117.

[Tiskin, 2004] Tiskin, A. (2004). Synchronisation-efficient parallel all-pairs shortest paths computation. In Work in progress.

[Tithi et al., 2015] Tithi, J. J., Ganapathi, P., Talati, A., Aggarwal, S., and Chowdhury, R. (2015). High-performance energy-efficient recursive dynamic programming for bioinformatics using matrix-multiplication-like flexible kernels. Parallel and Distributed Processing Symposium.

[Treibig et al., 2010] Treibig, J., Hager, G., and Wellein, G. (2010). Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In Proc. ICPPW, pages 207–216.

[Tzenakis et al., 2013] Tzenakis, G., Papatriantafyllou, A., Vandierendonck, H., Pratikakis, P., and Nikolopoulos, D. S. (2013). Bddt: Block-level dynamic dependence analysis for task-based parallelism. In Advanced Parallel Processing Technologies, pages 17–31.

[Ullman and Yannakakis, 1990] Ullman, J. D. and Yannakakis, M. (1990). The input/output complexity of transitive closure. In ACM SIGMOD Record, volume 19, pages 44–53.

[Ullman and Yannakakis, 1991] Ullman, J. D. and Yannakakis, M. (1991). The input/output complexity of transitive closure. Annals of Mathematics and Artificial Intelligence, 3(2-4):331–360.

[Ungerboeck, 1982] Ungerboeck, G. (1982). Channel coding with multilevel/phase signals. IEEE Transactions on Information Theory, pages 28(1):55–67.

[Venkataraman et al., 2003] Venkataraman, G., Sahni, S., and Mukhopadhyaya, S. (2003). A blocked all-pairs shortest-paths algorithm. Journal of Experimental Algorithmics (JEA), 8:2–2.

[Vishkin, 1984] Vishkin, U. (1984). A parallel-design distributed-implementation (pddi) general-purpose computer. Theoretical Computer Science, pages 32(1):157–172.

[Viterbi, 1967] Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Transactions on Information Theory, pages 13(2):260–269.

[Viterbi, 1971] Viterbi, A. J. (1971). Convolutional codes and their performance in communication systems. IEEE Transactions on Communication Technology, pages 19(5):751–772.

[Volkov and Demmel, 2008] Volkov, V. and Demmel, J. (2008). Lu, qr and cholesky factorizations using vector capabilities of gpus. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May, pages 2008–49.

[Vuillemin, 1978] Vuillemin, J. (1978). A data structure for manipulating priority queues. Communications of the ACM, pages 21(4):309–315.

[Wagner and Fischer, 1974] Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. Journal of the ACM (JACM), pages 21(1):168–173.

[Warshall, 1962] Warshall, S. (1962). A theorem on boolean matrices. JACM, pages 9(1):11–12.

[Waterman et al., 1995] Waterman, M. S. et al. (1995). Introduction to computational biology: maps, sequences and genomes. Chapman & Hall Ltd.

[Williams, 1964] Williams, J. W. J. (1964). Heapsort. Communications of the ACM, pages 7(6):347–348.

[Wolf, 1992] Wolf, M. E. (1992). Improving locality and parallelism in nested loops. PhD thesis, Citeseer.

[Wu et al., 2012] Wu, C.-C., Wei, K.-C., and Lin, T.-H. (2012). Optimizing dynamic programming on graphics processing units via data reuse and data prefetch with inter-block barrier synchronization. In International Conference on Parallel and Distributed Systems, pages 45–52.

[Wyllie, 1979] Wyllie, J. C. (1979). The complexity of parallel computations. Technical report, Cornell University.

[Xiao et al., 2009] Xiao, S., Aji, A. M., and Feng, W.-c. (2009). On the robust mapping of dynamic programming onto a graphics processing unit. In International Conference on Parallel and Distributed Systems, pages 26–33.

[Yoon et al., 2005] Yoon, S.-E., Lindstrom, P., Pascucci, V., and Manocha, D. (2005). Cache-oblivious mesh layouts. In ACM Transactions on Graphics (TOG), pages 24(3):886–893.

[Yotov et al., 2007] Yotov, K., Roeder, T., Pingali, K., Gunnels, J., and Gustavson, F. (2007). An experimental comparison of cache-oblivious and cache-conscious programs. In ACM symposium on Parallel algorithms and architectures, pages 93–104.

[Younger, 1967] Younger, D. H. (1967). Recognition and parsing of context-free languages in time n 3. Information and control, pages 10(2):189–208.