The Dissertation Committee for Rezaul Alam Chowdhury
certifies that this is the approved version of the following dissertation:

Algorithms and Data Structures for Cache-efficient Computation: Theory and Experimental Evaluation

Committee:

Vijaya Ramachandran, Supervisor

Matteo Frigo

Adam Klivans

Keshav Pingali

Greg Plaxton

Tandy Warnow

Algorithms and Data Structures for Cache-efficient Computation: Theory and Experimental Evaluation

by

Rezaul Alam Chowdhury, B.Sc.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2007

For

My parents

(Azam Chowdhury & Sajeda Kohinoor)

and

My sisters

(Tanzina Chowdhury & Tasnuva Chowdhury)

Acknowledgments

First and foremost, I would like to thank my advisor Vijaya Ramachandran. This thesis would not have been possible without her active support, guidance and encouragement.

I would like to express my thanks to the members of my dissertation committee for their valuable comments and suggestions, and also for making my defence a stress-free experience. Special thanks to Matteo Frigo and Keshav Pingali for many useful discussions. I wish to thank Lingling Tong, Hai-Son Le, David Lan Roche and Mo Chen for helping me with my experiments while they worked on their undergraduate honors theses at UT.

I thank the Department of Computer Sciences at UT Austin for supporting me with an MCD graduate fellowship. Thanks also go to NSF as my research was also supported by NSF Grants CCR-9988160 and CCF-0514876, and NSF CISE Research Infrastructure Grant EIA-0303609.

I am indebted to my undergraduate advisor Mohammad Kaykobad. He guided me into the world of research. He believed in me even when I did not believe in myself. The time I spent with him and Suman Nath during my undergraduate years is among the most precious in my life. I cannot thank them enough.

I do not know how to express my gratitude to my parents and my sisters. They had to go through a very difficult time during my absence. Still their support for me has been unconditional and unwavering. They always cared for me, and never complained when I failed to take care of them. I am also indebted to my grandmother, uncle, aunts and my cousins for always being there for me. I could not have completed my Ph.D. without their active support.

I must also thank my former roommate Uttiya Chowdhury. He was no less than an elder brother to me. In times of need I could always rely on him (and still can). I also thank Qumrul Ahsan and my friend Peter Djeu for extending their helping hands whenever I needed one. Thanks to my officemate Thomas Wahl for many interesting discussions.

Finally, I would like to express my gratitude to a friend who lived so far away, and yet cared so much! This caring and sharing helped me get through some of the most difficult times of my life, and kept me sane enough to earn this Ph.D.!

REZAUL ALAM CHOWDHURY

The University of Texas at Austin
August 2007

Algorithms and Data Structures for Cache-efficient Computation: Theory and Experimental Evaluation

Publication No. ____________________

Rezaul Alam Chowdhury, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Vijaya Ramachandran

The ideal-cache model is an abstraction of the memory hierarchy in modern computers which facilitates the design of algorithms that can use the caches (i.e., memory levels) in the hierarchy efficiently without using the knowledge of cache parameters. In addition to possibly running faster than traditional flat-memory algorithms due to reduced cache-misses, these cache-oblivious algorithms are also system-independent and thus more portable than cache-aware algorithms. These algorithms are useful both in applications that work on massive datasets and in applications that run on small-memory systems such as handheld devices.

The major contribution of this dissertation is a number of new cache-efficient

and cache-oblivious algorithms and data structures for problems in three different domains: graph algorithms, problems in the Gaussian Elimination Paradigm (GEP), and problems with dynamic programming algorithms. Among graph problems we concentrate on shortest path computation, and for the computation-intensive problems in the latter two domains we also present efficient parallelizations of our cache-oblivious algorithms for distributed and shared caches. We perform extensive experimental study of most of our algorithms, and compare them with best known existing algorithms and software.

In the area of graph algorithms and data structures, we introduce the first efficient cache-oblivious priority queue supporting *Decrease-Key* operations, and use it to obtain the first non-trivial cache-oblivious single-source shortest path algorithms for both directed and undirected graphs with general non-negative edge-weights. Our experimental results show that shortest path computation using a light-weight version of this new priority queue is faster than using highly optimized traditional priority queues even when the computation is in-core. We also present several new cache-efficient exact and approximate all-pairs shortest path algorithms for both weighted and unweighted undirected graphs.

The Gaussian Elimination Paradigm (GEP) includes many important practical problems with constructs similar to that in Gaussian elimination without pivoting, e.g., Floyd-Warshall's all-pairs shortest path, LU decomposition without pivoting, matrix multiplication, etc. We present a general cache-oblivious framework for cache-efficient sequential and parallel solution of any problem in GEP. Our experimental results comparing our cache-oblivious algorithms with industrial-strength cache-aware BLAS (i.e., Basic Linear Algebra Subprogram) code suggest that our GEP framework offers an attractive trade-off between efficiency and portability.

In the domain of dynamic programs, we present efficient cache-oblivious sequential and parallel algorithms for a number of important dynamic programs in bioinformatics including optimal pairwise sequence alignment, median of three sequences, and RNA secondary structure prediction with and without (simple) pseudoknots. All our algorithms improve significantly over the cache complexity of earlier algorithms, and either match or improve over their space complexity. We empirically compare most of our algorithms with the best publicly available code written by others, and our experimental results indicate that our algorithms run faster than these software.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mama says they was magic shoes.
They could take me anywhere.
(Forrest Gump)

Massive datasets appear in a wide range of applications including database systems [76, 105], spatial databases and geographic information systems (GIS) [38, 68, 86, 108], computational biology [67, 129, 130], VLSI design [14], physics and geophysics [36, 125], communications [12, 26], computer graphics and virtual reality [49, 108], and meteorology [36]. Efficient processing of these datasets requires a computer with a fast memory large enough to hold the entire input. For fundamental physical reasons, however, memory cannot be fast and large at the same time (see, e.g., [110]). Instead, in modern computers large access latencies of large memories are amortized by organizing the memory in a hierarchy with registers in the lowest level followed by several levels of caches (L1, L2 and possibly L3), RAM, and disk, with the access time and size of each level increasing with its depth, and using block transfers between adjacent levels.

An algorithm that performs well on memory hierarchies typically has the feature that whenever a block is brought into a faster level of memory it contains as much useful data as possible ('spatial locality'), and also that as much useful work as possible is performed on this data before it is written back to a slower level ('temporal locality'). Caching and prefetching heuristics have been developed in order to reduce the number of cache misses[1] on this hierarchy. However, these methods are general-purpose in nature and thus in general, cannot take full advantage of the locality inherent in an algorithm. Therefore, an algorithm must rearrange its memory accesses explicitly in order to maximize its cache performance.

[1]A cache miss refers to the situation where the referenced block does not reside in the current memory level and must be fetched from a higher level.

The presence of caches with larger access latencies deeper in the memory hierarchy motivates the use of cache-efficient algorithms for all input sizes. Another motivation comes from the emergence of a wide variety of handheld devices like mobile phones, PDAs and handheld computers, GPS navigation systems, gaming consoles and media players. Some of these devices (e.g., mobile phones, PDAs etc.) are designed as multi-purpose devices and they run all sorts of applications. These multi-purpose devices now outsell laptop/desktop computers combined [69]. However, since caches occupy valuable chip-area, these devices tend to have very small caches (i.e., DRAM and RAM). Another reason for having small caches is to keep the price of the device low and thus make it affordable for the mass population (e.g., cell phones). Therefore, algorithms running on these devices must be cache-efficient even if the dataset is small.

1.1 The Two-level I/O Model

The *two-level I/O model* [3] is a simple abstraction of the memory hierarchy that consists of a *cache* (or *internal memory*) of size M, and an arbitrarily large *main memory* (or *external memory*) partitioned into blocks of size B. An algorithm is said to have caused a *cache-miss* (or *page fault*) if it references a block that does not reside in the cache and must be fetched from the main memory. The *cache complexity* (or *I/O complexity*) of an algorithm is measured in terms of the number of cache-misses it incurs and thus the number of block transfers or I/O operations it causes. This is a simple model that successfully captures the situation where I/O operations between two levels of the memory hierarchy dominate the running time of the algorithm.

Two basic I/O bounds are known for this model: the number of I/Os needed to read N contiguous data items from external memory is $scan(N) = \Theta\left(1 + \frac{N}{B}\right)$ and that for sorting N data items is $sort(N) = \Theta\left(1 + \frac{N}{B} + \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ [3]. For most realistic values of M, B and N, $scan(N) < sort(N) \ll N$. Further, permuting N elements according to a given permutation takes $\Theta\left(\min(N, sort(N))\right)$ I/Os [3] which is $\Theta\left(sort(N)\right)$ for all practical values of N, M and B. This represents a fundamental difference between the flat memory (RAM) and I/O models, since N elements can be permuted in $\Theta\left(N\right)$ time in the RAM model whereas sorting them requires $\Theta\left(N \log N\right)$ time.

A major disadvantage of the two-level model is that algorithms often crucially depend on the knowledge of the parameters of two particular levels of the memory hierarchy and thus do not adapt well when the parameters change. Thus these algorithms cannot simultaneously adapt to all levels of a multi-level hierarchy, and in order to run efficiently on other machines they must have access to the cache parameters of the new system which are not always easily available. Moreover, modern operating systems typically run several concurrent threads that share the same cache, and hence the entire cache is not always available to any particular application and the size of the available cache can change during runtime without the knowledge of the application. A similar situation arises at the hardware level when multicore processors with shared caches are used.

1.2 The Ideal-cache Model & Cache-oblivious Algorithms

An algorithm is *cache-oblivious* if it contains no variables dependent on hardware parameters, such as cache size M and block transfer size B, that need to be tuned in order to optimize its cache complexity [52]. The *ideal-cache model* [52] is an extension of the two-level I/O model with the additional feature that algorithms remain cache-oblivious. This seemingly simple extension has surprisingly powerful consequences. One consequence is that since the analysis of an algorithm in this model holds for any memory and block size, it holds for any two adjacent levels of a multi-level memory hierarchy [52]. Thus by reasoning about a simple two-level memory model we can, in fact, prove results for an arbitrary multi-level memory hierarchy. Another consequence is that the resulting algorithms are flexible and portable since they do not need to be tuned to cache parameters that are not always easily available.

This model makes the following four assumptions.

1. **Optimal Replacement.** Assumes an *optimal offline cache replacement policy* – the cache block to be accessed furthest in the future is chosen for replacement.

2. **Exactly Two Levels of Memory.** Assumes that there are exactly two memory levels as in the two-level I/O model.

3. **Automatic Replacement.** Assumes that whenever a data item that is not stored in the internal memory is requested, the external memory block containing that item is automatically transferred to internal memory by the OS/hardware, and the algorithm designer need not worry about it while designing the algorithm.

4. **Full Associativity.** Assumes that when a block is fetched from the external memory it can be placed anywhere in the internal memory.

While assumption 1 is practically unrealizable, LRU and FIFO, the cache replacement policies mostly used in practice, allow for a constant factor approximation of the optimal strategy at the cost of only a constant factor wastage of the cache space [113]. Assumption 2 simplifies the model, but can be effectively removed by making several realistic assumptions about the memory hierarchy. Firstly, memory levels are assumed to satisfy the *inclusion property* – level i stores only a subset of the elements stored in level $i+1$, where level 1 is the level nearest to the CPU. Secondly, the size of level $i+1$ cache is assumed to be strictly larger than that of level i cache. While assumption 3 seems reasonable, assumption 4 does not, since in practice, caches are either direct-mapped[2] or have very limited associativity[3] such as 2 or 4, and usually not more than 16. But it has been shown in [52] that assumptions 3 and 4 can be efficiently implemented in software by using LRU cache replacement based on 2-universal hashing.

Cache-oblivious algorithms sometimes require a *tall cache* (i.e., require that cache size, $M = \Omega\left(B^2\right)$, where B is the block size) for cache-efficiency, which is not a severe restriction since most practical caches are tall.

The scanning and sorting bounds ($scan(N)$ and $sort(N)$) for the two-level I/O model (see Section 1.1) continue to hold for the ideal-cache model [52]. However, a tall cache is required for the sorting bound to hold. The optimal $\Theta\left(\min(N, sort(N))\right)$ bound for permuting N elements in the two-level I/O model cannot be achieved in the ideal-cache model [23]. Permutation in this model requires either $\mathcal{O}\left(sort(N)\right)$ or $\mathcal{O}\left(N\right)$ block transfers.

1.3 Scope of the Dissertation and Our Contributions

The central theme of this dissertation is the development of cache-efficient and cache-oblivious algorithms and data structures for the following three problem domains.

(*i*) Graph problems,

(*ii*) GEP (Gaussian elimination paradigm) problems (e.g., path computations over closed semirings, Gaussian elimination without pivoting, etc.), and

(*iii*) Problems with dynamic programming algorithms.

Among graph problems our emphasis is on shortest path problems, and for computationally expensive GEP and dynamic programming problems our goal is to design parallel cache-oblivious algorithms whenever feasible. In addition to producing theoretical results, we perform extensive experimental evaluation of our algorithms against existing algorithms.

1.3.1 Cache-efficient Graph Algorithms & Data Structures

Massive graphs arise in a wide variety of applications involving huge data sets. One example of huge data sets is AT&T's phone-record database with an estimated

[2]In direct-mapped caches each main-memory block can only be placed at a fixed location in the cache.

[3]In a c-way set-associative cache each block can be placed only at a fixed set of c locations in the cache.

growth of 20 terabytes a year [12, 26]. Besides using the data for billing purposes researchers would like to use it for understanding the network usage better and thus enabling the carriers to optimize their operations. For this purpose the data is often viewed as a massive directed multigraph with the telephone numbers as the nodes, and phone calls representing directed edges connecting nodes. The edges are weighted by the time and duration of the corresponding calls.

The hyperlinked landscape of the *World Wide Web* is also represented by a massive digraph with web pages as nodes and hyperlinks as edges. The number of nodes in this graph (known as the *Web Graph*) is in the order of billions at present, and is growing rapidly with time. Diameter, and connected and strongly connected components of this graph represent meaningful entities, and this graph is also useful for searching, browsing and mining the web.

Massive graphs also arise in *Geographic Information Systems* (GIS), and many common GIS problems can be formulated as standard graph problems [12]. The most commonly used GIS package, Arc/Info [9], contains functions for computing DFS, BSF, and minimum spanning trees, and also shortest paths and connected components.

Algorithms that handle graphs too large to fit in internal-memory must exploit the locality of data and computation in order to reduce costly page-faults. Since disks are far too slow compared to RAM, I/O-efficient algorithms must be used in order to ensure that the applications produce results in reasonable time.

In addition to applications that work on massive graphs, new graph applications are now emerging that run on small-memory devices. Examples of such devices include handheld GPS navigation systems and modern portable gaming consoles. Not only do these devices have limited memory (e.g., SONY PSP has only 32 MB RAM), they must also limit power dissipated due to cache misses. Hence, these devices need cache-efficient algorithms even for graphs of moderate size.

Our Results.

In the initial half of the thesis (i.e., Chapters 3 – 5) we consider shortest path problems on graphs – both single-source and all-pairs. These computational problems typically have high degree of spatial locality, but very little temporal locality. Consequently, our algorithms and data structures for these problems extensively use scanning and sorting primitives for exploiting spatial locality. We attempt to solve two major problems encountered in cache-efficient and cache-oblivious shortest path computation: (*i*) lack of cache-oblivious priority queues with *Decrease-Keys*, and (*ii*) unstructured accesses to adjacency lists of the input graph.

We begin with the introduction of the buffer heap (in Chapter 3) – the first cache-oblivious priority queue supporting *Decrease-Key* operations and matching the performance bounds of its cache-aware counterpart. A buffer heap supports

Delete, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}\left(\frac{1}{B}\log_2\frac{N}{M}\right)$ amortized cache-misses each, where N is the number of items in the data structure, B is the block transfer size, and M is the size of the cache. We use this data structure to obtain the first cache-oblivious single-source shortest path algorithms for both directed and undirected graphs with general edge-weights. These two algorithms incur $\mathcal{O}\left(\left(n+\frac{m}{B}\right)\cdot\log_2\frac{n}{B}\right)$ and $\mathcal{O}\left(n+\frac{m}{B}\log_2\frac{n}{M}\right)$ cache-misses, respectively, where n is the number of nodes and m is the number of edges in the graph. Both algorithms match the performance bounds of their best cache-aware counterparts. We also introduce the notion of a 'slim data structure' in which only a very small portion of the data structure can be retained in the cache between data structural operations. We show that buffer heaps in this 'slim' setting can be used to obtain an all-pairs shortest path algorithm with improved cache performance for graphs with arbitrary edge-weights.

In Chapter 4 we present the results of an experimental study on how cache-efficient priority queues improve the performance of some shortest path algorithms. We consider both in-core and out-of-core computations. Our experimental results suggest that shortest path computation with a light-weight version of our cache-oblivious buffer heap is often faster than that with highly optimized traditional flat-memory priority queues even when the computation is in-core.

Next we consider the all-pairs shortest path (APSP) problem on unweighted and bounded-weight undirected graphs (in Chapter 5). We use various techniques to reduce unstructured accesses to adjacency lists, and consequently obtain APSP algorithms with improved cache-miss bounds. We design the first cache-oblivious APSP algorithm for unweighted graphs matching the cache complexity of its cache-aware counterpart. On a graph with n nodes and m edges this algorithm incurs only $\mathcal{O}(n\cdot sort(m))$ cache-misses, where $sort(m)$ is the number of cache-misses incurred while sorting m items. We also present the first cache-efficient approximate APSP algorithms for unweighted graphs. Our exact APSP algorithm for bounded-weight graphs is based on a hierarchical clustering technique, and it is the first non-trivial cache-efficient algorithm for the problem.

1.3.2 The Cache-oblivious Gaussian Elimination Paradigm

We use the term *GEP* or the *Gaussian Elimination Paradigm* to refer to a class of triply nested loops similar to the one in Gaussian elimination without pivoting. Many important practical problems belong to this category including path computations over closed semirings [4] (e.g., Floyd-Warshall's all-pairs shortest path [48], transitive closure [128]), Gaussian elimination and LU decomposition without pivoting [37], and matrix multiplication. The all-pairs shortest path problem arises in a wide range of application areas including network routing, distributed computing and robotics. Gaussian elimination without pivoting is used in the solution of systems of linear equations and LU decomposition of symmetric positive-definite or diagonally dominant real matrices [37]. Matrix multiplication has numerous practical applications and is at the heart of scientific computing [59, 103, 116, 66, 102, 60].

All GEP problems have a high degree of both temporal and spatial locality, and hence algorithms for these problems must exploit both types of locality for efficient execution.

Our Results.

In Chapter 6 we present a general framework for efficient cache-oblivious execution of problems in the Gaussian Elimination Paradigm. We show that several important problems in this class (e.g, Gaussian elimination w/o pivoting, Floyd-Warshall's APSP, square matrix multiplication, etc.) can be solved in-place using our cache-oblivious framework, and further with a modest amount of extra space our framework can solve any GEP instance cache-efficiently. On input $n \times n$ matrices, our framework performs $\mathcal{O}\left(n^3\right)$ work and incurs $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses, where M is the size of the cache and B is the block transfer size. We also present a parallel implementation of our framework that terminates in $\mathcal{O}\left(\frac{n^3}{p} + n\log^2 n\right)$ parallel steps on p processors, and provide scheduling policies for cache-efficient execution of this implementation separately on parallel machines with distributed and shared caches. We discuss potential application of our framework in optimizing compilers as a cache-oblivious tiling technique.

In Chapter 7 we present extensive experimental results on our cache-oblivious framework for GEP, which we believe is one of the first attempts in literature to compare cache-oblivious code with high-performance industrial-strength cache-aware code. We consider both in-core and out-of-core, sequential and parallel implementations of our framework, and compare our in-core sequential algorithms for square matrix multiplication and Gaussian elimination w/o pivoting with finely-tuned cache-aware BLAS (Basic Linear Algebra Subprograms) code. The results indicate that our cache-oblivious framework offers an attractive trade-off between efficiency and portability.

1.3.3 Cache-oblivious Dynamic Programming

Dynamic programming (DP) [17, 114] is a powerful algorithmic technique which when applicable, allows one to solve combinatorial problems over an exponential search space in polynomial time and space. It is useful in a wide variety of application areas including stochastic systems analysis, operations research, combinatorics of discrete structures, biosequence analysis, flow problems, parsing of ambiguous languages etc. [58]. Dynamic programming is extensively used in biosequence analysis, such as in protein homology search, gene structure prediction, motif search, analysis of repetitive genomic elements, RNA secondary structure prediction, interpretation of mass spectrometry data, etc. [67, 46, 16, 129]. In [46], a recent textbook on biological sequence analysis, the authors list 11 applications of dynamic programming in bioinformatics in its introductory chapter with many more in chapters that follow.

Dynamic programming is based on two key ideas [45, 17]: (1) the *principle of optimality* and (2) recursion on the principle of optimality. The principle of optimality states that an optimal solution to a problem contains within it optimal solutions to subproblems. Recursion on the principle of optimality says that while the optimal solution to a subproblem might not be known, it can be determined by applying the principle of optimality on the subsequent subsubproblems recursively. The technique is very similar to the *divide-and-conquer* strategy. Unlike the divide-and-conquer strategy, however, dynamic programming is applicable when the subproblems are not independent, i.e., when the subproblems share subsubproblems. A dynamic programming algorithm avoids recomputing the solution to a subsubproblem every time it is encountered by saving the solution to the subsubproblem in a table the first time it is solved.

Standard implementations of most dynamic programming algorithms take full advantage of the spatial locality of the data since they mostly perform sequential read/write operations. These implementations, however, often fail to exploit the temporal locality inherent in the recursive nature of the solution. Therefore, there is room for significant improvement in the cache usage of these algorithms, and consequently also their running times. Moreover, since these algorithms are often quite expensive in terms of computation, parallel cache-efficient implementations of these algorithms are often desirable.

Our Results.

In Chapter 8 we present a general cache-oblivious dynamic programming framework that gives efficient cache-oblivious sequential and parallel algorithms for a number of important dynamic programming problems in bioinformatics including *optimal pairwise global sequence alignment* and *median of three sequences* (both with affine gap costs), and *RNA secondary structure prediction with simple pseudoknots*. For problems requiring solutions to d-dimensional recurrences ($d = 2$ for pairwise alignment, and $d = 3$ for the median problem; see Chapter 8 for details), our cache-oblivious algorithm performs $\mathcal{O}\left(n^d\right)$ work, uses $\mathcal{O}\left(n^{d-1}\right)$ space, incurs $\mathcal{O}\left(\frac{n^d}{BM^{\frac{1}{d-1}}}\right)$ cache-misses and terminates in $\mathcal{O}\left(\frac{n^d}{p} + dn\right)$ parallel steps, where n is the length of each input sequence, M is the size of the cache, B is the block transfer size, and p is the number of parallel processors. Given an RNA sequence of length n, our algorithm predicts an RNA secondary structure with simple pseudoknots in $\mathcal{O}\left(n^4\right)$ work, $\mathcal{O}\left(n^2\right)$ space, $\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$ cache-misses and $\mathcal{O}\left(\frac{n^4}{p} + n\log^2 n\right)$ parallel steps. We also present cache-oblivious sequential and parallel algorithms for *optimal pairwise alignment with general gap costs*. Our sequential algorithm runs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space, and incurs $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses, while its parallel implementation executes $\mathcal{O}\left(\frac{n^3}{p}+n^{\log_2 3}\right)$ parallel steps. All our algorithms improve significantly over the cache-efficiency of earlier algorithms, while matching the best-known time complexity, and matching or improving the best-known space complexity of the problem.

In Chapter 9 we perform extensive experimental evaluation of our cache-oblivious algorithms for optimal pairwise sequence alignment, the median problem, and RNA secondary structure prediction with simple pseudoknots. For the first two problems we compare our algorithms with the best publicly available code written by others, and conclude that our algorithms run faster than these software. Our parallel algorithms show good speed-up as the number of processors increase.

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows. In Chapter 2 we describe known results and the key issues we will address for each of the three problem domains we consider in this thesis.

In Chapters 3 – 5 we present our results on cache-efficient graph algorithms and data structures. Chapter 3 describes our theoretical results on cache-oblivious priority queues and cache-efficient shortest path computation using our new priority queue data structure. Chapter 4 presents our experimental results on how the cache-efficiency of priority queues affects the performance of some shortest path algorithms. In Chapter 5 we describe several theoretical results on cache-efficient all-pairs shortest path computation.

Chapter 6 presents our theoretical results on the cache-oblivious Gaussian Elimination Paradigm (GEP), followed by Chapter 7 which contains our experimental results on GEP.

In Chapter 8 we present our cache-oblivious dynamic programming results, and in Chapter 9 we include an experimental study on the performance of several algorithms presented in Chapter 8.

Finally, in Chapter 10 we offer some concluding remarks.

Chapter 2

Background

The possession of knowledge does not kill
the sense of wonder and mystery.
There is always more mystery.
(Anais Nin)

In this chapter we put the results in this dissertation in context by providing a survey of major known results on cache-efficient graph algorithms and data structures, the cache-oblivious Gaussian Elimination Paradigm, and cache-oblivious dynamic programming problems. For each topic we also discuss the major issues we address in subsequent chapters.

2.1 Cache-efficient Graph Algorithms and Data Structures

First we briefly describe the major known results on cache-efficient graph algorithms and data structures. We then discuss the key problems encountered in designing cache-efficient shortest path algorithms, some of which are addressed in Chapters 3 and 5.

2.1.1 Basic Notations & Definitions

By $G = (V, E, w)$ we denote a (directed or undirected) graph with vertex set V, edge set E, and a non-negative real-valued weight function w over E. By n and m we denote the size of V and E, respectively. We assume that E is given either as an unordered sequence of edges or as an adjacency list. An unordered sequence of edges can be converted to adjacency list format in $\mathcal{O}\left(sort(m)\right)$ I/Os using a sorting step.

The SSSP Problem. The *single-source shortest path* (SSSP) problem is one of the most fundamental and important combinatorial optimization problems from both a theoretical and a practical point of view. Given a (directed or undirected) graph $G = (V, E, w)$, and a distinguished vertex $s \in V$, the SSSP problem seeks to find a path of minimum total edge-weight from s to every reachable vertex $v \in V$. For unweighted graphs this problem is also called the *breadth-first search* (BFS) problem.

The APSP Problem. Given a (directed or undirected) graph $G = (V, E, w)$, the *all-pairs shortest path* (APSP) problem seeks to find a path of minimum total edge-weight between every pair of vertices in V. The *diameter* of G is the longest shortest distance between any pair of vertices in G. For unweighted graphs the APSP problem is also called the all-pairs breadth-first search (AP-BFS) problem.

Connected Components & Minimum Spanning Forest. Given an undirected graph $G = (V, E, w)$ and the *connected components* (CC) problem asks for an enumeration of maximal subsets of V such that for every pair of vertices $u, v \in V$ there is a path between u and v in G. In the *minimum spanning forest* (MSF) problem the objective is to find a spanning forest of G with a minimum total edge weight.

2.1.2 Known Results

The solution of almost any graph problem involves somehow permuting the n vertices and m edges of the graph, and hence the lower bound on permutation implies that $\mathcal{O}(\min(n, sort(n)))$ (which is $\Theta(sort(n))$ for all practical cases; see Sections 1.1 and 1.2 in Chapter 1) is a general lower bound on the number of I/O operations needed to solve most graph problems. Though in recent years considerable efforts have been devoted to developing efficient graph algorithms for external memory (see [77, 121, 21, 127] for recent surveys), not many of them are known to match the lower bound. We summarize the most important results below. Recall that B is the block transfer size and M is the size of the cache.

Cache-aware Results

The first work on external memory graph algorithms is due to Ullman and Yannakakis [123], where they considered the I/O complexity of the transitive closure problem.

Chiang et al. [31] considered a wide variety of graph problems for several of which they obtained optimal I/O bounds. They developed the first cache-optimal (matching the permutation lower bound) algorithm for list ranking (the problem of sorting the elements in a linked list stored unordered on disk) which is the most fundamental I/O graph problem. Using this algorithm and PRAM techniques $\mathcal{O}(sort(n))$ I/O algorithms can be developed for most problems on trees, such as computing an Euler tour, breadth-first search (BFS), depth-first search (DFS), centroid decomposition, and expression tree evaluation [31]. The best known external DFS algorithm for directed graphs that uses $\mathcal{O}\left(n + \frac{mn}{BM}\right)$ I/Os, is also due to Chiang et al. [31].

Arge [10] developed the *buffer tree technique*, and showed how to use this technique to obtain a priority queue supporting *Insert* and *Delete-Min* operations in $\mathcal{O}\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ amortized I/Os each, where N is the number of operations performed (or the total number of elements inserted into the queue).

Kumar & Schwabe [83] developed graph algorithms based on amortized data structures for binary heaps and tournament trees. While their cache-efficient binary heap supports the same operations in the same I/O bounds as does the priority queue by Arge [10], their cache-efficient tournament tree also supports *Update* (or *Decrease-Key*) operations. However, the tournament tree requires $\mathcal{O}\left(\frac{1}{B}\log_2\frac{N}{B}\right)$ amortized I/Os for each operation. Using cache-efficient tournament trees they developed the first and best known cache-efficient single-source shortest path (SSSP) algorithm for undirected graphs requiring $\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{m}{B}\right)$ I/Os. However, using the technique in [31] for handling visited vertices, undirected SSSP can be solved in $\mathcal{O}\left(n + \frac{mn}{BM} + sort(m)\right)$ I/Os.

Munagala & Ranade [92] gave improved algorithms for connectivity and BFS in undirected graphs requiring $\mathcal{O}\left(sort(m)\cdot\log\log\frac{nB}{m}\right)$ and $\mathcal{O}\left(n + sort(m)\right)$ I/Os respectively. Later Arge et al. [12] extended this approach to compute MSF in $\mathcal{O}\left(sort(m)\cdot\log\log\frac{nB}{m}\right)$ I/Os.

Buchsbaum et al. [25] developed the *buffered repository tree* to obtain the best known external DFS algorithm for directed graphs using $\mathcal{O}\left((n + \frac{m}{B})\cdot\log_2\frac{n}{B} + sort(m)\right)$ I/Os. Using a cache-efficient tournament tree [83] as the priority queue and a buffered repository tree for remembering visited vertices directed SSSP can be solved in $\mathcal{O}\left((n + \frac{m}{B})\cdot\log_2\frac{n}{B} + sort(m)\right)$ which is weaker than the known upper bound for undirected SSSP.

Mehlhorn & Meyer [88] reduced the I/O cost of accessing the adjacency lists during an undirected BFS from $\mathcal{O}(n)$ to $\mathcal{O}\left(\sqrt{\frac{mn}{B}}\right)$ when $m < nB$, and later Meyer & Zeh [89] obtained a slightly weaker result for undirected SSSP on graphs with bounded edge-weights.

Cache-oblivious Results

Arge et al. [11] introduced the first cache-oblivious priority queue supporting *Insert* and *Delete-Min* operations in optimal $\mathcal{O}\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os each, where N is the number of operations performed on the queue. Using this priority queue they solved the list ranking problem cache-obliviously using $\mathcal{O}\left(sort(n)\right)$ I/Os which immediately implies $\mathcal{O}\left(sort(n)\right)$ I/O cache-oblivious algorithms for tree problems such as the Euler tour, BFS, DFS, and centroid decomposition. They presented directed BFS and

DFS algorithms incurring $\mathcal{O}\left((n + \frac{m}{B}) \cdot \log_2 \frac{n}{B} + sort(m)\right)$ I/Os each, undirected BFS requiring $\mathcal{O}\left(n + sort(m)\right)$ I/Os, and undirected MSF algorithm with I/O complexity $\mathcal{O}\left(\min(n + sort(m), sort(m) \cdot \log\log n)\right)$. All their algorithms match the cache complexity of the best known cache-aware algorithms under tall cache assumption. Later Brodal & Fagerberg [22] introduced another cache-oblivious priority queue known as the *Funnel Heap* supporting the same operations in the same amortized bounds as does the priority queue by Arge et al. [11].

2.1.3 Key Issues in Cache-efficient Shortest Path Computation

As pointed out in [77], the key problems encountered in developing cache-efficient shortest path algorithms are: (a) lack of cache-efficient priority queues supporting *Decrease-Key* operations, (b) unstructured indexed accesses to adjacency lists, and (c) remembering visited vertices.

(a) Cache-efficient Priority Queue with Decrease-Keys. Virtually all internal memory SSSP algorithms work by maintaining an upper bound on the shortest distance (a *tentative distance*) to every vertex from the source vertex and visiting the vertices in a one-by-one fashion (or by groups) in non-decreasing order of tentative distances. The next vertex (or group of vertices) to be visited is the one with the smallest tentative distance extracted from the set of unvisited vertices kept in a priority queue Q. After a vertex (or a group of vertices) has been extracted from Q each of its unvisited neighbors is either inserted into Q with a finite tentative distance or gets its tentative distance updated if it already resides in Q. Therefore, in addition to supporting *Insert* and *Delete-Min* operations, Q needs to support efficient *Decrease-Key* operations.

Though the cache-aware *tournament tree* supports *Decrease-Key* operations cache-efficiently, no such cache-oblivious data structure was known prior to our work on cache-oblivious *buffer heap* (see Chapter 3).

(b) Unstructured Accesses to Adjacency Lists. Virtually all external memory graph traversal (BFS, DFS, SSSP) algorithms require $\Theta\left(n + \frac{m}{B}\right)$ block transfers to access the adjacency lists and this is a bottleneck for these algorithms. Though this bound has been improved slightly for undirected graphs with unweighted edges [88] and bounded-weight edges [89], improvement is achieved only for very sparse graphs. However, no such results are known for directed graphs or graphs with general edge-weights.

In Chapters 3 (Section 3.4.3) and 5 we use various techniques to reduce unstructured accesses to adjacency lists for the APSP problem.

(c) Remembering Visited Vertices. Shortest path algorithms need to remember the vertices whose shortest paths from the source have already been determined in order to avoid recomputing the shortest paths to those vertices in future. In undirected graphs this problem can be avoided by using an auxiliary priority queue [83, 77]. In directed graphs keeping track of visited vertices costs $\mathcal{O}\left(n \log n + \frac{m}{B} \log n\right)$ cache-misses using a *buffered repository tree* (BRT) [25, 32]. The BRT structure maintains $\mathcal{O}(m)$ elements under the operations *Insert* and *Extract* which are supported in $\mathcal{O}\left(\frac{1}{B} \log_2 n\right)$ and $\mathcal{O}(\log_2 n)$ amortized cache-misses, respectively. An SSSP algorithm performs n *Extract* and m *Insert* operations on this structure incurring $\mathcal{O}(n \log n)$ and $\mathcal{O}\left(\frac{m}{B} \log_2 n\right)$ block transfers, respectively. The I/O cost of remembering visited vertices is one of the major bottlenecks in shortest path computation in directed graphs.

2.2 The Cache-oblivious Gaussian Elimination Paradigm

We discuss known cache-oblivious algorithms for problems in the Gaussian Elimination Paradigm (GEP). The key issues in designing cache-efficient algorithms for GEP problems are similar to those arising during the design of cache-efficient dynamic programming algorithms and hence are discussed in Section 2.3.

2.2.1 Known Results

A cache-oblivious dynamic programming algorithm for Floyd-Warshall's APSP algorithm is given in [95] (also in [39]). The algorithm runs in $\mathcal{O}\left(n^3\right)$ time and incurs $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses. Experimental results show that on some architectures the algorithm runs up to 10 times faster than the standard Floyd-Warshall algorithm even when the entire input matrix fits into the RAM.

Though Gaussian elimination, LU decomposition and matrix multiplication are not dynamic programming algorithms, these computations have structural similarity to Floyd-Warshall's APSP (see Chapter 6). Known cache-oblivious algorithms for Gaussian elimination for solving systems of linear equations are based on LU decomposition. In [134, 19] cache-oblivious algorithms performing $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O operations are given for LU decomposition without pivoting, while the algorithm in [120] performs LU decomposition with partial pivoting within the same I/O bound. These algorithms use matrix multiplication and solution of triangular linear systems as subroutines.

An $\mathcal{O}(mnp)$ time and $\mathcal{O}\left(m + n + p + \frac{mn+np+mp}{B} + \frac{mnp}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix is given in [52].

Our major contribution in this area is a unified framework that gives efficient cache-oblivious algorithms for all problems above and possibly many others (see Chapter 6) typically matching the best performance bounds for the corresponding problem.

2.3 Cache-oblivious Dynamic Programming

We first give a brief overview of known results on cache-oblivious dynamic programming and then list the key issues one should address in designing these algorithms.

2.3.1 Known Results

In [30] an $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithm based on Valiant's context-free language recognition algorithm [124], is given for simple-DP that includes algorithms for RNA secondary structure prediction [78], matrix chain multiplication, optimal polygon triangulation and optimal binary search tree construction. A similar algorithm for simple-DP is also given in [117], and in [118] the algorithm is extended for cache-efficient execution on a multicore programming model based on IBM Cyclops64.

The cache-oblivious stencil computation technique presented in [54] can be used as a dynamic programming algorithm for computing the length of a longest common subsequence of two sequences of length n each in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space and $\mathcal{O}\left(\frac{n^2}{BM}\right)$ I/Os. This method, however, does not compute the subsequence.

2.3.2 Key Issues

Two of the major issues in developing efficient cache-oblivious dynamic programming algorithms are as follows. Both issues are addressed in Chapters 6 and 8.

(a) Exploiting Both Temporal and Spatial Locality. Standard implementations of dynamic programming algorithms often fully exploit the spatial locality of data since they mostly perform repeated sequential read/write operations on the dynamic programming table. On some architectures sequential scans receive good support from prefetchers. However, scanning the entire table over and over again means that no significant portion of the table is retained in the cache for reuse, i.e, any temporal locality inherent in the computation is ignored. Therefore, significant improvement in the cache usage of these algorithms can be achieved if the temporal locality can be exploited without giving up on the spatial locality.

(b) Exploiting Parallelism with Cache-efficiency. Parallelization is often desirable in order to cope with the high computational cost of dynamic programming algorithms. An open question is how to achieve both parallelism and cache-efficiency simultaneously. One must also take into account that unlike in the sequential setting, caches can now be either distributed or shared, and different approaches might be needed for handling these two types of caches. We discuss this issue in some more detail in the next section.

2.3.3 Caches on Parallel Machines

Symmetric Multiprocessors or SMPs are one of the most common multiprocessor computer architectures in use today. On an SMP two or more identical processors are connected to a single shared cache or main memory. Now-a-days Chip Multiprocessors or CMPs are also becoming commonplace. On a CMP multiple processors or cores are placed on a single chip, and each core is accompanied with its own on-chip private L1 cache. These CMPs which are also known as multicores, also have a large on-chip L2 cache shared among all processors.

On a parallel machine with a shared cache, a cache-miss occurs when a processor reads or writes a data item that is not in the shared cache. In order to reduce such cache-misses algorithms or scheduling policies must be designed in such a way that processors executing in parallel share cache blocks as much as possible. In contrast, on a parallel machine with distributed or private caches, all processors working in parallel should access disjoint sets of cache blocks in order to reduce cache-misses caused by transferring blocks back and forth between private caches. Thus for good cache performance on shared and distributed caches algorithms and schedulers need to employ different techniques for data sharing. The situation becomes even more complicated on CMPs if the goal is to reduce cache misses for both distributed (L1) and shared (L2) caches. In Chapters 6 and 8 we focus on reducing cache-misses on shared and distributed caches separately.

Chapter 3

Cache-oblivious Buffer Heap and its Applications

The distance is nothing;
it is only the first step that is difficult.
(Marie Anne du Deffand)

In this chapter we present the *buffer heap*, a cache-oblivious priority queue that supports *Delete*, *Delete-Min*, and *Decrease-Key* operations in $\mathcal{O}\left(\frac{1}{B}\log_2\frac{N}{M}\right)$ amortized block transfers from main memory, where M and B are the (unknown) cache size and block-size, respectively, and N is the number of elements in the queue. We assume that the *Decrease-Key* operation only verifies that the element does not exist in the priority queue with a smaller key value, and hence it also supports the *Insert* operation in the same amortized bound. The amortized time bound for each operation is $\mathcal{O}(\log N)$.

Using the buffer heap we present cache-oblivious algorithms for undirected and directed single-source shortest path (SSSP) problems for graphs with non-negative real edge-weights. On a graph with n vertices and m edges, our algorithm for the undirected case performs $\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{n}{M}\right)$ block transfers and for the directed case performs $\mathcal{O}\left(\left(n + \frac{m}{B}\right)\cdot\log_2\frac{n}{B}\right)$ block transfers. Running time of both algorithms is $\mathcal{O}\big((m+n)\cdot\log n\big)$.

For both priority queues with *Decrease-Key* operation, and for SSSP problems on general graphs, our results give the first non-trivial cache-oblivious bounds. Our results, though not known to be optimal, provide substantial improvements over known trivial bounds.

We also introduce the notion of a *slim data structure* which captures the situation when only a limited portion of the cache which we call a *slim cache*, is available to the data structure to retain data between data structural operations. We show that a buffer heap automatically adapts to such an environment and supports all operations in $\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2\frac{N}{\lambda}\right)$ amortized block transfers each when the size of the slim cache is λ. We use buffer heaps in this setting to improve the cache complexity of the cache-aware all-pairs shortest path (APSP) problem on weighted undirected graphs.

3.1 Introduction

The single-source shortest path (SSSP) and the all-pairs shortest path (APSP) problems are among the most important combinatorial optimization problems with numerous practical applications (see Chapter 1 for definitions). Under the traditional *von Neumann Model* of computation which assumes a single layer of memory with uniform access cost, the SSSP problem on a directed graph can be solved efficiently in $\mathcal{O}(m + n \log n)$ time by Dijkstra's algorithm [43] implemented using a *Fibonacci heap* [51]. For undirected graphs the problem can also be solved in $\mathcal{O}(m\alpha(m,n) + n\min(\log n, \log\log \rho))$ time [99], where ρ is the ratio of the maximum and the minimum edge-weights in G, and $\alpha(m,n)$ is a certain natural inverse of Ackermann's function that evaluates to a small constant for all practical values of m and n. Faster algorithms exist for special classes of graphs and graphs with restricted edge-weights. Efficient APSP algorithms have also been developed for this model [136].

As explained in Chapter 1, modern computers with deep memory hierarchies differ significantly from the original von Neumann architecture, and demand cache-efficient algorithms.

3.1.1 Cache-aware Shortest Path Algorithms

In recent years there has been considerable research on developing cache-efficient graph algorithms (see [127, 77] for recent surveys). Several cache-efficient SSSP algorithms have been developed [31, 83, 77, 89]. As explained in Section 2.1.3 of Chapter 2, in addition to a mechanism to remember visited vertices, cache-efficient implementations of virtually all SSSP algorithms require cache-efficient priority queues supporting *Decrease-Key* operations.

Major known SSSP results for the two-level I/O model are summarized in Table 3.3 under the caption "Cache-aware Results". Kumar & Schwabe [83] were the first to develop a cache-efficient version of Dijkstra's SSSP algorithm for undirected graphs. They use a tournament tree as a priority queue and perform some extra book-keeping using an auxiliary priority queue in order to handle visited vertices. A cache-efficient tournament tree supports a sequence of k *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}\left(\frac{k}{B}\log_2\frac{n}{M}\right)$ block transfers leading to an SSSP algorithm incurring $\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{n}{M}\right)$ cache-misses. The *phase approach* used in [31] implements a priority queue with *Decrease-Key*s indirectly and results in an undirected SSSP algorithm that beats Kumar & Schwabe's algorithm when $n = \mathcal{O}\left(M\log_2\frac{n}{M}\right)$, i.e., the set of vertices is not too large compared to the size of the cache. In [89] Meyer & Zeh developed another undirected SSSP algorithm that works on graphs with real edge-weights, but its performance depends on ρ, the ratio of the largest and the smallest edge-weights in the graph. This algorithm outperforms Kumar &

Schwabe's algorithm for sparse graphs, i.e., when $m = \mathcal{O}\left(\frac{B}{\log_2 \rho} \cdot n\right)$. This algorithm uses a hierarchical decomposition technique to reduce random accesses to adjacency lists, and a priority queue called the *bucket heap* that is specifically designed for this purpose. The bucket heap supports a sequence of k *Delete*, *Delete-Min* (*Batched-Delete-Min*) and *Decrease-Key* operations in $\mathcal{O}\left(sort(k) + \frac{k}{B}\log_2 \rho\right)$ cache-misses.

For directed graphs the survey paper [127] mentions a cache complexity of $\mathcal{O}\left((n + \frac{m}{B}) \cdot \log_2 \frac{n}{B}\right)$ for SSSP using a tournament tree. Using the phase approach directed SSSP can be solved in $\mathcal{O}\left(n + \frac{mn}{BM}\log_2 \frac{n}{B}\right)$ block transfers [31, 77].

A straight-forward method of computing APSP is to simply run an SSSP algorithm from each of the n vertices of the graph. Arge et al. [13] proposed a cache-aware APSP algorithm for undirected graphs with general non-negative edge-weights that performs $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}}\log n + sort(m)\right)\right)$ block transfers when $m = \mathcal{O}\left(\frac{B}{\log n} \cdot n\right)$. They use a priority queue structure called the *multi-tournament-tree* which is created by bundling together a number of cache-efficient tournament trees. The use of this structure reduces unstructured accesses to adjacency lists at the expense of increasing the cost of each priority queue operation.

3.1.2 Cache-oblivious Shortest Path Algorithms

The cache-oblivious priority queue introduced by Arge et al. [11] and the *funnel heap* introduced by Brodal & Fagerberg [22] support *Insert* and *Delete-Min* in amortized optimal $\mathcal{O}\left(\frac{1}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$ cache-misses, where N is the number of elements in the queue, but they do not support *Decrease-Keys*. Prior to our work no non-trivial cache-oblivious results were known for priority queue with *Decrease-Key*s or for SSSP on graphs. Very recently, however, Allulli et al. [7] obtained a cache-oblivious SSSP algorithm for undirected sparse graphs with bounded edge-weights by extending the cache-aware algorithm in [89] which outperforms our algorithm when $m = \mathcal{O}\left(\frac{B}{\log_2 \rho} \cdot n\right)$ and $\rho = 2^{o(B)}$, where ρ is the ratio of the largest and the smallest edge-weights.

I/O Model	Priority Queue	*Decrease-Key*	*Delete*	*Delete-Min*
Cache-aware	Tournament Tree [83]	$\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$		
Cache-oblivious	Buffer Heap **(our result)** (see also [24])	$\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$		

Table 3.1: Amortized cache complexities for priority queues with *Decrease-Keys*. (N = number of items in the queue)

I/O Model	Slim Priority Queue	*Decrease-Key*	*Delete*	*Delete-Min*
Cache-aware	Slim Tournament Tree $\left[1 \leq \lambda \leq \frac{B}{2}\right]$ (component of multi-tournament-tree [13])	$\mathcal{O}\left(\frac{1}{\lambda}\log_2 N\right)$		
Cache-oblivious	Slim Buffer Heap $[1 \leq \lambda \leq M]$ **(our result)**	$\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda}\right)$		

Table 3.2: Amortized cache complexities for slim priority queues with *Decrease-Keys*. (λ = slim cache size, N = # items)

3.1.3 Our Results

Majority of the results included in this chapter were presented in two conference papers [32, 33].

We introduce the *buffer heap*, the first cache-oblivious priority queue to support *Decrease-Key* operations. Independently of our work a similar data structure was also presented in [24]. The buffer heap matches the cache complexity of the cache-aware tournament tree (see Table 3.1), and we use it to obtain the first cache-oblivious SSSP algorithms for weighted undirected and directed graphs matching the cache performance of their cache-aware counterparts (see Table 3.3). Our cache-miss bounds for SSSP problems are not very impressive for sparse graphs, but they do provide dramatic improvements for moderately dense graphs. For example, for undirected graphs, if $m \geq \frac{nB}{\log_2\left(\frac{n}{B}\right)}$ our algorithm reduces the number of cache-misses by a factor of $\frac{B}{\log_2\left(\frac{n}{B}\right)}$ over the naïve method. For directed graphs, we obtain the same improvement if $m \geq nB$.

We also introduce the notion of a *slim data structure*. This notion captures the scenario where only a limited portion of the cache is available to store data from the data structure; it is assumed, however, that while executing an individual operation of the data structure, the entire cache is available for the computation. We describe and analyze the *slim buffer heap* which is a slim data structure based on the buffer heap (see Table 3.2 for a comparison with the only other similar data structure known), and use it to improve the cache performance of the cache-aware APSP algorithm for undirected graphs with general non-negative edge-weights given in [13] to $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}} + sort(m)\right)\right)$ when $m = \mathcal{O}\left(\frac{nB}{\log^2 n}\right)$ (see Table 3.3). Recall that $sort(m)$ is the cache complexity of sorting m data items. For general values of m our algorithm performs $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}} + \frac{m}{B}\log\frac{m}{B}\right)\right)$ block transfers. We also believe that the notion of a slim data structure is of independent interest.

In this chapter we show that the slim buffer heap can be made oblivious of the slim cache size without sacrificing its performance. In fact, we show that when a regular buffer heap (i.e., a buffer heap which is not restricted to using a slim cache) is run in an environment that limits the amount λ of cache space available to it to store data between data structural operations, it automatically adapts to this new environment and matches the performance bounds of a slim buffer heap with a slim cache of size λ.

Problem	Cache-aware Results	Cache-oblivious Results
Weighted Undirected SSSP	$\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{n}{M}\right)$ [83] $\mathcal{O}\left(n + \frac{mn}{BM} + sort(m)\right)$ [31, 77] $\mathcal{O}\left(\sqrt{\frac{mn}{B}\log_2\rho} + sort(m+n)\log_2\log_2\frac{nB}{m}\right)$ [89]	$\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{n}{M}\right)$ **(our result)** (see also [24])
Weighted Directed SSSP	$\mathcal{O}\left(\left(n + \frac{m}{B}\right)\cdot\log_2\frac{n}{B}\right)$ [127] $\mathcal{O}\left(n + \frac{mn}{BM}\log_2\frac{n}{B}\right)$ [31, 77]	$\mathcal{O}\left(\left(n + \frac{m}{B}\right)\cdot\log_2\frac{n}{B}\right)$ $\left[M = \Omega\left(B^2\right)\right]$ **(our result)**
Weighted Undirected APSP	$\mathcal{O}\left(n\cdot\left(\sqrt{\frac{mn}{B}\log_2 n} + sort(m)\right)\right)$ [13] $\mathcal{O}\left(n\cdot\left(\sqrt{\frac{mn}{B}} + sort(m)\right)\right)$ **(our result)**	$\mathcal{O}\left(n\cdot\left(n + \frac{m}{B}\log_2\frac{n}{M}\right)\right)$ **(derived from our undirected SSSP result above)**

Table 3.3: Cache complexities for SSSP and APSP problems on weighted graphs. ($n = |V|$, $m = |E|$)

3.1.4 Organization of the Chapter

In Section 3.2, we define a slim data structure. In Section 3.3, we present the cache-oblivious buffer heap as a slim data structure, prove the correctness of its implementation and analyze its cache and time complexities. In Section 3.4, we discuss three major applications of buffer heap. In Sections 3.4.1 and 3.4.2 we use the buffer heap to obtain cache-oblivious SSSP algorithms for weighted undirected and directed graphs, respectively. In Section 3.4.3 we describe the application of buffer heap in obtaining an improved cache-aware APSP algorithm for weighted undirected graphs. Finally, we present some concluding remarks in Section 3.5.

3.2 Slim Data Structures

A *slim data structure* is a data structure with a fixed-size footprint in the cache. The area in the cache that holds the footprint is called the *slim cache*. By $DS(\lambda)$ we denote a data structure DS, in which a portion of size $\Theta(\lambda)$ is kept in the slim cache. We continue to assume the behavior of the two-level I/O model, namely **(a)** the size of the cache is M and **(b)** data is transferred between the cache and the main memory in blocks of size B. Thus $1 \leq \lambda \leq M$; and the data structural operations must assume that the portion of the data structure that is not stored in the slim cache is stored in a main memory divided into blocks of size B, and thus accessing anything outside the slim cache may cause cache-misses. While executing a data structural operation the operation can use all free cache space for temporary computation, but after the operation completes only the data in the slim cache is preserved for reuse by the next operation on the data structure.

Some existing data structures can be viewed trivially as slim data structures. For example, Arge et al. [13] analyzed each component tournament tree of the *multi-tournament-tree* as supporting *Decrease-Key*, *Delete* and *Delete-Min* operations in $\mathcal{O}\left(\frac{1}{\lambda}\log N\right)$ amortized cache-misses each for $1 \leq \lambda \leq \frac{B}{2}$; this can be viewed as a slim data structure for this range of values for λ.

Although our main motivation behind introducing the notion of slim data structures was to obtain the APSP result in Section 3.4.3, we believe that the need for slim data structures could arise in other applications. A typical application would be one in which a number of data structures need to be kept in the cache simultaneously, and thus only a limited portion of the cache can be dedicated to each data structure.

In the next section we present our cache-oblivious buffer heap, and analyze its performance as a slim data structure.

3.3 The Buffer Heap

In this section we present the *Buffer Heap*, a cache-oblivious priority queue that supports *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$ amortized cache-misses each, where N is the number of items in the priority queue. A *Delete*(x) operation deletes element x from the queue if it exists and a *Delete-Min*() operation retrieves and deletes an element with the minimum key from the queue. A *Decrease-Key*(x, k_x) operation inserts the element x with key k_x into the queue if x does not already exist in the queue, otherwise it replaces the smallest key k'_x of x in the queue with k_x provided $k_x < k'_x$, and deletes all remaining keys of x in the queue. For simplicity of exposition, we assume that all keys in the data structure are distinct.

When analyzed as a slim data structure with a slim cache of size λ, we show that a buffer heap supports each of its three operations in $\mathcal{O}\left(\frac{1}{\lambda}+\frac{1}{B}\log_2\frac{N}{\lambda}\right)$ amortized cache-misses. The buffer heap, however, remains oblivious of the parameter λ; the external application using the data structure may choose to maintain a slim cache, i.e., impose a restriction on the value of λ. When a buffer heap is restricted to use a slim cache, we call it a *Slim Buffer Heap* and denote it by $SBH(\lambda)$, otherwise we call it a *Regular Buffer Heap*. Note that since a buffer heap is not aware of the existence of a slim cache, both types of buffer heap (slim and regular) have exactly the same implementation, the only difference is in their analysis. A regular buffer heap can be viewed as a slim buffer heap with a slim cache of size $\lambda = \Theta(M) = \Omega(B)$.

A regular buffer heap matches the cache complexity of a tournament tree [83], its only cache-aware counterpart that supports the same operations. It has been shown in [13] that a slim version of the tournament tree (a component of the multi-tournament-tree introduced in [13]) supports *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}\left(\frac{1}{\lambda}\log N\right)$ amortized cache-misses each when restricted to use a slim cache of size $\lambda \in \left[1, \frac{B}{2}\right]$. Hence, a slim buffer heap improves over the cache complexity of a slim tournament tree.

3.3.1 Structure

A buffer heap on N items consists of $r = 1 + \lceil \log_2 N \rceil$ levels. For $0 \le i \le r-1$, level i consists of an *element buffer* B_i and an *update buffer* U_i. Each element in B_i is of the form (x, k_x), where x is the element id and k_x is its key. Each update or operation in U_i is augmented with a timestamp indicating the time of its insertion into the data structure.
At any time, the following invariants are maintained:

Invariant 3.3.1.

(a) *Each* B_i $(0 \le i < r)$ *contains at most* 2^i *elements.*
(b) *Each* U_i $(0 \le i < r)$ *contains at most* 2^i *updates.*

Invariant 3.3.2.

(a) *Key of every element in* B_i $(0 \le i < r-1)$ *is no larger than the key of any element in* B_{i+1}.

(b) *All updates applicable to* B_i $(0 \le i < r-1)$ *that are not yet applied, reside in* $U_0, U_1, \ldots, U_i$.

Invariant 3.3.3.

(a) *Elements in each* B_i *are kept sorted in ascending order by element id.*

(b) *Updates in each* U_i *are divided into (a constant number of) segments with updates in each segment sorted in ascending order by element id and timestamp.*

All buffers are initially empty.

3.3.2 Layout

The element buffers are stored in a stack S_B with elements of B_i placed above elements of B_j for all $i < j$. Elements of the same B_i occupy contiguous space in the stack with an element (x_1, k_1) stored above another element (x_2, k_2) provided $x_1 < x_2$. Similarly, update buffers are placed in another stack S_U where updates in any U_i are stored above those in all U_j with $j > i$. Updates in a single buffer occupy a contiguous region in the stack. For $0 \leq i \leq r-1$, the segments of U_i are stored one above another in the stack, and updates in each segment are stored sorted from top to bottom first by element id and then by timestamp. An array A_s of size r stores information on the buffers. For $0 \leq i \leq r-1$, $A_s[i]$ contains the number of elements in B_i, and the number of segments in U_i along with the number of updates in each segment.

The buffer heap uses $\mathcal{O}(N)$ space.

3.3.3 Operations

In this section we describe how *Delete*, *Delete-Min* and *Decrease-Key* operations are implemented.

A *Decrease-Key* operation is performed by the Decrease-Key function (i.e., Function 3.3.1) which inserts it into U_0 augmented with the current timestamp. Further processing is deferred to the next *Delete-Min* operation except that the Fix-U function may be called to restore invariant 3.3.1(b) (i.e., overflowing update buffers) for the structure. A *Delete* operation is performed by the Delete function (i.e., Function 3.3.2) in exactly the same way.

The Fix-U function uses a function called Apply-Updates. When called with a parameter i, Apply-Updates (i.e., Function 3.3.5) applies the updates in U_i on the elements of B_i, and empties U_i by moving the updates from U_i to U_{i+1}. It also moves any overflowing elements from B_i to U_{i+1} as *Sink* operations. A *Sink*(x, k_x) operation is used to move an element (x, k_x) from B_i to B_{i+1} through U_{i+1}.

The Fix-U function (i.e., Function 3.3.6) is called with parameter i when U_i overflows. This function starts at level i and continues calling Apply-Updates on each successive level until it reaches a level j such that U_{j+1} does not overflow when Apply-Updates(j) completes execution. It collects all elements left in $B_i, B_{i+2}, \ldots, B_j$ in a temporary buffer B' and returns B' leaving these element buffers empty.

Every call to Fix-U is followed by a call to the Redistribute function (i.e., Function 3.3.7) which redistributes the elements returned by Fix-U to the shallowest element buffers.

The Delete-Min function (i.e., Function 3.3.3) executes a *Delete-Min* operation by first calling the Find-Min function to find an element with the minimum key in the data structure, and then calling the DELETE function to delete this element.

The FIND-MIN function (i.e., Function 3.3.4) works by finding the shallowest element buffer B_k that is left non-empty after applying the updates in U_k (by calling APPLY-UPDATES). The FIX-U function is then called to fix overflowing update buffers, if any. The elements left in B_k along with the elements returned by FIX-U are distributed to the shallowest element buffers by calling REDISTRIBUTE.

After each operation the RECONSTRUCT function (i.e., Function 3.3.8) is called. This function reconstructs the entire data structure periodically. It remembers the number of elements N_e in the structure immediately after the last reconstruction, and keeps track of the number of new operations N_o performed since then. Initially N_e is set to 0. When $N_o = \lfloor \frac{N_e}{2} \rfloor + 1$, the data structure is rebuilt by calling APPLY-UPDATES for each level, emptying the update buffers and distributing the remaining elements to the shallowest possible levels. The objective of the function is to ensure that the number of levels r in the structure is always within ± 1 of $\log_2 N$, where N is the current number of elements in the structure. This invariant is maintained because r can decrease by at most 1 since the last reconstruction (this happens if all $\lfloor \frac{N_e}{2} \rfloor + 1$ operations are *Delete* or *Delete-Min* operations), and can increase by at most 1 (if all those operations are *Decrease-Key*s).

Correctness

We prove the correctness of all buffer heap operations below.

Lemma 3.3.1. *Buffer heap correctly supports three external-memory priority queue operations, namely,* Decrease-Key*,* Delete *and* Delete-Min *operations, on its elements.*

Proof. We will prove that the DECREASE-KEY/DELETE function correctly inserts the corresponding *Decrease-Key*/*Delete* operation into the buffer heap, and the DELETE-MIN function correctly extracts the element with the minimum key from the buffer heap, while correctly applying all relevant *Decrease-Key* and *Delete* operations, and maintaining all invariants.

Before proving the correctness of the three functions mentioned above we must establish the correctness of APPLY-UPDATES and FIX-U which are called as subroutines by all of them. The APPLY-UPDATES function is at the core of all buffer heap functionality.

<u>APPLY-UPDATES</u>. When called with parameter i, APPLY-UPDATES applies all updates in U_i on the elements in B_i under the assumption that all invariants hold initially except possibly invariant 3.3.1(b) for U_i. All U_j for $0 \leq j < i$ are assumed to be empty.

FUNCTION **3.3.1.** DECREASE-KEY(x, k_x)

[Inserts a *Decrease-Key* operation into the structure, that decreases the key of element x to k_x. If x does not already exist in the structure, this operation results in the insertion of x with key k_x.**]**

1. insert the operation into U_0 augmented with current timestamp maintaining inv. 3.3.3(b)

2. $B' \leftarrow \emptyset$, $i \leftarrow 0$ $\{$*list* B' *stores elements returned by* FIX-U$\}$
FIX-U(i, B') $\{$*fix* U_i *(i.e, restore invariant 3.3.1(b)) in case of overflow*$\}$

3. REDISTRIBUTE(B') $\{$*redistribute elements in* B' *to shallowest element buffers*$\}$

4. RECONSTRUCT( ) $\{$*reconstruct the data structure periodically*$\}$

DECREASE-KEY ENDS

FUNCTION **3.3.2.** DELETE(x)

[Inserts a *Delete* operation into the structure, that deletes element x from the structure if exists.**]**

Same as Function 3.3.1 (DECREASE-KEY) above

DELETE ENDS

FUNCTION **3.3.3.** DELETE-MIN( ) **[**Extracts element with the smallest key from the structure.**]**

1. $(x, k_x) \leftarrow$ FIND-MIN( ) $\{$*find the element with the minimum key*$\}$

2. ***if*** $k_x \neq +\infty$ ***then*** DELETE(x) $\{$*delete* x *from the data structure if nonempty*$\}$

3. ***return*** (x, k_x)

DELETE-MIN ENDS

FUNCTION **3.3.4.** FIND-MIN( ) **[**Returns the element with the smallest key in the structure.**]**

1. $i \leftarrow -1$
repeat
(i) $i \leftarrow i + 1$
(ii) APPLY-UPDATES(i) $\{$*apply the updates in* U_i *on the elements in* $B_i\}$
until ($|B_i| > 0$) $\vee$ ($i = r - 1$)

2. ***if*** $|B_i| = 0$ ***then*** $\{$*the data structure has become empty*$\}$
(i) $(x, k_x) \leftarrow (\_\ , +\infty)$, $r \leftarrow 1$ $\{$*will return* $+\infty$ *as the minimum key*$\}$

3. ***else*** $\{$*the data structure is nonempty*$\}$
(i) $B' \leftarrow B_i$, $i \leftarrow i + 1$
FIX-U(i, B') $\{$*fix* U_i *(i.e, restore invariant 3.3.1(b)) in case of overflow*$\}$
(ii) REDISTRIBUTE(B') $\{$*redistribute elements in* B' *to shallowest element buffers*$\}$
(iii) $(x, k_x) \leftarrow$ the element in B_0 $\{B_0$ *has the element with the minimum key*$\}$

4. ***return*** (x, k_x)

FIND-MIN ENDS

Function **3.3.5.** Apply-Updates(i)

[Applies the updates in U_i on the elements in B_i, move remaining updates from U_i to U_{i+1} if $i < r-1$, and after applying the updates moves overflowing elements from B_i to U_{i+1} as *Sink*s.

Preconditions: All invariants hold except possibly 3.3.1(b) for U_i. All U_j, $j \in [0, i-1]$ are empty.

Postconditions: All invariants hold except possibly 3.3.1(b) for U_{i+1}. All U_j, $j \in [0, i]$ are empty.**]**

1. merge the segments of U_i

2. ***if*** ($|B_i| = 0$) $\wedge$ ($i < r-1$) ***then*** {*if i is not the last level and B_i is empty*}

  (i) empty U_i by moving the contents of U_i as a new segment of U_{i+1}

3. ***else***

  (i) ***if*** $i = r-1$ ***then*** $k \leftarrow +\infty$ ***else*** $k \leftarrow$ largest key in B_i

  (ii) scan B_i and U_i simultaneously, and for each $op \in U_i$: {*apply the updates in U_i on B_i*}

    (a) ***if*** op = *Delete*(x) ***then*** remove any element (x, k_x) from B_i if exists

    (b) ***if*** op = *Decrease-Key*(x, k_x)/*Sink*(x, k_x) ***then***

      - replace any $(x, k'_x) \in B_i$ with $(x, \min(k_x, k'_x))$

      - copy (x, k_x) to B_i if no (x, k'_x) exists in B_i and $k_x \leq k$

  (iii) ***if*** $i < r-1$ ***then*** {*move appropriate updates from U_i to U_{i+1}*}

    (a) copy each *Decrease-Key*(x, k_x) in U_i, not applied in step 3(ii)(b) to U_{i+1}

    (b) for each *Delete*(x) and each *Decrease-Key*(x, k_x) in U_i that was applied in step 3(ii)(b) copy a *Delete*(x) to U_{i+1}

  (iv) ***if*** $|B_i| > 2^i$ ***then*** {*restore invariant 3.3.1(a) if violated*}

    (a) ***if*** $i = r-1$ ***then*** $r \leftarrow r+1$

    (b) keep the 2^i elements with the smallest 2^i keys in B_i and move each remaining element (x, k_x) to U_{i+1} as *Sink*(x, k_x)

  (v) $U_i \leftarrow \emptyset$

Apply-Updates Ends

Function **3.3.6.** Fix-U(i, B')

[Fixes all overflowing update buffers in levels i and up. Update buffer U_i overflows if $|U_i| > 2^i$ (see invariant 3.3.1(b)). For each overflowing U_i collects contents of B_i in B' after applying U_i on B_i.

Preconditions: All invariants hold except invariant 3.3.1(b) for U_i. All U_j for $0 \leq j < i$ are empty.

Postconditions: All invariants hold. If k is the largest index for which the ***while*** loop in line 1 was executed, then all U_j for $0 \leq j \leq k$ are empty. The contents of all B_j for $i \leq j \leq k$ after applying all applicable updates on them are collected in B' leaving those buffers empty.**]**

1. ***while*** ($i < r$) $\wedge$ $\left(|U_i| > 2^i \right)$ ***do***

  (i) Apply-Updates(i) {*apply the updates in U_i on the elements in B_i*}

  (ii) empty B_i by merging it with B' {*collect in B' the elements remaining in B_i*}

  (iii) $i \leftarrow i+1$

Fix-U Ends

FUNCTION **3.3.7.** REDISTRIBUTE(B')

[Distributes the elements in B' to the shallowest element buffers maintaining invariants 3.3.1(a), 3.3.2(a) and 3.3.3(a).

Preconditions: All invariants hold. All B_i and U_i with $0 \le i \le k$ are empty, where k is the smallest integer such that $2^{k+1} - 1 \ge |B'|$. No key value in the data structure is smaller than any key value in B'.

Postconditions: All invariants hold. All update buffers remain unchanged, but $\bigcup_{i=0}^{k} B_i = B'$.**]**

1. $i \leftarrow$ largest integer such that $2^i - 1 < |B'|$

2. ***while*** $i \ge 0$ ***do***

  (i) move $|B'| - 2^i + 1$ elements with the largest $|B'| - 2^i + 1$ keys from B' to B_i maintaining invariant 3.3.3(a)

  (ii) $i \leftarrow i - 1$

REDISTRIBUTE ENDS

FUNCTION **3.3.8.** RECONSTRUCT( )

[Reconstructs the data structure when $N_o = \lfloor \frac{N_e}{2} \rfloor + 1$, where N_e is the number of elements in the data structure immediately after the last reconstruction ($N_e = 0$ initially), and N_o is the number of operations since the last reconstruction/initialization of the data structure.**]**

1. ***if*** $N_o = \lfloor \frac{N_e}{2} \rfloor + 1$ ***then***

  (i) $B' \leftarrow \emptyset$
  for $i \leftarrow 0$ ***to*** $r - 1$ ***do***

    (a) APPLY-UPDATES(i) {*apply the updates in* U_i *on the elements in* B_i}

    (b) merge B_i with B' {*collect in* B' *the elements remaining in* B_i}
    $B_i \leftarrow \emptyset$

  (ii) REDISTRIBUTE(B') {*redistribute elements in* B' *to shallowest element buffers*}

  (iii) $r \leftarrow i$, where i is the largest level such that $|B_i| > 0$

RECONSTRUCT ENDS

Observe that since invariant 3.3.2(b) holds initially and for $0 \le j < i$, $|U_j| = 0$, all updates applicable to B_i must reside in U_i. For each element x, this function considers all updates in U_i that are applicable to x in increasing order of timestamp, i.e., in the order in which they were inserted into the data structure. For each such $op \in U_i$ taken in order APPLY-UPDATES does the following.

• $\underline{op = Delete(\ x\ )}$: If any (x, k_x) exists in B_i it is deleted. If this element did not exist in B_i initially then it must have been inserted into B_i by a *Decrease-Key*(x, k_x)/*Sink*(x, k_x) operation in U_i earlier in the order. Irrespective of whether this *Delete*(x) operation was able to delete an element from B_i or not, it is moved to U_{i+1} without changing its timestamp which ensures that any remaining occurrence of x in the data structure inserted by operations with earlier timestamps is deleted.

• $op = \textit{Decrease-Key}(\ x,\ k_x\ )$: If some (x, k'_x) appears in B_i it is replaced with $(x, \min(k_x, k'_x))$. However, if element x does not appear in B_i, (x, k_x) is inserted into B_i provided $k_x \leq k$, where k is the largest key in B_i ($k = +\infty$ if i is the last level). Observe that if x initially existed in B_i but does not exist now, then it must have been deleted by some *Delete*(x) operation in U_i earlier in the order. Since each *Decrease-Key*(x, k_x) operation that cannot be applied to B_i must have $k_x > k$, it must be applicable to some element buffer in $B_{i+1}, B_{i+2}, \ldots, B_{r-1}$, and so it is moved to U_{i+1} in order to ensure that it is applied to the appropriate element buffer. For each *Decrease-Key*(x, k_x) operation that is applied to B_i, we copy a *Delete*(x) operation with the same timestamp to U_{i+1} so that all occurrences of x in $B_{i+1}, B_{i+2}, \ldots, B_{r-1}$ inserted by *Decrease-Key*(x, k_x) /*Sink*(x, k_x) operations with earlier timestamps are deleted.

• $op = \textit{Sink}(\ x,\ k_x\ )$: If some (x, k'_x) appears in B_i it is replaced with $(x, \min(k_x, k'_x))$, otherwise (x, k_x) is inserted into B_i. Since a *Sink*(x, k_x) operation is used to move element (x, k_x) from B_{i-1} to B_i, we will always have $k_x \leq k$, where k is the largest key in B_i ($k = +\infty$ if i is the last level), and so these updates are not applicable to element buffers in higher levels, i.e., Apply-Updates does not need to carry these updates to U_{i+1}.

Clearly, Apply-Updates never violates invariants 3.3.2 and 3.3.3. However, it can violate invariant 3.3.1(a) if $|B_i| > 2^i$ holds after the updates. It fixes this violation by keeping only the 2^i elements with the smallest 2^i keys in B_i and moving the remaining elements to U_{i+1} as *Sink* operations. Each such overflowing item (x, k_x) is moved to U_{i+1} as a *Sink*(x, k_x) operation with the current timestamp so that existing operations in the data structure cannot prevent this operation from inserting (x, k_x) into B_{i+1}.

Thus after the function terminates all invariants continue to hold except possibly invariant 3.3.1(b) for U_{i+1}. Since all updates from U_i are either moved to U_{i+1} or discarded, $|U_j| = 0$ holds for $j \in [0, i]$.

Fix-U. This function is called with parameter i when U_i overflows. It makes the same assumptions as Apply-Updates. Starting from level i it continues to call Apply-Updates for each level until it reaches a level j such that U_{j+1} does not overflow when Apply-Updates(j) terminates, i.e., the data structure does not have any overflowing update buffers and thus all invariants hold. For $i \leq k \leq j$, this function collects in a temporary buffer B' the contents of each B_j after applying U_j to it leaving B_j empty, and returns B'. The correctness of Fix-U follows directly from the correctness of Apply-Updates.

Decrease-Key(x, k_x)/Delete(x). The function inserts the corresponding *Decrease-Key*(x, k_x) /*Delete*(x) operation into U_0 augmented with the current timestamp so that it is treated by the data structure as the most recent operation. This insertion does not violate any invariants except possibly invariant 3.3.1(b) for U_0, i.e., U_0 overflows. This violation is fixed by calling FIX-U with parameter $i = 0$. Upon return from FIX-U all invariants hold. The set B' of elements returned by FIX-U does not have any key value larger than any key in the data structure, and FIX-U leaves enough empty element buffers at the shallowest possible levels so that the elements in B' can be distributed to those buffers without violating any invariant. The REDISTRIBUTE function performs this distribution. The RECONSTRUCT function reconstructs the entire data structure periodically. Thus the correctness of DECREASE-KEY/DELETE follows from the correctness of FIX-U, REDISTRIBUTE and RECONSTRUCT. We have already argued the correctness of FIX-U. The proofs of correctness of REDISTRIBUTE and RECONSTRUCT are straight-forward and hence are omitted.

DELETE-MIN( ). The DELETE-MIN function first calls FIND-MIN in order to find the element with the minimum key in the entire data structure, and then calls DELETE in order to delete this element. We have already argued correctness of DELETE, and hence we only need to prove FIND-MIN correct.

Observe that if invariant 3.3.2 holds, the smallest level k such that B_k is non-empty after applying all updates in $U_0, U_1, \ldots, U_k$ on B_k will contain the element with the smallest key in the entire data structure. The FIND-MIN function builds on this observation. Starting from level 0 it calls APPLY-UPDATES for each level until it reaches the first level k with $|B_k| \neq 0$ upon return from APPLY-UPDATES. At this point all invariants hold except possibly invariant 3.3.1(b) for U_{k+1}. The overflowing U_{k+1} is fixed by calling FIX-U for level $k+1$. All elements returned by FIX-U along with the contents of B_k are distributed to the shallowest possible element buffers by REDISTRIBUTE. At this point all invariants hold, B_0 contains exactly one element and U_0 is empty. Therefore, the element in B_0 which is returned by FIND-MIN is, indeed, the element with the smallest key. ∎

Cache Complexity

In this section we will view the buffer heap as a slim data structure with a slim cache of size $\Theta(\lambda)$ and denote it by $SBH(\lambda)$. The slim cache is assumed to be large enough to store $B_0, B_1, \ldots, B_t$ and $U_0, U_1, \ldots, U_{t+1}$, where $t = \log(\lambda + 1) - 1$. The remaining buffers reside in external memory.

The following two observations will be useful in our analyses.

Observation 3.3.1. *For $i \in [1, r-1]$,*

(a) *Each* Sink *operation in U_i can be mapped to a unique* Decrease-Key/Sink *operation that existed in U_{i-1} but does not exist in U_i; and*

(b) *U_i cannot contain more* Sink *operations than* Delete *operations.*

It is not difficult to see that Observation 3.3.1(a) is valid since each *Sink* operation in U_i is generated by an element evicted from B_{i-1} due to overflow, and each eviction from B_{i-1} can be viewed as caused by a unique *Decrease-Key*/*Sink* operation in U_{i-1} that inserted an element into B_{i-1}. After the insertion the responsible *Decrease-Key*/*Sink* operation ceases to exist: if it is a *Decrease-Key* operation it is converted to a *Delete* operation, and if it is a *Sink* operation it is simply discarded. The implication of Observation 3.3.1(a) is that every existing *Sink* operation in the queue can be traced back to a unique *Decrease-Key* operation following a chain of *Sink*s.

We know that the unique *Decrease-Key* operation responsible for the generation of any given *Sink* operation in U_i was converted to a *Delete* at the time it was applied on an element buffer, and it is not difficult to see that this *Delete* operation must now reside in U_i. Thus each *Sink* operation in U_i maps to a unique *Delete* operation in U_i, and Observation 3.3.1(b) follows.

The following lemma which implies that merging the segments of U_i (in line 1 of Apply-Updates) incurs only $\mathcal{O}\left(\frac{1}{B}\right)$ amortized cache-misses per operation in U_i, will be crucial in proving the cache-complexity of buffer heap operations.

Lemma 3.3.2. *For $1 \leq i \leq r-1$, every empty U_i receives batches of updates at most a constant number of times before U_i is applied on B_i and emptied again.*

Proof. Since $|U_1| \leq 2$, U_1 cannot receive more than two batches of updates before it overflows, and thus the lemma holds for $i = 1$. Hence, for the rest of proof we will assume $i > 1$.

Update buffer U_i receives at most two batches of updates whenever the execution of a Decrease-Key/Delete/Delete-Min function reaches level $i-1$. If the execution continues and reaches level i then U_i is applied on B_i, and thus emptied. If the execution terminates at level $i-1$ but leaves B_{i-1} empty, the next time an execution reaches level $i-1$ will continue to level i and empty U_i. Therefore, it suffices to consider only executions that terminate at level $i-1$ and leave B_{i-1} nonempty. Let $\mathcal{E}$ be such an execution. We will show that $\mathcal{E}$ increases the number of updates in U_i by at least 2^{i-2} which implies that executions can terminate at level $i-1$ at most four times without emptying B_{i-1} before U_i overflows (since $|U_i| \leq 2^i$) and is thus emptied by Fix-U.

For $j \in [0, r-1]$, let u_j and u'_j denote the number of updates in U_j immediately before the start of $\mathcal{E}$ and immediately after the termination of $\mathcal{E}$, respectively, and let $\delta u_j = u'_j - u_j$. For $j \in [0, i-1]$, we denote by u''_j the number of updates in U_j immediately before $\mathcal{E}$ reaches level j (i.e., $\mathcal{E}$ has already pushed all updates and overflowing elements from level $j-1$ to level j if $j > 0$). Let b'_j ($j \in [0, r-1]$) be the number of elements in B_j immediately after $\mathcal{E}$ terminates. We will prove the following.

$$(\ i > 1 \ ) \wedge (\ b'_{i-1} \neq 0 \ ) \Rightarrow (\ \delta u_i \geq 2^{i-2} \ ) \tag{3.3.1}$$

Now in order to establish equation 3.3.1 we consider the following two cases.

Case 1 $\left(u''_{i-1} < 2^{i-1}\right)$**:** Let $\mathcal{E}'$ be the last execution before $\mathcal{E}$ that reached level $i-1$ (and possibly continued to higher levels). Execution $\mathcal{E}$ has reached level $i-1$ because all B_j, $j \in [0, i-2]$ have become empty which were left full by $\mathcal{E}'$. Hence, at least $\sum_{j=0}^{i-2} 2^j = 2^{i-1} - 1$ elements have been deleted from the structure since $\mathcal{E}'$ completed execution, i.e., u''_{i-1} includes at least $2^{i-1} - 1 \geq 2^{i-2}$ *Delete* operations all of which will be moved to U_i and thus $\delta u_i \geq 2^{i-2}$.

Case 2 $\left(u''_{i-1} \geq 2^{i-1}\right)$**:** Since an update buffer cannot contain more *Sink* operations than *Delete* operations (see Observation 3.3.1(b)), u''_{i-1} includes at least $\frac{2^{i-1}}{2} = 2^{i-2}$ *Delete*/*Decrease-Key* operations and thus $\delta u_i \geq 2^{i-2}$.

Hence, equation 3.3.1 and consequently the lemma follow. ■

The following lemma gives the cache complexity of the operations supported by a slim buffer heap:

Lemma 3.3.3. *A slim buffer heap with a slim cache of size λ (i.e., $SBH(\lambda)$) supports* Delete, Delete-Min *and* Decrease-Key *operations in $\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda}\right)$ amortized cache-misses each, where N is the number of elements in the structure.*

Proof. For $0 \leq i \leq r-1$, let u_i be the number of operations in U_i and let d_i be the number of *Decrease-Key* operations among them. By Δ we denote the number of *Decrease-Key*, *Delete* and *Delete-Min* operations performed on the data structure since its last construction/reconstruction. If H is the current state of $SBH(\lambda)$, we define the *potential* of H as follows:

$$\Phi(H) = \sum_{i=0}^{r-1} \left(\frac{1}{B} \cdot (r-i) \ + \ \frac{2}{\lambda} \cdot \frac{1}{2^{\max(i-t,0)}} \right) \cdot (u_i + d_i) \ + \ \left(\frac{r}{B} + \frac{1}{\lambda} \right) \cdot \Delta,$$

where $t = \log(\lambda + 1) - 1$.

As in the original I/O analysis of Buffer Heap operations in [32], the key observation is that operations in update buffers always move downward and at each level they participate in a constant number of scans. The first term under the summation in $\Phi(H)$ captures this flow of data. The main reason for adding the second term is to ensure that after every $\Theta(\lambda)$ new operations enough potential is accumulated to account for the extra cache-miss in accessing data outside the slim cache. Also $\Phi(H)$ has been designed so that the potential gain due to a new *Decrease-Key* operation is more than that for a new *Delete* operation. This uneven distribution of potential is based on the observation that after a *Decrease-Key* operation has been applied successfully on some B_i it turns into a *Delete* operation and possibly generates an additional *Sink* operation in U_{i+1} (see Observation 3.3.1 and its implications). The last term in $\Phi(H)$ gathers potentials for the next reconstruction of the data structure.

We compute the amortized cost of each buffer heap operation below.

Reconstruction. Let us first consider the amortized cost of reconstruction (i.e., the RECONSTRUCT function). At the time of reconstruction $\Delta = \left\lfloor \frac{N_e}{2} \right\rfloor + 1$, where N_e is the number of elements in the structure immediately after the last reconstruction. Thus $\left\lceil \frac{N_e}{2} \right\rceil - 1 \leq \sum_{i=0}^{r-1} |B_i| \leq \left\lfloor \frac{3N_e}{2} \right\rfloor + 1$ implying $\Delta = \Theta\left(\sum_{i=0}^{r-1} |B_i|\right)$. If during the reconstruction operation no buffer outside the slim cache is accessed then no cache-miss occurs. Therefore, we will only consider the case in which some element buffer above level t is accessed. In that case $\Delta = \Omega(\lambda)$.

Accessing the first data outside the slim cache incurs $\mathcal{O}(1)$ cache-misses. The actual cache complexity of APPLY-UPDATES when called with a parameter i in step $1(i)(a)$ of RECONSTRUCT is $\mathcal{O}\left(\frac{|U_i|+|B_i|}{B}\right) = \mathcal{O}\left(\frac{\Delta}{B}\right)$, since the merge operations in step 1 of APPLY-UPDATES can be performed in $\mathcal{O}\left(\frac{|U_i|}{B}\right)$ cache-misses (implied by Lemma 3.3.2); steps $2(i)$, $3(i)$, $3(ii)$ and $3(iii)$ involve a constant number of scans of B_i and U_i incurring $\mathcal{O}\left(\frac{|U_i|+|B_i|}{B}\right)$ cache-misses; and step $3(iv)$ can be performed in $\mathcal{O}\left(\frac{|B_i|}{B}\right)$ cache-misses using a linear I/O selection algorithm [104]. The buffer B_i can be merged with B' in step $1(i)(b)$ of RECONSTRUCT in $\mathcal{O}\left(\frac{|B_i|+|B'|}{B}\right) = \mathcal{O}\left(\frac{\Delta}{B}\right)$ cache-misses. Therefore, the actual cache complexity of step $1(i)$ of RECONSTRUCT is $\mathcal{O}\left(1 + \frac{r}{B} \cdot \Delta\right)$. The actual cost of the REDISTRIBUTE function in step $1(ii)$ of RECONSTRUCT is $\mathcal{O}\left(\frac{r}{B} \cdot \Delta\right)$ since the ***while*** loop in step 2 of REDISTRIBUTE iterates $\mathcal{O}(r)$ times and in each iteration scans each element of B' at most a constant number of times if a linear I/O selection algorithm is used. Thus the actual cache complexity of reconstruction is $\mathcal{O}\left(1 + \frac{r}{B} \cdot \Delta\right)$.

Since all update buffers are emptied during reconstruction and $\Delta = \Omega(\lambda)$, the potential drop is $\Omega\left(\left(\frac{1}{\lambda} + \frac{r}{B}\right) \cdot \Delta\right) = \Omega\left(1 + \frac{r}{B} \cdot \Delta\right)$. Thus the amortized cost of reconstruction is $\mathcal{O}\left(1 + \frac{r}{B} \cdot \Delta\right) - \Omega\left(1 + \frac{r}{B} \cdot \Delta\right) \leq 0$.

Decrease-Key/Delete. The increase in potential due to the insertion of a *Decrease-Key* operation into U_0 is $\frac{5}{\lambda} + \frac{3}{B} \cdot r$, and due to the insertion of a *Delete* operation is $\frac{3}{\lambda} + \frac{2}{B} \cdot r$. If no element buffer of level higher than t is accessed in step 2 of the DECREASE-KEY/DELETE function then no cache-miss occurs (except in the RECONSTRUCT function in step 4 whose amortized cost has already been shown to be ≤ 0). So we only need to consider the case when a B_i with $i > t$ is accessed.

Let j be the largest value of i for which the ***while*** loop in step 1 of FIX-U was executed. The actual cost of APPLY-UPDATES when called with a parameter i in step $1(i)$ of FIX-U is $\mathcal{O}\left(\frac{|U_i|+|B_i|}{B}\right) = \mathcal{O}\left(\frac{2^i}{B}\right)$. The buffer B_i can be merged with B' in step $1(ii)$ of FIX-U in $\mathcal{O}\left(\frac{|B_i|+|B'|}{B}\right) = \mathcal{O}\left(\frac{2^i}{B}\right)$ cache-misses. Therefore, FIX-U incurs at most $\sum_{i=0}^{j}\mathcal{O}\left(\frac{2^i}{B}\right) = \mathcal{O}\left(\frac{2^j}{B}\right)$ cache-misses in total. Also $|B'| = \mathcal{O}\left(2^j\right)$ when FIX-U returns. Hence, the actual number of cache-misses incurred by REDISTRIBUTE in step 3 of DECREASE-KEY/DELETE for redistributing the elements in B' is at most $\mathcal{O}\left(\frac{2^j}{B}\right) + \sum_{i=0}^{j}\mathcal{O}\left(\frac{2^i}{B}\right) = \mathcal{O}\left(\frac{2^j}{B}\right)$ assuming a linear I/O selection algorithm is used. Therefore, including the $\mathcal{O}(1)$ cache-misses incurred for accessing the first data outside the slim cache, the actual cost of steps 1–3 of a *Decrease-Key/Delete* operation is $\mathcal{O}\left(1 + \frac{2^j}{B}\right)$.

Since U_j was full before APPLY-UPDATES was called in step $1(i)$ of FIX-U, the drop of potential due to the movement of these $|U_j| \geq 2^j$ updates to U_{j+1} is $\Omega\left(2^j \cdot \left(\frac{2}{\lambda} \cdot \frac{1}{2^{j+1-t}} + \frac{1}{B}\right)\right) = \Omega\left(1 + \frac{2^j}{B}\right)$. Therefore, this potential drop can compensate for the actual cost of executing steps 1–3 of DECREASE-KEY/DELETE.

Thus the amortized cost of a *Decrease-Key/Delete* operation is $\mathcal{O}\left(\frac{1}{\lambda} + \frac{r}{B}\right) = \mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 N\right)$. But since accessing the first t levels incurs no cache-misses, the amortized cost is $\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\{\log_2 N - t\}\right) = \mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda}\right)$.

***Delete-Min*.** The DELETE-MIN function calls the FIND-MIN function followed by a possible call to the DELETE function. We have already shown that the amortized cost of a *Delete* operation is $\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda}\right)$. We will show below that the amortized cost of finding the minumum is ≤ 0.

Let j be the largest value of i for which APPLY-UPDATES(i) was called by FIND-MIN. If $|U_j| \geq 2^j$ immediately before APPLY-UPDATES(j) was called (i.e., called inside FIX-U in step $3(i)$ of FIND-MIN), then the analysis is similar to that for *Decrease-Key/Delete* operation. Hence, here we will only consider the case when $|U_j| < 2^j$, i.e., APPLY-UPDATES(j) was called in step $1(ii)$ of FIND-MIN.

As before, we will assume that $j > t$. In this case, using an analysis similar to that for *Decrease-Key/Delete*, one can show that the actual cache complexity of FIND-MIN is $\mathcal{O}\left(1 + \frac{2^j}{B}\right)$.

Let b_j be the number of elements in B_j before APPLY-UPDATES(j) was called. Then in order to compute the potential drop we need to consider the following two cases.

(i) $\underline{b_j > 0}$: Observe that in this case the last REDISTRIBUTE function call that distributed elements from level j or higher must have left $B_0, B_1, \ldots, B_{j-1}$ completely full, and hence at least $\sum_{i=0}^{j-1}2^i = 2^j - 1$ elements have been deleted from the structure since last time B_j was accessed. Therefore, immediately before the current call to APPLY-UPDATES(j), U_j must have included at least $2^j - 1$ *Delete* operations, all of which were moved to U_{j+1}. Hence, the potential drop due to the movement of these operations is $\Omega\left((2^j - 1) \cdot \left(\frac{2}{\lambda} \cdot \frac{1}{2^{j+1-t}} + \frac{1}{B}\right)\right) = \Omega\left(1 + \frac{2^j}{B}\right)$.

(ii) $b_j = 0$: This can only happen when $j = r - 1$. Observe that level j was created due to an overflow in B_{j-1} and the overflowing elements from B_{j-1} was pushed into U_j as *Sink* operations. Therefore, at least 2^{j-1} elements have been deleted from the structure since this level was created, and as in case *(i)* this implies a potential drop of $\Omega\left(1 + \frac{2^j}{B}\right)$.

The amortized cost of FIND-MIN is thus $\mathcal{O}\left(1 + \frac{2^j}{B}\right) - \Omega\left(1 + \frac{2^j}{B}\right) \leq 0$.

Therefore, a *Delete-Min* operation incurs $\mathcal{O}\left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{N}{\lambda}\right)$ amortized cache-misses. ■

The following corollary follows by replacing λ with $\Theta(M) = \Omega(B)$ in Lemma 3.3.3.

Corollary 3.3.1. *A buffer heap supports* Delete*,* Delete-Min *and* Decrease-Key *operations in* $\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$ *amortized cache-misses each using* $O(N)$ *space, where* N *is the current number of elements in the structure.*

Time Complexity

The internal memory time complexities of slim buffer heap operations turn out to be independent of M, B and the slim cache size λ, and are given by the following lemma.

Lemma 3.3.4. *A slim buffer heap supports* Delete*,* Delete-Min *and* Decrease-Key *operations in* $\mathcal{O}(\log N)$ *amortized time each, where* N *is the number of elements in the structure.*

Proof. The proof uses the following potential function:

$$\Phi'(H) = \sum_{i=0}^{r-1} (r - i) \cdot (u_i + d_i) \;+\; r \cdot \Delta,$$

where H is the current state of the data structure, and u_i, d_i and Δ are as defined in the proof of Lemma 3.3.3.

The rest of the proof is similar to that of Lemma 3.3.3 but is simpler, and hence is omitted. ■

Additional Priority Queue Operations

It is straight-forward to augment a slim buffer heap with the following priority queue operations without changing its performance bounds.

Change-Key(x, k_x). This operation changes the key value of element x to k_x, and is implemented by performing a *Delete*(x) operation immediately followed by a *Decrease-Key*(x, k_x) operation. If $k_x \leq k'_x$, where k'_x is the old key of x, then the *Delete* operation acts simply like the *Delete* operation generated by the *Decrease-Key* operation immediately after its application, and thus works correctly. If $k_x > k'_x$, then the *Delete* operation first deletes x, after which the *Decrease-Key* operation reinserts it with the new key value. Since the *Delete* operation has a smaller timestamp than the *Decrease-Key* it cannot delete the new key value inserted by the *Decrease-Key*, and hence works correctly.

Relative-Increase(x, δ_x). This operation increases the key value of x by δ_x if it exists in the priority queue. It is implemented in the same way as the *Change-Key* operation above, but the *Decrease-Key* operation does not know the key value k_x initially and instead knows δ_x. However, as soon as the *Delete*(x) operation preceding the *Decrease-Key* finds the element x, k_x is updated to $k'_x + \delta_x$, where k'_x is the old key value of x discovered by the *Delete*. The *Decrease-Key* operation is then applied as usual.

3.4 Buffer Heap Applications

In this section we discuss three major applications of buffer heap. In Sections 3.4.1 and 3.4.2 we consider cache-oblivious SSSP algorithms for weighted undirected and directed graphs, respectively. These algorithms use regular buffer heaps, that is they do not impose any restriction on the size of the slim cache (i.e., assume slim cache size, $\lambda = \Theta(M) = \Omega(B)$). In Section 3.4.3 we discuss a cache-aware APSP algorithm for weighted undirected graphs. This algorithm uses a data structure built on slim buffer heaps.

3.4.1 Cache-oblivious Undirected SSSP

The cache-aware undirected SSSP algorithm by Kumar & Schwabe [83] (see [77] for a description and proof of correctness) can be made cache-oblivious by replacing both the primary and the auxiliary cache-aware priority queues used in that algorithm with buffer heaps. The primary priority queue is used to perform the standard operations for shortest path computation, and the auxiliary priority queue is used to correct for spurious updates performed on the primary priority queue. The auxiliary priority queue treats edges, instead of vertices, as its elements, and whenever a vertex with final distance $d[u]$ is settled, for each $(u, v) \in E$, a *Decrease-Key*$((u, v), d[u] + w(u, v))$ operation is performed on the auxiliary priority queue. The resulting cache-oblivious algorithm, i.e., Kumar & Schwabe's algorithm with buffer heaps, is given in Function 3.4.1 (Undirected-SSSP).

FUNCTION **3.4.1.** UNDIRECTED-SSSP($G,\ w,\ s,\ d$)

{*Kumar & Schwabe's algorithm [83] with buffer heap*}

[Given an undirected graph G with vertex set V (each vertex is identified with a unique integer in $[1, |V|)$, edge set E, a weight function $w : E \rightarrow \Re$ and a source vertex $s \in V$, this function cache-obliviously computes the shortest distance from s to each vertex $v \in V$ and stores it in $d[v]$.**]**

1. perform the following initializations:

- (i) $Q \leftarrow \emptyset$, $Q' \leftarrow \emptyset$ {*Q and Q' are both regular buffer heaps; Q contains items of the form (x, k_x) and Q' contains items of the form $((x, y), k_{x,y})$*}
- (ii) ***for*** each $v \in V$ ***do*** $d[v] \leftarrow +\infty$
- (iii) DECREASE-KEY$_{(Q)}$($s,\ 0$) {*insert vertex s with key (i.e., distance) 0 into Q*}

2. ***while*** $Q \neq \emptyset$ ***do***

- (i) $(u, k) \leftarrow$ FIND-MIN$_{(Q)}$( ), $((u', v'), k') \leftarrow$ FIND-MIN$_{(Q')}$( )
- (ii) ***if*** $k \leq k'$ ***then*** {*a new shortest distance (k) has been found* }
  - (a) DELETE$_{(Q)}$(u), $d[u] \leftarrow k$ {*k is the shortest distance from s to u*}
  - (b) ***for*** each $(u, v) \in E$ ***do***
    - DECREASE-KEY$_{(Q)}$($v,\ d[u] + w(u, v)$) {*relax edge (u, v)*}
    - DECREASE-KEY$_{(Q')}$($(u, v),\ d[u] + w(u, v)$) {*guard for spurious update on u*}

  else {*$k > k'$: shortest distance to u' has already been computed*}
  - (a) DELETE$_{(Q)}$(u'), DELETE$_{(Q')}$((u', v')) {*remove spurious vertex u'*}

UNDIRECTED-SSSP ENDS

Cache Complexity. The algorithm incurs $\mathcal{O}\left(\frac{m}{B}\log_2\frac{n}{M}\right)$ cache-misses for the $\mathcal{O}(m)$ priority queue operations it performs. In addition to that it incurs $\mathcal{O}\left(n + \frac{m}{B}\right)$ cache-misses for accessing $\mathcal{O}(n)$ adjacency lists. The cache complexity of the algorithm is thus $\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{n}{M}\right)$.

3.4.2 Cache-oblivious Directed SSSP

In this section we describe a cache-oblivious implementation of Dijkstra's directed SSSP algorithm [43] with a regular buffer heap used as a priority queue. Additionally, we use a cache-oblivious *Buffered Repository Tree*[1] (BRT) described in [11], in order to prevent any vertex whose shortest distance from the source vertex has already been determined, from being reinserted into the priority queue. A BRT maintains $\mathcal{O}(m)$ elements with keys in the range $[1 \ldots n]$ under the operations *Insert*(v, u) and *Extract*(u). An *Insert*(v, u) operation inserts a new element v with key u into the BRT, while an *Extract*(u) operation reports and deletes from the data structure

[1]Buffered Repository Trees have been used for breadth-first search and depth-first search in the cache-aware setting in [25] and in the cache-oblivious setting in [11]

all elements v with key u. The *Insert* and *Extract* operations are supported in $\mathcal{O}\left(\frac{1}{B}\log_2 n\right)$ and $\mathcal{O}\left(\log_2 n\right)$ amortized cache-misses, respectively (or in $\mathcal{O}\left(\frac{1}{B}\log_2 \frac{n}{B}\right)$ and $\mathcal{O}\left(\log_2 \frac{n}{B}\right)$ amortized cache-misses, respectively, assuming a tall cache).

The resulting cache-oblivious implementation of Dijkstra's algorithm is given in Function 3.4.2 (DIRECTED-SSSP).

FUNCTION **3.4.2.** DIRECTED-SSSP(G, w, s, d)

[Given a directed graph G with vertex set V (each vertex is identified with a unique integer in $[1, |V|]$), edge set E, a weight function $w : E \rightarrow \Re$ and a source vertex $s \in V$, this function cache-obliviously computes the shortest distance from s to each vertex $v \in V$ and stores it in $d[v]$.**]**

1. ***for*** each $v \in V$ ***do***
   $L_v \leftarrow \{\ u \mid (u, v) \in E\ \}$ $\{L_v$ *is the set of vertices from which* v *has an incoming edge*$\}$
   $L'_v \leftarrow \{\ \langle u, w(v, u)\rangle \mid (v, u) \in E\ \}$ $\{L'_v$ *is the set of vertices to which* v *has an outgoing edge*$\}$
   sort the items in both L_v and L'_v by vertex number

2. perform the following initializations:

   (i) $Q \leftarrow \emptyset,\ D \leftarrow \emptyset$ $\{Q$ *is a regular buffer heap that contains items of the form* (x, k_x) *and* D *is a BRT capable of containing key values in the range* $[1 \ldots |V|]\}$
   (ii) ***for*** each $v \in V$ ***do*** $d[v] \leftarrow +\infty$
   (iii) DECREASE-KEY$_{(Q)}$(s, 0) $\{$*insert vertex* s *with key (i.e., distance) 0 into* $Q\}$

3. ***while*** $Q \neq \emptyset$ ***do***

   (i) $(u, k) \leftarrow$ DELETE-MIN$_{(Q)}$( ), $d[u] \leftarrow k$ $\{k$ *is the shortest distance from* s *to* $u\}$
   (ii) $L''_u \leftarrow$ EXTRACT$_{(D)}$(u) $\{$*set of settled vertices to which* u *has an outgoing edge*$\}$
   sort L''_u by vertex number
   (iii) scan L'_u and L''_u simultaneously and ***for*** each $v \in L'_u$ such that $v \notin L''_u$ ***do***
   DECREASE-KEY$_{(Q)}$(v, $k + w(u, v)$) $\{$*relax edge* (u, v) *to the yet-to-settle vertex* $v\}$
   (iv) ***for*** each $v \in L_u$ ***do***
   INSERT$_{(D)}$(u, v) $\{$*mark neighbor* u *of* v *as settled*$\}$

DIRECTED-SSSP ENDS

Correctness. A standard implementation of Dijkstra's directed SSSP algorithm is through the use of a priority-queue Q with *Decrease-Key*. Priority-queue Q stores all vertices that are not yet *settled* (i.e., vertices whose shortest path length from the source vertex has not yet been finalized), and in each iteration of the algorithm, a vertex u is extracted from Q with a *Delete-Min* operation. The vertex u is provably settled at this point, and for each edge (u, v) such that v is not settled, i.e., such that v is on Q, a suitable *Decrease-Key* operation is performed on v in Q.

Our implementation of Dijkstra's algorithm (DIRECTED-SSSP) differs from the standard implementation in two ways, both with an eye to improving cache-efficiency. Firstly, we use a regular buffer heap instead of a standard priority queue. Secondly, instead of accessing a vertex directly in order to determine whether it is settled or not, we use a BRT D to perform these operations cache-efficiently and thus avoid a potential cache-miss during each such operation.

Since we have already proved the correctness of buffer heap (see Lemma 3.3.1), if we simply replace Q with a regular buffer heap in the standard implementation of Dijkstra's algorithm the implementation will still be correct. For $i \in [1, n]$, let u'_i denote the i-th vertex extracted from the priority queue in this implementation, and let V'_i be the set of vertices on which *Decrease-Key* operations are performed immediately after this extraction. Let u_i and V_i have similar definitions for DIRECTED-SSSP. Therefore, assuming the correctness of BRT operations (see [11]), correctness of DIRECTED-SSSP will follow if we can prove the following claim.

Claim 3.4.1. *For $i \in [1, n]$, $u_i = u'_i$ and $V_i = V'_i$.*

Proof. Let $S_i = \{ u_j \mid 1 \leq j \leq i \}$ for $i \in [0, n]$. Then clearly $V'_i = \{ v \mid (u'_i, v) \in E \wedge v \notin S_{i-1} \}$.

Since $u_1 = u'_1 = s$ and D is initially empty, the claim trivially holds for $i = 1$. Now suppose it holds up to some value $j \in [0, n-1]$ of i. We will show that it holds for $i = j + 1$.

Since the claim holds for all $i \leq j$, immediately before the extraction of the $(j+1)$-th vertex from the priority queue, the state of the priority queue in both implementations, i.e., the standard implementation with buffer heap and DIRECTED-SSSP, are exactly the same. Hence, $u_{j+1} = u'_{j+1}$.

Let U_{j+1} be the set of vertices extracted from D in iteration $j + 1$ of the ***while*** loop in DIRECTED-SSSP. Since the claim was true up to iteration j, for each $v \in S_j$ with $(u_{j+1}, v) \in E$, an element u_{j+1} with key value v was inserted into D in step 3(iv) at some point during the first j iterations. Hence, $U_{j+1} \supseteq \{ v \mid (u_{j+1}, v) \in E \wedge v \in S_j \}$. Again since D was initially empty and only settled vertices insert items into it, $U_{j+1} = \{ v \mid (u_{j+1}, v) \in E \wedge v \in S_j \}$. Therefore, $V_{j+1} = \{ v \mid (u_{j+1}, v) \in E \} \setminus U_{j+1} = V'_{j+1}$.

Hence, the claim holds for all $i \in [1, n]$. ■

Therefore, DIRECTED-SSSP is a correct implementation of Dijkstra's algorithm.

Cache Complexity. The following lemma gives the cache-complexity of DIRECTED-SSSP.

Lemma 3.4.1. *Single source shortest paths in a directed graph can be computed cache-obliviously in $\mathcal{O}\left((n + \frac{m}{B}) \cdot \log_2 \frac{m}{B}\right)$ cache-misses using a buffer heap under the tall cache assumption.*

Proof. In step 1, all sets L_v and L'_v can be generated with their items in appropriately sorted order after a constant number of sorting and scanning phases incurring $\mathcal{O}\left(n + \frac{m}{B}\log_2\frac{m}{B}\right)$ cache-misses.

In step 3, the algorithm performs n *Delete-Min* and m *Decrease-Key* operations on Q, and n *Extract* and m *Insert* operations on D incurring $\mathcal{O}\left(\frac{m+n}{B}\log_2\frac{n}{M}\right)$ and $\mathcal{O}\left(n\log_2\frac{n}{B} + \frac{m}{B}\log_2\frac{n}{B}\right)$ cache-misses, respectively. All lists in step 3(ii) can be sorted in $\mathcal{O}\left(\frac{m}{B}\log_2\frac{n}{B}\right)$ cache-misses in total, and the total cache-misses incurred by all scans in steps 3(iii) and 3(iv) is $\mathcal{O}\left(n + \frac{m}{B}\right)$.

Therefore, overall cache complexity of Directed-SSSP is $\mathcal{O}\left(\left(n + \frac{m}{B}\right)\cdot\log_2\frac{m}{B}\right)$. ∎

Directed SSSP with Cache-oblivious Tournament Tree. In Appendix A we present the *cache-oblivious tournament tree* (COTT) which supports the same set of operations (*Delete*, *Delete-Min* and *Decrease-Key*) as the buffer heap. Although COTT has weaker bounds than buffer heap, it is a simpler data structure, and can be used instead of buffer heap in the directed SSSP algorithm to achieve the same level of cache-efficiency as with buffer heap.

3.4.3 Cache-aware Undirected APSP

In this section we introduce a compound priority queue data structure based on slim buffer heap, called the *Multi-Buffer-Heap* (MBH), and use this structure for cache-efficient computation of APSP on an undirected graph with general non-negative edge-weights.

A multi-buffer-heap is constructed as follows. Let $\lambda < B$ and let $L = \frac{B}{\lambda}$. We pack the slim caches of $\Theta(L)$ slim buffer heaps $SBH(\lambda)$ into a single cache block. We call this block the *multi-slim-cache* and the resulting structure a *multi-buffer-heap*. By the analysis in section 3.3.3 this structure supports *Delete*, *Delete-Min* and *Decrease-Key* operations on each of its component slim buffer heaps in $\mathcal{O}\left(\frac{L}{B} + \frac{1}{B}\log_2\frac{NL}{B}\right)$ amortized cache-misses each.

For computing APSP we take the approach described in [13]. It solves APSP by working on all n underlying SSSP problems simultaneously, and each individual SSSP problem is solved using Kumar & Schwabe's algorithm for weighted undirected graphs [83]. For $1 \le i \le n$, this approach requires a priority queue pair (Q_i, Q'_i), where the i-th pair belongs to the i-th SSSP problem. These n priority queue pairs are implemented using $\Theta(\frac{n}{L})$ multi-buffer-heaps. The algorithm proceeds in n rounds. In each round it loads the multi-slim-cache of each MBH, and for each MBH extracts a settled vertex with minimum distance from each of the $\Theta(L)$ priority queue pairs it stores. It sorts the extracted vertices by vertex indices. It then scans this sorted vertex list and the sorted sequences of adjacency lists in parallel to retrieve the adjacency lists of the settled vertices of this round. Another sorting phase moves all adjacency lists to be applied to the same MBH together. Then all necessary *Decrease-Key* operations are performed by cycling through the multi-buffer-heaps once again. At the end of the algorithm the extracted vertices along with their computed distance values are sorted to produce the final distance matrix.

Cache Complexity. In each round the multi-slim-caches of all multi-buffer-heaps are loaded into the cache in $\mathcal{O}\left(\frac{n}{L}\right)$ cache-misses. Accessing all required adjacency lists over $\mathcal{O}(n)$ rounds incurs $\mathcal{O}(n \cdot sort(m))$ cache-misses, and a total of $\mathcal{O}\big(mn \cdot \left(\frac{1}{\lambda} + \frac{1}{B}\log_2 \frac{n}{\lambda}\right)\big)$ cache-misses are incurred by all $\mathcal{O}(mn)$ priority queue operations performed by this algorithm. The final distance matrix can be sorted in $\mathcal{O}(n \cdot sort(n))$ cache-misses. Thus the total cache complexity of this algorithm is $\mathcal{O}\Big(n \cdot \Big(\frac{n}{L} + \frac{m}{\lambda} + \frac{m}{B}\log_2 \frac{n}{\lambda} + sort(m)\Big)\Big)$. Using $L = \sqrt{\frac{nB}{m}} \geq 1$, we obtain the following:

Lemma 3.4.2. *Using multi-buffer-heaps, APSP on undirected graphs with non-negative real edge weights can be solved in* $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}} + sort(m)\right)\right)$ *cache-misses and* $\mathcal{O}\left(n^2\right)$ *space when* $m \leq \frac{nB}{(\log n)^2}$.

In conjunction with the cache-efficient APSP algorithm for sufficiently dense graphs implied by the SSSP results in [83, 32] we obtain the following corollary.

Corollary 3.4.1. *APSP on an undirected graph with non-negative real edge weights can be solved in* $\mathcal{O}\left(n \cdot \left(\sqrt{\frac{mn}{B}} + \frac{m}{B}\log\frac{n}{B}\right)\right)$ *cache-misses and* $\mathcal{O}\left(n^2\right)$ *space. The number of cache-misses is reduced to* $\mathcal{O}\left(\frac{mn}{B}\log\frac{n}{B}\right)$ *when* $m \geq \frac{nB}{\left(\log\frac{n}{B}\right)^2}$.

3.5 Conclusion

In this chapter we presented the buffer heap, the first cache-oblivious priority queue that supports *Decrease-Key* operations and used it to obtain the first cache-oblivious SSSP algorithms for weighted undirected and directed graphs, and an improved cache-aware APSP algorithm for weighted undirected graphs. All our cache-oblivious results match the cache complexity of their best cache-aware counterparts. However, open questions still remain. For example:

1. The only known lower bound on the cache complexity of cache-oblivious priority queue operations is $\Omega\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{M}\right)$ amortized which is trivially derived from the sorting lower bound. The buffer heap improves the upper bound from trivial $\mathcal{O}(\log N)$ to $\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$ amortized. But there is still a gap between this new upper bound and known lower bound. An open problem is to eliminate this gap.

2. The known cache-miss lower bound for the SSSP problem is $\Omega\left(\frac{m}{n} \cdot sort(n)\right)$ [92]. Though our SSSP algorithms improve significantly over known upper bounds, they are not known to be optimal.

- The n term in the cache complexity of our cache-oblivious undirected SSSP algorithm results from unstructured accesses to adjacency lists. Though some progress has been made in reducing this overhead for bounded-weight graphs [7, 89], nothing is known for graphs with general edge-weights.
- The $n \log n$ term in the cache complexity of our cache-oblivious directed SSSP algorithm results from the overhead of remembering visited vertices. Perhaps a completely new technique for handling this problem will be able to reduce this overhead significantly.

3. The $n\sqrt{\frac{mn}{B}}$ term in the cache complexity of the weighted undirected APSP algorithm described in Section 3.4.3 arises from unstructured accesses to adjacency lists. Though we show in Chapter 5 that we can get rid of this term completely for unweighted undirected graphs, achieving the same for weighted graphs still remains an open question.

Chapter 4

Experimental Results: Priority Queues for Efficient SSSP Computation

Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.
(Isaac Asimov)

A key ingredient in a cache-efficient implementation of Dijkstra's single-source shortest path (SSSP) algorithm is the cache-efficiency of the priority-queue it uses. In this chapter we report the results of an experimental study on how the cache-efficiency of priority queues affects the performance of Dijkstra's SSSP algorithm.

We implemented two different versions of the cache-oblivious buffer heap and used them in three slightly different versions of Dijkstra's algorithm. We compared the performance of these three algorithms with the performance of Dijkstra's algorithm using several traditional priority queues, as well as some highly optimized cache-aware priority queues.

Our experimental results suggest that on low-diameter real-weighted sparse graphs (i.e., $\mathcal{G}_{n,m}$ and power-law), a streamlined version of buffer heap is a better choice than any other priority queue we used in our experiments, except for the highly optimized cache-aware sequence heap which runs faster. However, our streamlined buffer heap which we call auxiliary buffer heap, is cache-oblivious and also simpler to implement than sequence heap. When the computation is out-of-core, an external-memory version of Dijkstra's algorithm implemented with a buffer heap and an auxiliary buffer heap performs the best.

For high-diameter graphs of geometric nature such as real-world road networks, Dijkstra's algorithm with traditional priority queues perform almost as well as any cache-efficient priority queue when the computation is in-core.

4.1 Introduction

Dijkstra's single-source shortest path (SSSP) algorithm is the most widely used algorithm for computing shortest paths in practice, and virtually all other SSSP algorithms are directly or indirectly based on Dijkstra's algorithm. A key ingredient in the efficient execution of Dijkstra's algorithm is the efficiency of the priority queue it uses. In this chapter we present an experimental study on how the cache-efficiency of priority queues affects the performance of several cache-efficient and traditional implementations of Dijkstra's algorithm.

We consider the following three implementations of Dijkstra's algorithm.

Dijkstra-Dec. The standard implementation of Dijkstra's algorithm that uses a priority queue with *Decrease-Key* operations (see Function B.0.2 in the Appendix).

Dijkstra-NoDec. An implementation that uses a priority queue with only *Insert* and *Delete-Min* operations (see Function B.0.3 in the Appendix).

Dijkstra-Ext. An external-memory implementation for undirected graphs that uses two priority queues: one with *Decrease-Key* operations, and the other one supporting only *Insert* and *Delete-Min* operations (see Function 3.4.1 in Chapter 3).

We also include the following Dijkstra implementation which was used as the benchmark solver for the "*9th DIMACS Implementation Challenge – Shortest Paths*" [1].

Dijkstra-Buckets. An implementation of Dijkstra's algorithm with a priority queue based on a bucketing structure. This algorithm works only on graphs with integer edge-weights.

We implemented the following cache-oblivious priority queues.

Buffer Heap (cache-oblivious). Our cache-oblivious priority queue with *Decrease-Key* operations introduced in Chapter 3.

Auxiliary Buffer Heap (cache-oblivious). A streamlined version of buffer heap that supports only *Insert* and *Delete-Min* operations.

We also coded the following two traditional priority queues with *Decrease-Keys*.

Standard Binary Heap [132]. Perhaps the most widely used priority queue.

Pairing Heap [50]. Considered to be the most efficient pointer based priority queue in practice and very simple to implement.

The above two priority queues were studied in [90] along with several others in the context of implementing the Prim-Dijkstra MST algorithm which has the same structure as Dijkstra's SSSP algorithm, and pairing heap was always found to be superior to all other priority queues considered in that experiment while standard binary heap performed better than the remaining priority queues in most cases.

We include the following two highly optimized array-based priority queues implemented by Peter Sanders [109]. They do not support *Decrease-Key* operations.

Bottom-up Binary Heap [131]. This is a highly optimized binary heap implementation that uses a bottom-up *Delete-Min* heuristic. The code is optimized not to have any redundant memory accesses or computations even in its assembler code.

Aligned 4-ary Heap [84] (cache-aware). This implementation also uses a bottom-up *Delete-Min* heuristic, and data is aligned to cache blocks to reduce cache-misses. It is a cache-aware priority queue.

It has been shown in [84] that array-based priority queues like the binary heap and the 4-ary heap outperform pointer-based heap data structures on modern architectures.

Finally, we include the following highly optimized cache-aware priority queue implemented by Peter Sanders [109]. It has been shown in [109] that it outperforms both bottom-up binary heap and the aligned 4-ary heap.

Sequence Heap [109] (cache-aware). This priority queue is based on a multi-way merging technique like other external-memory priority queues, but highly optimized to have excellent performance in cached memory.

We performed our experiments on undirected $\mathcal{G}_{n,m}$ and directed power-law graphs, as well as on some real-world graphs from the benchmark instances of the 9th DIMACS Implementation Challenge [1].

The experiments in this chapter were performed in collaboration with undergraduate students Lingling Tong [122], David Lan Roche [85] and Mo Chen.

4.1.1 Summary of Experimental Results

We describe the highlights of our experimental results in Section 4.5. Briefly here are the conclusions of our experimental study:

- For low-diameter real-weighted sparse graphs such as $\mathcal{G}_{n,m}$ and power-law:
  - When the computation was in-core, the "no Decrease-Key" variant of Dijkstra's algorithm (i.e., DIJKSTRA-NODEC) with an auxiliary buffer heap as a priority queue ran faster than all other implementations with traditional priority queues including the DIMACS solver (DIJKSTRA-BUCKETS). However, DIJKSTRA-NODEC implemented with the highly optimized cache-aware sequence heap ran 20-35% faster than the auxiliary buffer heap implementation.

- – When the computation was out-of-core, the external-memory implementation of Dijkstra's algorithm (DIJKSTRA-EXT) with the buffer heap and the auxiliary buffer heap as priority queues performed the best.

- For high-diameter graphs of geometric nature such as real-world road networks, Dijkstra's algorithm with traditional priority queues performed almost as well as any cache-efficient priority queue when the computation was in-core.

4.1.2 Organization of the Chapter

In Section 4.2 we give an overview of the priority queues we used in our experiments and in Section 4.3 the three different ways in which Dijkstra's algorithm was run using these priority queues. We discuss our experimental setup in Section 4.4, and in Section 4.5 we discuss our experimental results.

Priority Queue	*Insert/Decrease-Key*	*Delete*	*Delete-Min*
Standard Binary Heap [132] (worst-case bounds)	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Two-Pass Pairing Heap [50, 97]	$\mathcal{O}\left(2^{2\sqrt{\log\log N}}\right)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Buffer Heap (Chapter 3)	$\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$	$\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$	$\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$

Table 4.1: Amortized I/O bounds for priority queues with *Decrease-Keys* (N = # items in queue, B = block size, M = size of the cache/internal-memory).

Priority Queue	*Insert/Delete-Min*
Bottom-up Binary Heap [131] (worst-case bounds)	$\mathcal{O}(\log_2 N)$
Aligned 4-ary Heap [84] (worst-case bounds)	$\mathcal{O}(\log_4 N)$
Sequence Heap [109] (cache-aware)	$\mathcal{O}\left(\frac{1}{B}\log_k\frac{N}{m}\right)$ where, $k = \Theta(M/B)$ and $m = \Theta(M)$
Auxiliary Buffer Heap (cache-oblivious, this chapter)	$\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$

Table 4.2: Amortized I/O bounds for priority queues without *Decrease-Keys* (N = # items in queue, B = block size, M = size of the cache/internal-memory).

4.2 Overview of Priority Queues

I/O complexities of all priority queues in our experiments are listed in Tables 4.1 and 4.2.

4.2.1 Internal-Memory Priority Queues

We implemented standard binary heap and pairing heap. Both were implemented so that they allocate and deallocate space from at most two arrays. Pointers were reduced to indices to the array. This allows us to limit the amount of internal-memory available to the priority queue during out-of-core computations using STXXL (see Section 4.4).

Standard Binary Heap

We implemented the standard binary heap [132, 37]. However, we allocated nodes from a separate array and the heap stores only indices to those nodes. This ensures that whenever a new element is inserted all necessary information about it is stored at a location which remains static as long as the element remains in the heap. The pointer/index to this location is returned to the calling application (i.e., the SSSP algorithm). For future updates (i.e., *Decrease-Key* operations) on the element the calling application supplies this index to the binary heap data structure so that the data structure can perform the update by accessing all necessary information on that element directly.

Standard binary heap supports *Insert*, *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}(\log N)$ time and I/O bounds each.

Pairing Heap

We implemented two variants of the pairing heap: two-pass and multi-pass [50]. These two implementations differ only in the way the *Delete-Min* operation is performed. The two-pass variant has better theoretical bounds than the multi-pass variant and also ran faster in our experiments. We have also implemented the auxiliary two-pass and the auxiliary multi-pass variants of the pairing heap [115]. In our experiments the auxiliary variants of the pairing heap ran only marginally (around 4%) faster than the corresponding primary variants, and so we will report results only for two-pass pairing heap.

The two-pass pairing heap supports *Delete* and *Delete-Min* operations in $\mathcal{O}(\log N)$ and *Decrease-Key* operations in $\mathcal{O}\left(2^{\sqrt{\log\log N}}\right)$ amortized time and I/O bounds each. The two-pass variant also has a known amortized lower bound of $\Omega(\log\log N)$ time and I/O for the *Decrease-Key* operation.

Bottom-up Binary Heap

The *bottom-up binary heap* is a variant of binary heap which uses a bottom-up *Delete-Min* heuristic [131]. Compared to the traditional binary heap implementation, this variant performs only half the number of comparisons on an average during a *Delete-Min* operation.

We used the highly optimized implementation of bottom-up binary heap by Peter Sanders [109] that has no redundant memory accesses or computations even in its assembler code. This implementation supports only *Insert* and *Delete-Min* operations.

4.2.2 Cache-aware Priority Queues

Both of the following two priority queues support only *Insert* and *Delete-Min* operations.

Aligned 4-ary Heap

The *aligned 4-ary heap* is an array-based priority queue like the binary heap, and in [84] it was shown to outperform pointer based heap data structures. This is a cache-aware priority queue that needs to know the block size B of the cache, and aligns its data to cache blocks. The data alignment ensures that accessing any data element incurs at most one cache-miss. It supports *Insert* and *Delete-Min* operations in $\mathcal{O}(\log_4 N)$ worst-case time and $\mathcal{O}(\log_4 N)$ amortized block transfers each. In our experiments we used the optimized implementation of aligned 4-ary heap by Peter Sanders with the bottom-up *Delete-Min* heuristic [131].

Sequence Heap

The *sequence heap* is a cache-aware priority queue developed by Peter Sanders [109]. The priority queue is based on k-way merging for some appropriate k. When the cache is fully associative, k is chosen to be $\Theta\left(\frac{M}{B}\right)$, and for some $m = \Theta(M)$ and $R = \lceil \log_k \frac{N}{m} \rceil$, it can perform N *Insert* and up to N *Delete-Min* operations in $\frac{2R}{B} + \mathcal{O}\left(\frac{1}{k} + \frac{\log k}{m}\right)$ amortized cache-misses and $\mathcal{O}(\log N + \log R + \log m + 1)$ amortized time each. For α-way set associative caches k is reduced by $\mathcal{O}\left(B^{\frac{1}{\alpha}}\right)$.

We used Peter Sanders' own highly optimized implementation of the sequence heap accompanied with the paper [109]. In this implementation the k-way merging procedure is implemented using a *loser tree* [81]. The sequence heap maintains a small bottom-up binary heap known as the *insertion heap*, for quick insertion into the data structure. When the insertion heap gets full it is merged with the main heap. A small sorted *deletion buffer* is also maintained for quick deletion from the data structure. Peter Sanders suggests that 128 is a good value for k for most current architectures, and we used this value in our experiments.

4.2.3 Cache-oblivious Buffer Heap and Auxiliary Buffer Heap

Buffer Heap

In our implementation of the buffer heap we made several design choices which do not necessarily guarantee the best worst-case behavior but perform reasonably well in practice. One of our major goals was to keep the implementation as simple as possible.

Size of Update Buffer. Our preliminary experiments suggested that the overhead of maintaining the theoretical invariant on the sizes of update buffers slows down the operations by about 50%. Therefore, we chose to keep the sizes of update buffers unrestricted. This increases the amortized time per operation to $\mathcal{O}\left(\frac{1}{B}\log_2 N\right)$ (up from $\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$).

Periodic Reconstruction. Theoretically, periodic reconstruction ensures that the I/O complexity of buffer heap operations depends on the number of items currently in the data structure and not on the total number of operations performed on it. However, in our preliminary experiments the overhead of periodic reconstruction slowed down the buffer heap operations by about 33%. Therefore, we chose not to reconstruct the data structure periodically.

Sorting U_0. We used randomized quicksort [73, 37] to sort the contents of U_0 when a *Delete-Min* operation is performed. Randomized quicksort is simple, in-place and with high probability runs in $\mathcal{O}(N \log N)$ time and performs $\mathcal{O}\left(\frac{N}{B}\log_2 N\right)$ I/O operations on a sequence of N elements and thus does not degrade the asymptotic performance of buffer heap. We chose not to use optimal cache-oblivious sorting algorithms [52] for the purpose since they are complicated to implement, have more overhead and are not essential in order to guarantee the I/O bounds of buffer heap.

Selection Algorithm for Redistribution. A selection algorithm is required to partition the elements collected for redistribution after a *Delete-Min* operation is performed. We used the classical randomized selection algorithm [72, 37] (the one typically used in randomized quicksort) which is in-place and runs in $\Theta(N)$ expected time and performs $\Theta\left(\frac{N}{B}\right)$ expected I/O operations on a sequence of N elements.

Single Stack Implementation. We stored all buffers in a single stack with the update buffer on top of the element buffer of the same level. All temporary space was allocated on top of the stack. Using a single stack instead of multiple arrays has the advantage that it can benefit more from the limited number of prefetchers available for the caches. This also allowed us to limit the amount of internal-memory available to the data structure during our out-of-core experiments using STXXL (see Section 4.4).

Pipelining. For each level i we visit during a *Delete-Min* operation we first merge the segments of U_i and then apply this merged sequence on B_i (see Section 4.2.3). In our implementation of buffer heap we pipelined the output of the merge phase directly to the application phase which saved several extra scans over the updates.

Detailed description of most of our design choices for the buffer heap is available in the undergraduate honors thesis of Lingling Tong [122].

Auxiliary Buffer Heap

We designed the *auxiliary buffer heap*, which is a streamlined version of the buffer heap that supports only *Insert* and *Delete-Min* operations. The amortized I/O complexity of each operation in our implementation is $\mathcal{O}\left(\frac{1}{B}\log_2 N\right)$ and these operations have low overhead compared to buffer heap operations. Major features of this implementation are as follows.

No Selection. In auxiliary buffer heap the contents of all buffers are kept sorted by key value instead of element id and time stamp as in buffer heap. As a result during the redistribution step after a *Delete-Min* operation we do not need to use a selection algorithm to partition the elements which saves a constant factor in I/O complexity.

Insertion and Delete-Min Buffers. We maintain two small constant size buffers: one for insertions and the other one for deletions (i.e., *Delete-Mins*). Whenever a new element is inserted into the priority queue we simply collect it in the *insertion buffer*. The *delete-min buffer* holds the smallest few elements in the priority queue (not considering the elements in the insertion buffer) in sorted order. Whenever the insertion buffer gets full or a *Delete-Min* operation needs to be performed, the insertion buffer is sorted and all its elements with key value larger than the largest key in the delete-min buffer are pushed into the priority queue. The remaining elements in the insertion buffer are merged with the delete-min buffer. If the delete-min buffer becomes full in the process, the overflowing elements are inserted into the priority queue. If the deletion buffer becomes empty, we fill it up (not necessarily to its capacity) with elements from the priority queue during the next *Delete-Min* operation.

Efficient Merge. We use the optimized 3-way merge technique described in [109] for merging the contents of an update buffer with the (at most two) segments of the update buffer in the next higher level.

Less Space. Auxiliary buffer heap uses less space than buffer heap since it does not need to store timestamps with each element.

The Auxiliary buffer heap also incorporates all relevant optimizations that were applied on buffer heap.

Implementation	Base Routine	Priority Queue(s)	I/O Complexity
BinH	Dijkstra-Dec	Standard Binary Heap	$\mathcal{O}(m + (n + D) \cdot \log n)$
PairH	Dijkstra-Dec	Two-Pass Pairing Heap	$\mathcal{O}\left(m + n \cdot \log n + D \cdot 2^{2\sqrt{\log\log n}}\right)$
BH	Dijkstra-Dec	Buffer Heap	$\mathcal{O}\left(m + \frac{n+D}{B} \cdot \log(n + D)\right)$
FBinH	Dijkstra-NoDec	Bottom-up Binary Heap	$\mathcal{O}(m + (n + D) \cdot \log(n + D))$
Al4H	Dijkstra-NoDec	Aligned 4-ary Heap	$\mathcal{O}(m + (n + D) \cdot \log(n + D))$
SeqH	Dijkstra-NoDec	Sequence Heap	$\mathcal{O}\left(m + \frac{n+D}{B} \cdot \log_k \frac{n}{m}\right)$, where, $k = \Theta(M/B)$ and $m = \Theta(M)$
Aux-BH	Dijkstra-NoDec	Auxiliary Buffer Heap	$\mathcal{O}\left(m + \frac{n+D}{B} \cdot \log(n + D)\right)$
Dual-BH	Dijkstra-Ext (undirected graphs only)	Buffer Heap & Auxiliary Buffer Heap	$\mathcal{O}\left(n + \frac{n+m}{B} \cdot \log m\right)$
DIMACS	Dijkstra-Bucket (integer edge-weights)	Buckets with Caliber Heuristic	$\mathcal{O}(m + n)$ (expected)

Table 4.3: Different implementations of Dijkstra's algorithm evaluated in this chapter, where D ($\leq m$) is the number of *Decrease-Key*s performed by Dijkstra-Dec and B is the block size.

4.3 Choice of Algorithms for the SSSP problem

For our study we chose to consider only the performance of Dijkstra algorithm since it is the most widely used SSSP algorithm. We implemented three versions of Dijkstra's algorithm, the versions differing in the way the priority queue was used. More detailed descriptions of all three versions are given in Appendix B. Table 4.3 lists the I/O complexities of the versions we implemented.

> Dijkstra-Dec. This is the standard implementation of Dijkstra's algorithm using a priority queue that supports *Decrease-Key* operations. This method can also be used with a priority queue that does not support *Decrease-Key* but supports *Delete*, which takes a pointer to an element in the priority queue and deletes it.
>
> Dijkstra-NoDec. This is a slightly modified version of Dijkstra's algorithm that works with a priority queue that only supports *Delete-Min* and *Insert* operations. We implemented this with the priority queues in Table 4.2. We did

not implement the provably optimal cache-oblivious priority queues in [11, 22] since they appear to be more complicated to implement.

DIJKSTRA-EXT for *undirected graphs*. This is an implementation of Dijkstra's algorithm on undirected graphs that uses two external-memory priority queues, of which at least one supports the *Decrease-Key* operation (see Function 3.4.1 in Chapter 3). This gives the best theoretical I/O complexity for Dijkstra's algorithm on sparse undirected graphs.

We also included the following implementation of Dijkstra's algorithm which was used as the benchmark solver for the "*9th DIMACS Implementation Challenge – Shortest Paths*" [1].

DIJKSTRA-BUCKETS This implementation uses a priority queue based on a bucketing structure along with a heuristic to speed-up execution, and works only on graphs with integer edge-weights.

4.4 Experimental Set-up

We ran all our experiments on a dual processor 3.06 GHz Intel Xeon shared memory machine with 4 GB of RAM and running Ubuntu Linux 5.10 "Breezy Badger". Each processor had an 8 KB L1 data cache (4-way set associative) and a shared on-chip 512 KB unified L2 cache (8-way). For both caches the block transfer size (i.e., cache line size) was 64 bytes.

The machine was connected to a 73.5 GB 10K RPM Fujitsu MAP3735NC hard disk with an 8 MB data buffer. The average seek time for reads and writes were 4.5 and 5.0 ms, respectively. The maximum data transfer rate (to/from media) was 106.9 MB/s.

We used the *Cachegrind* profiler [112] for simulating cache effects. The machine was exclusively used for experiments (i.e., no other programs were running on it), and in the absence of explicit instructions for using both processors, our programs were automatically ran on a single processor.

We implemented all algorithms in C++ using a uniform programming style, and compiled using the g++ 3.3.4 compiler with optimization level -O3.

For out-of-core experiments we used STXXL library version 0.9. The STXXL library [40, 41] is an implementation of the C++ standard template library STL for external memory computations, and is used primarily for experimentation with huge data sets. The STXXL library maintains its own fully associative cache in RAM with pages from the disk. We compiled STXXL with DIRECT-I/O turned on, which ensures that the OS does not cache the data read from or written to the hard disk. We also configured STXXL (more specifically the STXXL vectors we used) to use LRU as the paging strategy.

We store the entire graph in a single vector so that the total amount of internal-memory available to the graph during out-of-core computations can be regulated by changing the STXXL parameters of the vector. The initial portion of the vector stores information on the vertices in increasing order of vertex id (each vertex is assumed to have a unique integer id from 1 to n) and the remaining portion stores the adjacency lists of the vertices in the same order. We store two pieces of information for each vertex: its distance value from the source vertex and a pointer to its adjacency list. For SSSP algorithms based on internal-memory priority queues with *Decrease-Keys* (e.g., standard binary heap and pairing heap) we also store the pointer returned by the priority queue when the vertex is inserted into it for the first time. This pointer is used by all subsequent *Decrease-Key* operations performed on the vertex. For each edge in the adjacency list of a vertex we store the other endpoint of the edge and the edge-weight. Each undirected edge (u, v) is stored twice: once in the adjacency list of u and again in the adjacency list of v. For each graph we use a one-time preprocessing step that puts the graph in the format described above.

Graph Classes Considered.

We ran our experiments on three classes of graphs. The synthetic graphs were generated using the generators (PR [98] and GT [15]) contributed by *9th DIMACS Implementation Challenge* participants [1, 15, 98].

Undirected $\mathcal{G}_{n,m}$ (PR [98]). The $\mathcal{G}_{n,m}$ distribution chooses a graph uniformly at random from all graphs with n labeled vertices and m edges [47]. Such a graph can be constructed by choosing m random edges with equal probability (and with replacement) from all possible $(n \times n - n)$ edges (the PR-generator avoids choosing self-loops).

Directed Power-Law Graphs (GT [15]). The GT-generator generates random graphs with power-law degree distributions and small-world characteristics using the *recursive matrix* (R-MAT) graph model [28]. The model has four non-zero parameters a, b, c and d with $a + b + c + d = 1$. Given the number of vertices n and the number of edges m, the GT-generator starts off with an empty $n \times n$ adjacency matrix for the graph, recursively divides the matrix into four quadrants, and distributes the edges to the top-left, top-right, bottom-left and bottom-right quadrants with probabilities a, b, c and d, respectively. It has been conjectured in [28] that many real-world graphs have $a : b \approx 3 : 1$, $a : c \approx 3 : 1$ and $a \geq d$, and accordingly we have used $a = 0.45$, $b = c = 0.15$ and $d = 0.25$ which are also the default values used by the GT-generator. The resulting graph is directed.

Undirected U.S. Road Networks ([111]). These are undirected weighted graphs representing the road networks of 50 U.S. states and the District of Columbia. Edge weights are given both as the spatial distance between the endpoints (i.e., the great circle distance in meters between the endpoints) and as the travel time between them (i.e., spatial distance divided by some average speed that depends on the road category). Merging all networks produces a graph containing about 24 million nodes and 29 million edges.

4.5 Experimental Results

Unless specified otherwise, all experimental results presented in this section are averages of three independent runs from three random sources on a randomly chosen graph from the graph class under consideration, and they do not include the cost of the one-time preprocessing step that puts the graph in the format described in Section 4.4. A brief overview of the results for $\mathcal{G}_{n,m}$ is as follows:

- When the computation is in-core Aux-BH runs consistently faster than all other implementations except for the highly optimized cache-aware SeqH. Though BH runs slower than all implementations based on DIJKSTRA-NODEC, it is the fastest among the implementations based on DIJKSTRA-DEC. Moreover, for graphs of fixed size, the denser the graph, the narrower the performance gap between BH and DIJKSTRA-NODEC-based implementations.

  On the other extreme, Dual-BH is the slowest of all implementations, typically running 8 to 16 times slower than all other implementations.

- The story is different for out-of-core computations. When the computation is fully external (i.e., neither the vertex set nor the priority queues completely fit in internal-memory), and the graph is not too dense, all three buffer heap based implementations (BH, Aux-BH and Dual-BH) run faster than all other implementations with Dual-BH running the fastest (we did not include SeqH in our out-of-core experiments due to the difficulty in making it compatible with STXXL).

  We must mention here that due to time constraints we performed our out-of-core experiments on small instances of $\mathcal{G}_{n,m}$ and kept the block size B and internal-memory size M artificially small (i.e., $B = 4$ KB and $M = 4$ MB). In practice, B and M have values in the range of megabytes and gigabytes, respectively, and we expect that for such practical values of B and M Dual-BH will handily beat all other implementations for edge densities that are likely to occur in sparse graphs that arise in practice. However, though Dual-BH is the fastest for out-of-core computations, it will still be very slow in a practical setting unless its theoretical I/O complexity is improved even further.

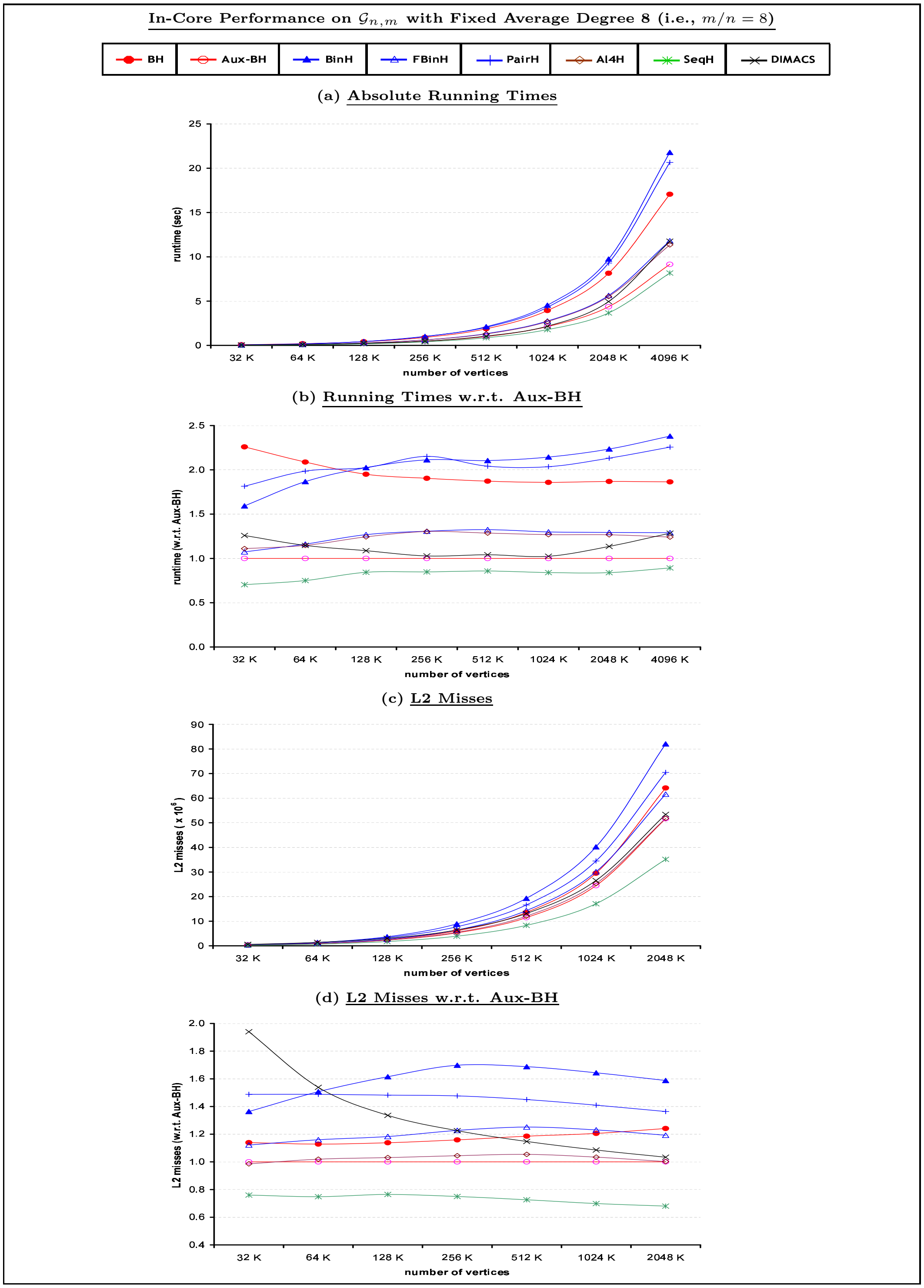

Figure 4.1: In-core performance of algorithms on $\mathcal{G}_{n,m}$ with fixed average degree 8.

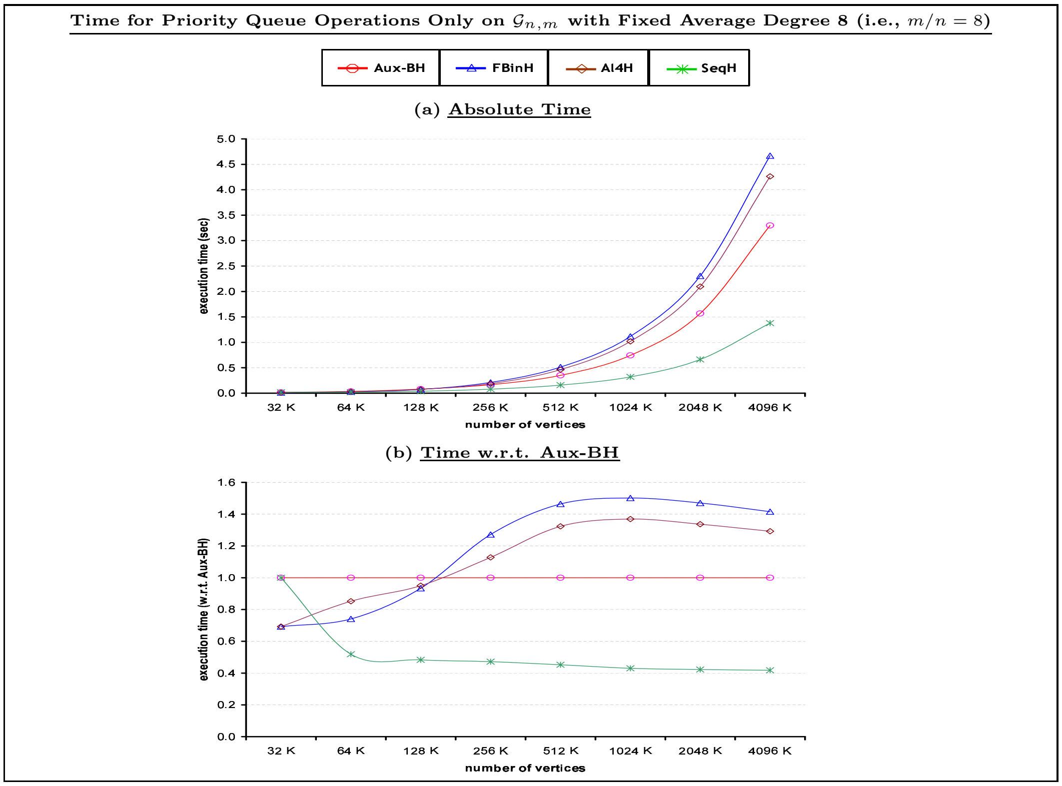

Figure 4.2: Time for priority queue operations only on $\mathcal{G}_{n,m}$ with fixed avg. degree 8.

4.5.1 In-Core Results for $\mathcal{G}_{n,m}$

We consider graphs in which we keep the average degree of vertices fixed and vary the number of edges (by varying the number of vertices), and also graphs in which we keep the number of edges fixed and vary the average degree of vertices (again by varying the number of vertices).

$\mathcal{G}_{n,m}$ with Fixed Average Degree

Figure 4.1 shows the in-core performance of all implementations (except Dual-BH which was much slower than all others in-core) on $\mathcal{G}_{n,m}$ with a fixed average degree 8, i.e., $\frac{m}{n} = 8$.

Running Times. Figures 4.1(a) and 4.1(b) plot the running times of the implementations as n is varied from 2^{15} to 2^{22}. In this range SeqH consistently performed better than all other implementations, and Aux-BH was consistently the second fastest. The SeqH implementation ran around 25% faster than Aux-BH. The DIMACS solver ran faster than the remaining implementations, and was up to 25% slower than Aux-BH. Both FBinH and Al4H ran at almost the same speed, and

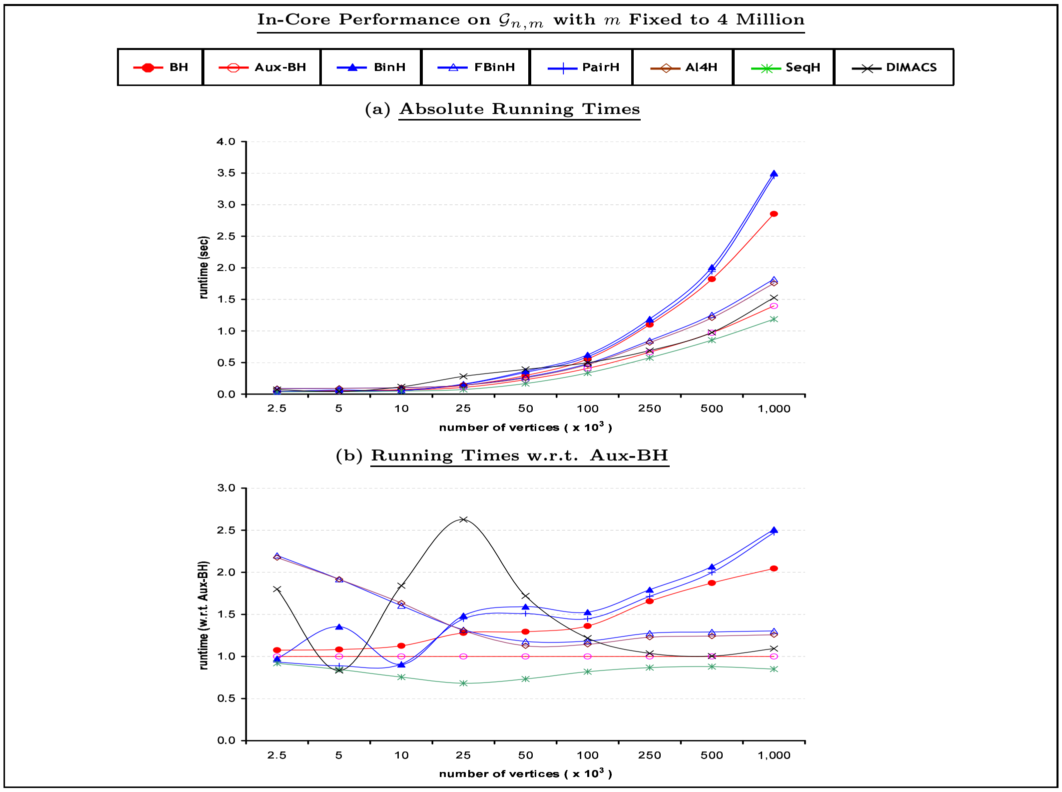

Figure 4.3: In-core performance of algorithms on $\mathcal{G}_{n,m}$ with m fixed to 4 million.

were consistently 25% slower than Aux-BH. The three implementations based on priority queues with *Decrease-Keys*, namely BH, BinH and PairH, ran at least 50% slower than all other implementations. Among these three implementations BH was the fastest for $n \geq 128\ K$, and run up to 25% faster than the remaining two. The slowest of all implementations was BinH.

Cache Performance. Figures 4.1(c) and 4.1(d) plot the L2 cache misses incurred by different implementations (except Dual-BH). As expected, cache-aware SeqH incurred the fewest cache-misses followed by Aux-BH. The BH implementation incurred more cache-misses than Aux-BH, but almost always fewer than the remaining implementations including FBinH which uses the highly optimized bottom-up binary heap.

Figure 4.1(d) shows that as n grows larger the cache performances of BH degrade with respect to BinH and PairH which can be explained as follows. In practice, both BinH and PairH support *Decrease-Key* operations more efficiently than *Delete-Min* operations, while for buffer heap that is not the case. Since all Dijkstra-Dec-based implementations perform exactly the same number of *Decrease-Key* operations (our experimental results suggest that this number is $\approx 0.8n$ for $\mathcal{G}_{n,m}$ with average

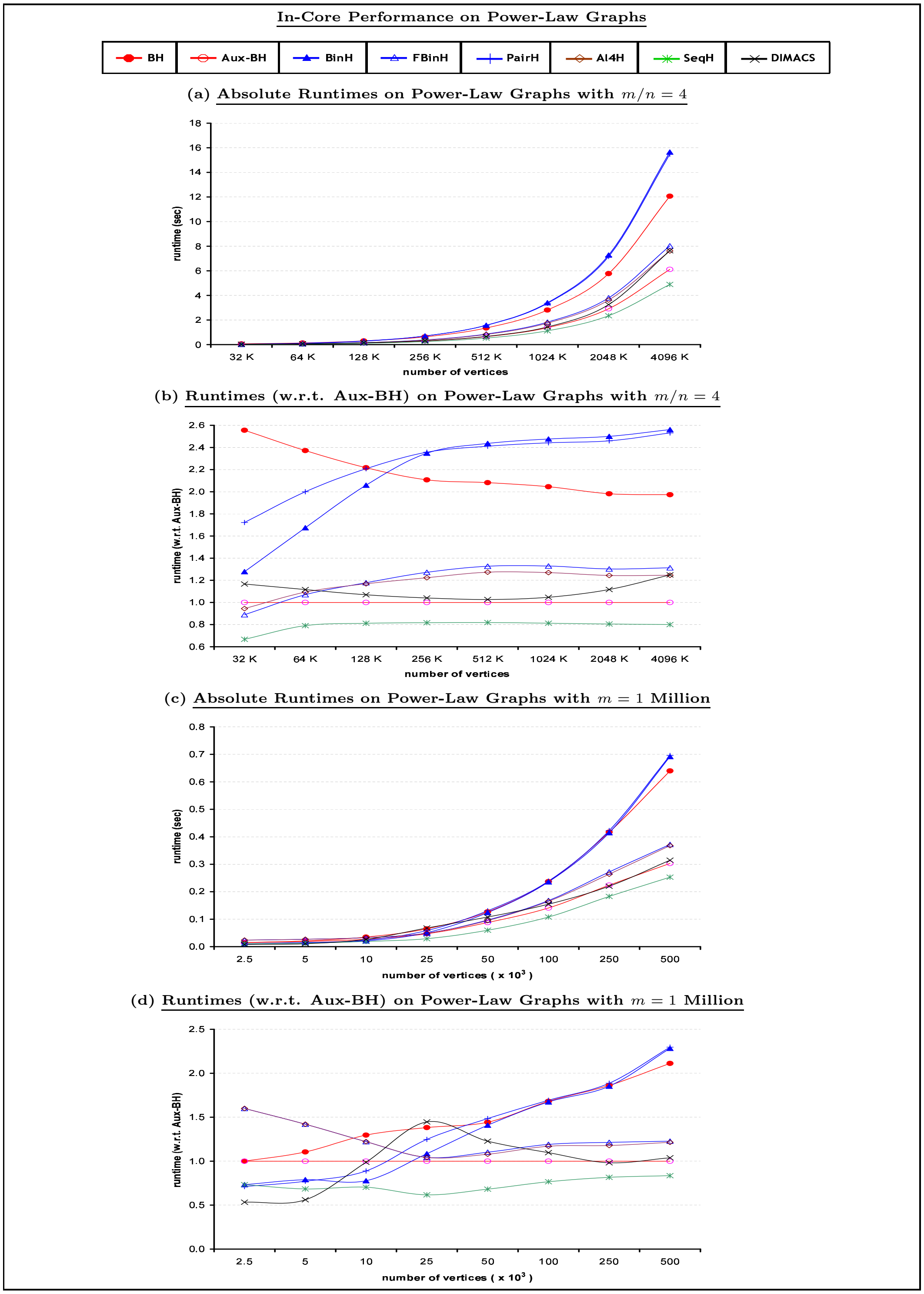

Figure 4.4: In-core performance of algorithms on power-law graphs.

degree 8), and with the increase of n the cache misses incurred by a *Decrease-Key* operation on a buffer heap increases at a more rapid rate than that on any internal-memory priority queue in our experiment, the cache-performance of BH degrades as a whole with respect to that of BinH and PairH as n increases. We believe that the cache behavior of Aux-BH can also be explained similarly. Surprisingly, however, Figure 4.1(b) shows that the running times of both BH and Aux-BH improve with respect to most other implementations as n increases. We believe this happens because of the prefetchers in Intel Xeon. As the operations of both buffer heap and the auxiliary buffer heap involve only sequential scans they benefit more from the prefetchers than the internal-memory heaps. The cachegrind profiler does not take hardware prefetching into account and as a result, Figures 4.1(c) and 4.1(d) failed to reveal their impact.

Priority Queue Performance Only. For Aux-BH, FBinH, Al4H and SeqH, Figures 4.2(a) and 4.2(b) plot the total time taken for executing only the priority queue operations while computing SSSP with DIJKSTRA-NODEC. We observe that sequence heap operations are more than 2 times faster than auxiliary buffer operations, and auxiliary buffer heap operations are about 40-50% faster than bottom-up binary heap and aligned 4-ary heap operations. However, since SSSP computation has other overheads (e.g., cache-misses due to unstructured accesses to adjacency lists), too, SSSP computation with sequence heap is only 25% faster than that with auxiliary buffer heap which, in turn, is only around 25% faster than SSSP computation with bottom-up binary heap and aligned 4-ary heap (see Figures 4.1(a) and 4.1(b)).

$\mathcal{G}_{n,m}$ with Fixed Number of Edges

Figures 4.3(a) and 4.3(b) plot running times as the number of vertices is increased from 2500 to 1 million while keeping m fixed to 4 million (i.e. average degree is decreased from 1600 down to 4). As before, SeqH consistently ran the fastest followed by Aux-BH. Though initially AuxBH ran only slightly faster than BH, its performance improved as n increased. This observation can be explained as follows. Table B.2 (in Appendix B) shows that DIJKSTRA-DEC (and hence BH) performs $2n + D$ priority queue operations and DIJKSTRA-NODEC (and thus Aux-BH) performs $2n + 2D$ priority queue operations, where D is the number of *Decrease-Key* operations performed by DIJKSTRA-DEC. However, empirical evidence (which we have not included in this dissertation) suggest that DIJKSTRA-DEC performs slightly more than $\mathcal{O}(n \log m)$ *Decrease-Key* and *Insert* operations in addition to n *Delete-Min* operations, that is, $D \approx \mathcal{O}(n \log m)$. Hence, for smaller n, the difference between the number of priority queue operations performed by BH and Aux-BH is also smaller, and the cost of other overheads (e.g., accessing the adjacency lists) dominate (see Table B.1 in Appendix B). As a result, the performance gap between BH and Aux-BH is narrower for smaller n. The relative performance of BH and

FBinH/Al4H can be explained similarly. As the average degree of the graph decreases down to 160, performance of the DIMACS solver degrades significantly, but after that its performance improves dramatically.

Remarks on Dual-BH. In all our in-core experiments, Dual-BH ran considerably slower than all other implementations which can be attributed to the fact that it performs significantly more priority queue operations compared to any of them (see Table B.2 in Appendix B). For example, in our experiments on $\mathcal{G}_{n,m}$ with average degree 8, Dual-BH consistently performed at least 6 times more priority queue operations than all other implementations.

4.5.2 In-Core Results for Power-Law Graphs

In Figures 4.4(a) and 4.4(b), we plot the running times of different implementations on power-law graphs with fixed average degree 4 as the number of vertices is varied. Figures 4.4(c) and 4.4(d) plot running times as the average degree of the graph is varied by keeping the number of edges fixed to 1 million. The trends in both cases are similar to those observed for $\mathcal{G}_{n,m}$ in Section 4.5.1.

4.5.3 Out-of-Core Results for $\mathcal{G}_{n,m}$

We report our experimental results for *fully external* computation where neither the vertex set nor the edge set completely fits in internal memory, and the priority queue is too large to fit in internal memory. We fixed the amount of internal memory available to store the graph and the priority queue separately by fixing the STXXL parameters of the corresponding vectors. We fixed the block size B to 4 KB and internal-memory size M to 4 MB. We have not included SeqH in these experiments because it was quite difficult to reimplement it in order to make it compatible with STXXL.

Figures 4.5(a) and 4.5(b) show I/O wait time of different implementations on $\mathcal{G}_{n,m}$ as the average degree of the graph is varied while keeping the number of edges fixed to 2 million. The Dual-BH which ran considerably slower than all other implementations in a similar in-core experiment (see Section 4.5.1), performed the best in the current experiment. As Table 4.3 shows Dual-BH is the most I/O-efficient implementation for sparse graphs, and in the current setting the block size B and the access latency of the external memory are large enough to hide the overheads of Dual-BH which were exposed during in-core experiments. When the graph is not too dense ($m/n < 40$), Dual-BH was slightly faster than both BH and Aux-BH, and around 30-40% faster than FBinH and A4lH. We believe this is due to their $\mathcal{O}\left(m\right)$ I/O overhead for accessing the graph data structure compared to only $\mathcal{O}\left(n + \frac{m}{B}\right)$ I/O operations performed by Dual-BH for the same (see Table B.1 in Appendix B). Figures 4.5(c) and 4.5(d) which plot the number of blocks transferred

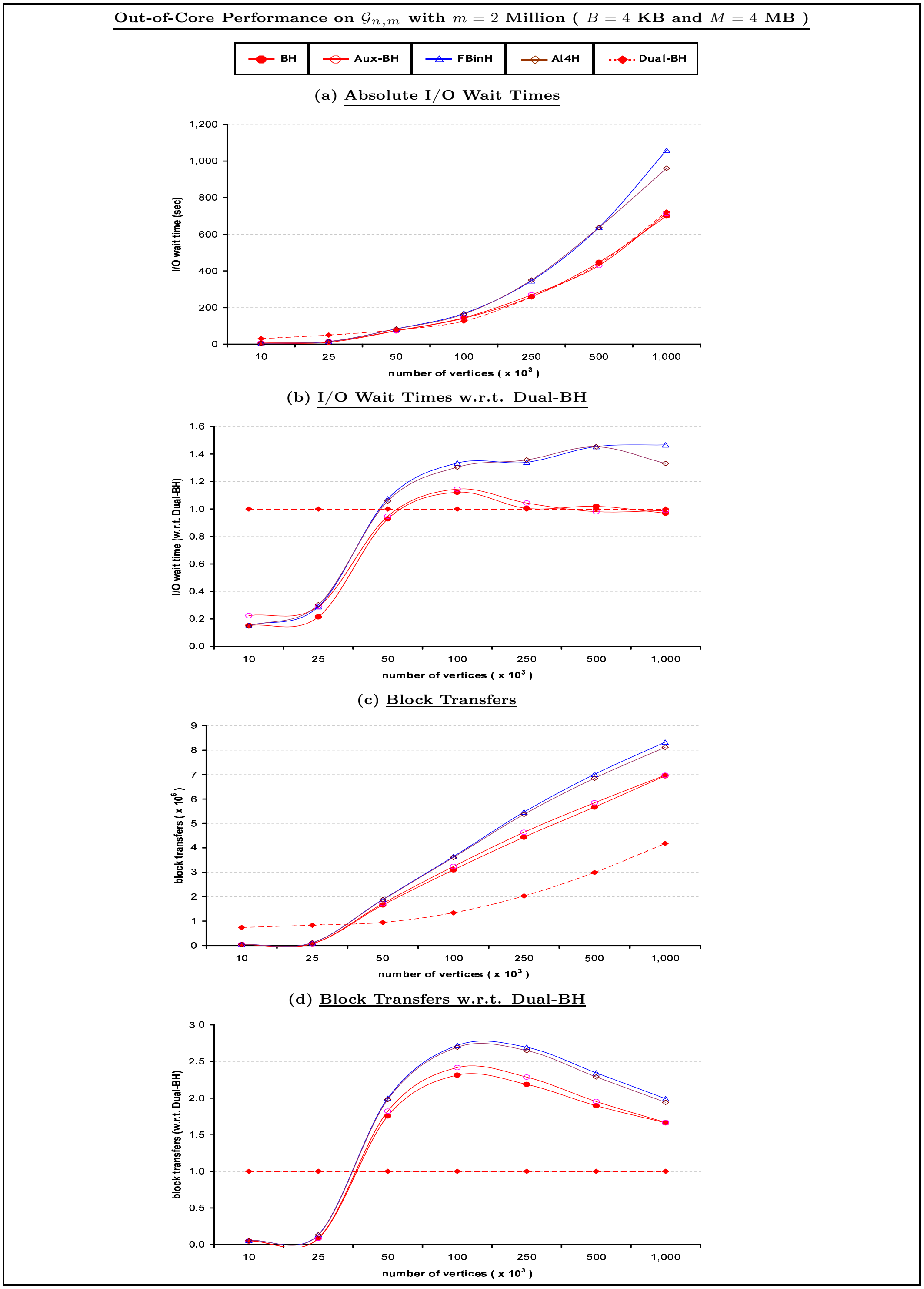

Figure 4.5: Out-of-core performance on $\mathcal{G}_{n,m}$ with m fixed to 2 million.

by different implementations support our claim. They show that both BH and Aux-BH perform more than 2 times more block transfers than Dual-BH, while FBinH and Al4H perform more than 2.5 times more.

We believe that the performance of Dual-BH will improve even further compared to other implementations under more memory intensive situation.

4.5.4 Performance on Real-World Graphs

In Table 4.4 we tabulate the performance of different implementations on road networks of different regions of the United States. These regional networks were downloaded from the DIMACS Challenge 9 website [1]. We chose the physical distance between the endpoints as the weight of each edge.

The SeqH implementation was the fastest followed by FBinH. Though for smaller networks Aux-BH was slower than the DIMACS benchmark code, it ran faster for larger networks. For the largest networks in the set, namely Central USA, Aux-BH was only 7% slower than SeqH. Among all implementations BH was the slowest.

The road networks are almost planar and their edge distributions ensure that nodes are connected to only nearby nodes on the plane. Since Dijkstra's algorithm only stores fringe nodes in the priority queue, the nature of these graphs keep the size of the priority queue very small. Empirical evidence suggests that in our experiments the priority queues were so small in size that they often fit in L2 cache. In the absence of any significant savings in L2 misses, other overheads in the implementations of buffer heap caused BH run slower than all internal-memory heap based implementations. As the size of the graph grew the performance of BH improved with respect to other implementations which makes sense since the size of the priority queue tends to grow as the graph becomes larger.

Region (n, m)	Running Time in Milliseconds Aux-BH	FBinH	Al4H	SeqH	BinH	PairH	BH	DIMACS
New York City (0.26×10^6, 0.73×10^6)	91	82	94	76	104	156	336	82
San Francisco Bay Area (0.32×10^6, 0.80×10^6)	106	96	107	87	120	179	381	94
Colorado (0.44×10^6, 1.06×10^6)	149	132	149	121	171	242	532	137
Florida (1.07×10^6, 2.71×10^6)	374	336	379	310	441	612	1,328	353
Northwest USA (1.21×10^6, 2.84×10^6)	445	420	456	381	540	761	1,578	455
Northeast USA (1.52×10^6, 3.90×10^6)	614	579	633	529	738	1,037	2,084	645
California and Nevada (1.89×10^6, 4.66×10^6)	746	688	783	645	876	1,266	2,531	791
Great Lakes (2.76×10^6, 6.89×10^6)	1,138	1,063	1,172	986	1,329	1,958	3,779	1,213
Eastern USA (3.60×10^6, 8.78×10^6)	1,642	1,563	1,686	1,453	1,966	2,731	5,150	1,842
Western USA (6.26×10^6, 15.25×10^6)	3,139	3,019	3,210	2,792	3,698	5,096	9,665	3,654
Central USA (14.08×10^6, 34.29×10^6)	12,109	11,854	12,341	11,326	13,965	17,236	27,450	15,225

Table 4.4: Running time (in milliseconds) of each implementation on US road networks (TIGER/Line). The datasets were downloaded from DIMACS Challenge 9 website [1].

Chapter 5

Cache-efficient Unweighted and Bounded-weight APSP

One day Alice came to a fork in the road
and saw a Cheshire cat in a tree.
Which road do I take? she asked.
Where do you want to go? was his response.
I don't know, Alice answered.
Then, said the cat, it doesn't matter.
(Lewis Carroll in *Alice in Wonderland*)

In this chapter we present several new cache-efficient algorithms for finding all-pairs shortest paths in an n-node, m-edge undirected graph. For all-pairs shortest paths and diameter in unweighted undirected graphs we present cache-oblivious algorithms with $\mathcal{O}(n \cdot sort(m))$ cache-misses, where $sort(m)$ is the number of cache-misses incurred for sorting m items. We also present efficient cache-aware algorithms that find paths between all pairs of vertices in an unweighted graph with lengths within a small additive constant of the shortest path length. For weighted undirected graphs we present a cache-aware exact algorithm that incurs $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{2}{3}}} + n \cdot sort(m)\right)$ cache-misses, where $\beta = \left(\frac{n}{m \log \rho}\right) \cdot B$, ρ is the ratio of the largest and the smallest edge-weights, and B is the block size between the cache and the main memory. The algorithm assumes $\beta \geq 1$.

All of our results improve earlier results known for these problems. For approximate all-pairs shortest paths we provide the first nontrivial results. Our diameter result uses $\mathcal{O}(m + n)$ extra space, and all of our other algorithms use $\mathcal{O}(n^2)$ space.

5.1 Introduction

In Chapter 3 we presented a cache-efficient algorithm for solving the all-pairs shortest path (APSP) problem on undirected graphs with general edge-weights. In this chapter we consider undirected graphs with unweighted and bounded-weight edges, and provide algorithms with provably better cache performance.

Recall that given a (directed or undirected) graph G with vertex set V, edge set E, and a non-negative real-valued weight function w over E, the APSP problem seeks to find a path of minimum total edge-weight between every pair of vertices in V. For unweighted graphs the APSP problem is also called the all-pairs breadth-first-search (AP-BFS) problem.

5.1.1 Cache-aware APSP Algorithms

The simplest method of computing AP-BFS (or APSP) is to simply run a BFS (or *single source shortest path* (SSSP) algorithm, respectively) from each of the n vertices of the graph. On unweighted undirected graphs BFS can be performed in $\mathcal{O}\left(\min\{n, \sqrt{\frac{mn}{B}}\} + sort(m)\right)$ block transfers [92, 88]. On undirected graphs with real edge-weights the SSSP problem can also be solved in $\mathcal{O}\left(\sqrt{\frac{nm}{B}} \cdot \log \rho + sort(m) + MST(n, m)\right)$ cache-misses [89], where ρ is the ratio of the largest and the smallest edge-weights, and $MST(n, m)$ is the cache complexity of computing a minimum spanning tree on a graph with n vertices and m edges. The cache complexity of AP-BFS (or APSP) is obtained by multiplying the cache complexity of BFS (or SSSP, resp.) by n.

Arge et al. [13] proposed a cache-aware algorithm for AP-BFS on undirected graphs that incurs only $\mathcal{O}\left(n \cdot sort(m)\right)$ cache-misses. Their algorithm works by clustering nearby vertices in the graph, and running concurrent BFS from all vertices of the same cluster. This same algorithm can be used to compute unweighted diameter of the graph in the same cache-miss bound and $\mathcal{O}\left(\sqrt{mnB}\right)$ additional space. They also present another algorithm for computing the unweighted diameter of sparse graphs in $\mathcal{O}\left(sort(kn^2 B^{\frac{1}{k}})\right)$ cache-misses and $\mathcal{O}\left(kn\right)$ space for any integer k, $3 \leq k \leq \log B$.

For undirected graphs with general non-negative edge-weights Arge et al. [13] proposed an APSP algorithm that incurs only $\mathcal{O}\left(n \cdot (\sqrt{\frac{mn}{B}} \cdot \log n + sort(m))\right)$ cache-misses provided $m = \mathcal{O}\left(\frac{B}{\log n} \cdot n\right)$. They use a priority queue structure called the *Multi-Tournament-Tree* which is created by bundling together a number of cache-efficient *Tournament Trees* [83]. This reduces unstructured accesses to adjacency lists at the expense of increasing the cost of each priority queue operation.

5.1.2 Cache-oblivious APSP Algorithms

No non-trivial cache-oblivious algorithm is known for the AP-BFS and the APSP problems except for the method of running BFS and SSSP, respectively, from each of the n vertices. In this model, BFS on an undirected graph can be performed in $\mathcal{O}\left(\sqrt{\frac{nm}{B}} + \frac{n}{B} \log \frac{m}{B} + MST(n, m)\right)$ cache-misses [24], and SSSP on an undirected graph with non-negative real-valued edge-weights can be solved in $\mathcal{O}\left(n + \frac{m}{B}\log\frac{n}{M}\right)$ cache-misses using our cache-oblivious buffer heap (see Chapter 3). Very recently, Allulli et al. [7] obtained a cache-oblivious SSSP algorithm for undirected sparse graphs with real edge-weights by extending the cache-aware algorithm in [89] which incurs $\mathcal{O}\left(\sqrt{\frac{nm}{B}\cdot\log\rho} + \frac{n}{B}\log\frac{m}{B} + MST(n,m)\right)$ cache-misses. The cache complexity of the corresponding all-pairs version of the problem is obtained by multiplying the cache complexity of the single-source version by n.

Problem	Known	New
Unweighted APSP	cache-misses $\mathcal{O}\left(n\cdot sort(m)\right)$ extra space $\mathcal{O}\left(\sqrt{mnB}\right)$ ([13])	**cache-oblivious** cache-misses $\mathcal{O}\left(n\cdot sort(m)\right)$ extra space $\mathcal{O}\left(n\right)$
Approximate unweighted APSP with additive error $2(k-1)$ (int $k\in[2,\log n]$)	none	$\mathcal{O}\left(\frac{\mathcal{T}(m,n,k)}{B}\right)$ if $\log n \geq \frac{B}{8}$, and $\mathcal{O}\left(\frac{\mathcal{T}(m,n,k)}{B} + \left(\frac{\mathcal{T}(m,n,k)}{B}\cdot\frac{n}{k}\right)^{\frac{2}{3}}\right)$ otherwise, where, $\mathcal{T}(m,n,k) = \mathcal{O}\left(k\left(\frac{m}{n\log n}\right)^{\frac{1}{k}} n^2\log n\right)$ is the runtime of the original flat-memory implementation of the algorithm (see [44]), and $m \geq n\log n$
Bounded-weight APSP	$\mathcal{O}\left(\frac{n^2}{\beta^{\frac{1}{2}}} + n\cdot sort(m)\right)$, when $m = \mathcal{O}\left(\frac{B}{\log\rho}\cdot n\right)$; here $\beta = \left(\frac{n}{m\log\rho}\right)\cdot B \geq 1$ (trivial using [89])	$\mathcal{O}\left(\frac{n^2}{\beta^{\frac{2}{3}}} + n\cdot sort(m)\right)$, when $m = \mathcal{O}\left(\frac{B}{\log\rho}\cdot n\right)$; $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{3}{4}}}\cdot\frac{1}{\log^{\frac{1}{4}}\rho} + n\cdot sort(m)\right)$, when $m = \mathcal{O}\left(\frac{B}{\max\{\log\rho,\log^2 n\}}\cdot n\right)$

Table 5.1: Cache-miss bounds for APSP problems on undirected graphs. Here, ρ is the ratio of the largest and the smallest edge-weights in the graph. All algorithms are cache-aware unless specified otherwise.

5.1.3 Our Results

Majority of the results included in this chapter were presented in a conference paper [33], and are tabulated in Table 5.1.

In Section 5.2 we present a simple cache-oblivious algorithm for computing AP-BFS on unweighted undirected graphs in $\mathcal{O}\left(n\cdot sort(m)\right)$ cache-misses, matching the cache complexity of its cache-aware counterpart [13]. We use this algorithm to compute the diameter of an unweighted undirected graph in the same cache-miss bound and $\mathcal{O}\left(n+m\right)$ space. Our cache-oblivious algorithm is arguably simpler than the cache-aware algorithm in [13] and it has a better space bound for computing the diameter.

In Section 5.3 we present the first nontrivial cache-efficient algorithm to compute approximate APSP on unweighted undirected graphs with small additive error. Our algorithm is based on a flat-memory algorithm in [44] that runs in $\mathcal{T}(m,n,k) = \mathcal{O}\left(k \cdot \left(\frac{m}{n \log n}\right)^{\frac{1}{k}} \cdot n^2 \log n\right)$ time, and produces estimated distances with an additive error of at most $2(k-1)$, where $k \in [2, \log n]$ is an integer. Our algorithm has the same error bounds, is cache-aware, and incurs $\mathcal{O}\left(\frac{\mathcal{T}(m,n,k)}{B} + \left(\frac{\mathcal{T}(m,n,k)}{B} \cdot \frac{n}{k}\right)^{\frac{2}{3}}\right)$ cache-misses provided $m \geq n \log n$. Additionally, if $\log n \geq \frac{B}{8}$ also holds, which is typically the case for shallower levels of the memory hierarchy (e.g., L1 and L2), the algorithm incurs only $\mathcal{O}\left(\frac{\mathcal{T}(m,n,k)}{B}\right)$ cache-misses. Our approximate algorithm performs fewer block transfers than the $\mathcal{O}\left(n \cdot sort(m)\right)$ I/O exact AP-BFS algorithm when $m > \max\left\{k^{\frac{k}{k-1}}, \left(\frac{B}{\log n}\right)^{\frac{k}{3k-2}}\right\} \cdot n \log n$.

In Section 5.4, we present a cache-aware APSP algorithm for weighted undirected graphs incurring $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{2}{3}}} + n \cdot sort(m)\right)$ cache-misses, where $\beta = \left(\frac{n}{m \log \rho}\right) \cdot B$, and ρ is the ratio of the largest and the smallest edge-weights in the graph. The algorithm assumes that $\beta \geq 1$. The best known previous bound of $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{1}{2}}} + n \cdot sort(m)\right)$ is obtained trivially by running Meyer & Zeh's bounded-weight undirected SSSP algorithm [89] once from every vertex. Our algorithm also builds on Meyer & Zeh's SSSP algorithm and improves over the trivial bound when $m = \mathcal{O}\left(\frac{B}{\log \rho} \cdot n\right)$. We also show that the cache-complexity of the algorithm can be further improved to $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{3}{4}}} \cdot \frac{1}{\log^{\frac{1}{4}} \rho} + n \cdot sort(m)\right)$ provided $m = \mathcal{O}\left(\frac{B}{\max\{\ \log \rho,\ \log^2 n\ \}} \cdot n\right)$.

The results in Sections 5.2 and 5.3 appeared in the conference paper [33], while the result in Section 5.4 is new.

5.1.4 Organization of the Chapter

In Section 5.2 we present our cache-oblivious exact APSP algorithm, and in Section 5.3 our cache-aware approximate APSP algorithm for unweighted undirected graphs. In Section 5.4 we describe our cache-aware APSP algorithm for weighted undirected graphs.

5.2 Cache-oblivious APSP and Diameter for Unweighted Undirected Graphs

In this section we present a cache-oblivious algorithm for computing all-pairs shortest paths and diameter in an unweighted undirected graph. The algorithm uses Munagala and Ranade's cache-oblivious BFS algorithm [92] as a subroutine which we describe first.

5.2.1 Munagala and Ranade's Cache-oblivious BFS Algorithm

Given an unweighted undirected graph $G = (V, E)$ and a source vertex $s \in V$, Munagala & Ranade's algorithm [92] computes the BFS level of each vertex $v \in V$ with respect to s. For $i \in [0, n-1]$, let $L(i)$ denote the set of vertices at BFS level i, and let $N(L(i))$ be the set of vertices adjacent to the vertices in $L(i)$. The set $L(-1)$ is assumed to be empty. The algorithm starts by setting $L(0) \leftarrow \{\ s\ \}$, and then for $1 \leq i \leq n-1$, incrementally computes $L(i)$ from $L(i-1)$, $L(i-2)$ and $N(L(i-1))$ assuming that $L(i-1)$ and $L(i-2)$ have already been computed.

Function 5.2.1 implements the algorithm. Summing up the cache-misses incurred by different steps we obtain $\mathcal{O}\big(\sum_{i=1}^{n-1}\left(|L(i)| + sort\left(|N(L(i-1))|\right)\right) + sort(|Q|)\big)$. Since $|Q| = \sum_{i=0}^{n-1}|L(i)| = n$, $\sum_{i=0}^{n-2}|N(L(i))| \leq m$ and $\sum_{i=0}^{n-2} sort\left(|N(L(i))|\right) \leq sort\left(\sum_{i=0}^{n-2}|N(L(i))|\right)$, the cache complexity of the algorithm reduces to $\mathcal{O}\big(n + sort(m)\big)$. The algorithm uses $\mathcal{O}\left(n + m\right)$ space.

5.2.2 Cache-oblivious APSP for Unweighted Undirected Graphs

In this section we describe a cache-oblivious APSP algorithm for unweighted undirected graphs which incurs only $\mathcal{O}\left(n \cdot sort(m)\right)$ cache-misses.

Let $G = (V, E)$ be an unweighted undirected graph. By $d(u, v)$ we denote the shortest distance between $u, v \in V$. Our algorithm is based on the following observation which follows from triangle inequality and the fact that $d(u, v) = d(v, u)$ in an undirected graph:

Observation 5.2.1. *For any three vertices u, v and x in G,*

$$d(u, x) - d(u, v) \leq d(v, x) \leq d(u, x) + d(u, v).$$

Suppose for some $u \in V$ we have already computed $d(u, x)$ for $\forall x \in V$. We sort the adjacency lists of G in non-decreasing order by $d(u, \cdot)$, and by $A(r)$ we denote the portion of this sorted list containing adjacency lists of vertices x with $d(u, x) = r$. Now if v is another vertex in V then Observation 5.2.1 implies that the adjacency list of any vertex x with $d(v, x) = i$, must reside in some $A(r)$, where $r \in [i -$

Function **5.2.1.** MR-BFS($G,\ s$) $\{$*Munagala & Ranade's cache-oblivious BFS algorithm [92]*$\}$

[Given an unweighted undirected graph G with vertex set V, edge set E, and a source vertex $s \in V$, this function cache-obliviously computes the BFS level of each vertex $v \in V$ w.r.t. s, and store it in $d[v]$. Let $n = |V|$ and $m = |E|$.**]**

1. $\forall_{v \in V}\ d[v] \leftarrow \infty$
$L(-1) \leftarrow \emptyset,\ L(0) \leftarrow \{\ s\ \}$ $\{$*for* $i \in [0, n-1]$, $L(i)$ *will contain the vertices at BFS level* $i\}$
$S \leftarrow \{\ (s, 0)\ \}$ $\{$*at the start of iteration* $i \in [1, n]$ *of the* ***for*** *loop in step 2,* $S = \{\ (v,\ j)\ |\ \ j \in [0, i-1] \wedge v \in L(j)\ \}\}$

2. ***for*** $i \leftarrow 1$ ***to*** $n - 1$ ***do***

- (i) Construct $N(L(i-1))$ by concatenating the adjacency lists of the vertices in $L(i-1)$ $\{$*for a set of vertices* S, $N(S)$ *is a multiset of all vertices adjacent to* $S\}$
- (ii) Remove duplicates from $N(L(i-1))$ by sorting it by vertex indices, followed by a, scan, and let $L'(i)$ be the resulting set
- (iii) Construct $L(i) = L'(i) \setminus \{L(i-1) \cup L(i-2)\}$ by scanning $L'(i)$, $L(i-1)$ and $L(i-2)$
- (iv) Construct $S \leftarrow S \cup \{\ (v,\ i)\ |\ \ v \in L(i)\ \}$ by a single scan of $L(i)$ and appending to S

3. Update $d[v]$ for all $v \in V$ by sorting S by vertex indices, followed by a scan

4. ***return*** d

MR-BFS Ends

Function **5.2.2.** Incremental-BFS($G,\ u,\ v,\ d(u, \cdot)$)

[Given an unweighted undirected graph G with vertex set V, edge set E, two vertices $u, v \in V$, and $d(u, w)$ for all $w \in V$, this algorithm computes $d(v, w)$ for all $w \in V$. We assume that E is given as a set of adjacency lists, and by $Adj(x)$ we denote the adjacency list of $x \in V$. Let $n = |V|$ and $m = |E|$.**]**

1. Sort E: for any $x, y \in V$, $Adj(x)$ is placed before $Adj(y)$ provided $\langle\ d(u, x), x\ \rangle < \langle\ d(u, y), y\ \rangle$. Let E' be the sorted E. Let $A(r)$, $r \in [0, n)$, be the section of E' containing $Adj(x)$ of $\forall x \in V$ with $d(u, x) = r$.

2. Compute $d(v, x)$ for $\forall x \in V$ by calling MR-BFS($G' = (V, E'),\ v$), but step $2(a)$ of MR-BFS replaced with:

$N(L(i-1)) \leftarrow \emptyset$
for $r \leftarrow \max\{\ 0,\ i - 1 - d(u, v)\ \}$ ***to*** $\min\{\ n - 1,\ i - 1 + d(u, v)\ \}$ ***do***
Scan $L(i-1)$ and $A(r)$ to append $\{\ Adj(x)\ |\ x \in L(i-1)\ \wedge\ Adj(x) \in A(r)\ \}$ to $N(L(i-1))$

3. ***return*** $d(v, \cdot)$

Incremental-BFS Ends

Function **5.2.3.** AP-BFS(G)

[Given an unweighted undirected graph G with vertex set V and edge set E, this function cache-obliviously computes the shortest distance $d(u,v)$ between every pair of vertices $u, v \in V$. Let $n = |V|$ and $m = |E|$.]

1. Perform the following initializations:

  (i) $T \leftarrow$ a spanning tree of G

  (ii) $ET \leftarrow$ an *Euler Tour of* T

  (iii) Mark the first occurrence of each vertex on ET.
  Let $v_1, v_2, \ldots, v_n$ be the marked vertices in the order they appear on ET.

2. $d(v_1, \cdot) \leftarrow$ MR-BFS(G, v_1)

3. ***for*** $i \leftarrow 2$ ***to*** n ***do***
  $d(v_i, \cdot) \leftarrow$ Incremental-BFS(G, v_{i-1}, v_i , $d(v_{i-1}, \cdot)$)

4. ***return*** d

AP-BFS Ends

$d(u,v), i + d(u,v)]$. Therefore, we can compute $d(v,x)$ for $\forall x \in V$ by calling MR-BFS with source vertex v, but in step $2(a)$ of MR-BFS, instead of constructing $N(L(i-1))$ by $|L(i-1)|$ independent accesses to the adjacency lists of G, we construct $N(L(i-1))$ by simultaneously scanning $L(i-1)$ and $A(r)$ for $r \in [\max\{\ 0,\ i-1-d(u,v)\ \}, \min\{\ n-1,\ i-1+d(u,v)\ \}]$. The resulting algorithm (Incremental-BFS) is given as Function 5.2.2.

Step 1 of Incremental-BFS incurs $\mathcal{O}\left(sort(m)\right)$ cache-misses. In step 2 each $A(r)$ is scanned $\mathcal{O}\left(d(u,v)\right)$ times, and so is each $L(i-1)$. Therefore, the number of cache-misses caused by step 2 is $\mathcal{O}\Big(\frac{1}{B} \cdot \Big(\sum_{i=1}^{n} |L(i-1)| + \sum_{r=0}^{n-1} |A(r)|\Big) \cdot d(u,v) + sort(m)\Big)$. Since each vertex appears in exactly one $L(i-1)$ and each edge appears in exactly one $A(r)$, we have $\sum_{i=1}^{n} |L(i-1)| = n \leq m$ and $\sum_{r=0}^{n-1} |A(r)| = m$. Thus the cache complexity of step 2 reduces to $\mathcal{O}\left(\frac{m}{B} \cdot d(u,v) + sort(m)\right)$ which is also the overall cache complexity of Incremental-BFS.

Since Incremental-BFS is actually an implementation of Munagala and Ranade's algorithm [92], its correctness follows from the correctness of that algorithm, and from Observation 5.2.1 which guarantees that the adjacency lists of all $x \in L(i-1)$ in step 2 of Incremental-BFS are found in the set of $A(r)$'s scanned.

We can use Incremental-BFS to perform BFS cache-efficiently from all $v \in V$. The following observation each part of which follows trivially from the properties of spanning trees, Euler tours and shortest paths, is central to this extension:

Observation 5.2.2. *If ET is an Euler tour of a spanning tree of an unweighted undirected graph $G = (V, E)$ with n vertices, then*

(a) *number of edges between any two $x, y \in V$ on ET is an upper bound on*

$d(x, y)$ *in* G,

(b) ET *has* $\mathcal{O}(n)$ *edges, and*

(c) *each vertex* $x \in V$ *appears at least once in* ET.

The extension is outlined in Function 5.2.3 (AP-BFS) which works as follows. It first constructs an Euler tour ET of a spanning tree of the input graph G, and then marks the first occurrence of each vertex of G on ET. Let $v_1, v_2, \ldots, v_n$ be the marked vertices in the order they appear on ET. The algorithm first calls MR-BFS to compute $d(v_1, x)$ for $\forall x \in V$. Then for $2 \leq i \leq n$, it calls INCREMENTAL-BFS to compute $d(v_i, x)$ for $\forall x \in V$ using the $d(v_{i-1}, \cdot)$ values computed in the previous step.

Correctness. Correctness of AP-BFS follows from the correctness of MR-BFS and INCREMENTAL-BFS. Moreover, Observation 5.2.2(c) ensures that BFS will be performed from each $v \in V$.

Space Complexity. Since the algorithm outputs $\Theta\left(n^2\right)$ pairwise distances it uses $\Theta\left(n^2\right)$ space.

Cache Complexity. Steps 1(a) and 1(b) of AP-BFS can be performed cache-obliviously in $\mathcal{O}\left(\min\left\{n + sort(m), sort(m) \cdot \log_2 \log_2 n\right\}\right)$ and $\mathcal{O}\left(sort(n)\right)$ cache-misses, respectively [11], while step 1(c) incurs $\mathcal{O}\left(sort(E)\right)$ cache-misses. The number of cache-misses incurred by steps 2 and 3 are $\mathcal{O}\left(n + sort(m)\right)$ and $\mathcal{O}\left(\frac{m}{B}\sum_{i=2}^{n} d(v_{i-1}, v_i) + n \cdot sort(m)\right)$, respectively. Since by Observations 5.2.2(a) and 5.2.2(b) we have $\sum_{i=2}^{n} d(v_{i-1}, v_i) = \mathcal{O}(n)$, the cache complexity of step 3, and consequently of the entire algorithm reduces to $\mathcal{O}\left(n \cdot sort(m)\right)$.

5.2.3 Cache-oblivious Unweighted Diameter for Undirected Graphs

The AP-BFS algorithm can be used to find the unweighted diameter of an undirected graph cache-obliviously in $\mathcal{O}\left(n \cdot sort(m)\right)$ cache-misses. We no longer need to output all $\Theta\left(n^2\right)$ pairwise distances, and each iteration of step 3 of AP-BFS only requires the $\Theta(n)$ distances computed in the previous iteration or in step 2. Thus the space requirement is only $\Theta(n)$ in addition to the $\mathcal{O}(m)$ space required to handle the adjacency lists.

5.3 Cache-aware Approximate APSP for Unweighted Undirected Graphs

In this section we present a family of efficient cache-aware algorithms APPROX-AP-BFS$_k$ for approximating all distances in an unweighted undirected graph with an additive error of at most $2(k-1)$, where $k \in [2, \log n]$ is an integer. The error is one sided. If $\delta(u, v)$ denotes the shortest distance between any two vertices

u and v in the graph, and $\widehat{\delta}(u,v)$ denotes the estimated distance between u and v produced by the algorithm, then $\delta(u,v) \leq \widehat{\delta}(u,v) \leq \delta(u,v) + 2(k-1)$. This family of algorithms is the cache-efficient version of the family of $\mathcal{T}(m,n,k) = \mathcal{O}\left(k \cdot \left(\frac{m}{n \log n}\right)^{\frac{1}{k}} \cdot n^2 \log n\right)$ time approximate shortest path algorithms ($\mathbf{apasp}_k$) introduced by Dor et al. [44] which is the most efficient algorithm available for solving the problem in the traditional flat-memory model. The function Approx-AP-BFS$_k$ incurs $\mathcal{O}\left(\frac{\mathcal{T}(m,n,k)}{B} + \left(\frac{\mathcal{T}(m,n,k)}{B} \cdot \frac{n}{k}\right)^{\frac{2}{3}}\right)$ cache-misses provided $m \geq n \log n$. Additionally, if $\log n \geq \frac{B}{8}$ also holds, the cache-complexity of the algorithm reduces to $\mathcal{O}\left(\frac{\mathcal{T}(m,n,k)}{B}\right)$.

5.3.1 Dor et al.'s Approximate AP-BFS for Flat-Memory Model

The approximate APSP algorithm ($\mathbf{apasp}_k$) for the traditional flat-memory model given in [44], receives an unweighted undirected graph $G = (V, E)$ as input, and outputs an approximate shortest distance $\widehat{\delta}(u,v)$ between every pair of vertices $u, v \in V$ with a positive additive error of at most $2(k-1)$.

Function 5.3.1 (DHZ-Approx-AP-BFS$_k$) gives a high level overview of the algorithm. Recall that a set of vertices D is said to dominate a set U if every vertex in U has a neighbor in D (see step 2(a) of the algorithm). The algorithm maintains the invariant that after the i-th iteration of the outermost ***for*** loop in step 3, the approximate distance computed by the algorithm from each $u \in D_i$ to each $v \in V$ has an additive error of at most $2(i-1)$. Thus after the k-th iteration a surplus $2(k-1)$ distance is computed between every pair of vertices in G.

5.3.2 Our Cache-efficient Algorithm

Our algorithm adapts the Dor et al. algorithm (DHZ-Approx-AP-BFS$_k$) to obtain a cache-efficient implementation. In our adaptation use the same sequence of values for $\langle s_1, s_2, \ldots, s_{k-1}\rangle$ as in the original algorithm. In Section 5.3.3 we describe a cache-efficient implementation of step 2 of DHZ-Approx-AP-BFS$_k$.

The cache complexity of DHZ-Approx-AP-BFS$_k$ depends on the cache-efficiency of the SSSP algorithm used in step 3. Therefore, we replace each SSSP algorithm with a more cache-efficient BFS algorithm by transforming each $G_i(u)$ to an unweighted graph $G'_i(u)$ of comparable size. But in order to preserve the shortest distances from u to other vertices in $G_i(u)$, we replace the weighted edges of $G_i(u)$ with a set of *directed* unweighted edges. This makes the graph $G'_i(u)$ partially directed, and we modify existing external-memory undirected BFS algorithms to handle the partial directedness in $G'_i(u)$ efficiently. This is described in section 5.3.4.

There are two ways to apply the BFS: either we can run an independent

---

FUNCTION **5.3.1.** DHZ-APPROX-AP-BFS$_k$(G)

{*Dor et al.'s internal-memory approximate AP-BFS algorithm [44]*}

[Given an unweighted undirected graph G with vertex set V and edge set E, this function computes an approximate shortest distance $\widehat{\delta}(u, v)$ between every pair of vertices $u, v \in V$ with a positive additive error of at most $2(k-1)$. We assume that for any vertex $v \in V$, $deg(v)$ denotes its degree. Let $n = |V|$ and $m = |E|$.**]**

1. ***for*** $i \leftarrow 1$ ***to*** $k - 1$ ***do*** $s_i \leftarrow \frac{m}{n}\left(\frac{n \log n}{m}\right)^{\frac{i}{k}}$

2. Decompose G to produce the following sets:

- (i) $D_1, D_2, \ldots, D_k$, where $D_k = V$, and for $i \in [1, k)$, $D_i \subseteq V$ dominates $\forall v \in V$ with $deg(v) \geq s_i$
- (ii) $E_1, E_2, \ldots, E_k$, where $E_1 = E$, and
  for $i \in (1, k]$, $E_i = \{(u, v) | (u, v) \in E \wedge \min\{deg(u), deg(v)\} \leq s_{i-1}\}$
- (iii) $E^* \subseteq E$ bearing witness that each D_i dominates the vertices of degree at least s_i

3. ***for*** $i \leftarrow 1$ ***to*** k ***do***

- (i) ***for each*** $u \in D_i$ ***do***
  - (a) Construct weighted graph $G_i(u) = (V, E_i \cup E^* \cup (\{u\} \times V))$ with the following edge-weights:
    - edges in $E_i \cup E^*$ are assigned unit weight
    - each edge (u, v) in $\{u\} \times V$ is assigned the current known shortest u to v distance as weight
  - (b) Run SSSP from u on $G_i(u)$

4. Return the smallest distance computed between every pair of vertices in step 3

DHZ-APPROX-AP-BFS$_k$ ENDS

BFS from each $u \in D_i$ as in step 3 of DHZ-APPROX-AP-BFS$_k$, or we can run BFS incrementally from the vertices of D_i as in Section 5.2.2. Running independent BFS is more cache-efficient when $|D_i|$ is smaller (i.e., value of i is smaller), and incremental BFS is more cache-efficient when $G'_i(u)$ is sparser (i.e., value of i is larger). Therefore, we choose a value of i at which switching from independent BFS to incremental BFS minimizes the cache complexity of the entire algorithm. The overall algorithm is described in Section 5.3.5.

5.3.3 Cache-efficient Graph Decomposition

A set of vertices D is said to *dominate* a set U if every vertex in U has a neighbor in D. It has been shown in [5] that there is always a set of size $\mathcal{O}\left(\frac{n \log n}{s}\right)$ that dominates all the vertices of degree at least s in an undirected graph, and in [44] it has been shown that this set can be found deterministically in $\mathcal{O}(n + m)$ time. In this section we present a cache-efficient implementation of the greedy algorithm

FUNCTION **5.3.2.** DOMINATE(G, s)
[Given an undirected graph $G = (V, E)$ and a *degree threshold* s, this algorithm outputs a pair $\langle\ D,\ E^*\ \rangle$, where D is a set of size $\mathcal{O}\left(\frac{n \log n}{s}\right)$ that dominates the vertices of degree at least s in G, and $E^* \subseteq E$ is a set of size $\mathcal{O}(n)$ such that for every $u \in V$ with degree at least s, there is an edge $(u, v) \in E^*$ with $v \in D$. Here, $n = |V|$ and $m = |E|$.**]**

1. ***for each*** $u \in V$ ***do***
  $A_u \leftarrow \{\ v \mid (u, v) \in E\ \}$ $\{A_u$ *is the set of neighbors of* $u\}$
  $L_u \leftarrow \{\ v \mid (u, v) \in E\ \wedge\ deg(v) \geq s\ \}$ $\{L_u$ *is the set of neighbors of* u *with degree* $\geq s\}$
  Sort L_u by vertex indices

2. Perform the following initializations:
  (i) $D \leftarrow \emptyset$, $E^* \leftarrow \emptyset$
  (ii) $T \leftarrow \emptyset$ $\{T$ *is a BRT capable of containing key values in the range* $[1 \ldots n]\}$
  (iii) $Q \leftarrow \emptyset$ {*a buffer heap that supports* Insert*,* Delete-Max *and* Relative-Decrease}
   for each $u \in V$ ***do***
    INSERT$_{(Q)}$($\langle\ u, deg(u)\ \rangle$, $|L_u|$) {*insert each* $u \in V$ *into* Q *with* $|L_u|$ *as the key*}

3. $(\langle\ u, deg(u)\ \rangle, d) \leftarrow$ DELETE-MAX$_{(Q)}$( ) {*u has maximum number (d) of undominated neighbors with degree* $\geq s$}
 while $d > 0$ ***do***
  (i) $D \leftarrow D \cup \{u\}$
  (ii) $L'_u \leftarrow$ EXTRACT$_{(T)}$(u) {*extract all dominated neighbors of* u *with degree* $\geq s$}
   Sort L'_u by vertex indices
  (iii) $L''_u \leftarrow L'_u \setminus L_u$ $\{L''_u$ *is the set of undominated neighbors of* u *with degree* $\geq s\}$
  (iv) ***if*** $deg(u) \geq s$ ***then***
    for each $v \in A_u$ ***do***
     RELATIVE-DECREASE$_{(Q)}$(v, 1) {*v loses an undominated neighbors*}
     INSERT$_{(T)}$(u, v) {*mark neighbor u of v as* dominated}
  (v) ***for*** each $v \in L''_u$ ***do***
    $E^* \leftarrow E^* \cup \{(u, v)\}$ {*edge (u, v) is the witness that u dominates v*}
    for each $x \in A_v$ ***do***
     RELATIVE-DECREASE$_{(Q)}$(x, 1) {*v loses an undominated neighbors*}
     INSERT$_{(T)}$(v, x) {*mark neighbor v of x as* dominated}
  (vi) $(u, d) \leftarrow$ DELETE-MAX$_{(Q)}$( )

4. ***return*** $\langle\ D,\ E^*\ \rangle$
DOMINATE ENDS

FUNCTION **5.3.3.** DECOMPOSE(G, $\langle s_1, s_2, \ldots, s_{k-1} \rangle$)

[Given an undirected graph $G = (V, E)$ and a decreasing sequence $s_1, s_2, \ldots, s_{k-1}$ of degree thresholds, this algorithm outputs a sequence of edge sets $E_1 \supseteq E_2 \supseteq \ldots \supseteq E_k$, where $E_1 = E$ and for $1 < i \leq k$ the set E_i contains edges that touch vertices of degree at most s_{i-1}. It also outputs dominating sets $D_1, D_2, \ldots, D_k$, and an edge set E^*. For $1 \leq i < k$ the set D_i dominates all vertices of degree greater than s_i, while D_k is simply V. The set $E^* \subseteq E$ is such that if $deg(u) > s_i$ then there exists an edge $(u, v) \in E^*$ with $v \in D_i$, where $deg(u)$ denotes the degree of vertex u.]

1. Construct the lexicographically sorted list
$E' = \{ \langle u, v, d \rangle \mid (u, v) \in E \ \wedge \ d = \min\{ deg(u), deg(v) \} \}$

2. $E_1 \leftarrow E$
for $i \leftarrow 2$ ***to*** k ***do*** $E_i \leftarrow \{ (u, v) \mid \langle u, v, d \rangle \in E' \ \wedge \ d \leq s_{i-1} \}$

3. ***for*** $i \leftarrow 1$ ***to*** $k - 1$ ***do*** $\langle D_i, E_i^* \rangle \leftarrow$ DOMINATE(G, s_i)
$D_k \leftarrow V$, $E^* \leftarrow \cup_{i=1}^{k-1} E_i^*$

4. ***return*** $\langle \langle D_1, D_2, \ldots, D_k \rangle, \langle E_1, E_2, \ldots, E_k \rangle, E^* \rangle$

DECOMPOSE ENDS

---

described in [44] for computing this set. The resulting implementation, which we call DOMINATE (Function 5.3.2), incurs $\mathcal{O}\left(\left(n + \frac{m}{B}\right) \cdot \log n\right)$ cache-misses and runs in $\mathcal{O}\left((n + m) \cdot \log n\right)$ time.

The DOMINATE function receives an undirected graph $G = (V, E)$ and a *degree threshold* s as inputs, and outputs a pair (D, E^*), where D is a set of size $\mathcal{O}\left(\frac{n \log n}{s}\right)$ dominating the set of vertices of degree at least s in G, and $E^* \subseteq E[G]$ is a set of size $\mathcal{O}(n)$ such that for every $u \in V$ with degree at least s, there is an edge $(u, v) \in E^*$ with $v \in D$.

The greedy algorithm in [44] starts with an empty set D, and for each $u \in V$, a list of neighbors of u with degree at least s that are yet to be dominated by the vertices of D. Each vertex is inserted into a priority queue with the number of its 'yet to be dominated neighbors' as the key. The algorithm then enters a loop in each iteration of which a vertex with the largest key is extracted from the priority queue and added to D. If a vertex u (with degree $\geq s$) now becomes dominated we update the 'yet to be dominated neighbors' lists and keys of all neighbors of u.

In our cache-efficient implementation of the algorithm described above, we use our cache-oblivious buffer heap (see Chapter 3) as a priority queue which supports *Insert*, *Delete-Max* and *Relative-Decrease* operations in $\mathcal{O}(\log N)$ amortized time and $\mathcal{O}\left(\frac{1}{B} \log \frac{N}{M}\right)$ amortized cache-misses each, where N is the number of elements in the priority queue. An INSERT$_{(Q)}$(x, k_x) operation inserts the element x with key value k_x into the priority queue Q, a DELETE-MAX$_{(Q)}$( ) extracts an element with the largest key from Q, and a RELATIVE-DECREASE$_{(Q)}$(x, δ_x) operation decreases the current key of x in Q by $\delta_x \geq 0$.

Additionally, we use a *Buffered Repository Tree* (BRT) [11] which for each vertex of G keeps track of the set of its neighbors currently dominated by D. A BRT maintains $\mathcal{O}(m)$ elements with keys in the range $[1 \ldots n]$ under the operations *Insert* and *Extract*. An $\textsc{Insert}_{(T)}(\ v, u\ )$ operation inserts a new element v with key u into the BRT T, while an $\textsc{Extract}_{(T)}(\ u\ )$ operation reports and deletes from T all elements v with key u. The *Insert* and *Extract* operations are supported in $\mathcal{O}\left(\frac{1}{B}\log n\right)$ and $\mathcal{O}(\log n)$ amortized cache-misses, respectively. Amortized time bound of each operation is $\mathcal{O}(\log n)$.

In our cache-efficient implementation, i.e., in the DOMINATE function, we use the above two data structures as follows. For each $u \in V$, we first construct its adjacency list A_u and the set L_u of neighbors of u with degree at least s. We start with an empty dominating set D, and an empty BRT T which for each $u \in V$, will maintain the subset of L_u currently dominated by D. A buffer heap Q is used to keep track for each $u \in V$, the number of its neighbors in L_u that are not yet dominated by D. Initially, we insert each u into Q with $|L_u|$ as its key. Every time a vertex u with the maximum key is extracted from Q and the key is nonzero, we add u to D, and update Q and T to reflect this change. If $deg(u) \geq s$, then for each $v \in A_u$, we update Q and T to indicate that one more neighbor (u) of v is now dominated by D, that is, we decrease the key value of v in Q by 1, and insert u with key value v into T. We extract from T the set L'_u of neighbors of u that have degree at least s and are dominated by $D \setminus \{u\}$, and construct from it the set $L''_u = L_u \setminus L'_u$. Each vertex $v \in L''_u$ will be dominated by u through the edge (u, v), and we add (u, v) to E^*. For each $x \in A_v$, we also update the entries of x in Q and T to take into account that neighbor v of x is now dominated by D.

Correctness of DOMINATE follows from the correctness of the greedy algorithm in [44] which it implements, and also from the correctness of buffer heap operations (see Chapter 3) and the BRT structure (see [11]).

Cache Complexity of DOMINATE. Step 1 can be implemented in a constant number of sorting and scanning phases on V and E, and thus incurs $\mathcal{O}\big(sort(n + m) + scan(n + m)\big) = \mathcal{O}(sort(m))$ cache-misses. Steps 2 and 3 perform at most $2n + m$ operations on Q which cause $\mathcal{O}\left(\frac{m+n}{B}\log\frac{m+n}{M}\right) = \mathcal{O}\left(\frac{m}{B}\log\frac{m}{M}\right)$ cache-misses. At most n EXTRACT and m INSERT operations are performed on T in step 3 incurring $\mathcal{O}\left(\left(n + \frac{m}{B}\right)\log n\right)$ cache-misses. The sorting and scanning phases in step 3 incur $\mathcal{O}(sort(n + m) + scan(n + m)) = \mathcal{O}(sort(m))$ cache-misses. The overall cache complexity of DOMINATE is thus $\mathcal{O}\big(\big(n + \frac{m}{B}\big)\log n + \frac{m}{B}\log\frac{m}{M} + sort(m)\big) = \mathcal{O}\big(\big(n + \frac{m}{B}\big)\log n\big)$.

We present another function, called DECOMPOSE (see Function 5.3.3), which is a cache-efficient implementation of a function with the same name described in [44], and uses DOMINATE as a subroutine. The function receives an undirected graph $G = (V, E)$, and a decreasing sequence $s_1 > s_2 > \ldots > s_{k-1}$ of degree thresholds as inputs. It produces a decreasing sequence of edge sets $E_1 \supseteq E_2 \supseteq \ldots \supseteq E_k$, where

$E_1 = E$, and for $1 < i \leq k$, the set E_i contains edges that touch vertices of degree at most s_{i-1}. Clearly, $|E_i| \leq ns_{i-1}$ for $1 < i \leq k$. This function also produces a sequence of dominating sets $D_1, D_2, \ldots, D_k$, and an edge set E^*. For $1 \leq i < k$, the set D_i dominates all vertices of degree at least s_i, while D_k is simply V. The set $E^* \subseteq E$ is a set of edges such that if the degree of a vertex u is at least s_i then there exists an edge $(u, v) \in E^*$ with $v \in D_i$. Clearly, $|E^*| \leq kn$. The implementation is simple, and so we do not describe it here (see Function 5.3.3 for details).

Cache Complexity of DECOMPOSE. The sorted list E' in step 1 can be constructed in a constant number of sorting and scanning phases on V and E, and thus incurs $\mathcal{O}\left(sort(m)\right)$ cache-misses. Step 2 scans E' exactly $k-1$ times causing $\mathcal{O}\left(k \cdot \frac{m}{B}\right)$ cache-misses. Step 3 calls DOMINATE $k-1$ times resulting in $\mathcal{O}\big(k \cdot (n + \frac{m}{B}) \cdot \log n\big)$ cache-misses. The cache complexity of DECOMPOSE is thus $\mathcal{O}\big(k \cdot (n + \frac{m}{B}) \cdot \log n + k \cdot \frac{m}{B} + sort(m)\big) = \mathcal{O}\big(k \cdot (n + \frac{m}{B}) \cdot \log n\big)$.

5.3.4 Replacing SSSP with BFS for Cache-efficiency

In step 3 of DHZ-APPROX-AP-BFS$_k$, for $i = 1, 2, \ldots, k$, an SSSP algorithm is run from each $u \in D_i$ on a graph $G_i(u) = (V, E_i(u))$, where $E_i(u) = E_i \cup E^* \cup (\{u\} \times V)$. The edges $E_i \cup E^*$ are the original edges of the graph. But the edges $\{u\} \times V$ are not necessarily the edges of the input graph, and to such an edge (u, v) an weight of $\widehat{\delta}(u, v)$ is attached, where $\widehat{\delta}(u, v)$ is the current best known upper bound on the shortest distance from u to v in G. Initially, $\widehat{\delta}(u, v) = 1$ if $(u, v) \in E$ and $\widehat{\delta}(u, v) = \infty$ otherwise.

Since BFS can be made more cache-efficient than SSSP, we replace the SSSP in step 3 with a BFS algorithm. But this requires us to transform the weighted graph $G_i(u)$ into an unweighted graph of comparable size.

Transforming $G_i(u)$ into an Unweighted Graph. Since the distances we compute are non-negative integers smaller than n, we can, in fact, transform $G_i(u)$ into an unweighted graph $G'_i(u)$ by introducing $n-2$ new vertices along with at most $2n-3$ new unweighted directed edges instead of the weighted undirected edges of $\{u\} \times V$ while preserving the shortest distances from u to all other vertices in V. We introduce $n-2$ new vertices $v'_2, v'_3, \ldots, v'_{n-1}$, and introduce the directed edges $(u, v'_2), (v'_2, v'_3), (v'_3, v'_4), \ldots, (v'_{n-2}, v'_{n-1})$. For each $v \in V$ with $\widehat{\delta}(u, v) = 1$, we add a directed edge (u, v), and for each $v \in V$ with $2 \leq \widehat{\delta}(u, v) = t \leq n-1$, we add a directed edge (v'_t, v) (see Figure 5.1). The resulting graph $G'_i(u)$ is partially directed.

We have the following lemma:

Lemma 5.3.1. *The unweighted partially directed graph $G'_i(u)$ obtained from the weighted undirected graph $G_i(u) = (V, E_i(u))$ preserves the shortest distances from u to all other vertices in V.*

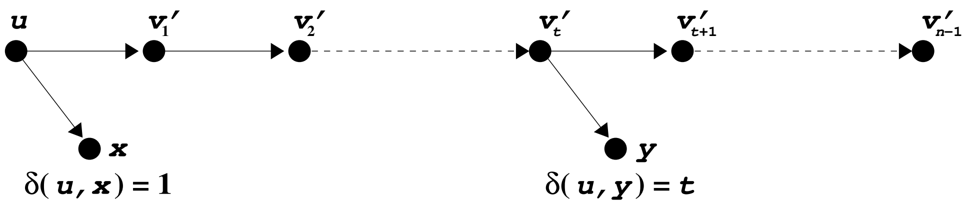

Figure 5.1: The directed unweighted edges that replace the undirected weighted edges of $G_i(u)$.

Proof. We observe that for every $v \in V$ to which a finite upper bound on the shortest distance from u is known, we introduce exactly one u to v path containing only directed edges. Let us call a path that contains no directed edges *old path*, and a path that contains at least one directed edge *new path*. Now suppose $G'_i(u)$ contains a shortest path from u to some vertex $v \in V$ that is shorter than the shortest u to v path in $G_i(u)$. This path must be of the form $P_1 \cdot P_2$, where P_1 is a subpath (new or old) from u to some vertex $x \in V$ $(x \neq u, v)$, and P_2 is a subpath (new) from x to v. But since u is the only entry point to the new directed edges introduced in $G'_i(u)$, P_2 must contain a path from u to v. Thus the path $P_1 \cdot P_2$ is not simple, and so cannot be a shortest path. ∎

Handling the Partial Directedness in $G'_i(u)$. We can modify the MR-BFS algorithm in section 5.2.1 to correctly handle the partial directedness in $G'_i(u)$ with only $\mathcal{O}(scan(m) + sort(n))$ additional cache-misses, and thus without degrading its cache complexity. The algorithm will receive $G'_i(u)$ as an undirected graph, and will implicitly handle the edges that are intended to be directed. It must ensure the following.

(a) $L(i)$ must not contain any other v'_j's except v'_{i+1}, and

(b) if BFS level of a vertex v is $< i$, any edge (v'_{i+1}, v) must not force v to be included in $L(i)$.

Ensuring (a) is straight-forward, but in order to ensure (b) we use an cache-optimal priority queue supporting *Insert* and *Delete-Min* [10] that keeps track of the visited vertices connected to the v'_j's. The modifications are detailed in Modified-MR-BFS. It performs at most one *Insert* and one *Delete-Min* for each edge of the form (v'_j, v), and thus causes $\mathcal{O}(sort(n))$ extra cache-misses [10]. An additional $\mathcal{O}(scan(m))$ cache-misses result from scanning the adjacency lists.

Correctness of Modified-MR-BFS. Since the correctness of MR-BFS has already been proved in section 5.2.1, in order to prove the correctness of Modified-MR-BFS we only need to show that it correctly handles the partial directedness

FUNCTION **5.3.4.** MODIFIED-MR-BFS($G'_i(u)$, u)

[The input graph $G'_i(u) = (V, E)$ is given as an undirected graph but with implicit directed edges as discussed in section 5.3.4. This algorithm is a version of Munagala & Ranade's BFS algorithm modified to perform BFS on this implicitly partially directed graph from the source vertex u. For each vertex $v \in V$, its BFS level is stored in $d[v]$. Let $n = |V|$ and $m = |E|$.**]**

1. $\forall_{v \in V}\ d[v] \leftarrow \infty$
$Q \leftarrow \emptyset$ {*a cache-optimal priority queue supporting* Insert, Find-Min *and* Delete-Min}
$L(-1) \leftarrow \emptyset$, $L(0) \leftarrow \{\ s\ \}$ {*for* $i \in [0, n-1]$, $L(i)$ *will contain the vertices at BFS level* i}
$S \leftarrow \{\ (s, 0)\ \}$ {*at the start of iteration* $i \in [1, n]$ *of the* ***for*** *loop in step 2,* $S = \{\ (v,\ j)\ |\ \ j \in [0, i-1] \wedge v \in L(j)\ \}$}

2. ***for*** $i \leftarrow 1$ ***to*** $n - 1$ ***do***

(*i*) ***for each*** $v \in L(i-1)$ ***do***
for each $(v, v'_{j+1}) \in A_v$ ***with*** $j \geq i$ ***do*** {A_v *is the adjacency list of* v}
INSERT$_{(Q)}$(v, j) {*insert* v *into* Q *with key* j}

(*ii*) $P \leftarrow \emptyset$, $(\ v, k\ ) \leftarrow$ FIND-MIN$_{(Q)}$( )
while $k = i$ ***do*** {*extract all vertices with key value* i *from* Q}
DELETE-MIN$_{(Q)}$( ), $P \leftarrow P \cup \{\ v\ \}$
$(\ v, k\ ) \leftarrow$ FIND-MIN$_{(Q)}$( )
Sort P by vertex indices

(*iii*) Construct $N(L(i-1))$ by concatenating the adjacency lists of the vertices in $L(i-1)$
{*for a set of vertices* S, $N(S)$ *is a multiset of all vertices adjacent to* S}

(*iv*) Remove duplicates and all v'_j's from $N(L(i-1))$,
and let $L'(i)$ be the resulting set

(*v*) Construct $L(i) = L'(i) \setminus \{L(i-1) \cup L(i-2) \cup P\} \cup \{\ v'_{i+1}\ \}$ by scanning the lists

(*vi*) Construct $S \leftarrow S \cup \{\ (v,\ i)\ |\ \ v \in L(i)\ \}$

3. Update $d[v]$ for all $v \in V$ by sorting S by vertex indices, followed by a scan

4. ***return*** d

MODIFIED-MR-BFS ENDS

implicit in its input graph. To show this we need to prove that if the BFS level of a vertex v is less than i, then no edge of the form (v'_{i+1}, v) can force v to be included in $L(i)$. In iteration i, we insert each vertex $v \in L(i-1)$ into Q with a key $j \geq i$ provided the edge (v'_{j+1}, v) exists. Then we extract from Q each vertex v that has an incoming edge from v'_{i+1}. These vertices must have BFS level less than i and are excluded from $L(i)$. Vertices can be extracted from Q using *Delete-Min* operations because initially (before iteration 1) Q is empty, and iteration i never inserts into Q any vertex with a key value less than i, and these insertions always precede the *Delete-Min* operations in the same iteration.

5.3.5 Cache-efficient Approximate AP-BFS

As pointed out in Section 5.3.2, there are two ways of applying the BFS in step 3 of DHZ-Approx-AP-BFS$_k$: either we can run BFS independently from each vertex in D_i as in DHZ-Approx-AP-BFS$_k$, or we can run BFS incrementally from the vertices of D_i using the strategy used in AP-BFS (see Section 5.2.2).

We present the algorithm Independent-BFS which when called with D_i as a parameter constructs the implicitly partially directed unweighted graph $G'_i(u)$ for each $u \in D_i$ and runs Mehlhorn & Meyer's sublinear I/O BFS algorithm [88] on $G'_i(u)$ from u. The cache complexity of Mehlhorn & Meyer's algorithm is $\mathcal{O}\big(\sqrt{\frac{mn}{B}} + \frac{m}{B} \cdot \log n\big)$ as opposed to the $\mathcal{O}\left(n + sort(m)\right)$ cache complexity of Munagala & Ranade's algorithm (MR-BFS in Section 5.2.1), and thus it performs better on sparse graphs. Mehlhorn & Meyer's algorithm is based on MR-BFS, and can be modified in exactly the same way to handle the implicit partial directedness in $G'_i(u)$. The cache complexity of Independent-BFS is thus $\mathcal{O}\left(D_i \cdot \left(\sqrt{\frac{n|E_i|}{B}} + \frac{|E_i|}{B} \cdot \log n\right)\right)$.

The algorithm Interdependent-BFS when called with parameter D_i, constructs $G'_i(u)$ for each $u \in D_i$, and then runs Modified-MR-BFS (Section 5.3.4) incrementally on $G'_i(u)$ from each u using the technique used in AP-BFS (Section 5.2.2). The main differences between Interdependent-BFS and AP-BFS are: Interdependent-BFS uses a different range for locating the adjacency lists, works on a slightly different graph in each iteration, each graph it works on is partially directed, and runs BFS only from the vertices in D_i. The cache complexity of Interdependent-BFS is $\mathcal{O}\left(\frac{|E_i|}{B} \cdot (n + i \cdot |D_i|) + |D_i| \cdot sort(|E_i|)\right)$.

We observe that running Independent-BFS in step 3 of DHZ-Approx-AP-BFS$_k$ is more cache-efficient when $|D_i|$ is smaller and $G'_i(u)$ is denser (i.e., value of i is smaller), and Interdependent-BFS is more cache-efficient when $|D_i|$ is larger and $G'_i(u)$ is sparser (i.e., value of i is larger). If we use Independent-BFS for all values of i, it will incur a total of $\mathcal{O}\left(\frac{n^2}{\sqrt{B}} + \frac{\mathcal{T}(m,n,k)}{B}\right)$ cache-misses, and running Interdependent-BFS for all values of i causes a total of $\mathcal{O}\left(\frac{mn}{B} + \frac{\mathcal{T}(m,n,k)}{B}\right)$ cache-misses, where $\mathcal{T}(m,n,k) = \mathcal{O}\left(k \cdot \left(\frac{m}{n \log n}\right)^{\frac{1}{k}} \cdot n^2 \log n\right)$. Therefore, we can do better if we take a hybrid approach: starting from $i = 1$ we run Independent-BFS up to some value l of i, and then we switch to Interdependent-BFS. We call this parameter l a *switching parameter*, and choose its value in order to minimize the cache complexity of the entire algorithm. The overall algorithm is given in Approx-AP-BFS$_k$.

Correctness of Approx-AP-BFS$_k$. The correctness of Approx-AP-BFS$_k$ follows from the following:

(*a*) correctness of DHZ-Approx-AP-BFS$_k$,

FUNCTION **5.3.5.** INDEPENDENT-BFS($G = (V, E),\ D_i,\ E_i,\ E^*,\ \mathcal{L}_\delta$)

[Invoked by APPROX-AP-BFS. See APPROX-AP-BFS for the definition of the parameters. Performs BFS independently from each vertex $u \in D_i$ on a graph constructed from V, E_i and E^*, and the information in the list $\mathcal{L}_\delta$ of current best upper bounds on all-pairs shortest distances in the original graph G. It updates $\mathcal{L}_\delta$ with the computed distances. Here, $n = |V|$ and $m = |E|$.**]**

1. $\mathcal{L}'_\delta \leftarrow \emptyset$
Sort D_i by vertex indices

2. ***for each*** $u \in D_i$ ***do***

(i) Retrieve from $\mathcal{L}_\delta$ the current best upper bound $\widehat{\delta}(u, v)$ on the shortest distance from u to each $v \in V$

(ii) Construct $G'_i(u) = (V', E')$ as follows:

(a) $V' \leftarrow V \cup \{v'_2, v'_3, \ldots, v'_{n-1}\}$

(b) $E' \leftarrow E_i \cup E^* \cup \{\ (u, v'_2)\ \} \cup \{\ (v'_t, v'_{t+1}) \mid t \in [2, n-2]\ \}$
$\cup \left\{\ (u, v) \mid v \in V\ \wedge\ \widehat{\delta}(u, v) = 1\ \right\}$
$\cup \left\{\ (v'_t, v) \mid v \in V\ \wedge\ 1 < \widehat{\delta}(u, v) = t < \infty\ \right\}$

(iii) Run on $G'_i(u)$ the sublinear I/O BFS in [88] modified to handle the partial directedness in the graph, and append the computed shortest distances to $\mathcal{L}'_\delta$

3. Update $\mathcal{L}_\delta$ by sorting $\mathcal{L}'_\delta$ appropriately and scanning the two lists simultaneously

4. ***return*** $\mathcal{L}_\delta$

INDEPENDENT-BFS ENDS

(b) lemma 5.3.1,

(c) correctness of MODIFIED-MR-BFS,

(d) correctness of Mehlhorn & Meyer's BFS algorithm [88] modified to handle the type partial directedness in the input graph as described in Section 5.3.4, and

(e) the guarantee that in step $3(d)$ of INTERDEPENDENT-BFS, all adjacency lists are found within the range searched.

Proof of (a) can be found in [44]. Proofs of (b) and (c) are given in Section 5.3.4. Proof of (d) follows from the proof of (c) since Mehlhorn & Meyer's algorithm builds on Munagala & Ranade's BFS algorithm [92] (MR-BFS in Section 5.2.1), and the modifications required are exactly the same.

We only need to prove (e). For $1 \le i \le k$, and $u, v \in V$, let $\delta_i(u, v)$ be the value of $\widehat{\delta}(u, v)$ after running BFS from all vertices of D_i. It has been shown in [44] that if $u \in D_i$ and $v \in V$, DHZ-APPROX-AP-BFS$_k$ maintains $\delta(u, v) \le \delta_i(u, v) \le \delta(u, v) + 2(i-1)$. The algorithm APPROX-AP-BFS$_k$ also clearly maintains this invariant up to level l. The first vertex in D_{l+1} also computes its distances with

FUNCTION **5.3.6.** INTERDEPENDENT-BFS($G = (V, E)$, D_i, E_i, E^*, $\langle\ v_1,\ v_2,\ \ldots,\ v_n\ \rangle$, $\mathcal{L}_\delta$)

[Invoked by APPROX-AP-BFS. See APPROX-AP-BFS for the definition of the parameters. Performs BFS from each $u \in D_i$ on a graph constructed from V, E_i and E^*, and the information in the list $\mathcal{L}_\delta$ of current best upper bounds on all-pairs shortest distances in G. BFS is performed on the vertices of D_i in the order they appear in $\langle v_1, v_2, \ldots, v_n\rangle$, and distance information obtained from the BFS run immediately preceding the current run is used to reduce cache-miss overhead. List $\mathcal{L}_\delta$ is updated with the computed distances. Here, $n = |V|$ and $m = |E|$.]

1. $\mathcal{L}'_\delta \leftarrow \emptyset$
   Permute D_i
   Let $\langle u_1, u_2, \ldots, u_{|D_i|}\rangle$ be the vertices in D_i in the order they appear in $\langle v_1, v_2, \ldots, v_n\rangle$

2. (i) Construct $G'_i(u_1) = (V'_1, E'_1)$ as in steps $2(a)$ and $2(b)$ of INDEPENDENT-BFS, but with u_1 instead of u
   (ii) $d(\ u_1,\ \cdot\ ) \leftarrow$ MODIFIED-MR-BFS($G'_i(u_1)$, u_1)
   Append $d(\ u_1,\ \cdot\ )$ to $\mathcal{L}'_\delta$

3. ***for*** $j \leftarrow 2$ ***to*** $|D_i|$ ***do***
   (i) Construct $G'_i(u_j) = (V'_j, E'_j)$ as in steps $2(a)$ and $2(b)$ of INDEPENDENT-BFS, but with u_j instead of u
   (ii) Sort $A' = \{\ (v'_r, v) \mid (v'_r, v) \in E'_j\ \}$ to place (v'_p, x) before (v'_q, y) provided $\langle p, x\rangle < \langle q, y\rangle$
   (iii) Sort $E' = E'_j \setminus A'$ to place (x, x') before (y, y') if $\langle d(u_{j-1}, x), x, x'\rangle < \langle d(u_{j-1}, y), y, y'\rangle$.
   Let $A(p)$ be the section of E' containing edges (x, x') for $\forall x \in V$ with $d(u_{j-1}, x) = p$.
   (iv) Find $d(u_j, \cdot)$ by calling MODIFIED-MR-BFS($(V'_j, E' \cup A'), u_j$),
   but compute each $N(L(q-1))$ as follows:
      $N(L(q-1)) \leftarrow \emptyset$, $\Delta \leftarrow d(u_{j-1}, u_j) + 2(i-1)$
      for $r \leftarrow \max\{\ 0,\ q-1-\Delta\ \}$ ***to*** $\min\{\ n-1,\ q-1+\Delta\ \}$ ***do***
         Scan $L(q-1)$, $A(r)$ and A', and append $\{\ y \mid x \in L(q-1)\ \wedge\ (x, y) \in A(r) \cup A'\ \}$ to $N(L(q-1))$
   Append the computed shortest distances to $\mathcal{L}'_\delta$

4. Update $\mathcal{L}_\delta$ by sorting $\mathcal{L}'_\delta$ appropriately and scanning the two lists simultaneously

5. ***return*** $\mathcal{L}_\delta$

INTERDEPENDENT-BFS ENDS

surplus error of at most $2l$. This is the base case. Now suppose all distances were calculated with surplus error of at most $2(i-1)$ up to the $(j-1)$-th vertex u_{j-1} of some D_i, where $1 < j \leq |D_i|$ and $l < i \leq k$. We will now prove that for $u_j \in D_i$ the adjacency lists will be found within the range in step $3(iv)$ of INTERDEPENDENT-BFS. Let v be any vertex in V. We have the following three invariants from DHZ-APPROX-AP-BFS$_k$:

FUNCTION **5.3.7.** APPROX-AP-BFS$_k$(G , l)

[Given an unweighted undirected graph $G = (V, E)$ and a switching parameter l, this algorithm computes the shortest distance between every pair of vertices in G with additive error of at most $2(k-1)$. Let $n = |V|$ and $m = |E|$.**]**

1. Perform the following initializations:

- (i) ***for*** $i \leftarrow 1$ ***to*** $k-1$ ***do*** $s_i \leftarrow \frac{m}{n}\left(\frac{n \log n}{m}\right)^{\frac{i}{k}}$
- (ii) $\langle\langle D_1, D_2, \ldots, D_k\rangle, \langle E_1, E_2, \ldots, E_k\rangle, E^*\rangle \leftarrow$ DECOMPOSE(G, $\langle s_1, s_2, \ldots, s_{k-1}\rangle$)
- (iii) Sort E so that (u_1, v_1) is placed before (u_2, v_2) provided $\langle\ u_1, v_1\ \rangle < \langle\ u_2, v_2\ \rangle$.
- (iv) Scan E to produce a sorted list $\mathcal{L}_\delta$ (with the same ordering as E) of approximate distances $\widehat{\delta}(u, v)$, where $u, v \in V$, and $\widehat{\delta}(u, v) \leftarrow 1$ if $(u, v) \in E$, $\widehat{\delta}(u, v) \leftarrow \infty$ otherwise.

2. ***for*** $i \leftarrow 1$ ***to*** l ***do***
$\mathcal{L}_\delta \leftarrow$ INDEPENDENT-BFS(G, D_i, E_i, E^*, $\mathcal{L}_\delta$)

3. (i) Perform the following initializations:

- (a) $T \leftarrow$ a spanning tree of G
- (b) $ET \leftarrow$ an *Euler Tour of* T
- (c) Mark the first occurrence of each vertex on ET.
  Let $v_1, v_2, \ldots, v_n$ be the marked vertices in the order they appear on ET.

(ii) ***for*** $i \leftarrow l+1$ ***to*** k ***do***
$\mathcal{L}_\delta \leftarrow$ INTERDEPENDENT-BFS(G, D_i, E_i, E^*, $\langle\ v_1,\ v_2,\ \ldots,\ v_n\ \rangle$, $\mathcal{L}_\delta$)

4. ***return*** $\mathcal{L}_\delta$

APPROX-AP-BFS$_k$ ENDS

$$\begin{array}{rcccl}
\delta(u_{j-1}, u_j) & \leq & \delta_i(u_{j-1}, u_j) & \leq & \delta(u_{j-1}, u_j) + 2(i-1) \\
\delta(u_{j-1}, v) & \leq & \delta_i(u_{j-1}, v) & \leq & \delta(u_{j-1}, v) + 2(i-1) \\
\delta(u_j, v) & \leq & \delta_i(u_j, v) & \leq & \delta(u_j, v) + 2(i-1)
\end{array}$$

We also have the following two triangle inequalities:

$$\begin{array}{rcl}
\delta(u_{j-1}, v) & \leq & \delta(u_{j-1}, u_j) + \delta(u_j, v) \\
\delta(u_j, v) & \leq & \delta(u_j, u_{j-1}) + \delta(u_{j-1}, v)
\end{array}$$

From the five inequalities above, we get

$$\delta_i(u_{j-1}, v) - \delta_i(u_{j-1}, u_j) - 2(i-1) \leq \delta_i(u_j, v) \leq \delta_i(u_{j-1}, v) + \delta_i(u_{j-1}, u_j) + 2(i-1)$$

Therefore the range used in step $3(iv)$ of INTERDEPENDENT-BFS is sufficient for finding the corresponding adjacency lists. Note that distances for the first vertex in any D_i, $l < i \leq k$, are computed correctly (with surplus error of at most $2(i-1)$) assuming that distances for all vertices in all D_j's with $1 \leq j \leq l$ are already computed correctly (with surplus error of at most $2(j-1)$).

Cache Complexity of Approx-AP-BFS$_k$. We note that $|D_i| = \mathcal{O}\left(\frac{n \log n}{s_i}\right)$ for $i \in [1, k-1]$, $|D_k| = n$, $|E_1| = m$, $|E_i| \leq n s_{i-1}$ for $i \in [2, k]$, and $|E^*| \leq kn$. We also have $s_i = \frac{m}{n} \cdot \alpha^i$ for $i \in [1, k-1]$, where $\alpha = (\frac{n \log n}{m})^{\frac{1}{k}}$. The assumption $m \geq n \log n$ ensures that the degree sequence $\langle\ s_1,\ s_2,\ \ldots,\ s_{k-1}\ \rangle$ is decreasing, and also that $|E^*| \leq n s_{k-1}$ for $k \in [2, \log n]$.

Cache complexity of step 1 is dominated by the cache performance of Decompose, and so this step incurs $\mathcal{O}\left(k \cdot \left(n + \frac{m}{B}\right) \cdot \log n\right)$ cache-misses.

For $i = 1$ to l, iteration i of step 2 calls Independent-BFS with D_i as a parameter which in turns runs the $\mathcal{O}\left(\sqrt{\frac{n|E_i|}{B}} + \frac{|E_i|}{B} \log n\right)$ I/O BFS algorithm by Mehlhorn & Meyer [88] (modified as outlined in Modified-MR-BFS to handle the partial directedness implicit in the input graph) from each vertex $u \in D_i$ on the graph $G'_i(u)$. Thus this step incurs $\mathcal{O}\left(\sum_{i=1}^{l} |D_i| \cdot \left(\sqrt{\frac{n|E_i|}{B}} + \frac{|E_i|}{B} \log n\right)\right) =$ $\mathcal{O}\left(\frac{l}{k} \cdot \frac{\mathcal{T}(m,n,k)}{B} + n^2 \log n \sqrt{\frac{n}{Bm\alpha^{l+1}}}\right)$ cache-misses, where $\mathcal{T}(m, n, k) = \mathcal{O}\big(k \cdot \left(\frac{m}{n \log n}\right)^{\frac{1}{k}} \cdot$ $n^2 \log n\big)$.

Cache complexity of step 3 is determined by the cache performance of step 3(ii). For $i = l+1$ to k, iteration i of step 3(ii) calls Interdependent-BFS with D_i as a parameter incurring $\mathcal{O}\left(\frac{|E_i|}{B} \cdot (n + i \cdot |D_i|) + |D_i| \cdot sort(|E_i|)\right)$ cache-misses. Cache complexity of step 3(b) is thus $\mathcal{O}\Big(\sum_{i=l+1}^{k} \left(\frac{|E_i|}{B} \cdot (n + i \cdot |D_i|) + |D_i| \cdot sort(|E_i|)\right)\Big)$ $= \mathcal{O}\big(\frac{k-l}{k} \cdot \frac{\mathcal{T}(m,n,k)}{B} + \frac{mn\alpha^{l-1}}{B}\big)$.

Therefore, the cache complexity of Approx-AP-BFS$_k$ is $\mathcal{O}\Big(\frac{\mathcal{T}(m,n,k)}{B} + \sqrt{\frac{n^3 \log^2 n}{Bm\alpha^{l+1}}} + \frac{mn\alpha^{l-1}}{B}\Big)$. We determine the optimal value of l by equating the last two terms of this expression, and get $l = \frac{\log(Bn^3 \log^2 n) - \log(m^3 \alpha)}{3 \log \alpha} + 1$. For this value of l the cache complexity of the algorithm is $\mathcal{O}\left(\frac{\mathcal{T}(m,n,k)}{B} + \left(\frac{\mathcal{T}(m,n,k)}{B} \cdot \frac{n}{k}\right)^{\frac{2}{3}}\right)$. Furthermore, if $\log n \geq \frac{B}{8}$ also holds, the cache complexity reduces to $\mathcal{O}\left(\frac{\mathcal{T}(m,n,k)}{B}\right)$.

5.4 Cache-aware APSP for Bounded-weight Undirected Graphs

In this Section, we present a cache-aware APSP algorithm for real-weighted undirected graphs. On a graph with n vertices and m edges our algorithm incurs

$\mathcal{O}\left(\frac{n^2}{\beta^{\frac{2}{3}}} + n \cdot sort(m)\right)$ cache-misses, where $\beta = \left(\frac{n}{m \log \rho}\right) \cdot B$ and ρ is the ratio of the largest and the smallest edge-weights in the graph. We assume that $\beta \geq 1$. Our algorithm builds on Meyer & Zeh's bounded-weight undirected SSSP algorithm [89], and improves over $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{1}{2}}} + n \cdot sort(m)\right)$, i.e., the best known previous cache-miss bound for the problem, when $m = \mathcal{O}\left(\frac{B}{\log \rho} \cdot n\right)$. We also show that the cache complexity of our algorithm can be further improved to $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{3}{4}}} \cdot \frac{1}{\log^{\frac{1}{4}} \rho} + n \cdot sort(m)\right)$ when $m = \mathcal{O}\left(\frac{B}{\max\{\ \log \rho,\ \log^2 n\ \}} \cdot n\right)$.

5.4.1 Meyer & Zeh's Bounded-weight Undirected SSSP Algorithm

Given a weighted undirected graph $G = (V, E)$ with the smallest and the largest edge-weights 1 and ρ, respectively, a source vertex $s \in V$, and a parameter μ, Meyer & Zeh's algorithm computes the shortest distance from s to all vertices in V in $\mathcal{O}\left(\frac{n}{\mu} + \frac{\mu m}{B} \log \rho + sort(m) \log \log \frac{nB}{m}\right)$ cache-misses. The algorithm has the following two phases.

<u>Clustering Phase</u>. The vertex set V is partitioned into a set of clusters $\{\ V_1, V_2, \ldots, V_q\ \}$, and the adjacency lists of the vertices in each cluster V_j are placed in a separate cluster file $\mathcal{F}_j$. The category of a cluster V_j is defined as the smallest integer i such that V_j is completely contained in a connected component of $G_i = (V, E_i)$, where $E_i = \{\ (u, v) \in E \mid w(u, v) < 2^i\ \}$. The computed cluster partition is guaranteed to have the following properties:

1. $q = \mathcal{O}\left(\frac{n}{\mu}\right)$,
2. if V_j is a category-i cluster then no edge (u, v) with $w(u, v) < 2^{i-1}$ exists such that $u \in V_j$ and $v \in V_j$, and
3. no category-i cluster has diameter greater than $2^i \mu$.

In fact, in order to maintain the correctness and the performance bounds of the algorithm it suffices to guarantee the following instead of property 3:

$3'$. If u is a vertex nearest to s in a cluster V_j, then no vertex in V_j is at a distance greater than $2^i \mu$ from u.

This phase incurs $\mathcal{O}\left(\frac{n}{B} \log \rho + sort(m) \log \log \frac{nB}{m}\right)$ cache-misses (see [89]).

<u>Shortest Path Phase</u>. At a very high level, this phase is similar to Dijkstra's SSSP algorithm. However, relaxation of edges of settled vertices are delayed as

much as possible, and the first time an edge of a vertex in some cluster V_j is relaxed the entire cluster file $\mathcal{F}_j$ is loaded into a set of hot pools. Then all edges from V_j that need to be relaxed in future can be extracted from the hot pools without any random access. If v is a vertex in V_j nearest to s and v has an edge of weight 1 then $\mathcal{F}_j$ is loaded as soon as v is settled.

The cache-complexity of this phase is $\mathcal{O}\left(\frac{n}{\mu} + \frac{\mu m}{B}\log\rho + sort(m)\right)$ (see [89]), where the $\frac{n}{\mu}$ term results from the $\mathcal{O}\left(\frac{n}{\mu}\right)$ random accesses needed to load the cluster files into the hot pools.

5.4.2 Our Bounded-weight Undirected APSP Algorithm

In order to perform APSP on a bounded-weight undirected graph using Meyer & Zeh's SSSP algorithm, we need to compute a cluster partition only once, and then execute the shortest path phase once from each of the n vertices. The resulting algorithm incurs $\mathcal{O}\left(n \cdot \left(\frac{n}{\mu} + \frac{\mu m}{B}\log\rho + sort(m)\right)\right)$ cache-misses, which reduces to $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{1}{2}}} + n \cdot sort(m)\right)$ for $\mu = \beta^{\frac{1}{2}}$, where $\beta = \left(\frac{n}{m\log\rho}\right) \cdot B \geq 1$.

We can improve the cache-complexity of the APSP algorithm described above by reducing the number of unstructured accesses required to load the cluster files into hot pools. In fact, for each of the n SSSP computations we can predict the exact order in which the cluster files will be loaded, and thus eliminate the unstructured accesses to the set of cluster files altogether. Our algorithm has three phases:

<u>Clustering Phase</u>. We compute a cluster partition $P = \left\{ V_1, V_2, \ldots, V_q \right\}$ of V in exactly the same way as in Meyer & Zeh's SSSP algorithm (see Section 5.4.1), where $q = \mathcal{O}\left(\frac{n}{\mu}\right)$.

<u>Prediction Phase</u>. For any $v \in V$ and $j \in [1, q]$, let $u_{v,j}$ be a vertex in V_j nearest to v, and let $\Delta_{v,j}$ be the shortest distance from v to $u_{v,j}$. We compute $u_{v,j}$ and $\Delta_{v,j}$ for all v and j as follows.

The computation in this phase proceeds in q iterations. In iteration $j \in [1, q]$, we compute $\Delta(v, V_j)$ for all $v \in V$. In this iteration, we construct a new graph $G' = (V', E')$ from G by adding a new vertex u to V and $|V_j|$ new unit weight undirected edges (u, v), $v \in V_j$ to E. Observe that the cluster partition P computed in the clustering phase remains valid for this new graph, i.e., properties 1, 2 and $3'$ of cluster partitions described in Section 5.4.1 are not violated. Therefore, we execute the shortest path phase of Meyer & Zeh's algorithm (see Section 5.4.1) from u on G' with cluster partition P, and compute the shortest distance $d(u, v)$ from u to all $v \in V$ in G'. Clearly, $\Delta_{v,j} = d(u, v) - 1$ for all $v \in V$. Let u' be the vertex following u on the shortest u to v path computed by the algorithm. Then u' is a vertex in V_j nearest to v, and we set $u_{v,j} = u'$.

After the $\Delta_{v,j}$ and $u_{v,j}$ values have been computed for all v and j, we sort them, and for each $v \in V$, construct a list $\mathcal{L}_v$ of $\langle\ j,\ u_{v,j},\ \Delta_{v,j}\ \rangle$ triplets sorted in non-decreasing order of $\Delta_{v,j}$ values.

<u>Shortest Path Phase</u>. In this phase, we consider each vertex $s \in V$ once and perform the following computation.

We first construct the graph $G_s = (V_s, E_s)$ from $G = (V, E)$ as follows. We add q new vertices $u_1, u_2, \ldots, u_q$ to V, and q new undirected edges $(u_1, u_{s,1})$, $(u_2, u_{s,2})$, …, $(u_q, u_{s,q})$ of unit weight to E. It is straight-forward to see that addition of these new vertices and edges do not change the shortest distance from s to any $v \in V$, and also that P remains a valid cluster partition of the vertices in G_s (i.e., properties 1, 2 and $3'$ mentioned in Section 5.4.1 are not violated). Using the $\mathcal{L}_s$ list constructed in the prediction phase, we construct a sorted list $\mathcal{L'}_s$ of cluster files in which cluster file $\mathcal{F}_i$ is placed ahead of $\mathcal{F}_j$ provided $\Delta_{s,i} < \Delta_{s,j}$. We now run the shortest path phase of Meyer & Zeh's SSSP algorithm (see Section 5.4.1) on G_s from s using P as the cluster partition, and thus compute the shortest distance from s to each vertex in V. Observe that since each $u_{s,j}$ has an edge (i.e., $(u_j, u_{s,j})$) of unit weight, the cluster file $\mathcal{F}_j$ will be loaded into the hot pools as soon as $u_{s,j}$ settles. Since the cluster files in $\mathcal{L'}_s$ are sorted in the order the $u_{s,j}$ vertices settle, loading all cluster files requires only a sequential scan of $\mathcal{L'}_s$.

The correctness of our algorithm directly follows from the correctness of Meyer & Zeh's SSSP algorithm [89] and the description above.

Cache Complexity. The clustering phase incurs $\mathcal{O}\left(\frac{n}{B}\log\rho + sort(m)\log\log\frac{nB}{m}\right)$ cache-misses (see [89]). In the prediction phase, each iteration of SSSP computation incurs $\mathcal{O}\left(\frac{n}{\mu} + \frac{\mu m}{B}\log\rho + sort(m)\right)$ cache-misses (see [89]), and there are $q = \mathcal{O}\left(\frac{n}{\mu}\right)$ such iterations. Constructing the L_v lists requires sorting $n \times q$ triplets and thus incurs $\mathcal{O}\left(sort(nq)\right) = \mathcal{O}\left(sort\left(\frac{n^2}{\mu}\right)\right)$ cache-misses. In each iteration of the shortest path phase, constructing the L'_s list requires a constant number of sorting and scanning steps involving the L_s list and the cluster files, and thus incurs $\mathcal{O}\left(sort(m)\right)$ cache-misses. The SSSP computation step can now load the cluster files in sequential scan of L'_s, and thus incurs only $\mathcal{O}\left(\frac{\mu m}{B}\log\rho + sort(m)\right)$ cache-misses.

The overall cache-complexity of the algorithm is thus $\mathcal{O}\left(\left(\frac{n}{\mu}\right)^2 + n \cdot \frac{\mu m}{B}\log\rho + n \cdot sort(m)\right)$. We choose $\mu = \beta^{\frac{1}{3}}$, where $\beta = \left(\frac{n}{m\log\rho}\right) \cdot B \geq 1$, and the cache-complexity reduces to $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{2}{3}}} + n \cdot sort(m)\right)$.

5.4.3 An Improved Algorithm

We can improve the cache performance of the APSP algorithm described in Section 5.4.2 even further by implementing the SSSP computations in its prediction phase using multi-buffer-heaps (see Section 3.4.3 of Chapter 3).

Multi-buffer-heaps (MBH) are constructed from slim buffer heaps. A slim buffer heap (SBH) is a priority queue that supports *Delete*, *Delete-Min* and *Decrease-Key* operations, but it is allowed to retain only a small portion of its data items in the cache between data structural operations. The area in the cache that holds these data items is called a slim cache. By λ we denote the size of the slim cache, and assume that $\lambda < B$. An MBH is constructed by packing the slim caches of $L = \frac{B}{\lambda}$ slim buffer heaps into a single cache block. This cache block is called the multi-slim-cache. This structure supports each priority queue operation on each of its component SBH in $\mathcal{O}\left(\frac{L}{B} + \frac{1}{B}\log_2 \frac{NL}{B}\right)$ amortized cache-misses, where N is the number of items in the SBH (see Chapter 3).

Now the improved prediction phase is implemented as follows. We work on the $q = \mathcal{O}\left(\frac{n}{\mu}\right)$ underlying SSSP problems (see Section 5.4.2) simultaneously (as in [13]), and solve each of them using Kumar & Schwabe's SSSP algorithm [83]. This approach requires a priority queue pair for each SSSP problem. These q priority queue pairs are implemented using $\frac{q}{L}$ multi-buffer-heaps. The phase proceeds in n rounds. In each round it loads the multi-slim-cache of each MBH, and extracts a settled vertex with the minimum distance from each of the L priority queue pairs it stores. The extracted vertices are sorted by vertex indices. The adjacency lists of the settled vertices of this round are extracted by scanning the sorted vertex list and the sorted sequences of adjacency lists in parallel. Another sorting phase moves all adjacency lists to be applied to the same MBH together. Then all necessary *Decrease-Key* operations are performed by cycling through the multi-buffer-heaps once again. At the end of the phase the $\mathcal{L}_v$ lists are constructed as in Section 5.4.2.

Cache Complexity. It is straight-forward to show that the prediction phase as described above incurs $\mathcal{O}\left(n \cdot \left(\frac{n}{\mu L} + \frac{mL}{\mu B} + \frac{m}{\mu B}\log_2 \frac{nL}{B} + sort(m)\right)\right)$ cache-misses (see Section 3.4.3 of Chapter 3 for details), which reduces to $\mathcal{O}\left(\frac{n^2}{\mu}\sqrt{\frac{m}{nB}} + n \cdot sort(m)\right)$ for $L = \sqrt{\frac{nB}{m}} > 1$ and $m = \mathcal{O}\left(\frac{B}{\log^2 n} \cdot n\right)$. Including the cache-misses incurred by the clustering and the shortest path phases (see Section 5.4.2), we get the overall cache complexity of the algorithm $\mathcal{O}\left(\frac{n^2}{\mu}\sqrt{\frac{m}{nB}} + n \cdot \frac{\mu m}{B}\log\rho + n \cdot sort(m)\right)$. We choose $\mu = \left(\frac{\beta}{\log\rho}\right)^{\frac{1}{4}}$, where $\beta = \left(\frac{n}{m\log\rho}\right) \cdot B \geq 1$, and the cache-complexity reduces to $\mathcal{O}\left(\frac{n^2}{\beta^{\frac{3}{4}}} \cdot \frac{1}{\log^{\frac{1}{4}}\rho} + n \cdot sort(m)\right)$ for $m = \mathcal{O}\left(\frac{B}{\max\{\ \log\rho,\ \log^2 n\ \}} \cdot n\right)$.

5.5 Conclusion

In this chapter we have presented the first cache-oblivious APSP algorithm for unweighted graphs, and also the the first nontrivial algorithms for approximate APSP on unweighted graphs, and exact APSP on bounded-weight graphs. All our results are for graphs with undirected edges. However, several open questions still exist. For example:

1. While it is not clear whether our exact AP-BFS algorithm in Section 5.2 is cache-optimal, our approximate AP-BFS algorithm in Section 5.3 and the exact APSP algorithm in Section 5.4 are very unlikely to be so. Therefore, it is highly likely that algorithms with better cache performance can still be developed for the latter two problems.

2. Efficient cache-oblivious algorithms are yet to be developed for the approximate AP-BFS and the bounded-weight exact APSP problems.

3. An obvious open question is: how to extend these algorithms to handle directed graphs cache-efficiently?

Chapter 6

The Cache-oblivious Gaussian Elimination Paradigm

What appear to be different elementary particles are
actually different "notes" on a fundamental string.
The universe – being composed of an enormous number of
these vibrating strings – is akin to a cosmic symphony.
(Brian Greene in *The Elegant Universe*)

In this chapter we introduce the cache-oblivious Gaussian Elimination Paradigm (GEP) to obtain efficient cache-oblivious algorithms for several important problems that have algorithms with triply-nested loops similar to those that occur in Gaussian elimination. These include Gaussian elimination and LU-decomposition without pivoting, all-pairs shortest paths and matrix multiplication.

We prove several important properties of the cache-oblivious framework for GEP, which we denote by I-GEP. We build on these results to obtain C-GEP, a completely general cache-oblivious implementation of GEP that applies to any code in GEP form, and which has the same time and I/O bounds as I-GEP, while using a modest amount of additional space. We then analyze a parallel implementation of the framework and its caching performance for both shared and distributed caches.

'Tiling', an important loop transformation technique employed by optimizing compilers in order to improve temporal locality in nested loops, is a cache-aware method that does not adapt to all levels in a multi-level memory hierarchy. The cache-oblivious GEP framework (either I-GEP or C-GEP) produces system-independent cache-efficient code for triply nested loops of the form that appears in Gaussian elimination without pivoting, and is potentially applicable to being used by optimizing compilers for loop transformation.

6.1 Introduction

In this chapter we use *GEP* or the *Gaussian Elimination Paradigm* to denote a class of problems that can be solved using a construct similar to the computation in Gaussian elimination without pivoting. Traditional algorithms that use this construct fully exploit the spatial locality of data but they fail to exploit the temporal locality, and they run in $\mathcal{O}\left(n^3\right)$ time, use $\mathcal{O}\left(n^2\right)$ space and incur $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses. We present a framework for in-place cache-oblivious execution of several important special cases of GEP including Gaussian elimination and LU-decomposition without pivoting, all-pairs shortest paths and matrix multiplication; this framework can also be adapted to solve important non-GEP dynamic programming problems such as a class of dynamic programs termed as 'simple-DP' [30] which includes algorithms for RNA secondary structure prediction [78], matrix chain multiplication and optimal binary search tree construction. This framework takes full advantage of both spatial and temporal locality of data to incur only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses while still running in $\mathcal{O}\left(n^3\right)$ time and without using any extra space. We denote this framework as I-GEP.

We establish several properties of I-GEP, and then build on these results to derive C-GEP, which has the same time and I/O bounds as I-GEP, but unlike I-GEP is a provably correct cache-oblivious implementation of GEP in its full generality. However, C-GEP uses a modest amount of extra space.

We also consider generalized versions of three major I-GEP applications: Gaussian elimination without pivoting, Floyd-Warshall's APSP, and matrix multiplication. Short proofs of correctness for these applications can be obtained using results we prove for I-GEP. We also provide an algorithm for transforming simple DP to I-GEP.

One potential application of I-GEP and C-GEP framework is in compiler optimizations for the memory hierarchy. 'Tiling' is a powerful loop transformation technique employed by optimizing compilers that improves temporal locality in nested loops. However, this technique is cache-aware, and thus does not produce machine-independent code nor does it adapt simultaneously to multiple levels of the memory hierarchy. In contrast, the cache-oblivious GEP framework produces I/O-efficient portable code for a form of triply nested loops that occurs frequently in practice.

6.1.1 The Gaussian Elimination Paradigm (GEP)

Let $c[1 \ldots n, 1 \ldots n]$ be an $n \times n$ matrix with entries chosen from an arbitrary set $\mathcal{S}$, and let $f : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ be an arbitrary function. The algorithm G given in Figure 6.1 modifies c by applying a given set of updates of the form $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$, where $i,j,k \in [1,n]$. By $\langle i,j,k \rangle$ $(1 \le i,j,k \le n)$ we

$G(c, 1, n)$

(The input $c[1 \ldots n, 1 \ldots n]$ is an $n \times n$ matrix. Function $f(\cdot, \cdot, \cdot, \cdot)$ is a problem-specific function, and Σ_G is a problem-specific set of updates to be applied on c.)

for $k \leftarrow 1$ ***to*** n ***do***
  for $i \leftarrow 1$ ***to*** n ***do***
    for $j \leftarrow 1$ ***to*** n ***do***
      if $\langle i, j, k \rangle \in \Sigma_G$ ***then*** $c[i, j] \leftarrow f(c[i, j], c[i, k], c[k, j], c[k, k])$

Figure 6.1: GEP: Triply nested ***for*** loops typifying code fragment with structural similarity to the computation in Gaussian elimination without pivoting.

$F(X, k_1, k_2)$

(X is a $2^q \times 2^q$ square submatrix of c such that $X[1, 1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$ for some integer $q \geq 0$. Function F assumes the following:
(a) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$
(b) $[i_1, i_2] \neq [k_1, k_2] \Rightarrow [i_1, i_2] \cap [k_1, k_2] = \emptyset$ and $[j_1, j_2] \neq [k_1, k_2] \Rightarrow [j_1, j_2] \cap [k_1, k_2] = \emptyset$.

The initial call to F is $F(c, 1, n)$ for an $n \times n$ input matrix c, where n is a power of 2.)

if $T_{X,[k_1,k_2]} \cap \Sigma_G = \emptyset$ ***then return***   {*Σ_G is the set of updates performed by* G *in Figure 6.1, and* $T_{X,[k_1,k_2]} = \{\langle i, j, k \rangle | i \in [i_1, i_2] \wedge j \in [j_1, j_2] \wedge k \in [k_1, k_2]\}$}
if $k_1 = k_2$ ***then***
  $c[i_1, j_1] \leftarrow f(c[i_1, j_1], c[i_1, k_1], c[k_1, j_1], c[k_1, k_1])$
else   {*The top-left, top-right, bottom-left and bottom-right quadrants of* X *are denoted by* X_{11}, X_{12}, X_{21} *and* X_{22}, *respectively.*}
  $k_m \leftarrow \lfloor \frac{k_1 + k_2}{2} \rfloor$
  $F(X_{11}, k_1, k_m)$, $F(X_{12}, k_1, k_m)$, $F(X_{21}, k_1, k_m)$, $F(X_{22}, k_1, k_m)$ {*forward pass*}
  $F(X_{22}, k_m + 1, k_2)$, $F(X_{21}, k_m + 1, k_2)$, $F(X_{12}, k_m + 1, k_2)$, $F(X_{11}, k_m + 1, k_2)$ {*backward pass*}

Figure 6.2: Cache-oblivious I-GEP. For several special cases of f and Σ_G in Figure 6.1, we show that F performs the same computation as G (see Section 6.3), though there are some cases of f and Σ_G where the computations return different results.

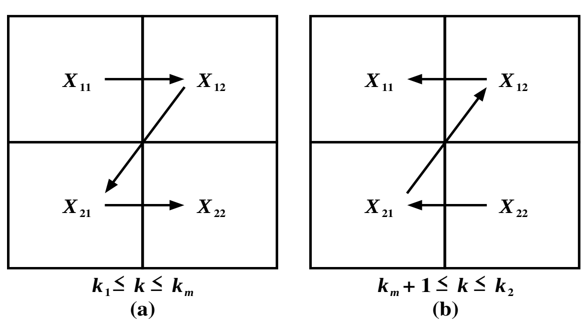

Figure 6.3: Processing order of quadrants of X by F: **(a)** forward pass, **(b)** backward pass.

denote an update of the form $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$, and we let Σ_G denote the set of such updates that the algorithm needs to perform.

In view of the structural similarity between the construct in G and the computation in Gaussian elimination without pivoting, we refer to this computation as the *Gaussian Elimination Paradigm* or *GEP*. Many practical problems fall in this category, for example: all-pairs shortest paths, LU decomposition, and Gaussian elimination without pivoting. Other problems can be solved using GEP through structural transformation, including simple dynamic program [30] and matrix multiplication.

We note the following properties of G, which are easily verified by inspection: Given Σ_G, G applies each $\langle i,j,k\rangle \in \Sigma_G$ on c exactly once, and in a specific order. Given any two distinct updates $\langle i_1,j_1,k_1\rangle \in \Sigma_G$ and $\langle i_2,j_2,k_2\rangle \in \Sigma_G$, the update $\langle i_1,j_1,k_1\rangle$ will be applied before $\langle i_2,j_2,k_2\rangle$ if $k_1 < k_2$, or if $k_1 = k_2$ and $i_1 < i_2$, or if $k_1 = k_2$ and $i_1 = i_2$ but $j_1 < j_2$.

The running time of G is $\mathcal{O}\left(n^3\right)$ provided both the test $\langle i,j,k\rangle \in \Sigma_G$ and the update $\langle i,j,k\rangle$ in line 4 can be performed in constant time. The cache complexity is $\mathcal{O}\left(\frac{n^3}{B}\right)$ provided the only cache misses, if any, incurred in line 4 are for accessing $c[i,j]$, $c[i,k]$, $c[k,j]$ and $c[k,k]$; i.e., neither the evaluation of $\langle i,j,k\rangle \in \Sigma_G$ nor the evaluation of f incurs any additional cache misses.

In the rest of the chapter we assume, without loss of generality, that $n = 2^q$ for some integer $q \geq 0$.

6.1.2 Related Work

Known cache-oblivious algorithms for Gaussian elimination for solving systems of linear equations are based on LU decomposition. In [134, 19] cache-oblivious algorithms performing $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O operations are given for LU decomposition without pivoting, while the algorithm in [120] performs LU decomposition with partial pivoting within the same I/O bound. These algorithms use matrix multiplication and solution of triangular linear systems as subroutines. Our algorithm for Gaussian elimination without pivoting (see Section 6.3.1) is not based on LU decomposition, i.e., it does not call subroutines for multiplying matrices or solving triangular linear systems, and is thus arguably simpler than existing algorithms.

An $\mathcal{O}\left(mnp\right)$ time and $\mathcal{O}\left(m + n + p + \frac{mn+np+mp}{B} + \frac{mnp}{B\sqrt{M}}\right)$ I/O algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix is given in [52].

In [30], an $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithm based on Valiant's context-free language recognition algorithm [124], is given for simple-DP.

A cache-oblivious algorithm for Floyd-Warshall's APSP algorithm is given in [95] (see also [39]). The algorithm runs in $\mathcal{O}\left(n^3\right)$ time and incurs $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses. Our I-GEP implementation of Floyd-Warshall's APSP (see Section 6.3.3) produces exactly the same algorithm.

The main attraction of the Gaussian Elimination Paradigm is that it unifies all problems mentioned above and possibly many others under the same framework, and presents a single I/O-efficient cache-oblivious solution (see C-GEP in Section 6.4) for all of them.

6.1.3 Organization of the Chapter

Majority of the results in this chapter appeared in two conference papers [34, 35].

In Section 6.2, we present and analyze an $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O in-place cache-oblivious algorithm, called I-GEP, which solves several important special cases of GEP. We prove some theorems relating the computation in I-GEP to the computation in GEP. In Section 6.3, we describe generalized versions of three major applications of I-GEP (Gaussian elimination without pivoting, matrix multiplication and Floyd-Warshall's APSP). Succinct proofs of correctness of these I-GEP implementations can be obtained using results from Section 6.2.

In Section 6.4, we present cache-oblivious C-GEP, which solves G in its full generality with the same time and I/O bounds as I-GEP, but uses n^2+n extra space (recall that n^2 is the size of the input/output matrix c). In Section 6.5 we present parallel I-GEP (and C-GEP) and analyze its performance on both distributed and shared caches.

We consider the potential application of the GEP framework in compiler optimizations in Section 6.6.

Section 6.7 contains detailed treatment of one additional application of I-GEP, namely the 'simple DP'. Finally, we present some concluding remarks in Section 6.8.

6.2 I-GEP: In-place Cache-oblivious Solution to Some Important Cases of GEP

In this section we analyze I-GEP, a recursive function F given in Figure 6.2 that is cache-oblivious, computes in-place, and is a provably correct implementation of GEP in Figure 6.1 for several important special cases of f and Σ_G including Floyd-Warshall's APSP, Gaussian elimination without pivoting and matrix multiplication. (This function F does not solve GEP in its full generality, however.) We call this implementation I-GEP to denote an initial attempt at a general cache-oblivious version of GEP as well as an in-place implementation, in contrast to the other implementation (C-GEP) which we give in Section 6.4 that solves GEP in its full generality but uses a modest amount of additional space.

The inputs to F are a square submatrix X of $c[1 \ldots n, 1 \ldots n]$, and two indices k_1 and k_2. The top-left cell of X corresponds to $c[i_1, j_1]$, and the bottom-right cell corresponds to $c[i_2, j_2]$. These indices satisfy the following constraints:

Input Conditions 6.2.1. *If $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$, k_1 and k_2 are the inputs to* F *in Figure 6.2, then*

(a) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ *for some integer* $q \geq 0$

(b) $[i_1, i_2] \neq [k_1, k_2] \quad \Rightarrow \quad [i_1, i_2] \cap [k_1, k_2] = \emptyset$ *and* $[j_1, j_2] \neq [k_1, k_2] \quad \Rightarrow \quad [j_1, j_2] \cap [k_1, k_2] = \emptyset$

Let $U \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $V \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$. Then for every entry $c[i, j] \in X$, $c[i, k]$ can be found in U and $c[k, j]$ can be found in V. Input condition **(a)** requires that X, U and V must all be square matrices of the same size. Input condition **(b)** requires that $\big(X \equiv U\big) \;\vee\; \big(X \cap U = \emptyset\big)$, i.e., either U overlaps X completely, or does not intersect X at all. Similar constraints are imposed on V, too.

The base case of F occurs when $k_1 = k_2$, and the function updates $c[i_1, j_1]$ to $f(c[i_1, j_1],\ c[i_1, k_1],\ c[k_1, j_1],\ c[k_1, k_1])$. Otherwise it splits X into four quadrants $(X_{11}, X_{12}, X_{21}$ and $X_{22})$, and recursively updates the entries in each quadrant in two passes: forward (line 6) and backward (line 7). The processing order of the quadrants are shown in Figure 6.3. The initial function call is $\mathrm{F}(c, 1, n)$.

6.2.1 Properties of I-GEP

We prove two theorems that reveal several important properties of F. Theorem 6.2.1 states that F and G are equivalent in terms of the updates applied, i.e., both of them apply exactly the same updates on the input matrix exactly the same number of times. The theorem also states that both F and G apply the updates applicable to any fixed entry in the input matrix in exactly the same order. However, it does not say anything about the total order of the updates. Theorem 2 identifies the exact states of $c[i, k]$, $c[k, j]$ and $c[k, k]$ (in terms of the updates applied on them) immediately before $c[i, j]$ is updated to $f(c[i, j],\ c[i, k],\ c[k, j],\ c[k, k])$. One implication of this theorem is that the total order of the updates as applied by F and G can be different.

Recall that in Section 6.1.1 we defined Σ_G to be the set of all updates $\langle i, j, k \rangle$ performed by the original GEP algorithm G in Figure 6.1. Analogously, for the transformed cache-oblivious algorithm F, let Σ_F be the set of all updates $\langle i, j, k \rangle$ performed by $\mathrm{F}(c, 1, n)$.

We assume that each instruction executed by F receives a unique time stamp, which is implemented by initializing a global variable t to 0 before the algorithm starts execution, and incrementing it by 1 each time an instruction is executed. By

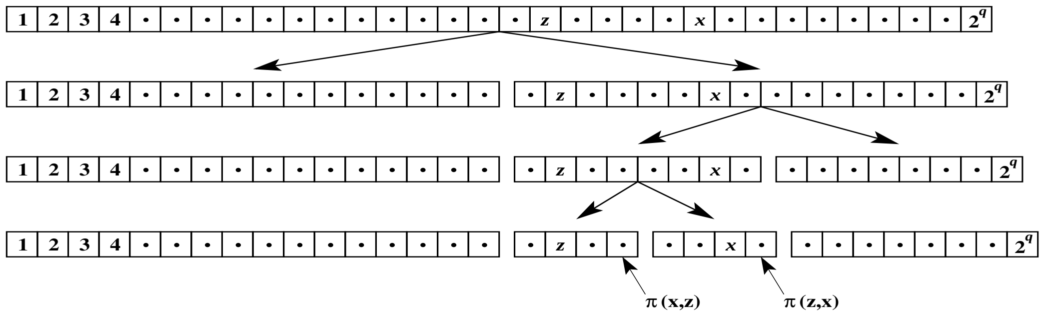

Figure 6.4: Evaluating $\pi(x,z)$ and $\pi(z,x)$ for $x > z$: Given $x, z \in [1, 2^q]$ such that $x > z$, we start with an initial sequence of 2^q consecutive integers in $[1, 2^q]$, and keep splitting the segment containing both x and z at midpoint until x and z fall into different segments. The largest integer in z's segment gives the value of $\pi(x,z)$, and that in x's segment gives the value of $\pi(z,x)$.

the quadruple $\langle i,j,k,t\rangle$ we denote an update $\langle i,j,k\rangle$ that was applied at time t. Let Π_F be the set of all updates $\langle i,j,k,t\rangle$ performed by $\mathrm{F}(c,1,n)$.

The following theorem states that F applies each update performed by G exactly once, and no other updates; it also identifies a partial order on the updates performed by F.

Theorem 6.2.1. *Let Σ_G, Σ_F and Π_F be the sets as defined above. Then*

(a) $\Sigma_F = \Sigma_G$, i.e., both F *and* G *perform the same set of updates;*

(b) $\langle i,j,k,t_1\rangle \in \Pi_F \wedge \langle i,j,k,t_2\rangle \in \Pi_F \Rightarrow t_1 = t_2$, i.e., function F *performs each update $\langle i,j,k\rangle$ at most once; and*

(c) $\langle i,j,k'_1,t_1\rangle \in \Pi_F \wedge \langle i,j,k'_2,t_2\rangle \in \Pi_F \wedge k'_2 > k'_1 \Rightarrow t_2 > t_1$, i.e., function F *updates each $c[i,j]$ in increasing order of k values.*

Proof. (Sketch.) $\langle i,j,k\rangle \in \Sigma_F \Rightarrow \langle i,j,k\rangle \in \Sigma_G$ holds by the check in line 1 of Figure 6.2.

The reverse direction of (a) can be proved by forward induction on q, while parts (b) and (c) can be proved by backward induction on q, where $n = 2^q$. ■

We now introduce some terminology as well as two functions π and δ which will be used later in this section to identify the exact states of $c[i,k]$, $c[k,j]$ and $c[k,k]$ at the time when F is about to apply $\langle i,j,k\rangle$ on $c[i,j]$.

Definition 6.2.1. *Let $n = 2^q$ for some integer $q > 0$.*

(a) An aligned subinterval *for n is an interval $[a,b]$ with $1 \le a \le b \le n$ such that $b - a + 1 = 2^r$ for some nonnegative integer $r \le q$ and $a = c \cdot 2^r + 1$ for some integer $c \ge 0$. The* width *of the aligned subinterval is 2^r.*

(b) An aligned subsquare *for n is a pair of aligned subintervals $[a,b],[a',b']$ with $b-a=b'-a'$.*

The following observation can be proved by (reverse) induction on r, starting with q, where $n=2^q$.

Observation 6.2.1. *Consider the call $F(c,1,n)$. Every recursive call is on an aligned subsquare of c, and every aligned subsquare of c of width 2^r for $r \le q$ is invoked in exactly $n/2^r$ recursive calls on disjoint aligned subintervals $[k_1,k_2]$ of length 2^r each.*

Definition 6.2.2. *Let x, y, and z be integers, $1 \le x,y,z \le n$.*

(a) For $x \neq z$ or $y \neq z$, we define $\delta(x,y,z)$ to be b for the largest aligned subsquare $[a,b],[a,b]$ that contains (z,z), but not (x,y). If $x=y=z$ we define $\delta(x,y,z)$ to be $z-1$.

We will refer to the $[a,b],[a,b]$ subsquare as the aligned subsquare $S(x,y,z)$ for z with respect to (x,y)*; analogously, $S'(x,y,z)$ is the largest aligned subsquare $[c,d],[c',d']$ that contains (x,y) but not (z,z).*

(b) For $x \neq z$, the aligned subinterval for z with respect to x*, $I(x,z)$, is the largest aligned subinterval $[a,b]$ that contains z but not x; similarly* the aligned subinterval for x with respect to z*, $I(z,x)$, is the largest aligned subinterval $[a',b']$ that contains x but not z;*

We define $\pi(x,z)$ to be the largest index b in the aligned subinterval $I(x,z)$ if $x \neq z$, and $\pi(x,z)=z-1$ if $x=z$.

Figures 6.4 and 6.5 illustrate the definitions of π and δ respectively. For completeness, more formal definitions of δ and π are given in Appendix C. The following observation summarizes some simple properties that follow from Definition 6.2.2.

Observation 6.2.2.

(a) If $x \neq z$ or $y \neq z$ then $\delta(x,y,z) \ge z$, and if $x \neq z$ then $\pi(x,z) \ge z$; $I(x,z)$ and $I(z,x)$ have the same length while $S(x,y,z)$ and $S'(x,y,z)$ have the same size; and $S(x,y,z)$ is always centered along the main diagonal while $S'(x,y,z)$ in general will not occur along the main diagonal.

(b) If $x=y=z$ then $\delta(x,y,z)=z-1$, and if $x=z$ then $\pi(x,z)=z-1$.

Part (a) in the following lemma will be used to pin down the state of $c[k,k]$ at the time when update $\langle i,j,k\rangle$ is about to be applied, and parts (b) and (c) can be used to pin down the states at that time of $c[i,k]$ and $c[k,j]$, respectively. As with Observation 6.2.1, this lemma can be proved by backward induction on q. As before the initial call is to $\mathrm{F}(c,1,n)$.

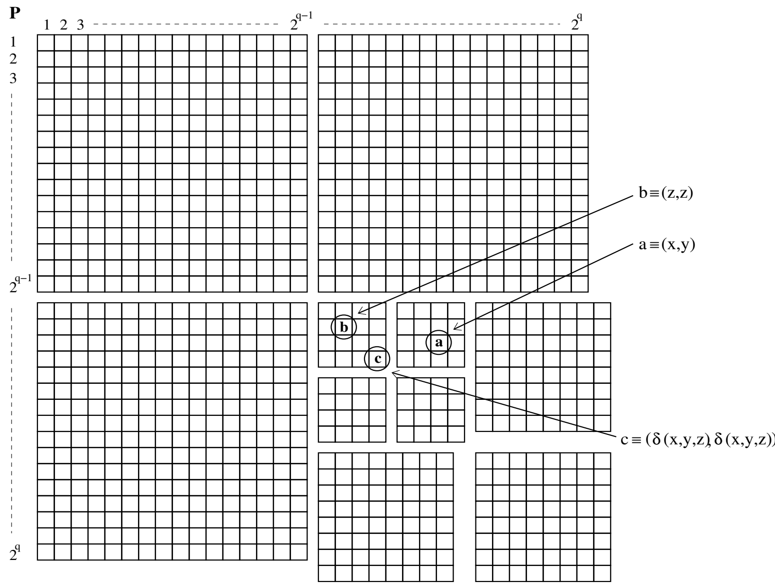

Figure 6.5: Evaluating $\delta(x, y, z)$: Given $x, y, z \in [1, 2^q]$ (where $q \in \mathbb{Z}^+$), such that $x \neq z \ \vee \ y \neq z$, we start with an initial square $P[1 \ldots 2^q, 1 \ldots 2^q]$, and keep splitting the square (initially the entire square P) containing both $P[x, y]$ and $P[z, z]$ into subsquares (quadrants) until $P[x, y]$ and $P[z, z]$ fall into different subsquares. The largest coordinate in $P[z, z]$'s subsquare at that point gives the value of $\delta(x, y, z)$.

Lemma 6.2.1. *Let i, j, k be integers, $1 \leq i, j, k \leq n$, with not all i, j, k having the same value.*

(a) There is a recursive call $\mathrm{F}(X, k_1, k_2)$ *with $k \in [k_1, k_2]$ in which the aligned subsquares $S(i, j, k)$ and $S'(i, j, k)$ will both occur as (different) subsquares of X being called in steps 6 and 7 of the I-GEP pseudocode. The aligned subsquare $S(i, j, k)$ will occur only as either X_{11} or X_{22} while $S'(i, j, k)$ can occur as any one of the four subsquares except that it is not the same as $S(i, j, k)$.*

If $S(i, j, k)$ occurs as X_{11} then $k \in [k_1, k_m]$ and $\delta(i, j, k) = k_m$; if $S(i, j, k)$ occurs as X_{22} then $k \in [k_m + 1, k_2]$ and $\delta(i, j, k) = k_2$.

(b) If $j \neq k$, let $T(i, j, k)$ be the largest aligned subsquare that contains (i, k) but not (i, j) and let $T'(i, j, k)$ be the largest aligned subsquare that contains (i, j) but not (i, k). There is a recursive call $\mathrm{F}(X, k'_1, k'_2)$ *with $k \in [k'_1, k'_2]$ in which the aligned subsquares $T(i, j, k)$ and $T'(i, j, k)$ will both occur as (different) subsquares of X being called in steps 6 and 7 of the I-GEP pseudocode. The set $\{T(i, j, k), T'(i, j, k)\}$ is either $\{X_{11}, X_{12}\}$ or $\{X_{21}, X_{22}\}$, and $\pi(j, k) = k'$, where k' is the largest integer such that (i, k') belongs to $T(i, j, k)$.*

(c) If $i \neq k$, let $R(i, j, k)$ be the largest aligned subsquare that contains (k, j)

but not (i,j) and let $R'(i,j,k)$ be the largest aligned subsquare that contains (i,j) but not (k,j). There is a recursive call $\mathrm{F}(X, k_1'', k_2'')$ *with $k \in [k_1'', k_2'']$ in which the aligned subsquares $R(i,j,k)$ and $R'(i,j,k)$ will both occur as (different) subsquares of X being called in steps 6 and 7 of the I-GEP pseudocode. The set $\{R(i,j,k), R'(i,j,k)\}$ is either $\{X_{11}, X_{21}\}$ or $\{X_{12}, X_{22}\}$, and $\pi(i,k) = k''$, where k'' is the largest integer such that (k'', j) belongs to $R(i,j,k)$.*

Let $c_k(i,j)$ denote the value of $c[i,j]$ after all updates $\langle i,j,k'\rangle \in \Sigma_G$ with $k' \leq k$ have been performed by F, and no other updates have been performed on it. We now present the second main theorem of this section.

Theorem 6.2.2. *Let δ and π be as defined in Definition 6.2.2. Then immediately before function* F *performs the update $\langle i,j,k\rangle$ (i.e., before it executes $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$), the following hold:*

- $c[i,j] = c_{k-1}(i,j)$,
- $c[i,k] = c_{\pi(j,k)}(i,k)$,
- $c[k,j] = c_{\pi(i,k)}(k,j)$,
- $c[k,k] = c_{\delta(i,j,k)}(k,k)$.

Proof. We prove each of the four claims by turn.

$\underline{c[i,j]}$: By Theorem 6.2.1, for any given $i,j \in [1,n]$ the value of $c[i,j]$ is updated in increasing value of k, hence at the time when update $\langle i,j,k\rangle$ is about to be applied, the state of $c[i,j]$ must equal $c_{k-1}(i,j)$.

$\underline{c[k,k]}$: Assume that either $k \neq i$ or $k \neq j$, and consider the state of $c[k,k]$ when update $\langle i,j,k\rangle$ is about to be applied. Let $S(i,j,k)$ and $S'(i,j,k)$ be as specified in Definition 6.2.2, and consider the recursive call $\mathrm{F}(X, k_1, k_2)$ with $k \in [k_1, k_2]$ in which $S(i,j,k)$ and $S'(i,j,k)$ are both called during the execution of lines 6 and 7 of the I-GEP code (this call exists as noted in Lemma 6.2.1). Also, as noted in Lemma 6.2.1, the aligned subsquare $S(i,j,k)$ (which contains position (k,k) but not (i,j)) will occur either as X_{11} or X_{22}.

If $S(i,j,k)$ occurs as X_{11} when it is invoked in the pseudocode, then by Lemma 6.2.1 we also know that $k \in [k_1, k_m]$, and $S'(i,j,k)$ will be invoked as X_{12}, X_{21} or X_{22} in the same recursive call. Thus, $c[k,k]$ will have been updated by all $\langle i,j,k'\rangle \in \Sigma_G$ for which $(k',k') \in S(i,j,k)$, before update $\langle i,j,k\rangle$ is applied to $c[i,j]$ in the forward pass. By Definition 6.2.2 the largest integer k' for which (k',k') belongs to $S(i,j,k)$ is $\delta(i,j,k)$. Hence the value of $c[k,k]$ that is used in update $\langle i,j,k\rangle$ is $c_{\delta(i,j,k)}(k,k)$.

Similarly, if $S(i,j,k)$ occurs as X_{22} when it is invoked in the pseudocode, then $k \in [k_m+1, k_2]$, and $S'(i,j,k)$ will be invoked as X_{11}, X_{12} or X_{21} in the same recursive call. Since the value of k is in the higher half of $[k_1, k_2]$, the update $\langle i, j, k \rangle$ will be performed in the backward pass in line 7, and hence $c[k, k]$ will have been updated by all $\langle i, j, k' \rangle \in \Sigma_G$ with $k' \leq k_2$. As above, by Definition 6.2.2, $\delta(i, j, k)$ is the largest value of k' for which (k', k') belongs to $S(i, j, k)$, which is k_2, hence the value of $c[k, k]$ that is used in update $\langle i, j, k \rangle$ is $c_{\delta(i,j,k)}(k, k)$.

Finally, if $i = j = k$, we have $c[k, k] = c_{k-1}(i, j) = c_{\delta(i,j,k)}(k, k)$ by definition of $\delta(i, j, k)$.

$\underline{c[i, k] \text{ and } c[k, j]}$: Similar to the proof for $c[k, k]$ but using parts (b) and (c) of Lemma 6.2.1. ■

6.2.2 Cache Complexity

Let $Q(n)$ be an upper bound on the number of block transfers performed by F on an input of size $n \times n$. It is not difficult to see that

$$Q(n) \leq \begin{cases} \mathcal{O}\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma M, \\ 8Q\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases} \tag{6.2.1}$$

where γ is the largest constant sufficiently small that four $\sqrt{\gamma M} \times \sqrt{\gamma M}$ submatrices fit in the cache. The solution to the recurrence is $Q(n) = \mathcal{O}\left(1 + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right) = \mathcal{O}\left(1 + \frac{n^2}{B} + \frac{n^3}{B\sqrt{M}}\right)$ (assuming a tall cache, i.e., $M = \Omega(B^2)$).

Since I-GEP can be used for multiplying matrices, it follows from the I/O lower bound of matrix multiplication [74] that the cache complexity of I-GEP is, in fact, tight for any algorithm that performs $\Theta\left(n^3\right)$ operations in order to implement the general version of the GEP computation as defined in Section 6.1.1.

Below we prove a more general upper bound on the cache-misses incurred by function F, which will be used in Section 6.7.1 to determine the cache complexity of the I-GEP implementation of 'Simple-DP'. The following theorem assumes that F is called on an $n \times n$ input matrix c (i.e., called as $\mathrm{F}(c, 1, n)$), but considers only the cache misses incurred for applying the updates $\langle i, j, k \rangle \in \Sigma_G$ with $c[i, j] \in X$, where X is an $m \times m$ submatrix of c. Thus the implicit assumption is that immediately before any such update $\langle i, j, k \rangle$ is applied on $c[i, j]$, each of $c[i, k]$, $c[k, j]$ and $c[k, k]$ has the correct value (as implied by Theorem 6.2.2) even if it does not belong to X.

Theorem 6.2.3. *Let X be an $m \times m$ submatrix of c, where c is the $n \times n$ input matrix to* F*. Then the number of cache misses incurred by* F *while applying all updates $\langle i, j, k \rangle \in \Sigma_G$ with $c[i, j] \in X$ is $\mathcal{O}\left(\frac{m^2 n}{B\sqrt{M}}\right)$, assuming that X is too large to fit in the cache, and that the cache is tall (i.e., $M = \Omega\left(B^2\right)$).*

Proof. Let X_e be an aligned subsquare of c of largest width 2^r such that four such subsquares completely fit in the cache, i.e., $\lambda \cdot 4^r \leq M < \lambda \cdot 4^{r+1}$ for some suitable constant λ. We know from Observation 6.2.1 that X_e will be invoked in exactly $\frac{n}{2^r}$ recursive calls of F on disjoint aligned subintervals of $[k_1, k_2]$ of length 2^r each. The number of cache misses incurred in fetching X_e into the cache along with all (at most three) other aligned subsquares required for updating the entries of X_e is $\mathcal{O}\left(2^r + \frac{2^r \times 2^r}{B}\right)$, since at most 1 cache miss will occur for accessing each row of the subsquares, and $\mathcal{O}\left(\frac{2^r \times 2^r}{B}\right)$ cache misses for scanning in all entries. Thus the total cache misses incurred by all $\frac{n}{2^r}$ recursive calls on X_e is $\mathcal{O}\left(\frac{n}{2^r} \times \left(2^r + \frac{2^r \times 2^r}{B}\right)\right) = \mathcal{O}\left(n + \frac{n\sqrt{M}}{B}\right)$, since $2^r \times 2^r = \Theta(M)$.

Now since X is an $m \times m$ subsquare of c, and all recursive calls on an $2^r \times 2^r$ subsquare incur $\mathcal{O}\left(n + \frac{n\sqrt{M}}{B}\right)$ cache misses, the number of cache misses incurred by all recursive calls updating the entries of X is $\Theta\left(\frac{m^2}{2^r \times 2^r}\right) \times \mathcal{O}\left(n + \frac{n\sqrt{M}}{B}\right) = \mathcal{O}\left(\frac{m^2 n}{B\sqrt{M}} + \frac{m^2 n}{M}\right) = \mathcal{O}\left(\frac{m^2 n}{B\sqrt{M}}\right)$ (since $M = \Omega\left(B^2\right)$). ■

6.2.3 Time and Space Complexities

Since I-GEP is in-place, its space complexity is determined by the size of its input matrices which is clearly $\Theta\left(n^2\right)$. Time complexity of I-GEP is given by the following recurrence relation, where $T(n)$ denotes the running time of I-GEP on an input of size $n \times n$.

$$T(n) \leq \begin{cases} \mathcal{O}(1) & \text{if } n \leq 1, \\ 8T\left(\frac{n}{2}\right) + \mathcal{O}(1) & \text{otherwise;} \end{cases} \tag{6.2.2}$$

Solving, we get $T(n) = \mathcal{O}\left(n^3\right)$.

6.2.4 Static Pruning of I-GEP

The test in line 1 of Figure 6.2 enables function F to decide during runtime whether the current recursive call is necessary or not, and thus avoid taking unnecessary branches in its recursion tree. However, if the update set Σ_G is available offline (which is usually the case), we can eliminate some of these unnecessary branchings from the code during the transformation of G to F, and thus save on some overhead. We can perform this type of static pruning of F as follows.

Recall that $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$ is the input submatrix, and $[k_1, k_2]$ is the range of k-values supplied to F, and they satisfy the input conditions 6.2.1. Let $U \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $V \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$. Then for every entry $c[i,j] \in X$, $c[i,k]$ can be found in U and $c[k,j]$ can be found in V. From input condition 6.2.1**(a)** we know that X, U and V must all be square matrices of the same dimensions. Input condition 6.2.1**(b)** requires that each of U and V either overlaps X completely, or does not intersect X at all. These conditions on the

F(X, k_1, k_2) $\quad$ {F $\in \{A, B_1, B_2, C_1, C_2, D_1, D_2, D_3, D_4\}$; *see column 1 of Figure 6.7.*}

(X is a $2^q \times 2^q$ square submatrix of c such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$ for some integer $q \geq 0$. Function F assumes the following:

(*a*) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$

(*b*) $[i_1, i_2] \neq [k_1, k_2] \;\Rightarrow\; [i_1, i_2] \cap [k_1, k_2] = \emptyset$ and $[j_1, j_2] \neq [k_1, k_2] \;\Rightarrow\; [j_1, j_2] \cap [k_1, k_2] = \emptyset$

(*c*) $P(F)$ (see Figure 6.8)

The initial call to F is A$(c, 1, n)$ for an $n \times n$ input matrix c, where n is a power of 2.)

1. ***if*** $T_{X,[k_1,k_2]} \cap \Sigma_G = \emptyset$ ***then return***

$\quad$ $\{T_{X,[k_1,k_2]} = \{\langle i,j,k\rangle | i \in [i_1, i_2] \wedge j \in [j_1, j_2] \wedge k \in [k_1, k_2]\}$, *and* Σ_G *is the set of updates performed by* G *in Figure 6.1*$\}$

2. ***if*** $k_1 = k_2$ ***then***

3. $\quad c[i_1, j_1] \leftarrow f(c[i_1, j_1], c[i_1, k_1], c[k_1, j_1], c[k_1, k_1])$

4. ***else*** $\quad$ {*The following function calls are determined from the table in Figure 6.7. The top-left, top-right, bottom-left and bottom-right quadrants of X are denoted by* X_{11}, X_{12}, X_{21} *and* X_{22}, *respectively.*}

5. $\quad k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$

6. $\quad F_{11}(X_{11}, k_1, k_m),\; F_{12}(X_{12}, k_1, k_m),\; F_{21}(X_{21}, k_1, k_m),\; F_{22}(X_{22}, k_1, k_m)$ {*forward pass*}

7. $\quad F'_{22}(X_{22}, k_m+1, k_2),\; F'_{21}(X_{21}, k_m+1, k_2),\; F'_{12}(X_{12}, k_m+1, k_2),\; F'_{11}(X_{11}, k_m+1, k_2)$ {*backward pass*}

Figure 6.6: Cache-oblivious I-GEP reproduced from Figure 6.2, but here F is assumed to be a template function that can be instantiated to any of the 9 functions given in Figure 6.8. The recursive calls in lines 6 and 7 are replaced with appropriate instantiations of F which can be determined from Figure 6.7.

inputs to F implies nine possible arrangements (i.e., relative positions) of X, U and V. For different arrangements of these matrices we give a different name to F. Figure 6.9 identifies each of the nine names (A, B_1, B_2, C_1, C_2, D_1, D_2, D_3 and D_4) with the corresponding arrangement of the matrices. Each of these nine functions will be called an instantiation of F. Given an instantiation F′ of F, Figure 6.8 expresses the corresponding arrangement of X, U and V as a relationship $P(F')$ among the indices i_1, i_2, j_1, j_2, k_1 and k_2. Function A assumes that both U and V overlap X, i.e., all required $c[i,k]$ and $c[k,j]$ values can be found in X. Functions B_1 and B_2 both assume that V and X overlap, but B_1 assumes that U lies to the left of X, and B_2 assumes that U lies to the right of X. Functions C_1 and C_2 are called when U and X overlap, but V and X do not. Function C_1 is called when V lies above X, C_2 is called otherwise. Functions D_1, D_2, D_3, and D_4 assume that neither U nor V overlap X, and each of them assumes different relative positions of U and V with respect to X.

In Figure 6.6 we reproduce F from Figure 6.2, but replace the recursive calls in lines 6 and 7 with instantiations of F. By F_{pq} $(p, q \in [1,2])$, we denote the

F	F_{11}	F_{12}	F_{21}	F_{22}	F'_{22}	F'_{21}	F'_{12}	F'_{11}
A	A	B_1	C_1	D_1	A	B_2	C_2	D_4
B_i $(i = 1, 2)$	B_i	B_i	D_i	D_i	B_i	B_i	D_{i+2}	D_{i+2}
C_i $(i = 1, 2)$	C_i	D_{2i-1}	C_i	D_{2i-1}	C_i	D_{2i}	C_i	D_{2i}
D_i $(i \in [1, 4])$	D_i	D_i	D_i	D_i	D_i	D_i	D_i	D_i

Figure 6.7: Functions recursively called by F in Figure 6.6.

F	$P(F)$
A	$i_1 = k_1 \wedge j_1 = k_1$
B_1	$i_1 = k_1 \wedge j_1 > k_2$
B_2	$i_1 = k_1 \wedge j_2 < k_1$
C_1	$i_1 > k_2 \wedge j_1 = k_1$
C_2	$i_2 < k_1 \wedge j_1 = k_1$
D_1	$i_1 > k_2 \wedge j_1 > k_2$
D_2	$i_1 > k_2 \wedge j_2 < k_1$
D_3	$i_2 < k_1 \wedge j_1 > k_2$
D_4	$i_2 < k_1 \wedge j_2 < k_1$

Figure 6.8: Function specific pre-condition $P(F)$ for F in Figure 6.2.

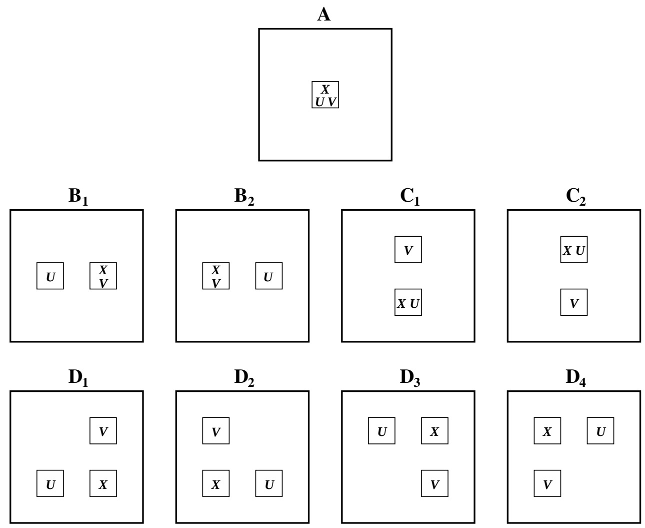

Figure 6.9: Relative positions of $Y \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $Z \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$ w.r.t. $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$ assumed by different instantiations of F.

instantiation of F that processes quadrant X_{pq} in the forward pass (line 6), and by F'_{pq} $(p, q \in [1, 2])$ we denote the same in the backward pass (line 7). For each of the nine instantiations of the calling function F, Figure 6.7 associates F_{pq} and F'_{pq} $(p, q \in [1, 2])$ with appropriate instantiations.

A given computation need not necessarily make all recursive calls in lines 6 and 7. Whether a specific recursive call to a function F' (say) will be made or not depends on $P(F')$ (see Figure 6.8) and the GEP instance at hand. For example, if $i \geq k$ holds for every update $\langle i, j, k \rangle \in \Sigma_G$, then we do not make any recursive call to function C_2 since the indices in the updates can never satisfy $P(C_2)$. The I-GEP implementation of the code for Gaussian elimination without pivoting can employ static pruning very effectively, in which case, we can eliminate all recursive calls except for those to A, B_1, C_1 and D_1.

The initial function call is $A(c, 1, n)$ (F instantiated to A).

6.3 Applications of Cache-oblivious I-GEP

In this section we consider I-GEP for three major GEP instances. Though the C-GEP implementation given in Section 6.4 works for all instances of f and Σ_G, it uses extra space, and is slightly more complicated than I-GEP. Our experimental results in Chapter 7 also show that I-GEP performs slightly better than both variants of C-GEP. Hence an I-GEP implementation is preferable to a C-GEP implementation if it can be proved to work correctly for a given GEP instance.

We consider the following applications of I-GEP in this section.

- In Section 6.3.1 we consider I-GEP for a class of applications that includes Gaussian elimination without pivoting, where we restrict Σ_G but allow f to be unrestricted.
- In Section 6.3.2 we consider a class of applications where we do not impose any restrictions on Σ_G, but restrict f to receive all its inputs except the first one (i.e., except $c[i, j]$) from matrices that remain unmodified throughout the computation. An important problem in this class is matrix multiplication.
- In Section 6.3.3 we consider I-GEP for path computations over closed semirings which includes Floyd-Warshall's APSP algorithm [48] and Warshall's algorithm for finding transitive closures [128]. For this class of problems we restrict both f and Σ_G.

At the end of this chapter (Section 6.7) we consider another application of I-GEP, namely the simple-DP.

6.3.1 Gaussian Elimination without Pivoting

Gaussian elimination without pivoting is used in the solution of systems of linear equations and LU decomposition of symmetric positive-definite or diagonally dominant real matrices [37]. We represent a system of $n-1$ equations in $n-1$ unknowns $(x_1, x_2, \ldots, x_{n-1})$ using an $n \times n$ matrix c, where the i'th $(1 \leq i < n)$ row represents the equation $a_{i,1}x_1 + a_{i,2}x_2 + \ldots + a_{i,n-1}x_{n-1} = b_i$. The method proceeds in two phases. In the first phase, an upper triangular matrix is constructed from c by successive elimination of variables from the equations. This phase requires $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(\frac{n^3}{B}\right)$ I/Os. In the second phase, the values of the unknowns are determined from this matrix by back substitution. It is straight-forward to implement this second phase in $\mathcal{O}\left(n^2\right)$ time and $\mathcal{O}\left(\frac{n^2}{B}\right)$ I/Os, so we will concentrate on the first phase.

The first phase is an instantiation of the GEP code in Figure 6.1. In Figure 6.10 we give a computation that is a more general form of the computation

G$(c, 1, n)$

(The input $c[1 \ldots n, 1 \ldots n]$ is an $n \times n$ matrix. Function $f(\cdot,\cdot,\cdot,\cdot)$ is a problem-specific function, and for Gaussian elimination without pivoting $f(x, u, v, w) = x - \frac{u}{w} \times v$.)

1. ***for*** $k \leftarrow 1$ ***to*** n ***do***
2.   ***for*** $i \leftarrow 1$ ***to*** n ***do***
3.   ***for*** $j \leftarrow 1$ ***to*** n ***do***
4.   ***if*** $(k \leq n-2) \wedge (k < i < n) \wedge (k < j)$ ***then*** $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$

Figure 6.10: A more general form of the first phase of Gaussian elimination without pivoting.

for $k \leftarrow 1$ ***to*** n ***do***
  for $i \leftarrow 1$ ***to*** n ***do***
    for $j \leftarrow 1$ ***to*** n ***do***
      $c[i,j] \leftarrow c[i,j]$
        $+ a[i,k] \times b[k,j]$

(a)

for $k \leftarrow 1$ ***to*** n ***do***
  for $i \leftarrow 1$ ***to*** n ***do***
    for $j \leftarrow 1$ ***to*** n ***do***
      if $\langle i,j,k\rangle \in \Sigma_G$ ***then***
        $c[i,j] \leftarrow f(c[i,j], a[i,k], b[k,j], d[k,k])$
          $\{a, b, d \neq c\}$

(b)

Figure 6.11: **(a)** Modified matrix multiplication algorithm, **(b)** A more general form of the algorithm in Figure 6.11(a).

in the first phase of Gaussian elimination without pivoting in the sense that the update function f in Figure 6.10 is arbitrary. The ***if*** condition in line 4 ensures that $i > k$ and $j > k$ hold for every update $\langle i, j, k\rangle$ applied on c, i.e., $\Sigma_G = \{\langle i, j, k\rangle : (1 \leq k \leq n-2) \wedge (k < i < n) \wedge (k < j \leq n)\}$.

The correctness of the I-GEP implementation of the code in Figure 6.10 can be proved by induction on k using Theorem 6.2.2 and by observing that each $c[i,j]$ $(1 \leq i, j \leq n)$ settles down (i.e., is never modified again) before it is ever used on the right hand side of an update.

As described in Section 6.2.4, we can apply static pruning on the resulting I-GEP implementation to remove unnecessary recursive calls from the pseudocode.

A similar method solves LU decomposition without pivoting within the same bounds. Both algorithms are in-place. Our algorithm for Gaussian elimination is arguably simpler than existing algorithms since it does not use LU decomposition as an intermediate step, and thus does not invoke subroutines for multiplying matrices or solving triangular linear systems, as is the case with other cache-oblivious algorithms for this problem [134, 19, 120].

6.3.2 Matrix Multiplication

We consider the problem of computing $C = A \times B$, where A, B and C are $n \times n$ matrices. Though standard matrix multiplication does not fall into GEP, it does after the small structural modification shown in Figure 6.11(a) (index k is in the outermost loop in the modified algorithm, while in the standard algorithm it is in the innermost loop); correctness of this transformed code is straight-forward.

The algorithm in Figure 6.11(b) generalizes the computation in step 4 of Figure 6.11(a) to update $c[i,j]$ to a new value that is an arbitrary function of $c[i,j], a[i,k], b[k,j]$ and $d[k,k]$, where matrices $a, b, d \neq c$.

The correctness of the I-GEP implementation of the code in Figure 6.11(b) follows from Theorem 6.2.1 and from the observation that matrices a, b and d remain unchanged throughout the computation.

6.3.3 Path Computations Over a Closed Semiring

An algebraic structure known as a *closed semiring* [4] serves as a general framework for solving path problems in directed graphs. In [4], 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 [48] and Warshall's algorithm for finding transitive closures [128] are instantiations of this algorithm.

Consider a directed graph $\mathcal{G} = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, and each edge (v_i, v_j) is labeled by an element $l(v_i, v_j)$ of some closed semiring $(S, \oplus, \odot, 0, 1)$. If $(v_i, v_j) \notin E$, $l(v_i, v_j)$ 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 $v_i, v_j \in V$, $c[i,j]$ 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 presense of cycles), $c[i,j]$ will still be well-defined due to the properties of a closed semiring.

The algorithm given in Figure 6.12(b), which is an instance of GEP, computes $c[i,j]$ for all pairs of vertices $v_i, v_j \in V$. This algorithm performs $\mathcal{O}\left(n^3\right)$ operations and uses $\mathcal{O}\left(n^2\right)$ space. Floyd-Warshall's APSP is a specialization of the algorithm in Figure 6.12(b) in that it performs computations over a particular closed semiring $(\Re, \min, +, +\infty, 0)$.

Correctness of I-GEP Implementation of Figure 6.12(b). Recall that $c_0(i,j)$ is the initial value of $c[i,j]$ received by the I-GEP function F in Figure 6.2, and $c_k(i,j)$ $(1 \leq i,j \leq n)$ denotes the value of $c[i,j]$ after all updates $\langle i,j,k' \rangle \in \Sigma_G$ with $k' \leq k$, and no other updates have been performed on it by F.

For $i,j \in [1,n]$ and $k \in [0,n]$, let $P^k_{i,j}$ denote the set of all paths from v_i to v_j with no intermediate vertex higher than v_k, and let $Q^k_{i,j}$ be the set of all paths from

Initial Values:

$$\forall_{1\le i,j\le n} c[i,j] = \begin{cases} 1 & \text{if } i = j, \\ l\,(v_i, v_j) & \text{otherwise.} \end{cases}$$

(a)

Computation of Path Costs:

```
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            c[i, j] ← c[i, j] ⊕ (c[i, k] ⊙ c[k, j])
```

(b)

Figure 6.12: Computation of path costs over a closed semiring $(S, \oplus, \odot, 0, 1)$: **(a)** Initialization of c, **(b)** Computation of path costs.

v_i to v_j that have contributed to the computation of $c_k(i, j)$.

The correctness of the I-GEP implementation of the code in Figure 6.12(b) follows from the following lemma, which can be proved by induction on k using Theorems 6.2.1 and 6.2.2.

Lemma 6.3.1. *For all* $i, j, k \in [1, n]$, $Q^k_{i,j} \supseteq P^k_{i,j}$.

Since for $i, j \in [1, n]$, $P^n_{i,j}$ contains all paths from v_i to v_j, we have $Q^n_{i,j} \subseteq P^n_{i,j}$, which when combined with $Q^n_{i,j} \supseteq P^n_{i,j}$ obtained from lemma 6.3.1, results in $Q^n_{i,j} = P^n_{i,j}$.

6.4 C-GEP: Extension of I-GEP to Full Generality

In order to express mathematical expressions with conditionals in compact form, in this section we will use *Iverson's convention* [75, 82, 62] for denoting values of Boolean expressions. In this convention we use $|\mathcal{E}|$ to denote the value of a Boolean expression $\mathcal{E}$, where $|\mathcal{E}| = 1$ if $\mathcal{E}$ is true and $|\mathcal{E}| = 0$ if $\mathcal{E}$ is false[1].

6.4.1 A Closer Look at I-GEP

Recall that $c_k(i, j)$ denotes the value of $c[i, j]$ after all updates $\langle i, j, k' \rangle \in \Sigma_G$ with $k' \le k$, and no other updates have been applied on $c[i, j]$ by F, where $i, j \in [1, n]$ and $k \in [0, n]$. Let $\hat{c}_k(i, j)$ be the corresponding value for G, i.e., let $\hat{c}_k(i, j)$ be the value of $c[i, j]$ immediately after the k-th iteration of the outer ***for*** loop in G, where $i, j \in [1, n]$ and $k \in [0, n]$.

In the following table, we tabulate the exact states of $c[i, j]$, $c[i, k]$, $c[k, j]$ and $c[k, k]$ immediately before G or F applies an update $\langle i, j, k \rangle \in \Sigma_G$. Entries in the 2nd column are determined by inspecting the code in Figure 6.1, while those in the 3rd column follows from Theorem 6.2.2.

[1] Iversion actually used $[\mathcal{E}]$ to denote the value of a Boolean expression $\mathcal{E}$. But we use $|\mathcal{E}|$ instead, since square brackets are used extensively for other purposes in this dissertation.

Cell	G	F
$c[i,j]$	$\hat{c}_{k-1}(i,j)$	$c_{k-1}(i,j)$
$c[i,k]$	$\hat{c}_{k-\lvert j\leq k\rvert}(i,k)$	$c_{\pi(j,k)}(i,k)$
$c[k,j]$	$\hat{c}_{k-\lvert i\leq k\rvert}(k,j)$	$c_{\pi(i,k)}(k,j)$
$c[k,k]$	$\hat{c}_{k-\lvert(i<k)\ \vee\ (i=k\ \wedge\ j\leq k)\rvert}(k,k)$	$c_{\delta(i,j,k)}(k,k)$

Table 6.1: States of $c[i,j]$, $c[i,k]$, $c[k,j]$ and $c[k,k]$ immediately before applying $\langle i,j,k\rangle \in \Sigma_G$.

It follows from Definition 6.2.2 that for $i,j < k$, $\pi(j,k) \neq k - |j \leq k|$, $\pi(i,k) \neq k-|i \leq k|$ and $\delta(i,j,k) \neq k-|(i<k) \vee (i=k \wedge j \leq k)|$. Therefore, though both G and F start with the same input matrix, at certain points in the computation F and G would supply different input values to f while applying the same update $\langle i,j,k\rangle \in \Sigma_G$, and consequently f could return different output values. Whether the final output matrix returned by the two algorithms are the same depends on f, Σ_G and the input values.

As an example, consider a 2×2 input matrix c, and let $\Sigma_G = \{\langle i,j,k\rangle | 1 \leq i,j,k \leq 2\}$. Then G will compute the entries in the following order: $\hat{c}_1(1,1)$, $\hat{c}_1(1,2)$, $\hat{c}_1(2,1)$, $\hat{c}_1(2,2)$, $\hat{c}_2(1,1)$, $\hat{c}_2(1,2)$, $\hat{c}_2(2,1)$, $\hat{c}_2(2,2)$; on the other hand, F will compute in the following order: $c_1(1,1)$, $c_1(1,2)$, $c_1(2,1)$, $c_1(2,2)$, $c_2(2,2)$, $c_2(2,1)$, $c_2(1,2)$, $c_2(1,1)$. Since both G and F use the same input matrix, the first 5 values computed by F will be correct, i.e., $c_1(1,1) = \hat{c}_1(1,1)$, $c_1(1,2) = \hat{c}_1(1,2)$, $c_1(2,1) = \hat{c}_1(2,1)$, $c_1(2,2) = \hat{c}_1(2,2)$ and $c_2(2,2) = \hat{c}_2(2,2)$. However, the next value, i.e., the final value of $c[2,1]$, computed by F is not necessarily correct, since F sets $c_2(2,1) \leftarrow f(c_1(2,1), c_2(2,2), c_1(2,1), c_2(2,2))$, while G sets $\hat{c}_2(2,1) \leftarrow f(\hat{c}_1(2,1), \hat{c}_1(2,2), \hat{c}_1(2,1), \hat{c}_1(2,2))$. For example, if initially $c[1,1] = c[1,2] = c[2,1] = 0$ and $c[2,2] = 1$, and f just returns the sum of its input values, then F will output $c[2,1] = 8$, while G will output $c[2,1] = 2$.

6.4.2 C-GEP using $4n^2$ Additional Space

We first define a quantity τ_{ij}, which plays a crucial role in the extension of I-GEP to the completely general C-GEP.

Definition 6.4.1. *For $1 \leq i,j,l \leq n$, we define $\tau_{ij}(l)$ to be the largest integer $l' \leq l$ such that $\langle i,j,l'\rangle \in \Sigma_G$ provided such an update exists, and 0 otherwise. More formally, for all $i,j,l \in [1,n]$, $\tau_{ij}(l) = \max_{\,l'} \{l' \mid l' \leq l \wedge \langle i,j,l'\rangle \in \Sigma_G \cup \{\langle i,j,0\rangle\}\}$.*

The significance of τ of can be explained as follows. We know from Theorem 6.2.1 that both F and G apply the updates $\langle i,j,k\rangle$ in increasing order of k values. Hence, at any point of time during the execution of F (or G) if $c[i,j]$ is in state $c_l(i,j)$ ($\hat{c}_l(i,j)$, resp.), where $l \neq 0$, then $\langle i,j,\tau_{ij}(l)\rangle$ is the update that has left $c[i,j]$ in this state. We also note the difference between π (defined in Definition

H(X, k_1, k_2)

(X is a $2^q \times 2^q$ submatrix of c such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$ for some integer $q \geq 0$. Matrices u_0, u_1, v_0 and v_1 are global, and each initialized to c before the initial call to H is made. Similar to F in Figure 6.2, H assumes the following:

(a) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$

(b) $[i_1, i_2] \neq [k_1, k_2] \Rightarrow [i_1, i_2] \cap [k_1, k_2] = \emptyset$ and $[j_1, j_2] \neq [k_1, k_2] \Rightarrow [j_1, j_2] \cap [k_1, k_2] = \emptyset$

The initial call to H is H$(c, 1, n)$ for an $n \times n$ input matrix c, where n is a power of 2.)

1. ***if*** $T_{X,[k_1,k_2]} \cap \Sigma_G = \emptyset$ ***then return***
   $\{T_{X,[k_1,k_2]} = \{\langle i, j, k\rangle | i \in [i_1, i_2] \wedge j \in [j_1, j_2] \wedge k \in [k_1, k_2]\}$, *and* Σ_G *is the set of updates performed by* G *in Figure 6.1*$\}$
2. ***if*** $k_1 = k_2$ ***then***
3. $\quad i \leftarrow i_1, \;\; j \leftarrow j_1, \;\; k \leftarrow k_1$
4. $\quad c[i,j] \leftarrow f(c[i,j], u_{|j>k|}[i,k], v_{|i>k|}[k,j], u_{|(i>k) \vee (i=k \wedge j>k)|}[k,k])$
5. $\quad$ ***if*** $k = \tau_{ij}(j-1)$ ***then*** $u_0[i,j] \leftarrow c[i,j]$
   $\{\tau_{ij}(l) = \max_{\;l'} \{l' \mid l' \leq l \wedge \langle i, j, l'\rangle \in \Sigma_G \cup \{\langle i, j, 0\rangle\}\}\}$
6. $\quad$ ***if*** $k = \tau_{ij}(j)$ ***then*** $u_1[i,j] \leftarrow c[i,j]$
7. $\quad$ ***if*** $k = \tau_{ij}(i-1)$ ***then*** $v_0[i,j] \leftarrow c[i,j]$
8. $\quad$ ***if*** $k = \tau_{ij}(i)$ ***then*** $v_1[i,j] \leftarrow c[i,j]$
9. ***else*** $\{$*The top-left, top-right, bottom-left and bottom-right quadrants of* X *are denoted by* X_{11}, X_{12}, X_{21} *and* X_{22}, *respectively.*$\}$
10. $\quad k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$
11. $\quad H(X_{11}, k_1, k_m), H(X_{12}, k_1, k_m), H(X_{21}, k_1, k_m), H(X_{22}, k_1, k_m)$ $\{$*forward pass*$\}$
12. $\quad H(X_{22}, k_m+1, k_2), H(X_{21}, k_m+1, k_2), H(X_{12}, k_m+1, k_2), H(X_{11}, k_m+1, k_2)$ $\{$*backward pass*$\}$

Figure 6.13: C-GEP: A cache-oblivious implementation of GEP (i.e., G in Figure 6.1) that works for all f and Σ_G.

6.2.2) and τ: we know from Theorem 6.2.2 that immediately before applying $\langle i, j, k\rangle$ function F finds $c[i,k]$ in state $c_{\pi(j,k)}(i,k)$, and from the definition of τ we know that $\langle i, k, \tau_{ik}(\pi(j,k))\rangle$ is the update that has left $c[i,k]$ in this state. Similar observation holds for δ defined in Definition 6.2.2.

We extend I-GEP to full generality by modifying F in Figure 6.2 so that it performs updates according to the second column of Table 6.1 instead of the third column. As described below, we achieve this by saving suitable intermediate values of the entries of c in auxiliary matrices as F generates them. Note that for all $i, j, k \in [1, n]$, F computes $c_{k-|j \leq k|}(i,k)$, $c_{k-|i \leq k|}(k,j)$ and $c_{k-|(i<k) \vee (i=k \wedge j \leq k)|}(k,k)$ before it computes $c_k(i,j)$ since we know from Observation 6.2.2 that $\pi(j,k) \geq k - |j \leq k|$, $\pi(i,k) \geq k - |i \leq k|$ and $\delta(i,j,k) \geq k - |(i < k) \vee (i = k \wedge j \leq k)|$ for all $i, j, k \in [1, n]$. However, these values could be overwritten before F needs to use them. In particular, we may lose certain key values as summarized in the observation below which follows from Theorem 6.2.1 and the definition of τ.

Observation 6.4.1. *Immediately before* F *applies the update* $\langle i, j, k \rangle \in \Sigma_G$*:*

(a) if $\tau_{ik}(\pi(j, k)) > k - |j \le k|$ *then* $c[i, k]$ *may not necessarily contain* $c_{k-|j\le k|}(i, k)$*;*

(b) if $\tau_{kj}(\pi(i, k)) > k - |i \le k|$ *then* $c[k, j]$ *may not necessarily contain* $c_{k-|i\le k|}(i, k)$*; and*

(c) if $\tau_{kk}(\delta(i, j, k)) > k - |(i < k) \ \vee \ (i = k \ \wedge \ j \le k)|$ *then* $c[k, k]$ *may not necessarily contain* $c_{k-|(i<k) \ \vee \ (i=k \ \wedge \ j\le k)|}(k, k)$*.*

If the condition in Observation 6.4.1(a) holds, we must save $c_{k-|j\le k|}(i, k)$ as soon as it is generated so that it can be used later by $\langle i, j, k \rangle$. However, $c_{k-|j\le k|}(i, k)$ is not necessarily generated by $\langle i, k, k - |j \le k| \rangle$ since this update may not exist in Σ_G in the first place. If $\tau_{ij}(k - |j \le k|) \ne 0$, then $\langle i, k, \tau_{ij}(k - |j \le k|) \rangle$ is the update that generates $c_{k-|j\le k|}(i, k)$, and we must save this value after applying this update and before some other update modifies it. If $\tau_{ij}(k - |j \le k|) = 0$, then $c_{k-|j\le k|}(i, k) = c_0(i, k)$, i.e., update $\langle i, j, k \rangle$ can use the initial value of $c[i, k]$. A similar argument applies to $c[k, j]$ and $c[k, k]$ as well.

Now in order to identify the intermediate values of each $c[i, j]$ that must be saved, consider the accesses made to $c[i, j]$ when executing the original GEP code in Figure 6.1.

Observation 6.4.2. *The GEP code in Figure 6.1 accesses each* $c[i, j]$*:*

(a) as $c[i, j]$ *at most once in each iteration of the outer **for** loop for applying updates* $\langle i, j, k \rangle \in \Sigma_G$*;*

(b) as $c[i, k]$ *only in the* j*-th iteration of the outer **for** loop, for applying updates* $\langle i, j', j \rangle \in \Sigma_G$ *for all* $j' \in [1, n]$*;*

(c) as $c[k, j]$ *only in the* i*-th iteration of the outer **for** loop, for applying updates* $\langle i', j, i \rangle \in \Sigma_G$ *for all* $i' \in [1, n]$*; and*

(d) if $i = j$, *as* $c[k, k]$ *in the* i*-th iteration of the outer **for** loop for applying updates* $\langle i', j', i \rangle \in \Sigma_G$ *for all* $i', j' \in [1, n]$*.*

The updates in Observation 6.4.2(a) do not need to be stored separately, since we know from Theorem 6.2.1 that both GEP and I-GEP apply the updates on a fixed $c[i, j]$ in exactly the same order.

Now consider the accesses to $c[i, j]$ in parts (b), (c) and (d) of Observation 6.4.2. By inspecting the code in Figure 6.1 (see also the 2nd column of Table 6.1), we observe that immediately before G applies the update $\langle i, j', j \rangle$ in Observation 6.4.2(b), $c[i, j] = \hat{c}_{j-1}(i, j) = \hat{c}_{\tau_{ij}(j-1)}(i, j)$ if $j' \le j$, and $c[i, j] = \hat{c}_j(i, j) = \hat{c}_{\tau_{ij}(j)}(i, j)$ otherwise. Similarly, immediately before applying the update $\langle i', j, i \rangle$ in Observation 6.4.2(c), $c[i, j] = \hat{c}_{i-1}(i, j) = \hat{c}_{\tau_{ij}(i-1)}(i, j)$ if $i' \le i$, and $c[i, j] = \hat{c}_i(i, j) = \hat{c}_{\tau_{ij}(i)}(i, j)$

otherwise. When G is about to apply an update $\langle i', j', i\rangle$ from Observation 6.4.2(d), $c[i,j] = \hat{c}_{i-1}(i,j) = \hat{c}_{\tau_{ij}(i-1)}(i,j)$ if $i' < i \ \vee \ (i' = i \ \wedge \ j' \leq j)$, and $c[i,j] = \hat{c}_i(i,j) = \hat{c}_{\tau_{ij}(i)}(i,j)$ otherwise.

Therefore, F must be modified to save the value of $c[i,j]$ immediately after applying the update $\langle i,j,k\rangle \in \Sigma_G$ for $k \in \{\tau_{ij}(i-1), \tau_{ij}(i), \tau_{ij}(j-1), \tau_{ij}(j)\}$. Observe that since there are exactly n^2 possible (i,j) pairs, we need to save at most $4n^2$ intermediate values.

In Figure 6.13 we present the modified version of F, which we call H. The algorithm has exactly the same structure as F, i.e., it accepts the same inputs as F (one square matrix X, and two integers k_1 and k_2) and assumes the same preconditions on inputs, it decomposes the input matrix in exactly the same way, and processes the submatrices in the same order using similar functions as F does. The only difference between F and H is in the way the updates are performed. In line 3, F updates $c[i,j]$ using entries directly from c, i.e., it updates $c[i,j]$ using whatever values $c[i,j]$, $c[i,k]$, $c[k,j]$ and $c[k,k]$ have at the time of the update. In contrast, H uses four $n \times n$ matrices u_0, u_1, v_0 and v_1 for saving appropriate intermediate values computed for the entries of c as discussed above, which it uses for future updates. We assume that each of the tests in lines 5–8 involving τ_{ij} can be performed in constant time without incurring any additional cache misses.

Cache Complexity & Running Time. The number of cache misses incurred by H can be described using the same recurrence relation (6.2.1) that was used to describe the cache misses incurred by F in Section 6.2, and hence the cache complexity remains the same, i.e., $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$. Function H also has the same $\mathcal{O}\left(n^3\right)$ running time as F, since it only incurs a constant overhead per update applied.

Correctness. Since Theorems 6.2.1 and 6.2.2 in Section 6.2 were proved based on the structural properties of F and not on the actual form of the updates, they continue to hold for H.

The correctness of H, i.e., that it correctly implements column 2 of Table 6.1 and thus G, follows directly from the following lemma, which can be proved by induction on k using Theorems 6.2.1 and 6.2.2, and by observing that H saves all required intermediate values in lines 5–8.

Lemma 6.4.1. *Immediately before* H *performs the update* $\langle i,j,k\rangle$*, the following hold:* $c[i,j] = \hat{c}_{k-1}(i,j)$*,* $v_{|j>k|}[i,k] = \hat{c}_{k-|j\leq k|}(i,k)$*,* $h_{|i>k|}[k,j] = \hat{c}_{k-|i\leq k|}(k,j)$ *and* $v_{|(i>k) \ \vee \ (i=k \ \wedge \ j>k)|}[k,k]) = \hat{c}_{k-|(i<k) \ \vee \ (i=k \ \wedge \ j\leq k)|}(k,k)$.

6.4.3 Reducing the Additional Space

We can reduce the amount of extra space used by H (Figure 6.13) by observing that at any point during the execution of H we do not need to store more than $n^2 + n$ intermediate values for future use. In fact, we will show that it is sufficient to use four $\frac{n}{2} \times \frac{n}{2}$ matrices and two vectors of length $\frac{n}{2}$ each for storing intermediate values, instead of using four $n \times n$ matrices.

Let $U \equiv u_0[1 \ldots n, 1 \ldots n]$, $\overline{U} \equiv u_1[1 \ldots n, 1 \ldots n]$, $V \equiv v_0[1 \ldots n, 1 \ldots n]$ and $\overline{V} \equiv v_1[1 \ldots n, 1 \ldots n]$. By U_{11}, U_{12}, U_{21} and U_{22} we denote the top-left, top-right, bottom-left and bottom-right quadrants of U, respectively. We identify the quadrants of $\overline{U}$, V and $\overline{V}$ similarly. For $i \in [1, 2]$, let D_i and $\overline{D}_i$ denote the diagonal entries of U_{ii} and $\overline{U}_{ii}$, respectively.

Now consider the initial call to H, i.e., $\mathrm{H}(X, k_1, k_2)$ where $X = c$, $k_1 = 1$ and $k_2 = n$. We show below that the forward pass in line 11 of this call can be implemented using only $n^2 + n$ extra space. A similar argument applies to the backward pass (line 12) as well.

The first recursive call $\mathrm{H}(X_{11}, k_1, k_2)$ in line 11 will generate U_{11}, $\overline{U}_{11}$, V_{11}, $\overline{V}_{11}$, D_1 and $\overline{D}_1$. The amount of extra space used by this recursive call is thus $n^2 + n$. The entries in U_{11} and V_{11}, however, will not be used by any future updates, and hence can be discarded. The second recursive call $\mathrm{H}(X_{12}, k_1, k_2)$ will use $\overline{U}_{11}$, D_1 and $\overline{D}_1$, and generate V_{12} and $\overline{V}_{12}$ in the space freed by discarding U_{11} and V_{11}. Each update $\langle i, j, k \rangle$ applied by this recursive call retrieves $u_{|j>k|}[i, k]$ from $\overline{U}_{11}$, $v_{|i>k|}[k, j]$ from V_{12} or $\overline{V}_{12}$, and $u_{|(i>k) \,\vee\, (i=k \,\wedge\, j>k)|}[k, k]$ from D_1 or $\overline{D}_1$. Upon return from $\mathrm{H}(X_{12}, k_1, k_2)$ we can discard the entries in $\overline{U}_{11}$ and V_{12} since they will not be required for any future updates. The next recursive call $\mathrm{H}(X_{12}, k_1, k_2)$ will use $\overline{V}_{11}$, D_1 and $\overline{D}_1$, and generate U_{21} and $\overline{U}_{21}$ in the space previously occupied by $\overline{U}_{11}$ and V_{12}. Each update performed by this recursive call retrieves $u_{|j>k|}[i, k]$ from U_{21} or $\overline{U}_{21}$, $v_{|i>k|}[k, j]$ from $\overline{V}_{11}$, and $u_{|(i>k) \,\vee\, (i=k \,\wedge\, j>k)|}[k, k]$ from D_1 or $\overline{D}_1$. The last function call $\mathrm{H}(X_{22}, k_1, k_2)$ in line 11 will use $\overline{U}_{21}$, $\overline{V}_{12}$, D_1 and $\overline{D}_1$ for updates, and will not generate any intermediate values. Thus line 11 can be implemented using only four additional $\frac{n}{2} \times \frac{n}{2}$ matrices and two vectors of length $\frac{n}{2}$ each.

Therefore, H can be implemented to work with any arbitrary f and arbitrary Σ_G at the expense of only $n^2 + n$ extra space. The running time and the cache complexity of this implementation remain $\mathcal{O}\left(n^3\right)$ and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$, respectively.

6.5 Parallel I-GEP and C-GEP

In this section we consider parallel implementations of I-GEP and C-GEP. It is not difficult to observe that the second and third calls to F in line 5 of the pseudocode for I-GEP given in Figure 6.2 can be performed in parallel while maintaining correctness and all properties we have established for I-GEP; similarly the second and third calls to F in line 6 can be performed in parallel. A similar observation holds for lines 11 and 12 of H. The resulting parallel code performs a sequence of 6 parallel calls (four calling F or H once and two calling F or H twice), and hence with p processors its parallel execution time is $\mathcal{O}\left(n^3/p + n^{\log_2 6}\right)$.

A(X, U, V, W)

(Each of X, U, V and W points to the same $2^q \times 2^q$ square submatrix of c for some integer $q \geq 0$. The initial call to A is A(c, c, c, c) for an $n \times n$ input matrix c, where n is assumed to be a power of 2.)

1. ***if*** $T_{XUV} \cap \Sigma_G = \emptyset$ ***then return*** { T_{XUV} = { *updates on* X *using* $(i,k) \in U$ *and* $(k,j) \in V$ }, *and* Σ_G *is the set of updates performed by iterative GEP*}
2. ***if*** X is a 1×1 matrix ***then*** $X \leftarrow f(X, U, V, W)$
   else {X_{11}, X_{12}, X_{21} *and* X_{22} *are the top-left, top-right, bottom-left and bottom-right quadrants of* X*, respectively.*}
3. A(X_{11}, U_{11}, V_{11}, W_{11})
4. ***parallel*** : B$_1$(X_{12}, U_{11}, V_{12}, W_{11}), C$_1$(X_{21}, U_{21}, V_{11}, W_{11})
5. D$_1$(X_{22}, U_{21}, V_{12}, W_{11})
6. A(X_{22}, U_{22}, V_{22}, W_{22})
7. ***parallel*** : B$_2$(X_{21}, U_{22}, V_{21}, W_{22}), C$_2$(X_{12}, U_{12}, V_{22}, W_{22})
8. D$_4$(X_{11}, U_{12}, V_{21}, W_{22})

B$_l$(X, U, V, W) { $l \in \{1, 2\}$ }

($X \equiv V \equiv c[i_1..i_2, j_1..j_2]$ and $U \equiv W \equiv c[i_1..i_2, k_1..k_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$, $[i_1, i_2] = [k_1, k_2]$ and $[j_1, j_2] \cap [k_1, k_2] = \emptyset$.)

1. ***if*** $T_{XUV} \cap \Sigma_G = \emptyset$ ***then return***
2. ***if*** X is a 1×1 matrix ***then***
   $X \leftarrow f(X, U, V, W)$
   else
3. ***parallel*** : B$_l$(X_{11}, U_{11}, V_{11}, W_{11})
   B$_l$(X_{12}, U_{11}, V_{12}, W_{11})
4. ***parallel*** : D$_l$(X_{21}, U_{21}, V_{11}, W_{11})
   D$_l$(X_{22}, U_{21}, V_{12}, W_{11})
5. ***parallel*** : B$_l$(X_{21}, U_{22}, V_{21}, W_{22})
   B$_l$(X_{22}, U_{22}, V_{22}, W_{22})
6. ***parallel*** : D$_{l+2}$(X_{11}, U_{12}, V_{21}, W_{22})
   D$_{l+2}$(X_{12}, U_{12}, V_{22}, W_{22})

C$_l$(X, U, V, W) { $l \in \{1, 2\}$ }

($X \equiv U \equiv c[i_1..i_2, j_1..j_2]$ and $V \equiv W \equiv c[k_1..k_2, j_1..j_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$, $[j_1, j_2] = [k_1, k_2]$ and $[i_1, i_2] \cap [k_1, k_2] = \emptyset$.)

1. ***if*** $T_{XUV} \cap \Sigma_G = \emptyset$ ***then return***
2. ***if*** X is a 1×1 matrix ***then***
   $X \leftarrow f(X, U, V, W)$
   else
3. ***parallel*** : C$_l$(X_{11}, U_{11}, V_{11}, W_{11})
   C$_l$(X_{21}, U_{21}, V_{11}, W_{11})
4. ***parallel*** : D$_{2l-1}$(X_{12}, U_{11}, V_{12}, W_{11})
   D$_{2l-1}$(X_{22}, U_{21}, V_{12}, W_{11})
5. ***parallel*** : C$_l$(X_{12}, U_{12}, V_{22}, W_{22})
   C$_l$(X_{22}, U_{22}, V_{22}, W_{22})
6. ***parallel*** : D$_{2l}$(X_{11}, U_{12}, V_{21}, W_{22})
   D$_{2l}$(X_{21}, U_{22}, V_{21}, W_{22})

D$_l$(X, U, V, W) { $l \in \{1, 2, 3, 4\}$ }

($X \equiv c[i_1..i_2, j_1..j_2]$, $U \equiv c[i_1..i_2, k_1..k_2]$, $V \equiv c[k_1..k_2, j_1..j_2]$ and $W \equiv c[k_1..k_2, k_1..k_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$, $[i_1, i_2] \cap [k_1, k_2] = \emptyset$, and $[j_1, j_2] \cap [k_1, k_2] = \emptyset$.)

1. ***if*** $T_{XUV} \cap \Sigma_G = \emptyset$ ***then return***
2. ***if*** X is a 1×1 matrix ***then*** $X \leftarrow f(X, U, V, W)$
   else
3. ***parallel*** : D$_l$(X_{11}, U_{11}, V_{11}, W_{11}), D$_l$(X_{12}, U_{11}, V_{12}, W_{11}),
   D$_l$(X_{21}, U_{21}, V_{11}, W_{11}), D$_l$(X_{22}, U_{21}, V_{12}, W_{11})
4. ***parallel*** : D$_l$(X_{11}, U_{12}, V_{21}, W_{22}), D$_l$(X_{12}, U_{12}, V_{22}, W_{22}),
   D$_l$(X_{21}, U_{22}, V_{21}, W_{22}), D$_l$(X_{22}, U_{22}, V_{22}, W_{22})

Figure 6.14: Multithreaded I-GEP. Initial call is A(c, c, c, c) on an $n \times n$ matrix c, where n is a power of 2.

In Figure 6.14 we present a better parallel implementation of I-GEP. In this figure we have explicitly referred to the different types of functions invoked by I-GEP based on the relative values of the i, j, and k intervals. Here we use the notation introduced in Section 6.2.4 and summarized in Figures 6.6, 6.7, 6.8 and 6.9. The four types of functions (i.e., A, B_l, C_l and D_l) differ in the amount and type of overlap the input matrices X, U and V have among them (note that only the diagonal entries of W are used). Function A assumes that all three matrices overlap, while function D_l expects completely non-overlapping matrices. Function B_l assumes that only X and V overlap, while C_l assumes overlap only between X and U. Intuitively, the less the overlap among the input matrices the more flexibility the function has in ordering its recursive calls, and thus leading to better parallelism. The initial call is to an input of type A, where the intervals for i, j and k are identical.

We now analyze the parallel execution time for I-GEP on function A. Let $T_A(n) = T_\infty$ denote the parallel running time when A is invoked with an unbounded number of processors on an $n \times n$ matrix. Let $T_B(n)$, $T_C(n)$ and $T_D(n)$ denote the same for B_i, C_i and D_i, respectively. We will assume for simplicity that $T_A(1) = T_B(1) = T_C(1) = T_D(1) = \mathcal{O}(1)$. Hence we have the following recurrences:

$$T_A(n) \leq 2(T_A(n/2) + \max\{T_B(n/2), T_C(n/2)\} + T_D(n/2)) + \mathcal{O}(1)$$

$$T_B(n) \leq 2(T_B(n/2) + T_D(n/2)) + \mathcal{O}(1)$$

$$T_C(n) \leq 2(T_C(n/2) + T_D(n/2)) + \mathcal{O}(1)$$

$$T_D(n) \leq 2T_D(n/2) + \mathcal{O}(1)$$

Solving these recurrences we obtain $T_\infty = \mathcal{O}\left(n \log^2 n\right)$, and thus the following theorem based on "Brent's theorem" [20]:

Theorem 6.5.1. *When executed with p processors, multithreaded I-GEP performs $T_1 = \mathcal{O}\left(n^3\right)$ work and terminates in $\frac{T_1}{p} + T_\infty = \mathcal{O}\left(\frac{n^3}{p} + n\log^2 n\right)$ parallel steps on an $n \times n$ input matrix.*

A similar parallel algorithm with the same parallel time bound applies to C-GEP.

For specific applications of I-GEP, the actual recursive function calls may not take the most general form analyzed above (see Section 6.2.4). For instance, only a subset of the calls are made for Gaussian elimination without pivoting. However, the parallel time bound remains the same as in Theorem 6.5.1 for this problem as well as for all-pairs shortest paths. On the other hand, for matrix multiplication, we can perform all four recursive calls in each of steps 5 and 6 of Figure 6.6 in parallel and hence the parallel time bound is reduced to $\mathcal{O}\left(\frac{n^3}{p} + n\right)$. Note that this matrix multiplication computation does not assume associativity of addition.

We have implemented this multithreaded version of I-GEP for Floyd-Warshall's APSP, square matrix multiplication and Gaussian elimination w/o pivoting in `pthreads`, and we report some experimental results in Chapter 7.

6.5.1 Cache Complexity

We first consider distributed caches, where each processor has its own private cache, and then a shared cache, where all processors share the same cache.

Distributed Caches. Part (b) of the following lemma is obtained by considering the schedule that executes each subproblem of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ entirely on a single processor. This schedule gives a better result than the one given in part (a) for the work-stealing scheduler Cilk [53]; the bound in part (a) is obtained by applying a result in [55] on the caching performance of parallel algorithms whose sequential cache complexity is a concave function of work.

Lemma 6.5.1. *Consider multithreaded I-GEP executed with p processors, each with a private cache of size M and block size B.*

(a) *When executed by Cilk, with high probability I-GEP incurs* $\mathcal{O}\Big(\frac{n^3}{B\sqrt{M}} + \frac{(p \cdot n \log^2 n)^{\frac{1}{3}} n^2}{B} + p \cdot n \log^2 n\Big)$ *cache misses.*

(b) *There exists a deterministic schedule which incurs only* $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \sqrt{p} \cdot \frac{n^2}{B}\right)$ *cache misses.*

Shared Caches. Here we consider the case when the p processors share a single cache of size M_p. Part (a) of lemma 6.5.2 below is obtained using a general result for shared caches given in [18] for a PDF *(parallel depth first search)* schedule. Better bounds are obtained in part (b) of lemma 6.5.2 through the following hybrid depth-first schedule.

Let $\mathcal{G}$ denote the computation DAG of I-GEP (i.e., function A), and let $\mathcal{C}(\mathcal{G})$ denote a new DAG obtained from $\mathcal{G}$ by contracting each subDAG of $\mathcal{G}$ corresponding to a recursive function call on an $r \times r$ submatrix to a supernode, where r is a power of 2 such that $\sqrt{p} \leq r < 2\sqrt{p}$. The subDAG in $\mathcal{G}$ corresponding to any supernode v is denoted by $\mathcal{S}(v)$.

Now the hybrid scheduling scheme is applied on $\mathcal{G}$ as follows. The scheduler executes the nodes (i.e., supernodes) of $\mathcal{C}(\mathcal{G})$ under 1DF-schedule [18]. However, for each supernode v, the scheduler uses a PDF-schedule with all p processors in order to execute the subDAG $\mathcal{S}(v)$ of $\mathcal{G}$ before moving to the next supernode. This leads to the following.

$\mathcal{F}(v)$	A	$\mathrm{B}_i \quad (i = 1, 2)$	$\mathrm{C}_i \quad (i = 1, 2)$	$\mathrm{D}_i \quad (i \in [1, 4])$
$n(\mathcal{F}(v))$	$\frac{n}{r}$	$\left(\frac{n}{r}\right)^2 - \frac{n}{r}$	$\left(\frac{n}{r}\right)^2 - \frac{n}{r}$	$\left(\frac{n}{r}\right)^3 - 2\left(\frac{n}{r}\right)^2 + \frac{n}{r}$
$s(\mathcal{F}(v))$	$\mathcal{O}\left(r \log^2 r\right)$	$\mathcal{O}\left(r \log r\right)$	$\mathcal{O}\left(r \log r\right)$	$\mathcal{O}\left(r\right)$

Table 6.2: Properties of supernodes in $\mathcal{C}(\mathcal{G})$: for a given supernode v, $\mathcal{F}(v)$ denotes the recursive function represented by subDAG $\mathcal{S}(v)$ in $\mathcal{G}$ while $n(\mathcal{F}(v))$ and $s(\mathcal{F}(v))$ denote the number of supernodes in $\mathcal{C}(\mathcal{G})$ representing $\mathcal{F}(v)$ and the number of parallel steps required to execute $\mathcal{F}(v)$, respectively.

Lemma 6.5.2. *For $p \geq 1$ let multithreaded I-GEP execute T_p parallel steps and incur Q_p cache misses with p processors and on a shared ideal cache of M_p blocks. Then*

(a) *With a* PDF*-schedule, $Q_p \leq Q_1$ if $M_p \geq M_1 + \Theta\left(pn \log^2 n\right)$.*

(b) *With the hybrid depth-first schedule,*

i. $Q_p \leq Q_1$ if $M_p \geq M_1 + \Theta\left(p\right)$,

ii. If $M_1 = M_p$ then $Q_p = O(Q_1)$ provided $p = O(M_p)$.

(c) *For both schedules, $T_p = \mathcal{O}\left(\frac{n^3}{p} + n \log^2 n\right)$.*

Proof. (sketch) (a) This part follows from a general result for shared caches given in [18] which states that under PDF-schedule $Q_p \leq Q_1$ provided $M_p \geq M_1 + \Theta\left(p \cdot T_\infty(n)\right)$.

($b.i$) Since for each supernode v in $\mathcal{C}(\mathcal{G})$ the subDAG $\mathcal{S}(v)$ in $\mathcal{G}$ accesses at most $\Theta\left(r^2\right)$ locations of the input matrix, when executing $\mathcal{S}(v)$ under PDF-schedule no more than $\Theta\left(r^2\right) = \Theta\left(p\right)$ nodes can become *premature* [18] simultaneously. Since supernodes are executed one at a time, having $M_p \geq M_1 + \Theta\left(p\right)$ ensures that there is always enough space in the shared cache to accommodate the premature nodes without ever incurring any extra cache misses. Therefore, $Q_p \leq Q_1$.

($b.ii$) Suppose $M_p = M_1 = M$. Since the hybrid schedule never creates more than $\Theta\left(p\right)$ simultaneous premature nodes (see part (a)), we can set aside $\Theta\left(p\right)$ locations in the shared cache for holding the premature nodes. The effective cache size thus reduces to $M - \Theta\left(p\right)$, and assuming $M - \Theta\left(p\right) = \Omega\left(M\right) \Rightarrow p = \mathcal{O}\left(M\right)$, the number of cache misses incurred by multithreaded I-GEP is $Q_p \leq \mathcal{O}\left(\frac{n^3}{B\sqrt{M - \Theta(p)}}\right) = \mathcal{O}\left(Q_1\right)$.

(c) The claimed parallel running time for PDF-schedule follows from the results in [18]. Therefore, we restrict our attention to the hybrid scheduler below.

Observe that $\mathcal{G}$ has $\Theta\left(n^3\right)$ nodes, and each subDAG in $\mathcal{G}$ corresponding to supernodes in $\mathcal{C}(\mathcal{G})$ has $\Theta\left(r^3\right)$ nodes. Therefore, $\mathcal{C}(\mathcal{G})$ has only $\Theta\left(\left(\frac{n}{r}\right)^3\right)$ nodes.

For a given supernode v, let $\mathcal{F}(v)$ denote the recursive function represented by subDAG $\mathcal{S}(v)$ in $\mathcal{G}$. Let $n(\mathcal{F}(v))$ and $s(\mathcal{F}(v))$ denote the number of supernodes in $\mathcal{C}(\mathcal{G})$ representing $\mathcal{F}(v)$ and the number of parallel steps required to execute $\mathcal{F}(v)$, respectively. The values of $n(\mathcal{F}(v))$ and $s(\mathcal{F}(v))$ for $\mathcal{F}(v) \in \{A, B_i, C_i, D_i\}$ are tabulated in Table 6.2 (the calculations are not difficult and are omitted for brevity). Therefore, the number of parallel steps required to execute all supernodes is $\sum_{\mathcal{F}(v)\in\{A,B_i,C_i,D_i\}} n(\mathcal{F}(v)) \times s(\mathcal{F}(v)) = \mathcal{O}\left(\frac{n^3}{r^2} + \frac{n^2}{r}\log r + n\log r\right) = \mathcal{O}\left(\frac{n^3}{p} + n\log^2 n\right)$ (since $p \leq n^2$).

Since $\mathcal{C}(\mathcal{G})$ has only $\Theta\left(\left(\frac{n}{r}\right)^3\right)$ nodes, the number of steps required to execute $\mathcal{C}(\mathcal{G})$ under 1DF-schedule is $\mathcal{O}\left(\left(\frac{n}{r}\right)^3\right) = \mathcal{O}\left(\frac{n^3}{p\sqrt{p}}\right)$. Therefore, the total number of parallel steps required to execute multithreaded I-GEP under the hybrid depth-first schedule is $\mathcal{O}\left(\frac{n^3}{p} + \frac{n^3}{p\sqrt{p}} + n\log^2 n\right) = \mathcal{O}\left(\frac{n^3}{p} + n\log^2 n\right) = \mathcal{O}\left(\frac{T_1}{p} + T_\infty\right)$ since $T_1 = n^3$ and $T_\infty = \mathcal{O}\left(n\log^2 n\right)$. ■

6.6 Cache-oblivious GEP and Compiler Optimization

'Tiling' is a powerful loop transformation technique employed by optimizing compilers for improving temporal locality in nested loops [91]. This transformation partitions the iteration-space of nested loops into a series of small polyhedral areas of a given *tile size* which are executed one after the other. Tiling a single loop replaces it by a pair of loops, and if the tile size is T then the inner loop iterates T times, and the outer loop has an increment equal to T (assuming that the original loop had unit increments). This transformation can be applied to arbitrarily deep nested loops. Figure 6.15(b) shows a tiled version of the triply nested loop shown in Figure 6.15(a) that occurs in matrix multiplication [91].

Cache performance of a tiled loop depends on the chosen tile size T. Choice of T, in turn, crucially depends on (1) the type of the cache (direct mapped or set associative), (2) cache size, (3) block transfer size (i.e., cache line size), and (4) the loop bounds [91, 133]. Thus tiling is a highly system-dependent technique. Moreover, since only a single tile size is chosen, tiling cannot be optimized for all levels of a memory hierarchy simultaneously.

The I-GEP code in Figure 6.2 and the C-GEP code given in Figure 6.13 can be viewed as cache-oblivious versions of tiling for the triply nested loops of the form as shown in Figure 6.1. The nested loop in Figure 6.1 has an $n \times n \times n$ iteration-space. Both I-GEP and C-GEP are initially invoked on this $n \times n \times n$ cube, and at each stage of recursion they partition the input cube into 8 equal-sized subcubes, and recursively process each subcube. Hence, at some stage of recursion, they are guaranteed to generate subcubes of size $T' \times T' \times T'$ such that $\frac{T}{2} < T' \leq T$, where T is the optimal tile size for any given level of the memory hierarchy. Thus for each level of the memory hierarchy both I-GEP and C-GEP cache-obliviously choose a

```
1. for i ← 1 to n do
2.    for j ← 1 to n do
3.       for k ← 1 to n do
4.          c[i, j] ← c[i, j]
                      + a[i, k] × b[k, j]
```

(a)

```
1. for i ← 1 to n by T do        {T is the tile size}
2.    for j ← 1 to n by T do
3.       for k ← 1 to n by T do
4.          for i' ← i to min(i + T − 1, n) do
5.             for j' ← j to min(j + T − 1, n) do
6.                for k' ← k to min(k + T − 1, n) do
7.                   c[i', j'] ← c[i', j'] + a[i', k'] × b[k', j']
```

(b)

Figure 6.15: **(a)** Traditional matrix multiplication algorithm, **(b)** Tiled version of the matrix multiplication algorithm of part (a) [91].

tile size that is within a constant factor of the optimal tile size for that level. We can, therefore, use I-GEP and C-GEP as cache-oblivious loop transformations for the memory hierarchy.

C-GEP. C-GEP is a legal transformation for any nested loop that conforms to the GEP format given in Figure 6.1. In order to apply this transformation the compiler must be able to evaluate $\tau_{ij}(i-1)$, $\tau_{ij}(i)$, $\tau_{ij}(j-1)$ and $\tau_{ij}(j)$ for all $i, j \in [1, n]$. For most practical problems this is straight-forward; for example, when $\Sigma_G = \{\langle i, j, k\rangle \mid i, j, k \in [1, n]\}$ which occurs in path computations over closed semirings (see Section 6.3.3), or even if the computation is not over a closed semiring, we have $\tau_{ij}(l) = l$ for all $i, j, l \in [1, n]$.

I-GEP. Though C-GEP is always a legal transformation for GEP loops, I-GEP is not. Due to the space overhead of C-GEP, I-GEP should be the transformation of choice wherever it is applicable. Moreover, experimental results (see Chapter 7) suggest that I-GEP outperforms C-GEP in both in-core and out-of-core computations.

We will now look at some general conditions under which I-GEP is a legal transformation for a given GEP code. Consider the general GEP code in Figure 6.1. Recall the definition of π from Section 6.2, and the definition of τ_{ij} from Section 6.4.2 (Definition 6.4.1). The following lemma follows from Observations 6.4.1 and 6.4.2 in Section 6.4.2, and also from the observation that I-GEP will correctly implement GEP if for each $c[i, j]$ and each update in Σ_G that uses $c[i, j]$ on the right hand side, $c[i, j]$ retains the correct value needed for that update until I-GEP applies the update.

Lemma 6.6.1. *If $\tau_{ij}(\pi(k, i)) \le i - |k \le i|$ for all $\langle i, k, j\rangle \in \Sigma_G$, and $\tau_{ij}(\pi(k, j)) \le j - |k \le j|$ for all $\langle k, j, i\rangle \in \Sigma_G$, then I-GEP is a legal transformation for the GEP code in Figure 6.1.*

6.7 An Additional Application of Cache-oblivious I-GEP

In Section 6.3 we considered three major applications of I-GEP. In this section we consider a class of dynamic programs called 'simple DP' [30] that includes important problems such as RNA secondary structure prediction, matrix chain multiplication and construction of optimal binary search trees. In Section 6.7.1 we show how simple DP can be decomposed into a sequence of I-GEP instances using a decomposition technique from [57], we argue the correctness of our decomposition and prove bounds on its cache performance.

6.7.1 Simple Dynamic Programs

In [30], the term *simple dynamic program* was used to denote a class of dynamic programming problems over a nonassociative semiring $(S,\ \min,\ +,\ \infty)$[2] which can be solved in $\mathcal{O}\left(n^3\right)$ time using the dynamic program shown in Figure 6.16. Its applications include RNA secondary structure prediction, optimal matrix chain multiplication, construction of optimal binary search trees, and optimal polygon triangulation. An $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithm based on Valiant's context-free language recognition algorithm [124] was given in [30] for this class of problems.

In this section, we consider a more general version of simple DP, which is called the *parenthesis problem* in [57], and is described as follows (the generalization comes from the additional term $w(i,k,j)$):

$$c[i,j] = \begin{cases} x_j & \text{if } 0 \le i = j-1 < n, \\ \min_{i<k<j}\left\{c[i,k] + c[k,j] + w(i,k,j)\right\} & \text{if } 0 \le i < j-1 < n; \end{cases} \tag{6.7.3}$$

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 incurring any cache misses.

We describe below a method that transforms the dynamic program given in 6.7.3 to a sequence of dynamic programs in GEP. In this method the upper triangular matrix c is decomposed into (forward) diagonal strips of horizontal width $n^{\frac{1}{4}}$, and the entries in c are computed one strip at a time starting from the largest (leftmost) strip. The computation for each strip involves min-plus matrix multiplication and dynamic programs that can be solved with cache-oblivious I-GEP. The resulting algorithm runs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/Os. Unlike I-GEP, however, this algorithm uses a modest amount $\left(\mathcal{O}\left(n^{1.75}\right) = o\left(n^2\right)\right)$ of extra space. This method is based on a parallel algorithm for the parenthesis problem given in [57]. There are,

[2] In a nonassociative semiring min is an associative, commutative and idempotent binary operator; + is a nonassociative and noncommutative binary operator; ∞ is the identity for min and annihilator for +; and the operators distribute over each other.

```
1. for i ← 0 to n − 1 do c[i, i + 1] ← x_{i+1}
2. for d ← 2 to n do
3.    for i ← 0 to n − d do
4.       j ← i + d,  c[i, j] ← ∞
5.       for k ← i + 1 to j − 1 do
6.          c[i, j] ← min{c[i, j], c[i, k] + c[k, j]}
```

Figure 6.16: The $\mathcal{O}\left(n^3\right)$ time simple DP algorithm.

however, two major differences between the transformation described here and the algorithm in [57].

(i) We reorder the execution of some of the steps in the algorithm (without affecting its correctness) for better space utilization.

(ii) We reduce computations involving 4-dimensional arrays to computations on 2-dimensional arrays (see step 2.1) so that GEP can be applied (since GEP works on a 2-dimensional input matrix).

We now describe the transformation. The recurrence relation in 6.7.3 can be viewed as computing a binary tree of minimum weight [126] in which

(a) each vertex is given a unique label (i, j), $0 \le i < j \le n$,

(b) leaves are labeled $(i, i + 1)$, $i \in [0, n - 1]$ in order from left to right with x_{i+1} being the weight of leaf $(i, i + 1)$, and

(c) each internal node (i, j) has weight $w(i, k, j)$, left child (i, k) and right child (k, j) for some $k \in [i+1, j-1]$, and its descendant leaves are labeled $(i', i'+1)$ for $i \le i' < j$.

For each (i, j) with $0 \le i < j \le n$, the dynamic program given by 6.7.3 computes in $c[i, j]$ the cost $g(i, j)$ of the optimal (i.e., minimum-weight) binary tree rooted at (i, j). A *partial tree* T is defined to be a tree rooted at some vertex (i, j) with the subtree rooted at one of its non-leaf nodes (r, s) deleted. Then (r, s) is said to be the *gap* of T. Let $L(r, j, i, j)$, $r > i$, be the cost of the partial tree rooted at (i, j) with gap (r, j) such that (r, j) is the right child of (i, j). Let $L(i, s, i, j)$, $s < j$, be defined similarly. Then $L(r, j, i, j) = g(i, r) + w(i, r, j)$, and $L(i, s, i, j) = g(s, j) + w(i, s, j)$. For all other cases $L(r, s, i, j)$ is assumed to be $+\infty$. Let $L^*(r, s, i, j)$ be the cost of the optimal partial tree rooted at (i, j) with gap (r, s).

Initially, $g(i, i + 1)$ is given for all $i \in [0, n - 1]$, and $g(i, j) = +\infty$ for all others, and $L^*(r, s, i, j)$ is initialized to $+\infty$ for all r, s, i, j.

Given two $n \times n \times n \times n$ 4-dimensional 'matrices' L_1 and L_2, the product $L_3 = L_1 L_2$ is defined for all $0 \le i \le r < s \le j \le n$ as in [57]:

$$L_3(r,s,i,j) = \min_{i \le k_1 \le r, s \le k_2 \le j} \{L_1(r,s,k_1,k_2) + L_2(k_1,k_2,i,j)\}$$

The upper triangular matrix c is decomposed into forward diagonal strips of horizontal width $n^{\frac{1}{4}}$ each, and the entries in c are computed using the following steps:

Step 1. The g values in the first (leftmost) strip of width $n^{\frac{1}{4}}$ is computed using Rytter's algorithm [107]. This takes $\mathcal{O}\left(n^{2.25}\log n\right)$ time and $\mathcal{O}\left(\frac{n^{2.25}\log n}{B}\right)$ I/Os since a straight-forward implementation of Rytter's algorithm involves only linear scans (i.e., no random accesses).

Now starting from the second strip the following two steps are executed for each strip until the last one. This is unlike [57], where the first step (Step 2.1) is executed for all strips, followed by the execution of the second step (Step 2.2) for all strips, thus optimizing parallel computation time. Interleaving these two steps as we do below allows space reuse in sequential computations, and thus reduces space requirement.

Step 2.1. We compute L^* of the strip (i.e., all entries $L^*(r,s,i,j)$, where both (i,j) and (r,s) belong to the strip, and $i \le r < s \le j$) recursively as follows. Let S be a strip of width ν (initially $\nu = n^{\frac{1}{4}}$), and let S_1 and S_2 be strips of width $\frac{\nu}{2}$ each such that they compose S, and the diagonals of S_1 are larger than those of S_2. We recursively compute L^* of S_1 followed by the recursive computation of L^* of S_2, and then we combine these results to compute L^* of S.

We set $L(r,j,i,j) = g(i,r) + w(i,r,j)$ and $L(i,s,i,j) = g(s,j) + w(i,s,j)$ initially. For these initializations $g(i,r)$ and $g(s,j)$ are retrieved from the first strip since $r - i \le n^{\frac{1}{4}}$ and $j - s \le n^{\frac{1}{4}}$.

Let G_S, G_1 and G_2 be L^* of S, S_1 and S_2, respectively, and let L_S be L in strip S. Then as shown in [57], $G_S = G_2 L_S G_1$. These multiplications involve computations using four dimensional arrays. We reduce these multiplications to computations using two dimensional arrays as follows.

There are at most $n\nu$ entries in a strip S of width ν. We assign an index to each entry. The first entry in the first row gets index 1, and then we assign indices using consecutive integers such that entries in higher-numbered rows get higher indices, and within the same row entries in higher-numbered columns get higher indices. Thus there are at most $n\nu$ indices. Now let X be an $n\nu \times n\nu$ matrix. We copy each entry from L_S corresponding to the strip S to X. Suppose $(i,j),(r,s) \in S$. Then if $r - i \le \nu$ and $j - s \le \nu$, we copy $L_S(r,s,i,j)$ to $X[id(i,j), id(r,s)]$, where $id(i,j)$ and $id(r,s)$ are the indices assigned to (i,j) and (r,s), respectively. Thus the entries from L_S corresponding to the strip S of horizontal width ν form a forward diagonal strip of horizontal width ν^2 in X. However, instead of allocating space for the entire $n\nu \times n\nu$ matrix X, we store this horizontal strip in an $n\nu \times \nu^2$ rectangular matrix. We apply similar transformations to strips S_1 and S_2, too. We can then easily multiply those larger strips in two dimensions.

There are several useful properties of the matrix multiplications performed in this step using the larger strips. First, the multiplication is min-plus, i.e., performing the same update on the same location several times do not affect the final result. Second, updates applicable on the same location can be applied in any order. Third, the updates are somewhat local, i.e., an entry in the output matrix depends only on entries that are horizontally or vertically at most at a distance ν^2 from the corresponding entry in the input matrix. Therefore, we divide the strip of width ν^2 in X into $\mathcal{O}\left(\frac{n}{\nu}\right)$ squares of size $2\nu^2 \times 2\nu^2$ each such that the last ν^2 rows of each square overlaps with the first ν^2 rows of the square below it.

Therefore, $G_S = G_2 L_S G_1$ can be computed using $\mathcal{O}\left(\frac{n}{\nu}\right)$ multiplications involving $2\nu^2 \times 2\nu^2$ matrices, each of which can be implemented cache-obliviously using I-GEP to incur only $\mathcal{O}\left(\frac{\nu^6}{B\sqrt{M}}\right)$ I/Os. For the entire strip the number of cache misses is thus $\mathcal{O}\left(\frac{n}{\nu} \times \frac{\nu^6}{B\sqrt{M}}\right) = \mathcal{O}\left(\frac{n\nu^5}{B\sqrt{M}}\right)$. For $\nu = n^{\frac{1}{4}}$, the I/O complexity is $\mathcal{O}\left(\frac{n^{2.25}}{B\sqrt{M}}\right)$. Since the number of cache misses decreases by a constant factor as width decreases, the total number of cache misses is $\mathcal{O}\left(\frac{n^{2.25}}{B\sqrt{M}}\right)$. The amount of extra space used is $\mathcal{O}\left(n\nu \times \nu^2\right) = \mathcal{O}\left(n^{1.75}\right)$, which is reused by this step for processing every strip of c.

Step 2.2. Let S be a strip of width ν for which we want to compute g, and let S' be the strip of width $l\nu$ for which g has already been computed (i.e., all previous strips). Then g values for strip S can be computed by the following steps. For $(i,j) \in S$,

$$g'(i,j) = \min_{j-l\nu \leq k \leq i+l\nu} \left\{ g(i,k) + g(k,j) + w(i,k,j) \right\}.$$

And for $(i,j),(r,s) \in S$,

$$g(i,j) = \min_{i \leq r,s \leq j} \left\{ g'(r,s) + L^*(r,s,i,j) \right\} \tag{6.7.4}$$

In the step for computing $g'(i,j)$ we only need to consider those (i,r) and (s,j) such that $(i,r),(s,j) \in S'$. The computation is again min-plus, and the output is updated using entries directly from the input which is unchanged. Therefore, we can implement this step cache-obliviously using I-GEP as in Section 6.3.2, and update only the entries in S. We observe that S can be completely covered by $\mathcal{O}\left(\frac{n}{\nu}\right)$ non-overlapping squares of size $\nu \times \nu$ each. From Theorem 6.2.3 we know that I-GEP incurs $\mathcal{O}\left(\frac{\nu^2 n}{B\sqrt{M}}\right)$ cache misses for updating only the entries of any particular $\nu \times \nu$ square. Hence, total number of cache misses incurred for computing g' for S is $\mathcal{O}\left(\frac{n}{\nu} \times \frac{\nu^2 n}{B\sqrt{M}}\right) = \mathcal{O}\left(\frac{n^2\nu}{B\sqrt{M}}\right)$. For $\nu = n^{\frac{1}{4}}$, the I/O complexity is thus $\mathcal{O}\left(\frac{n^{2.25}}{B\sqrt{M}}\right)$.

Now consider the step that computes g from g'. Before executing this step we copy the entries of g' corresponding to strip S to a linear array g'' such that for any $(r_1,s_1),(r_2,s_2) \in S$, $g'(r_1,s_1)$ appears before $g'(r_2,s_2)$ in g'' provided $r_1 < r_2$, or $r_1 = r_2$ and $s_1 < s_2$. Recall from Step 2.1 that for each $(i,j) \in S$, all $L^*(r,s,i,j)$ values with $i \le r, s \le j$ occupy a single row of width ν^2 in an $n\nu \times n\nu$ array X, and the ordering of the $L^*(r,s,i,j)$ values in that row is exactly similar to the ordering of the $g'(r,s)$ values in g''. Hence for every $(i,j) \in S$, we can compute $g(i,j)$ just by scanning the corresponding row in X and the relevant portion of length $\mathcal{O}\left(\nu^2\right)$ from the linear array g'' and pairing up appropriate entries from these two sources according to 6.7.4. Since there are $\mathcal{O}\left(n\nu\right)$ pairs of (i,j)'s in S, computing all g values for S will incur $\mathcal{O}\left(n\nu\left(1+\frac{\nu^2}{B}\right)\right) = \mathcal{O}\left(n\nu + \frac{n\nu^3}{B}\right)$ cache misses, which is $\mathcal{O}\left(n^{1.25} + \frac{n^{1.75}}{B}\right)$ for $\nu = n^{\frac{1}{4}}$.

Since there are $\mathcal{O}\left(n^{0.75}\right)$ strips of width $n^{\frac{1}{4}}$, step 2 will be executed $\mathcal{O}\left(n^{0.75}\right)$ times, and the total number of cache misses incurred by this step will be thus $\mathcal{O}\left(n^{0.75} \times \left(\frac{n^{2.25}}{B\sqrt{M}} + n^{1.25} + \frac{n^{1.75}}{B}\right)\right) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + n^2 + \frac{n^{2.5}}{B}\right)$. The I/O complexity of the entire algorithm is, therefore, $\mathcal{O}\left(\frac{n^{2.25}\log n}{B} + \frac{n^3}{B\sqrt{M}} + n^2 + \frac{n^{2.5}}{B}\right) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$, provided $n = \Omega(M)$ and $M = \Omega\left(B^2\right)$.

The correctness of the transformation described above follows from the correctness of the parallel algorithm for the parenthesis problem given in [57], and from the observation that interleaved execution of steps 2.1 and 2.2 for different strips as above (from the largest to the smallest strip) does not affect the correctness of the algorithm. We observe that step 2.1 correctly computes L^* values for every strip since it uses only the g values for the first strip which are computed in step 1 and thus always available. Step 2.2 correctly computes g values for the current strip since it uses only g values of larger strips and L^* values of the current strip, and the order in which we execute steps 2.1 and 2.2 ensures that these values are already computed.

6.8 Conclusion

We have presented a cache-oblivious framework for problems that can be solved using a construct similar to the computation in Gaussian elimination without pivoting (i.e., using a GE-type construct). We have proved that this framework can be used to obtain efficient in-place cache-oblivious algorithms for several important classes of practical problems. We have also shown that if we are allowed to use only $n^2 + n$ extra space, where n^2 is the size of the input matrix, we can obtain an efficient cache-oblivious algorithm for any problem that can be solved using a GE-type construct. In addition to the practical problems solvable using this framework, it also has the potential of being used by optimizing compilers for loop transformation [91].

However, many important open questions still exist. For example:

1. Can we extend cache-oblivious I-GEP to solve function G in Figure 6.1 in its

$\mathrm{G}_{ijk}(c, 1, n)$

(The input $c[1 \ldots n, 1 \ldots n]$ is an $n \times n$ matrix. Function $f(\cdot, \cdot, \cdot, \cdot)$ is a problem-specific function, and $\Sigma_{G_{ijk}}$ is a problem-specific set of updates to be applied on c.)

1. ***for*** $i \leftarrow 1$ ***to*** n ***do***
2. ***for*** $j \leftarrow 1$ ***to*** n ***do***
3. ***for*** $k \leftarrow 1$ ***to*** n ***do***
4. ***if*** $\langle i, j, k \rangle \in \Sigma_{G_{ijk}}$ ***then*** $c[i, j] \leftarrow f(c[i, j], c[i, k], c[k, j], c[k, k])$

$\mathrm{G}_{ikj}(c, 1, n)$

(The input $c[1 \ldots n, 1 \ldots n]$ is an $n \times n$ matrix. Function $f(\cdot, \cdot, \cdot, \cdot)$ is a problem-specific function, and $\Sigma_{G_{ikj}}$ is a problem-specific set of updates to be applied on c.)

1. ***for*** $i \leftarrow 1$ ***to*** n ***do***
2. ***for*** $k \leftarrow 1$ ***to*** n ***do***
3. ***for*** $j \leftarrow 1$ ***to*** n ***do***
4. ***if*** $\langle i, j, k \rangle \in \Sigma_{G_{ikj}}$ ***then*** $c[i, j] \leftarrow f(c[i, j], c[i, k], c[k, j], c[k, k])$

Figure 6.17: Two simple variants of GEP (Figure 6.1) obtained by rearranging the ***for*** loops.

full generality without using any extra space, or at least using $o\left(n^2\right)$ space?

2. Can we obtain general cache-oblivious frameworks for other variants of G (for example, for those shown in Figure 6.17)?

3. Are there simpler transformations of 'simple DP' (or the parenthesis problem) and the gap problem to GEP?

Chapter 7

Experimental Results: Gaussian Elimination Paradigm

You have to do the best with what God gave you.
(Mrs. Gump)

In this chapter we present extensive experimental results for both in-core and out-of-core performance of several algorithms derived from our cache-oblivious Gaussian Elimination Paradigm (GEP) introduced in Chapter 6. We consider three major applications of GEP, namely, square matrix multiplication, Gaussian elimination without pivoting and Floyd-Warshall's all-pairs shortest paths. We implement both sequential and parallel versions of our algorithms, and compare them with finely-tuned cache-aware BLAS code for matrix multiplication and Gaussian elimination without pivoting. Our results indicate that cache-oblivious GEP offers an attractive trade-off between efficiency and portability.

7.1 Introduction

In Chapter 6 we introduced a framework for efficient cache-oblivious execution of a class problems solvable using a construct similar to the computation in Gaussian elimination without pivoting. We denote this class as the Gaussian Elimination Paradigm (GEP). We presented an in-place cache-oblivious algorithm called I-GEP which solves several important special cases of GEP including Gaussian elimination and LU-decomposition without pivoting, Floyd-Warshall's all-pairs shortest paths and square matrix multiplication. We also presented C-GEP, which has the same time and I/O bounds as I-GEP, but unlike I-GEP can solve any instance of GEP. However, C-GEP uses a modest amount of extra space. We described and analyzed both the sequential and parallel implementations of I-GEP and C-GEP.

In this chapter we present empirical results showing that both I-GEP and

C-GEP significantly outperform traditional iterative implementations of GEP especially in out-of-core computations, although improvements in computation time are already realized during in-core computations. We also include some experimental results on our `pthreads` implementation of parallel I-GEP. Finally, we compare performance of I-GEP with that of highly optimized cache-aware BLAS routines for square matrix multiplication and Gaussian elimination without pivoting. Our results show that our implementation of I-GEP runs moderately slower than native BLAS; however, I-GEP performs fewer number of cache misses, is much simpler to code, easily supports `pthreads` and is portable across machines.

7.1.1 Organization of the Chapter

We describe our experimental setup in Section 7.2. In Section 7.3 we present all of our experimental results: in Section 7.3.1 we present results comparing C-GEP, I-GEP and GEP for Floyd-Warshall, in Section 7.3.2 results comparing I-GEP to BLAS routines, and in Section 7.3.3 experimental results on parallel I-GEP using `pthreads`. Finally, we include some concluding remarks on our findings in Section 6.8.

7.2 Experimental Setup

We ran our experiments on the three architectures listed in Table 7.1. Each machine can perform at most two double precision floating point operations per clock cycle. The *peak performance* of each machine is thus measured in terms of *GFLOPS* (or Giga FLoating point Operations Per Second) which is two times the clock speed of the machine in GHz.

Model	Processors	Speed	Peak GFLOPS (per proc)	L1 Cache	L2 Cache	RAM
Intel P4 Xeon	2	3.06 GHz	6.12	8 KB (4-way)	512 KB (8-way)	4 GB
AMD Opteron 250	2	2.4 GHz	4.8	64 KB (2-way)	1 MB (8-way)	4 GB
AMD Opteron 850	8 (4 dual-core)	2.2 GHz	4.4	64 KB (2-way)	1 MB (8-way)	32 GB

Table 7.1: Machines used for experiments. All block sizes (B) are 64 bytes.

The Intel P4 Xeon machine is also equipped with a 73.5 GB Fujitsu MAP3735NC hard disk (10K RPM, 4.5 ms avg. seek time, 64.1 to 107.86 MB/s data transfer rate) [2]. Our out-of-core experiments were run on this machine. All machines run Ubuntu Linux 5.10 "Breezy Badger". Each machine was exclusively used for experiments.

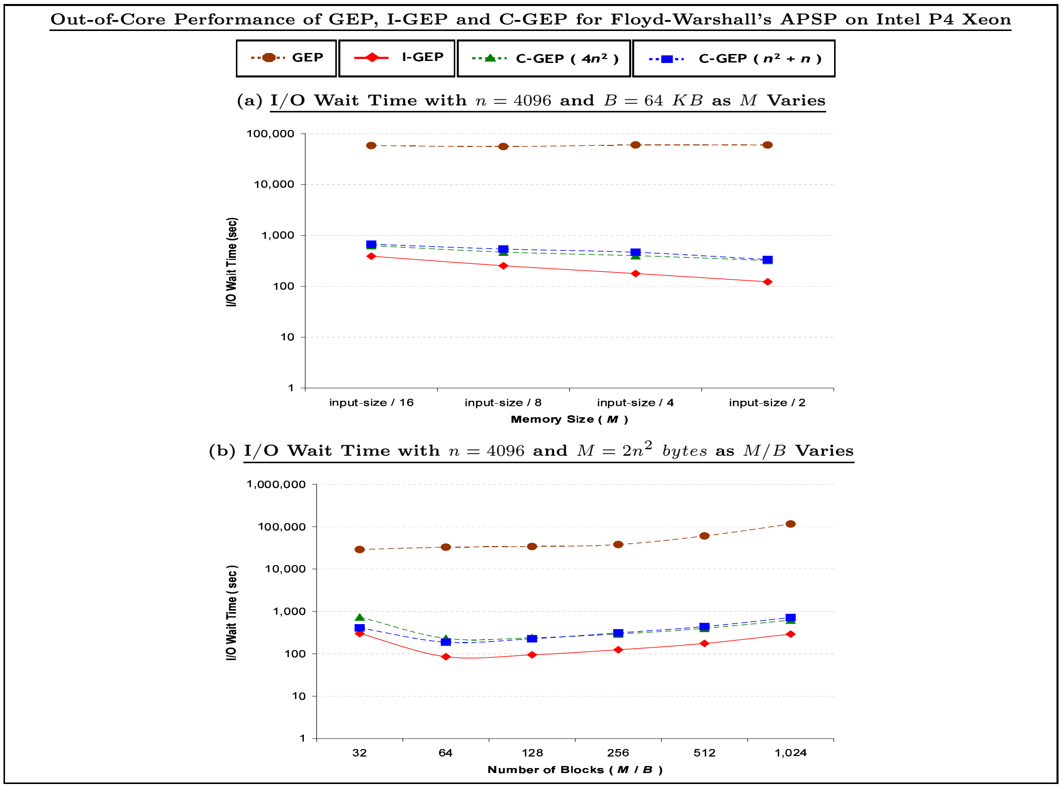

Figure 7.1: Comparison of out-of-core performance of GEP, I-GEP and C-GEP on Intel Pentium 4 Xeon with a fast hard disk (10K RPM, 4.5 ms avg. seek time, 64 to 107 MB/s transfer rate).

We used the *Cachegrind* profiler [112] for simulating cache effects. For in-core computations all algorithms were implemented in C using a uniform programming style and compiled using *gcc* 3.3.4 with optimization parameter -O3 and limited loop unrolling.

7.3 Experimental Results

We summarize our results below.

7.3.1 GEP, I-GEP and C-GEP for APSP

In this section we present experimental results comparing GEP, I-GEP and C-GEP implementations of Floyd-Warshall's APSP algorithm [48, 128] for both *in-core* and *out-of-core* computations.

Out-of-Core Computation. For out-of-core computations we implemented GEP,

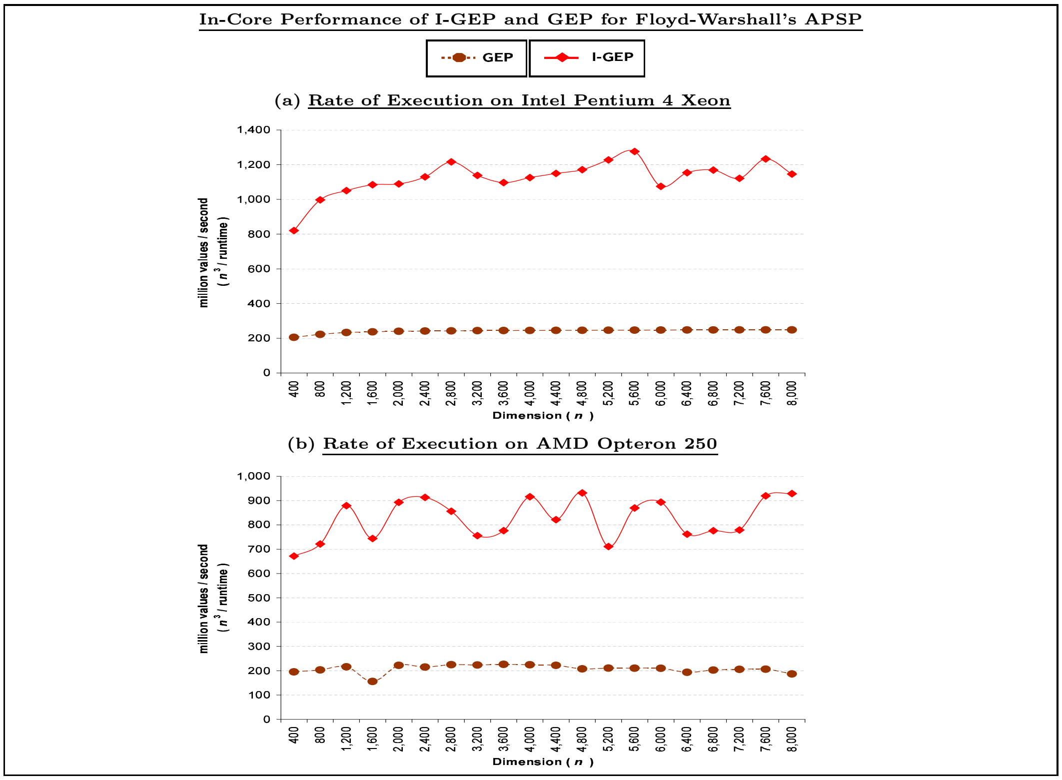

Figure 7.2: Comparison of I-GEP and GEP on Intel Xeon and AMD Opteron for computing Floyd-Warshall's all-pairs shortest paths. Both machines have two processors, but only one was used.

I-GEP and C-GEP in C++, and compiled using g++ 3.3.4 compiler with optimization level -O3 and STXXL software library version 0.9 [41] for external memory accesses. The STXXL library maintains its own fully associative cache in RAM with pages from the disk, and allows users set the size of the cache (M) and the block transfer size (B) manually. We compiled STXXL with DIRECT-I/O turned on so that the OS does not cache data from hard disk.

When the computation is out-of-core I/O wait times dominate computation times. In Figure 7.1(a) we keep n and B fixed and vary M. We observe that M has almost no effect on the I/O wait time of GEP while that of both I-GEP and C-GEP decrease as M increases. This result is consistent with theoretical predictions since the cache complexity of GEP is independent of M and that of I-GEP and C-GEP vary inversely with $\sqrt{M}$. In general, the I/O wait time of GEP is several hundred times more than that of I-GEP and C-GEP; for example, when only half of the input matrix fits in internal memory GEP waits 500 times more than I-GEP, and almost 180 times more than both variants of C-GEP. In Figure 7.1(b) we keep n

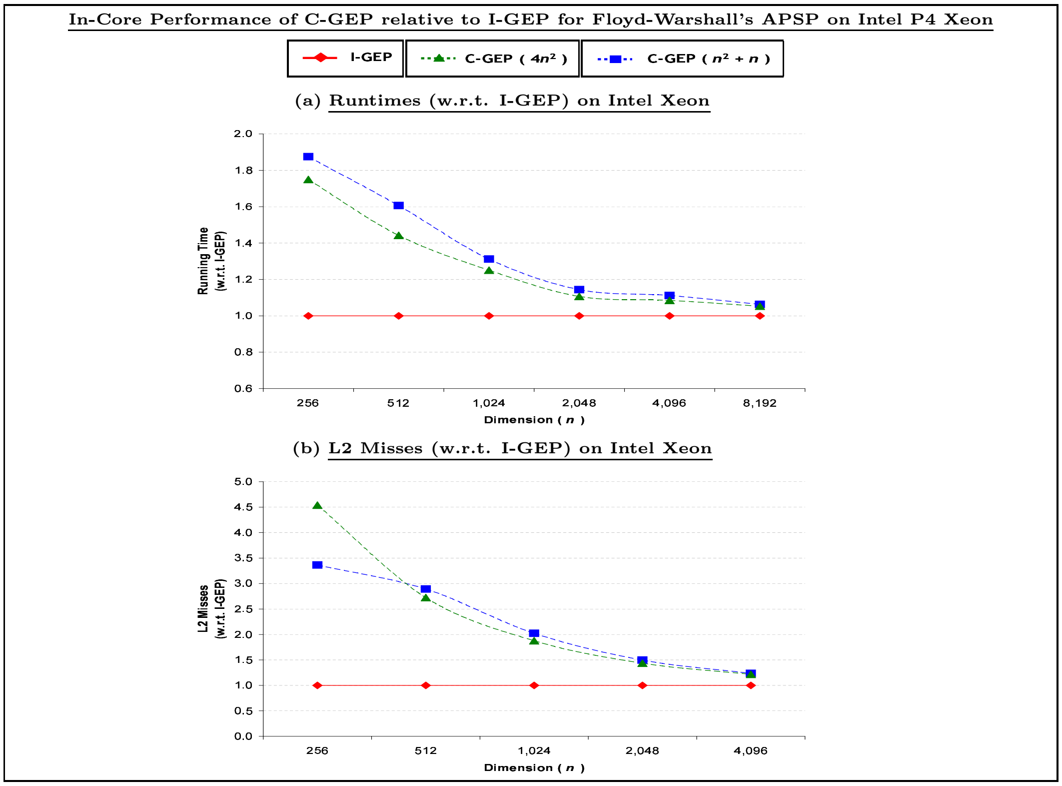

Figure 7.3: Comparison of in-core performance of I-GEP and C-GEP on Intel Pentium 4 Xeon.

and M fixed, and vary M/B (by varying B), and observe that in general, I/O wait times increase linearly with the increase of M/B. The theoretical I/O complexities of all these algorithms vary inversely with B, which explains the observed trend. However, when M/B is small, the number of page faults increases which affects the cache performance of all algorithms.

In-Core Computation. In Figure 7.2 we plot the performance of GEP and I-GEP on both Intel Xeon and AMD Opteron 250. We optimized I-GEP as described in Section 7.3.2. On Intel Xeon I-GEP runs around 5 times faster than GEP while on AMD Opteron it runs around 4 times faster.

In Figure 7.3 we plot the relative performance of I-GEP and C-GEP on Intel Xeon. As expected, both versions of C-GEP run slower and incur more L2 misses than I-GEP, since they perform more write operations. However, this overhead diminishes as n becomes larger. The $(n^2 + n)$-space variant of C-GEP performs slightly worse than the $4n^2$-space variant which we believe is due to the fact that the $(n^2 + n)$-space C-GEP needs to perform more initializations and reinitializations of the temporary matrices (i.e., u_0, u_1, v_0 and v_1) compared to the $4n^2$-space C-GEP.

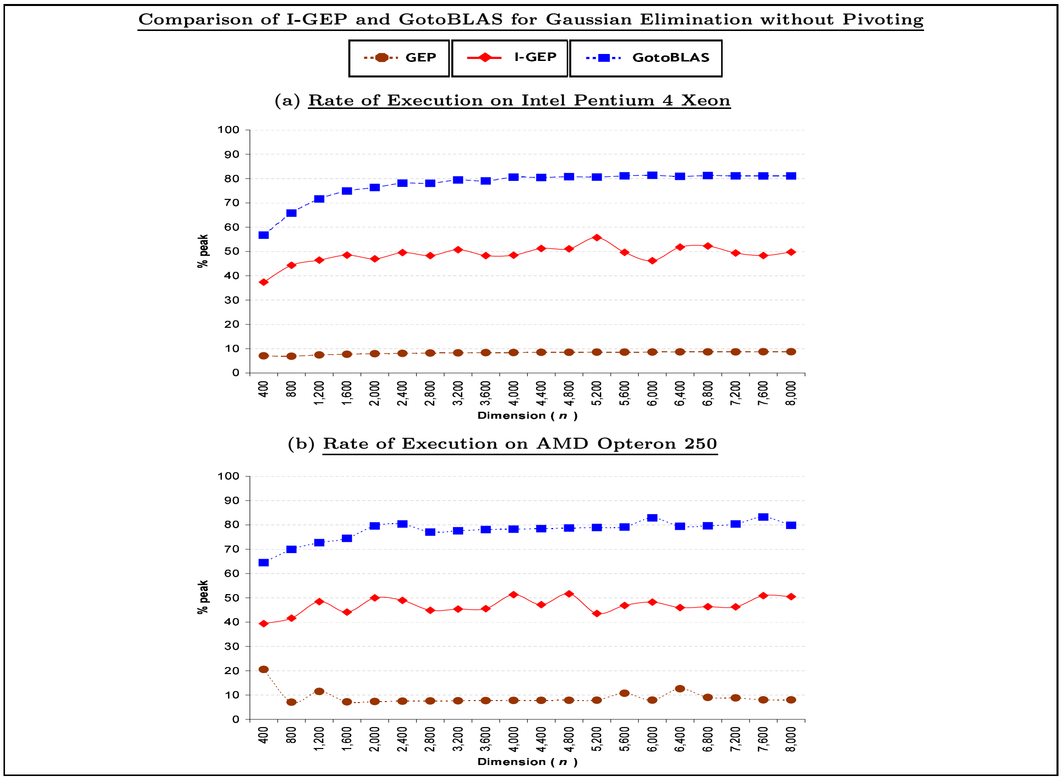

Figure 7.4: Comparison of I-GEP and GotoBLAS on Intel Xeon and AMD Opteron for performing Gaussian elimination without pivoting. Both machines have two processors, but only one was used.

7.3.2 Comparison of I-GEP and BLAS Routines

We compared the performance of our I-GEP code for square matrix multiplication and Gaussian elimination without pivoting on double precision floats with algorithms based on highly fine-tuned *Basic Linear Algebra Subprograms* (BLAS). We applied the following major optimizations on our basic I-GEP routines before the comparison:

– In order to reduce the overhead of recursion we solve the problem directly using a GEP-like iterative kernel as the input submatrix X received by the recursive functions becomes very small. We call the size of X at which we switch to the iterative kernel the *base-size*. On each machine the best value of *base-size*, i.e., for which the implementation ran the fastest, was determined empirically. On Intel Xeon it is 128×128 and on AMD Opteron it is 64×64.

– We use SSE2 ("Streaming SIMD Extension 2") instructions for increased throughput.

– For Gaussian elimination without pivoting we use a standard technique for

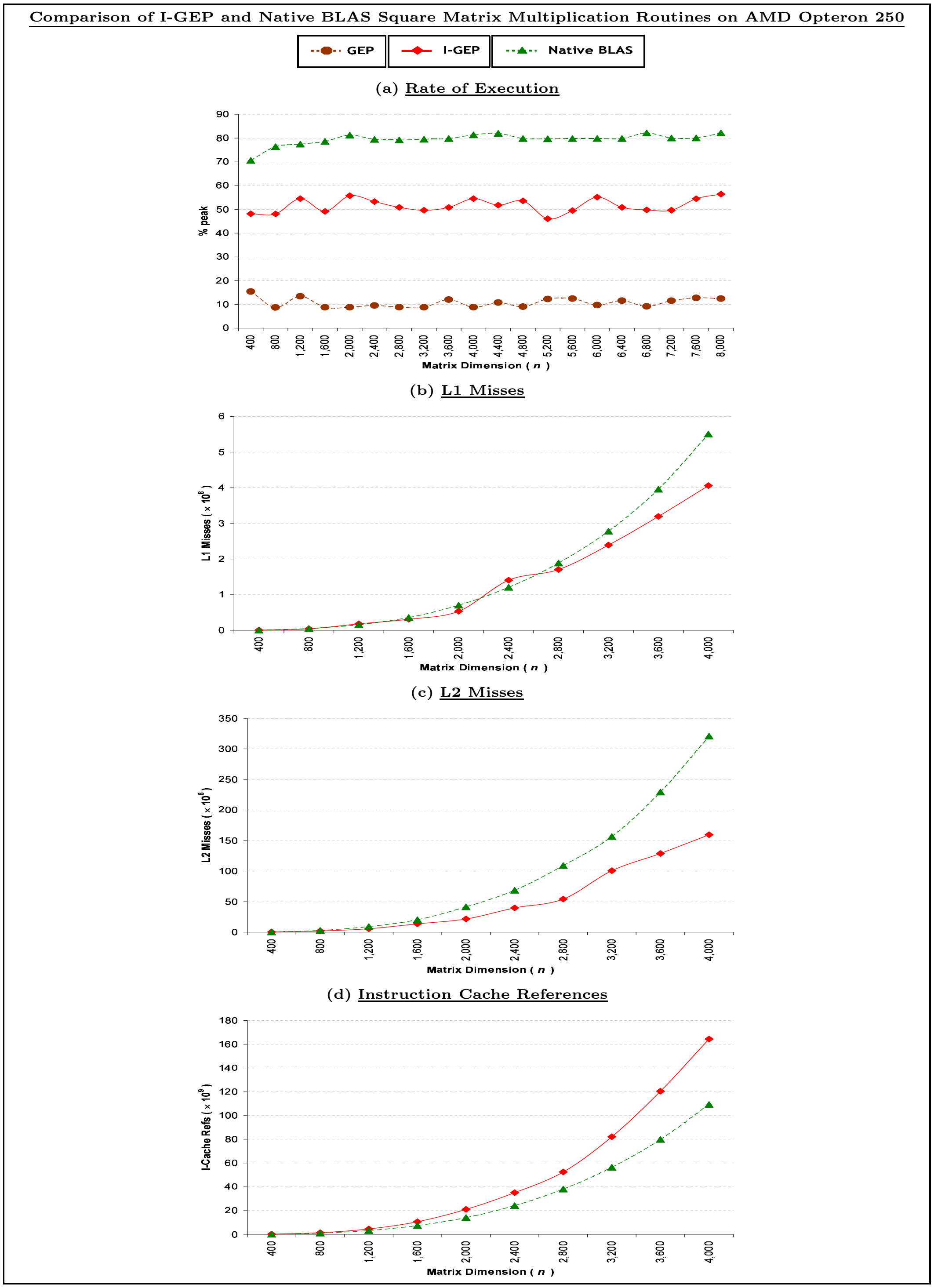

Figure 7.5: Comparison of I-GEP and native BLAS square matrix multiplication routines on a 2.4 GHz dual processor AMD Opteron 250 (one processor was used).

reducing the number of division operations to $o\left(n^3\right)$ (i.e., by moving the division operations out of the innermost loops).

– We use the *bit-interleaved* layout (e.g., see [52, 29]) for reduced TLB misses. More specifically, we arrange the base case size blocks in the bit-interleaved layout with data within the blocks arranged in row-major layout. We include the cost of converting to and from this format in the time bounds.

In Figure 7.5 we show the performance of square matrix multiplication on AMD Opteron 250 with GEP (an optimized version), I-GEP and *Native BLAS*, i.e., BLAS generated for the native machine using the automated empirical optimization tool ATLAS [102]. We report the results in '% peak', e.g., an algorithm executing at 'x% of peak' spends x% of its execution time performing floating point operations while remaining time is spent in other overheads including recursion, loops, cache misses, etc. From the plots in Figure 7.5 we observe:

– Native BLAS executes at 78–83% of peak while I-GEP executes at 50–56% of peak. Traditional GEP reaches only 9–13% of peak. The GotoBLAS [60] which is usually the fastest BLAS available for most machines (not shown in the plots) runs at 85–88% of peak.

– I-GEP incurs fewer L1 and L2 misses than native BLAS.

– I-GEP executes more instructions than native BLAS.

We obtained similar results on Intel P4 Xeon.

In Figure 7.4 we plot the performance of Gaussian elimination without pivoting using GEP, I-GEP and GotoBLAS [60] on both Intel Xeon and AMD Opteron 250. (Recall that GotoBLAS is the fastest BLAS available for most machines.) We used the LU decomposition (without pivoting) routine available in FLAME [66] to implement Gaussian elimination without pivoting using GotoBLAS. On both machines GotoBLAS executes at around 75–83% of peak while I-GEP runs at around 45–55% of peak. Traditional GEP reaches only 7–9% of peak.

Recursive square matrix multiplication using an iterative base case similar to our implementations is studied in [135]. The experimental results in [135] report performance level of only about 35% of peak for Intel P4 Xeon which is significantly lower than what we obtain for the same machine (50–58%). We conjecture that our improved performance is partly due to our use of SSE2 instructions, especially since [135] obtained performance levels of 60–75% for SUN UltraSPARC IIIi, IBM Power 5 and Intel Itanium 2 using FMA instructions. These latter results nicely complement our results for Intel P4 Xeon and AMD Opteron and further suggest that reasonable performance levels can be reached for square matrix multiplication on different architectures using relatively simple code that does not directly depend on cache parameters.

Both our implementations and the ones in [135] experimentally determined the best base-size since the overhead of recursion becomes excessive if the recursion extends all the way down to size 1. In [135] this is viewed as not being purely cache-oblivious; however we consider the fine-tuning of the base-size in recursive algorithms to be a standard optimization during implementation.

7.3.3 Multithreaded I-GEP

We implemented multithreaded I-GEP using the standard `pthreads` library. We varied the number of concurrent threads from 1 to 8 on an 8-processor AMD Opteron 850 and used I-GEP to perform matrix multiplication, Gaussian elimination without pivoting and Floyd-Warshall's APSP on input 5000×5000 matrices. Threads were created recursively as the computation progressed. We used the following simple scheduling policy. Whenever a thread created a child thread and the number of concurrent threads were still below the limit, we let the default scheduling policy on Linux assign the thread to a processor. Otherwise we suspended the parent (current) thread, ran the child thread to completion on the current processor and then resumed executing the parent thread on the same processor.

In Figure 7.6 we plot the speed-up factors achieved by multithreaded I-GEP over its unthreaded version as the number of concurrent threads is increased. For square matrix multiplication I-GEP speeds up by a factor of 6 when the number of concurrent threads increases from 1 to 8, while for Gaussian elimination without pivoting and Floyd-Warshall's APSP the speed-up factors are smaller, i.e., 5.33 and 5.73, respectively. As mentioned in Section 6.5, I-GEP for matrix multiplication has more parallelism than I-GEP for Gaussian elimination without pivoting and Floyd-Warshall's APSP, which could explain the better speed-up factor for matrix multiplication.

7.4 Discussion

We draw the following conclusions from our results:

– In our experiments I-GEP always outperformed both variants of C-GEP (see Section 7.3.1). The $4n^2$-space variant of C-GEP almost always outperformed the $(n^2 + n)$-space variant, and it is also easier to implement. Therefore, if disk space is not at a premium, the $4n^2$-space C-GEP should be used instead of the $(n^2 + n)$-space variant, and I-GEP is preferable to both variants of C-GEP whenever applicable.

– When the computation is in-core, I-GEP runs about 5–6 times faster than even some reasonably optimized versions of GEP. It has been reported in [94] that I-GEP runs slightly slower than GEP on Intel P4 Xeon for Floyd-Warshall's APSP when the prefetchers are turned on. We believe that we get dramatically better results for I-GEP in part because unlike [94] we arrange the entries of each base-case submatrix in a prefetcher-friendly layout, i.e., in row-major order (see Section

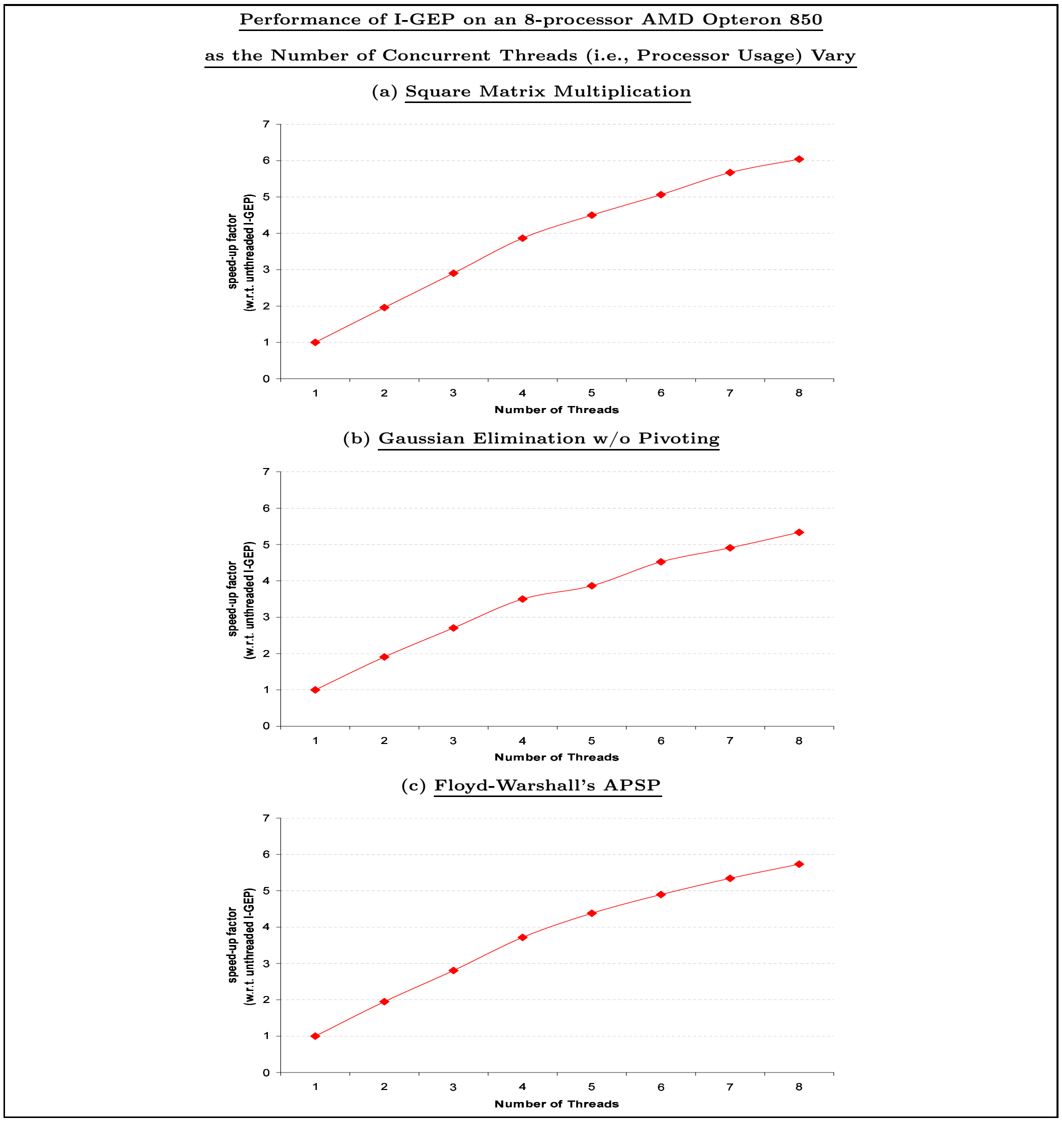

Figure 7.6: Performance of I-GEP on an 8-processor AMD Opteron 850 for square matrix multiplication, Gaussian elimination w/o pivoting and Floyd-Warshall's all-pairs shortest paths on 5000×5000 matrices as number of concurrent threads is varied.

7.3.2). Note that we include the cost of converting to and from this layout in the time bounds we report.

– BLAS routines run about 1.5 times faster than I-GEP. However, I-GEP is cache-oblivious and is implemented in a high level language, while BLAS routines are cache-aware and employ numerous low-level machine-specific optimizations in assembly language. The cache-miss results in Section 7.3.2 indicate that the cache performance of I-GEP is at least as good as that of native BLAS. Hence the performance gain of native BLAS over I-GEP is most likely due to optimizations other than cache-optimizations.

– Our I-GEP/C-GEP code for in-core computations can be used for out-of-core computations without any changes, while BLAS is optimized for in-core computations only.

– I-GEP/C-GEP can be parallelized very easily, and speeds up reasonably well as the number of processors (i.e., concurrent threads) increases. However, current systems offer very limited flexibility in scheduling tasks to processors, and we believe that performance of multithreaded I-GEP can be improved further if better scheduling policies are used.

Chapter 8

Cache-oblivious Dynamic Programs for Bioinformatics

I've got the world on a string, sittin' on a rainbow...
(Ted Koehler & Harold Arlen,
Recorded by Frank Sinatra)

In this chapter we present efficient cache-oblivious sequential and parallel algorithms for some well-studied string problems in bioinformatics:

1. *LCS problem.* The longest common subsequence between two given sequences;
2. *Pairwise alignment.* Optimal pairwise global sequence alignment using affine gap penalty;
3. *Median.* Optimal alignment of three sequences using affine gap penalty;
4. *RNA secondary structure prediction.* Maximizing number of base pairs in RNA secondary structure with simple pseudoknots and also without pseudoknots.
5. *Gap problem.* Optimal pairwise alignment with general gap costs. We also consider the *least weight subsequence problem* which can be viewed as a simpler version of the gap problem.

For each of these problems we present sequential cache-oblivious algorithms that match the best-known time complexity, match or improve the best-known space complexity, and improve significantly over the cache-efficiency of earlier algorithms. We also show that these algorithms are easily parallelizable, and we analyze their parallel performance.

Our methods are applicable to several other dynamic programs for string problems in bioinformatics including local alignment, generalized global alignment with intermittent similarities, multiple sequence alignment under several scoring functions such as 'sum-of-pairs' objective function and RNA secondary structure prediction with simple pseudoknots using energy functions based on adjacent base pairs.

In Chapter 9 we present experimental results on several of our cache-oblivious algorithms mentioned above.

8.1 Introduction

Algorithms for sequence alignment and for RNA secondary structure prediction are some of the most widely studied and widely-used methods in bioinformatics. Many of these are dynamic programming algorithms that run in polynomial time under the traditional *von Neumann Model* of computation which assumes a single layer of memory with uniform access cost, and many have been further improved in their space usage, mainly using a technique due to Hirschberg [70]. However, most of these algorithms are deficient with respect to cache-efficiency, and thus do not deliver the best performance on modern computers with multi-level memory hierarchy.

8.1.1 Our Results

In this chapter we consider two particular classes of dynamic programming problems: one with 'local dependencies', and the other with 'non-local dependencies'. In a dynamic program with local dependencies the value of each cell in the DP table depends only on values in adjacent cells, while in a dynamic program with non-local dependencies this property is violated for some or all cells in the DP table.

Dynamic Programming with Local Dependencies. We first present an efficient cache-oblivious framework for both sequential and parallel machines which solves a general class of recurrence relations in 2- and 3-dimensions that are amenable to solution by dynamic programs with local dependencies. In principle our framework can be generalized to any number of dimensions, although we study explicitly only the 2- and 3-dimensional cases. We use this framework to develop cache-oblivious algorithms for several well-known string problems in bioinformatics, and show that our algorithms are theoretically more cache-efficient than previous algorithms for these problems. We also show that our parallel cache-oblivious algorithms can be scheduled on both distributed and shared caches to achieve almost the same level of cache-efficiency as their sequential counterparts. The string problems we consider are:

- *LCS* (i.e., longest common subsequence): Given two sequences of length n each our cache-oblivious algorithm finds the longest common subsequence in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space and $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache-misses. On a machine with p processors a parallel implementation of this algorithm performs $\mathcal{O}\left(n^2\right)$ work and terminates in $\mathcal{O}\left(\frac{n^2}{p}+n\right)$ parallel steps.
- *Global pairwise alignment* with *affine* gap costs: On a pair of sequences of length n each our cache-oblivious algorithm runs in $\mathcal{O}\left(n^2\right)$ time, uses $\mathcal{O}\left(n\right)$ space and incurs $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache-misses. When executed with p processors a paral-

lel implementation of this algorithm performs $\mathcal{O}\left(n^2\right)$ work and terminates in $\mathcal{O}\left(\frac{n^2}{p}+n\right)$ parallel steps.

- *Median* (i.e., optimal alignment of three sequences) with *affine* gap costs: Our cache-oblivious algorithm runs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space, and incurs only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses on three sequences of length n each. On a machine with p processors a parallel version of our algorithm performs $\mathcal{O}\left(n^3\right)$ work and executes $\mathcal{O}\left(\frac{n^3}{p}+n\right)$ parallel steps.

- *RNA secondary structure prediction with simple pseudoknots*: On an RNA sequence of length n, our cache-oblivious algorithm runs in $\mathcal{O}\left(n^4\right)$ time, uses $\mathcal{O}\left(n^2\right)$ space and incurs $\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$ cache-misses, and parallel implementation of our algorithm performs $\mathcal{O}\left(n^4\right)$ work and terminates in $\mathcal{O}\left(\frac{n^4}{p}+n\log^2 n\right)$ parallel steps when executed with p processors.

Our cache-oblivious framework can also be used to obtain efficient cache-oblivious algorithms for several other string problems in bioinformatics including local alignment, generalized global alignment with intermittent similarities, multiple sequence alignment under several scoring functions such as 'sum-of-pairs' objective function and RNA secondary structure prediction with simple pseudoknots using energy functions based on adjacent base pairs.

Two features of our cache-oblivious framework are worth noting.

- Our cache-oblivious algorithms improve on the space usage of traditional dynamic programs for each of the problems we study, and match the space usage of the Hirschberg-style space-reduced versions [70] of these traditional dynamic programs. However, our space reduction is obtained through a divide-and-conquer strategy that is quite different from the method used in [70]. Briefly, our method computes the dynamic program table recursively in sub-blocks and stores only the computed values at the *boundary* of the sub-block. This results in the space saving, and we show that the stored values suffice to compute a solution with optimal value.

- Our algorithms are simpler than the space-reduced versions of the traditional dynamic programs, and hence are easier to code. Further, the recursive structure of our method gives rise to a good amount of parallelizism, which is also very easy to expose using standard parallel constructs such as `fork` and `join`.

Dynamic Programming with Non-local Dependencies. Among dynamic programs with non-local dependencies, we consider dynamic programming algorithms for the following problems:

- *Gap problem* (i.e., pairwise sequence alignment with *general* gap costs): On a pair of sequences of length n each, our cache-oblivious algorithm runs in $\mathcal{O}\left(n^3\right)$ time, uses $\mathcal{O}\left(n^2\right)$ space and incurs $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses. A parallel implementation of our algorithm performs $\mathcal{O}\left(n^3\right)$ work and terminates in $\mathcal{O}\left(\frac{n^3}{p} + n^{\log_2 3}\right)$ parallel steps on a machine with p processors. We present scheduling schemes for cache-efficient execution of the parallel algorithm on both distributed and shared caches.

- *Basic RNA secondary structure prediction problem* (i.e., w/o pseudoknots): We show that given an RNA sequence of length n, an RNA secondary structure (w/o pseudoknots) with the maximum number of base pairs can be found cache-obliviously within the same sequential time, space and cache-miss bounds as the gap problem described above.

Remarks. We note that often in practice biologists seek not a precise optimal solution but biologically significant solutions that may be sub-optimal under the optimization measure used to define the problem. However, often in such cases, an algorithm for the precise optimal solution can be used as a subroutine in conjunction with other methods that determine biological features not captured by the combinatorial problem specification. Therefore, our algorithms are likely to be of use to biologists even when biologically significant solutions are sought that are not necessarily optimal under our definition.

8.1.2 Organization of the Chapter

In Section 8.2 we describe our results on the cache-oblivious framework that we will use to solve the dynamic programming problems with local dependencies we consider in this chapter. In Section 8.2.1 we describe and analyze the 3-dimensional version of the framework, and in Section 8.2.2 we establish its I/O lower bound for arbitrary dimensions. In Section 8.2.3 we describe and analyze the parallel implementation of the framework for the 3-dimensional case, and present scheduling strategies for its cache-efficient execution on both shared and distributed caches. In Section 8.2.4 we describe how to use the cache-oblivious framework to obtain cache-oblivious algorithms for longest common subsequence, global pairwise sequence alignment, median and RNA secondary structure prediction with simple pseudoknots.

In Section 8.3 we consider two dynamic programming problems with non-local dependencies. In Section 8.3.1 we describe and analyze sequential and parallel cache-oblivious algorithms for solving the gap problem. In Section 8.3.2 we describe how to solve the basic RNA secondary structure (w/o pseudoknots) prediction problem cache-obliviously.

Preliminary versions of some of the results in this chapter, particularly the LCS result in Section 8.2.4, the I/O lower bound in Section 8.2.2 and the sequential algorithm for the gap problem in Section 8.3.1, appeared in a conference paper [34].

8.2 Cache-oblivious Dynamic Programs with Local Dependencies

Before providing a general recurrence defining the class of dynamic programs with local dependencies we will consider in this chapter, let us look at a simple motivating example.

A Motivating Example - The LCS DP. A sequence $Z = z_1 z_2 \ldots z_k$ is called a *subsequence* of another sequence $X = x_1 x_2 \ldots x_n$ if there exists a strictly increasing function $f : [1, 2, \ldots, k] \rightarrow [1, 2, \ldots, n]$ such that for all $i \in [1, k]$, $z_i = x_{f(i)}$. In the *Longest Common Subsequence* (LCS) problem we are given two input sequences, and we need to find a maximum-length subsequence common to both sequences.

Given two sequences $S_1 = s_{1,1} s_{1,2} \ldots s_{1,n}$ and $S_2 = s_{2,1} s_{2,2} \ldots s_{2,n}$ (for simplicity, we assume equal-length sequences here), we define $c[i_1, i_2]$ ($0 \leq i_1, i_2 \leq n$) to be the length of an LCS of $s_{1,1} s_{1,2} \ldots s_{1,i_1}$ and $s_{2,1} s_{2,2} \ldots s_{2,i_2}$. Then $c[n, n]$ is the length of an LCS of S_1 and S_2, and can be computed using the following recurrence relation (see, e.g., [37]):

$$c[i_1, i_2] = \begin{cases} 0 & \text{if } i_1 = 0 \text{ or } i_2 = 0, \\ c[i_1 - 1, i_2 - 1] + 1 & \text{if } i_1, i_2 > 0 \wedge s_{1,i_1} = s_{2,i_2}, \\ \max\{\ c[i_1, i_2 - 1],\ c[i_1 - 1, i_2]\ \} & \text{if } i_1, i_2 > 0 \wedge s_{1,i_1} \neq s_{2,i_2}. \end{cases} \tag{8.2.1}$$

We can rewrite the recurrence above in the following form:

$$c[i_1, i_2] = \begin{cases} h\left(\ \langle\ i_1,\ i_2\ \rangle\ \right) & \text{if } i_1 = 0 \text{ or } i_2 = 0, \\ f\left(\begin{array}{c} \langle\ i_1,\ i_2\ \rangle,\ \ \langle\ s_{1,i_1},\ s_{2,i_2}\ \rangle, \\ c[\ i_1 - 1 : i_1,\ i_2 - 1 : i_2\ ] \setminus c[i_1, i_2] \end{array} \right) & \text{otherwise.} \end{cases} \tag{8.2.2}$$

where $h(\cdot)$ is an initialization function that always returns 0, and $f(\cdot, \cdot, \cdot)$ is the function that computes the value of each cell based on the values in adjacent cells as follows.

$$f\left(\begin{array}{c} \langle\ i_1,\ i_2\ \rangle,\ \ \langle\ s_{1,i_1},\ s_{2,i_2}\ \rangle, \\ c[i_1 - 1 : i_1, i_2 - 1 : i_2] \setminus c[i_1, i_2] \end{array} \right) = \begin{cases} c[i_1 - 1, i_2 - 1] + 1 & \text{if } s_{1,i_1} = s_{2,i_2}, \\ \max \left\{ \begin{array}{c} c[i_1, i_2 - 1], \\ c[i_1 - 1, i_2] \end{array} \right\} & \text{otherwise.} \end{cases}$$

All computations above are performed in the domain of nonnegative integers (i.e., the set $\mathbb{N}$ of natural numbers).

Let us now extend recurrence 8.2.2 to arbitrary number of sequences and arbitrary functions h and f.

A General Framework for DPs with Local Dependencies. Suppose we are given the following.

- $d \geq 1$ sequences $S_i = s_{i,1}s_{i,2}\ldots s_{i,n}$, $1 \leq i \leq d$, of length n each, with symbols chosen from an arbitrary finite alphabet. We define the following (to be used later).
  - Given integers $i_j \in [0, n]$, $j \in [1, d]$, we denote by $\mathbf{i}$ the sequence of d integers $i_1, i_2, \ldots, i_d$; and by $\langle\ \mathbf{i}\ \rangle$ we denote the d-dimensional vector $\langle\ i_1, i_2, \ldots, i_d\ \rangle$.
  - By $\langle\ \mathbf{S_i}\ \rangle$ we denote the d-dimensional vector $\langle\ s_{1,i_1},\ s_{2,i_2}, \ldots,\ s_{d,i_d}\ \rangle$ containing the i_j-th symbol of S_j in j-th position, where each $i_j \in [1, n]$.
- An arbitrary set $\mathcal{U}$.
- An initialization function $h(\cdot)$ that accepts a vector $\langle\ \mathbf{i}\ \rangle$ as input and outputs an element from $\mathcal{U}$.
- A function $f(\cdot, \cdot, \cdot)$ that accepts vectors $\langle\ \mathbf{i}\ \rangle$ and $\langle\ \mathbf{S_i}\ \rangle$, and an ordered set of $2^d - 1$ elements from Σ, and returns an element of $\mathcal{U}$.

Now suppose $c[0:n, 0:n, \ldots, 0:n]$ is a d-dimensional matrix that can store elements from the given set $\mathcal{U}$, and we want to compute the entries of c using the following dynamic programming recurrence.

$$c[\ \mathbf{i}\ ] = \begin{cases} h\left(\ \langle\ \mathbf{i}\ \rangle\ \right) & \text{if } \exists\ i_j = 0, \\ f\left(\langle\ \mathbf{i}\ \rangle,\ \langle\ \mathbf{S_i}\ \rangle,\ c[i_1 - 1 : i_1, i_2 - 1 : i_2, \ldots, i_d - 1 : i_d] \setminus c[\ \mathbf{i}\ ]\right) & \text{otherwise.} \end{cases} \tag{8.2.3}$$

Function f can be arbitrary except that it is allowed to use exactly one cell from its third argument to compute the final value of $c[i_1, i_2, \ldots, i_d]$ (though it can consider all cells), and we call that specific cell the *parent cell* of $c[i_1, i_2, \ldots, i_d]$. We also assume that f does not access any memory locations in addition to those passed to it as inputs except possibly some constant size local variables.

Typically, two types of outputs are expected when evaluating this recurrence: (i) the value of $c[n, n, \ldots, n]$, and (ii) the traceback path starting from $c[n, n, \ldots, n]$. The *traceback path* from any cell $c[i_1, i_2, \ldots, i_d]$ is the path following the chain of parent cells through c that ends at some $c[i'_1, i'_2, \ldots, i'_d]$ with $\exists\ i'_j = 0$.

Each cell of c can have multiple fields and in that case f must compute a value for each field, though as before, it is allowed to use exactly one field from its third argument to compute the final value of any field in $c[i_1, i_2, \ldots, i_d]$. The definition of traceback path extends naturally to this case, i.e., when the cells have multiple fields.

Recurrence 8.2.3 can be evaluated iteratively in $\mathcal{O}\left(n^d\right)$ time, $\mathcal{O}\left(n^d\right)$ space and $\mathcal{O}\left(n^d/B\right)$ cache-misses. Though space can be reduced to $\mathcal{O}\left(n^{d-1}\right)$ using Hirschberg's technique [70], the cache-complexity remains unchanged if the traceback path must also be computed[1].

In Section 8.2.1 we present a cache-oblivious algorithm for solving the 3-dimensional version (i.e., $d = 3$) of recurrence 8.2.3 along with a traceback path in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses. It improves over the previous best cache-miss bound by at least a factor of $\sqrt{M}$, and reduces space requirement by a factor of n when compared with the traditional iterative solution. In Section 8.2.4 we use this algorithm to solve median of three sequences and RNA secondary structure prediction with simple pseudoknots.

A cache-oblivious algorithm which is similar to the algorithm for solving the 3-dimensional recurrence but is simpler, solves the 2-dimensional version (i.e., $d = 2$) of recurrence 8.2.3. We include this algorithm in Appendix D. This algorithm runs in $\mathcal{O}\left(n^2\right)$ time, uses $\mathcal{O}\left(n\right)$ space and incurs $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache misses. It improves over the previous best cache-complexity by a factor of at least M, and also improves over the space complexity of the standard iterative DP by a factor of n. In Section 8.2.4 we use this algorithm for solving the LCS problem and for global pairwise sequence alignment with affine gap costs.

8.2.1 Cache-oblivious Algorithm for Solving Recurrence 8.2.3 in 3D

Our algorithm works by decomposing the given cube $c[\,1:n,\ 1:n,\ 1:n\,]$ into smaller subcubes, and is based on the observation that for any such subcube we can recursively compute the entries on its *output boundary* (i.e., on its right, front and bottom boundaries) provided we know the entries on its *input boundary* (i.e., entries immediately outside of its left, back and top boundaries). Since the subcubes share boundaries, when the output boundaries of all subcubes are computed the problem of finding the traceback path through the entire cube is reduced to the problem of recursively finding the fragments of the path through the subcubes. Though we compute all $\Theta\left(n^3\right)$ entries of c, at any stage of recursion we only need to save the entries on the boundaries of the subcubes and thus use only $\mathcal{O}\left(n^2\right)$ space. The divide and conquer strategy also improves locality of computation and consequently leads to an efficient cache-oblivious algorithm.

---

[1] If a traceback path is not required it is easy to reduce space requirement of the iterative algorithm to $\mathcal{O}\left(n^{d-1}\right)$ even without using Hirschberg's technique (see, e.g., [37]), and when $M = \Omega\left(B^{d-1}\right)$, its cache-complexity can be improved to $\mathcal{O}\left(\frac{n^d}{BM^{\frac{1}{d-1}}}\right)$ using the cache-oblivious stencil-computation technique [54].

Function **8.2.1.** Compute-Boundary-3D($X,\ Y,\ Z,\ L,\ B,\ T$)

Input. Same as the input description of Compute-Traceback-Path-3D (Function 8.2.2 in Figure 8.2).

Output. Returns $\langle R, F, D\rangle$, where R ($\equiv Q[r, 1 : r, 1 : r]$), F ($\equiv Q[1 : r, r, 1 : r]$) and D ($\equiv Q[1 : r, 1 : r, r]$) are the right, front and bottom boundaries of $Q[1 : r, 1 : r, 1 : r]$, respectively.

1. ***if*** $r = 1$ ***then*** $R = F = D \leftarrow f\ (\ \langle\ u,\ v,\ w\ \rangle,\ \langle\ X,\ Y,\ Z\ \rangle,\ L \cup B \cup T\ )$
2. ***else***
3. Extract $L_{1,j,k}$ from L, $B_{i,1,k}$ from B, and $T_{i,j,1}$ from T, respectively, where $i, j, k \in [1, 2]$
4. *subcube*$[1 : 8] \leftarrow \langle\langle 1, 1, 1\rangle, \langle 2, 1, 1\rangle, \langle 1, 2, 1\rangle, \langle 2, 2, 1\rangle, \langle 1, 1, 2\rangle, \langle 2, 1, 2\rangle, \langle 1, 2, 2\rangle, \langle 2, 2, 2\rangle\rangle$
5. ***for*** $l \leftarrow 1$ ***to*** 8 ***do***
6. $\langle\ i,\ j,\ k\ \rangle \leftarrow$ *subcube*$[\ l\ ]$
   $\langle\ R_{ijk}, F_{ijk},\ D_{ijk}\ \rangle \leftarrow$ Compute-Boundary-3D($X_i,\ Y_j,\ Z_k,\ L'_{ijk},\ B'_{ijk},\ T'_{ijk}$)
7. Compose R from $R_{2,j,k}$, F from $F_{i,2,k}$, and D from $D_{i,j,2}$, where $i, j, k \in [1, 2]$
8. ***return*** $\langle\ R,\ F,\ D\ \rangle$

Compute-Boundary-3D Ends

Figure 8.1: Evaluating recurrence 8.2.3 cache-obliviously for $d = 3$ without a traceback path. For simplicity, we assume $n = 2^q$ for some integer $q \geq 0$. In initial call to Function D.0.5, $X = x_1x_2 \ldots x_n$, $Y = y_1y_2 \ldots y_n$, $Z = z_1z_2 \ldots z_n$, $L \equiv c[0, 0 : n, 0 : n]$, $B \equiv c[0 : n, 0, 0 : n]$ and $T \equiv c[0 : n, 0 : n, 0]$.

As noted before, Hirschberg's technique [70] can also be used to solve recurrence 8.2.3 along with a traceback path. Unlike our algorithm, however, Hirschberg's approach decomposes the problem into two subproblems of typically unequal size, and uses a complicated process involving the application of the traditional iterative DP in both forward and backward directions to perform the decomposition. The application of the iterative DP along with the fact that the subproblems are often unequal in size contributes to its inefficient cache usage. Moreover, the use of both forward and backward DP, and particularly the need for combining the results obtained from them complicates the implementation of Hirschberg's technique for multiple simultaneous recurrences or recurrences with multiple fields. In contrast, our algorithm always applies DP in one direction and thus is simpler to implement. A method for applying Hirschberg's space-reduction using forward-only DP is given in [46], but it involves repeated linear scans and thus is not cache-efficient.

We describe below the two parts of our algorithm. The pseudocode for these parts are given in Figures 8.1 and 8.2. The first part of our algorithm (i.e., the method for computing the output boundary) has some similarity to the cache-oblivious stencil computation algorithm described in [54] which, however, uses a different decomposition scheme.

Compute-Boundary-3D. Given the input boundary of $c[\ i_1 : i_2,\ j_1 : j_2,\ k_1 :$

FUNCTION **8.2.2.** COMPUTE-TRACEBACK-PATH-3D($X,\ Y,\ Z,\ L,\ B,\ T,\ P$)

Input. Here $r = |X| = |Y| = |Z| = 2^t$ for some nonnegative integer $t \leq q$, and $Q[0:r, 0:r, 0:r] \equiv c[u-1:u+r-1, v-1:v+r-1, w-1:w+r-1]$, $X = x_u x_{u+1} \ldots x_{u+r-1}$, $Y = y_v y_{v+1} \ldots y_{v+r-1}$ and $Z = z_w z_{w+1} \ldots z_{w+r-1}$ for some u, v and w ($1 \leq u, v, w \leq n-r+1$). The left, back and top boundaries of $Q[1:r, 1:r, 1:r]$ are stored in L ($\equiv Q[0, 0:r, 0:r]$), B ($\equiv Q[0:r, 0, 0:r]$) and T ($\equiv Q[0:r, 0:r, 0]$), respectively. Current traceback path is given in P.

Output. Returns the updated traceback path.

1. ***if*** $P \cap Q = \emptyset$ ***return*** P
2. ***if*** $r = 1$ ***then*** update P using recurrence 8.2.3
3. ***else*** { *For* $i, j, k \in [1, 2]$*, the left, right, front, back, top and bottom planes of subcube* Q_{ijk} *are denoted by* L_{ijk}, R_{ijk}, F_{ijk}, B_{ijk}, T_{ijk} *and* D_{ijk}*, respectively.* X_1 *and* X_2 *denote the 1st and the 2nd half of* X*, respectively (similarly for* Y *and* Z*).*}
4. Extract $L_{1,j,k}$ from L, $B_{i,1,k}$ from B, and $T_{i,j,1}$ from T, respectively, where $i, j, k \in [1, 2]$ { $L_{2,j,k} \equiv R_{1,j,k}$, $B_{i,2,k} \equiv F_{i,1,k}$ *and* $T_{i,j,2} \equiv D_{i,j,1}$ *for* $i, j, k \in [1, 2]$ }
5. *subcube*$[1:8] \leftarrow \langle\langle 1,1,1\rangle, \langle 2,1,1\rangle, \langle 1,2,1\rangle, \langle 2,2,1\rangle, \langle 1,1,2\rangle, \langle 2,1,2\rangle, \langle 1,2,2\rangle, \langle 2,2,2\rangle\rangle$
6. ***for*** $l \leftarrow 1$ ***to*** 7 ***do*** {*forward pass (compute boundaries)*}
7. $\langle\ i,\ j,\ k\ \rangle \leftarrow$ *subcube*$[\ l\ ]$
   $\langle\ R_{ijk}, F_{ijk},\ D_{ijk}\ \rangle \leftarrow$ COMPUTE-BOUNDARY-3D($X_i,\ Y_j,\ Z_k,\ L'_{ijk},\ B'_{ijk},\ T'_{ijk}$)
   { *if* $L_{ijk} \equiv Q[\ i',\ j_1:j_2,\ k_1:k_2\ ]$ *then* $L'_{ijk} \equiv Q[\ i',\ j_1-1:j_2,\ k_1-1:k_2\ ]$*; similarly for* B_{ijk} *and* T_{ijk}*.* }
8. ***for*** $l \leftarrow 8$ ***downto*** 1 ***do*** {*backward pass (compute traceback path)*}
9. $\langle\ i,\ j,\ k\ \rangle \leftarrow$ *subcube*$[\ l\ ]$
   $P \leftarrow$ COMPUTE-TRACEBACK-PATH-3D($X_i,\ Y_j,\ Z_k,\ L'_{ijk},\ B'_{ijk},\ T'_{ijk},\ P$)
10. ***return*** P

COMPUTE-TRACEBACK-PATH-3D ENDS

Figure 8.2: Evaluating recurrence 8.2.3 cache-obliviously for $d = 3$ along with a traceback path. For simplicity, we assume $n = 2^q$ for some integer $q \geq 0$. In initial call to Function 8.2.2, $X = x_1 x_2 \ldots x_n$, $Y = y_1 y_2 \ldots y_n$, $Z = z_1 z_2 \ldots z_n$, $L \equiv c[0, 0:n, 0:n]$, $B \equiv c[0:n, 0, 0:n]$, $T \equiv c[0:n, 0:n, 0]$ and $P = \langle (n, n, n) \rangle$.

k_2] this function (Function 8.2.1) recursively computes its output boundary. For simplicity of exposition we assume that $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$.

If $q = 0$, the function can compute the output boundary directly using recurrence 8.2.3, otherwise it decomposes its cubic computation space Q (initially $Q \equiv c[1:n, 1:n, 1:n]$) into 8 subcubes $Q_{i,j,k}$, $1 \leq i, j, k \leq 2$, where $Q_{i,j,k}$ denotes the subcube that is i-th from the left, j-th from the back and k-th from the top. It then computes the output boundary of each subcube recursively as the input boundary of the subcube becomes available during the process of computation. After all recursive calls terminate, the output boundary of Q is composed from the output boundaries of the subcubes.

Analysis. Let $I_1(n)$ be the cache-complexity of Compute-Boundary-3D on input sequences of length n each. Then

$$I_1(n) = \begin{cases} \mathcal{O}\left(n + \frac{n^2}{B}\right) & \text{if } n \leq \sqrt{\alpha M}, \\ 8I_1\left(\frac{n}{2}\right) + \mathcal{O}\left(n + \frac{n^2}{B}\right) & \text{otherwise;} \end{cases}$$

where α is the largest constant sufficiently small that computation involving three input sequences of length $\sqrt{\alpha M}$ each can be performed completely inside the cache. Solving the recurrence we obtain $I_1(n) = \mathcal{O}\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ for all n. It is straight-forward to show that the algorithm runs in $\mathcal{O}\left(n^3\right)$ time and uses $\mathcal{O}\left(n^2\right)$ space, and the cache complexity reduces to $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ when the cache is tall (i.e., $M = \Omega\left(B^2\right)$) and the space usage is too large for the cache (i.e., $n^2 = \Omega\left(M\right)$). In contrast, though the standard iterative dynamic programming approach for computing the output boundary has the same time and space complexities (see, e.g., [37] for a standard technique that allows the DP to be implemented in $\mathcal{O}\left(n^2\right)$ space), it incurs a factor of $\sqrt{M}$ more cache-misses.

Compute-Traceback-Path-3D. Given the input boundary of $c[\ i_1 : i_2,\ j_1 : j_2,\ k_1 : k_2\ ]$ and the entry point of the traceback path on the output boundary this function (Function 8.2.2) recursively computes the entire path. Recall that a traceback runs backwards, that is, it enters the cube through a point on the output boundary and exits through the input boundary.

If $q = 0$, the traceback path can be updated directly using recurrence 8.2.3, otherwise it performs two passes: forward and backward. In the forward pass it computes the output boundaries of all subcubes except $Q_{2,2,2}$ as in Compute-Boundary-3D. After this pass the algorithm knows the input boundaries of all eight subcubes, and the problem reduces to recursively extracting the fragments of the traceback path from each subcube and combining them. In the backward pass the algorithm starts at $Q_{2,2,2}$ and updates the traceback path by calling itself recursively on the subcubes in the reverse order of the forward pass. This backward order of the recursive calls is essential since in order to find the traceback path through a subcube the algorithm requires an entry point on its output boundary through which the path enters the subcube and initially this point is known for only one subcube. The subcubes are processed in the backward order because it ensures that the exit point of the traceback path from one subcube can be used as the entry point of the path to the next subcube in the sequence.

Analysis. Let $I_2(n)$ be the cache-complexity of Compute-Traceback-Path-3D on input sequences of length n each. We observe that though the algorithm calls itself recursively 8 times in the backward pass, at most 4 of those recursive calls will actually be executed and the rest will terminate at line 1 of the algorithm (see Figure 8.2)) since the traceback path cannot intersect more than 4 subcubes. Then,

$$I_2(n) = \begin{cases} \mathcal{O}\left(n + \frac{n^2}{B}\right) & \text{if } n \leq \sqrt{\gamma M}, \\ 4I_2\left(\frac{n}{2}\right) + 7I_1\left(\frac{n}{2}\right) + \mathcal{O}\left(n + \frac{n^2}{B}\right) & \text{otherwise;} \end{cases}$$

where γ is the largest constant sufficiently small that computation involving sequences of length $\sqrt{\gamma M}$ each can be performed completely inside the cache. Solving the recurrence we obtain $I_2(n) = \mathcal{O}\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ for all n. The algorithm runs in $\mathcal{O}\left(n^3\right)$ time and uses $\mathcal{O}\left(n^2\right)$ space, and $I_2(n)$ reduces to $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ when $M = \Omega\left(B^2\right)$ (i.e., the cache is tall) and $n^2 = \Omega\left(M\right)$ (i.e., space usage is too large for the cache). When compared with the cache-complexity of any existing algorithm for finding the traceback path our algorithm improves it by at least a factor of $\sqrt{M}$, and improves the space complexity by a factor of n when compared against the standard dynamic programming solution.

Our algorithm can be easily extended to handle lengths that are not powers of 2 within the same performance bounds. Thus we have the following theorem.

Theorem 8.2.1. *Given three sequences X, Y and Z of length n each, any recurrence relation of the same form as recurrence 8.2.3 with $d = 3$ can be solved and a traceback path can be computed in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ cache misses.*

If the cache is tall $\left(\text{i.e., } M = \Omega\left(B^2\right)\right)$ and the sequences are long $\left(\text{i.e., } n = \Omega\left(\sqrt{M}\right)\right)$, the cache complexity of the algorithm reduces to $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$.

8.2.2 I/O Lower Bound

In this section we prove that our cache-oblivious algorithms for solving recurrence 8.2.3 for $d = 2$ and $d = 3$ are cache-optimal by showing that the following theorem holds.

Theorem 8.2.2. *For any $d \geq 2$, any algorithm that implements the computation defined by recurrence 8.2.3, must perform $\Omega\left(\frac{n^d}{BM^{\frac{1}{d-1}}}\right)$ block transfers.*

The lower bound in Theorem 8.2.2 follows from the I/O lower bound proved by Hong & Kung [74] for executing the DAG obtained by taking the product of d directed line graphs. Let $L_1 = (V, E)$ be a directed line graph, where $V = \{\ 1, 2, \ldots, n\ \}$ and $E = \{\ (i, i+1) \mid i \in [1, n-1]\ \}$. Nodes in L_1 represent operations, and edges represent data-flow. The node with no incoming edges (i.e., node 1) is the unique

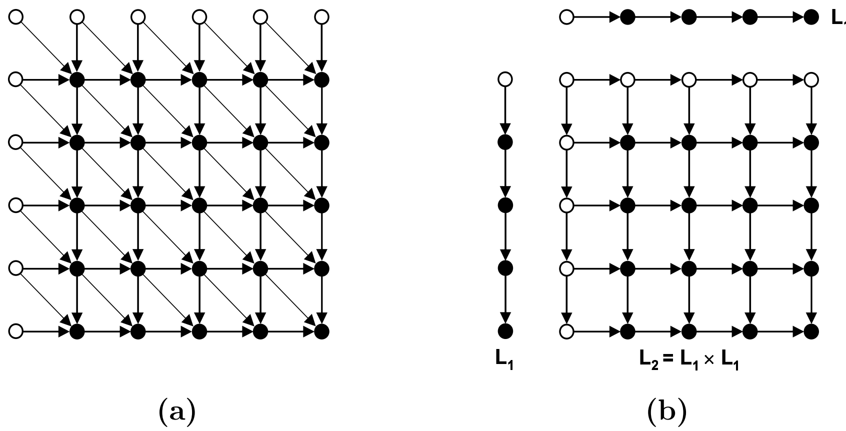

Figure 8.3: (*I/O lower bound for DP implementing recurrence 8.2.3*) **(a)** Computational DAG G implementing recurrence 8.2.3 for $d = 2$. The nodes colored *white* represent input nodes. **(b)** Product graph L_2 of two line graphs (L_1), which is a subDAG of DAG G shown in Figure 8.3(a). Hence, I/O lower bound for executing L_2 also holds for G.

input and the node with no outgoing edges (i.e., node n) is the unique *output*. For $d \geq 2$, L_d is obtained by taking the product of d such L_1's (see Figure 8.3(b) for an example with $d = 2$). Corollary 7.1 in [74] gives a lower bound on the number of I/O operations $\mathcal{Q}$ required to execute L_d.

Corollary 7.1 in [74]. *For the product L_d with $d \geq 2$, $\mathcal{Q} = \Omega\left(\frac{n^d}{M^{\frac{1}{d-1}}}\right)$.*

The corollary above assumes that data is transferred to and from the cache in blocks of size 1. For block size B, $\mathcal{Q} = \Omega\left(\frac{n^d}{BM^{\frac{1}{d-1}}}\right)$.

Now let us consider the computation DAG G given by recurrence 8.2.3 (Figure 8.3(a) shows an example for $d = 2$). It is easy to see that L_d is, in fact, a subDAG of G, and hence I/O lower bound for executing L_d also holds for G. Therefore, Theorem 8.2.2 follows from the corollary above under the assumption that data is transferred in blocks of size B.

8.2.3 Parallel Implementation of the Cache-oblivious Framework

The framework in Section 8.2 has a simple parallel implementation that for general d performs $\mathcal{O}\left(n^d\right)$ work, uses $\mathcal{O}\left(n^{d-1}\right)$ space, incurs $\mathcal{O}\left(\frac{n^d}{BM^{\frac{1}{d-1}}}\right)$ cache-misses and terminates in $\mathcal{O}\left(\frac{n^d}{p} + n^{\log_2(d+1)} \log n\right)$ parallel steps when run on p processors with private caches. This parallelization is based on the observation that some of the recursive function calls in the sequential algorithm can actually be executed in parallel; for example, in Compute-Boundary-3D (lines 5–6) and in the forward pass of Compute-Traceback-Path-3D (lines 6–7) quadrants $Q_{1,1,2}$, $Q_{1,2,1}$ and $Q_{2,1,1}$ can be evaluated in parallel followed by the parallel evaluation of quadrants $Q_{1,2,2}$, $Q_{2,1,2}$ and $Q_{2,2,1}$.

In this section we improve the parallel time complexity of our framework to $\mathcal{O}\left(\frac{n^d}{p} + nd\right)$ while keeping the other bounds unchanged from above. While the simple parallel implementation described above is both cache- and processor-oblivious, the improved algorithms require the knowledge of p. We present two different parallel implementations for the 3-dimensional case for distributed and shared caches, respectively. Implementation for general d is similar.

Distributed Caches

We consider a parallel machine with p processors with each processor having a private cache of size M and block size B.

Par-Compute-Boundary-3D. This function decomposes its cubic computation space Q ($\equiv c[\,1:n,\ 1:n,\ 1:n\,]$) into r^3 subcubes of size $\frac{n}{r} \times \frac{n}{r} \times \frac{n}{r}$ each, where $r = \Theta\left(\min(\sqrt{p}, n)\right)$. By $Q_{i,j,k}$ ($1 \leq i,j,k \leq r$) we denote the subcube that is i-th from the left boundary of Q, j-th from the back boundary and k-th from the top boundary. Then the computation progresses in $3r-2$ steps. In step t ($1 \leq t \leq 3r-2$) output boundaries of all $Q_{i,j,k}$ with $i+j+k = t+2$ are computed in parallel using a modified version of Compute-Boundary-3D which for each cell on the output boundary also computes the location where the traceback path from that cells hits the input boundary.

For $p \leq n^2$, there are $\Theta\left(\sqrt{p}\right)$ steps of parallel subcube computations requiring $\mathcal{O}\Big(\big(\frac{n}{\sqrt{p}}\big)^3\Big)$ time each, and thus the entire computation terminates in $\mathcal{O}\Big(\sqrt{p} \times \left(\frac{n}{\sqrt{p}}\right)^3\Big) = \mathcal{O}\left(\frac{n^3}{p}\right)$ parallel time. For $p > n^2$, there are $\Theta\left(n\right)$ steps of $\mathcal{O}\left(1\right)$ time each and the computation completes in $\mathcal{O}\left(n\right)$ parallel time. The parallel time complexity of the algorithm is thus $\mathcal{O}\left(\frac{n^3}{p} + n\right)$. It is straight-forward to show that the algorithm performs $\mathcal{O}\left(n^3\right)$ work and uses $\mathcal{O}\left(n^2\right)$ space.

Since there are r^3 calls to Compute-Boundary-3D on sequences of length $\frac{n}{r}$ and since each of them is executed on a single processor, total number of cache-misses incurred by all such calls is $r^3 \times \mathcal{O}\left(\frac{n}{r} + \frac{\left(\frac{n}{r}\right)^2}{B} + \frac{\left(\frac{n}{r}\right)^3}{M} + \frac{\left(\frac{n}{r}\right)^3}{B\sqrt{M}}\right) = \mathcal{O}\Big(r^2 \cdot n + r \cdot \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\Big) = \mathcal{O}\left(p \cdot n + \sqrt{p} \cdot \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$.

Par-Compute-Traceback-Path-3D. This function is similar to the sequential Compute-Traceback-Path-3D given in Section 8.2.1 except for the following differences.

Forward Pass: Instead of calling Compute-Boundary-3D described in Section 8.2.1 we call Par-Compute-Boundary-3D described in this section with all p processors on each of the 8 subcubes (including $Q_{2,2,2}$) sequentially.

Backward Pass: We know the cell through which the traceback path enters $Q_{2,2,2}$. Now using the extra information on traceback paths computed in the forward pass we can find in constant time where the traceback path hits all other (at most three) subcubes. Therefore, we can extract the fragments of the traceback path from all (at most four) subcubes by calling Par-Compute-Traceback-Path-3D on each of them with $\frac{p}{4}$ processors each.

Let $T_p(n)$ denote the parallel running time of Par-Compute-Traceback-Path-3D when called with p processors. Let $T'_p(n)$ denote the same for Par-Compute-Boundary-3D. Then for $p \le n^2$,

$$T_p(n) \le 8 \cdot T'_p\left(\frac{n}{2}\right) + +T_{\frac{p}{4}}\left(\frac{n}{2}\right) + \Theta\left(\frac{n^2}{p}\right)$$

Solving we obtain, $T_p(n) = \mathcal{O}\left(\frac{n^3}{p}\right)$. Therefore, for all values of p, $T_p(n) = \mathcal{O}\left(\frac{n^3}{p} + n\right)$. The algorithm performs $\mathcal{O}\left(n^3\right)$ work and uses $\mathcal{O}\left(n^2\right)$ space. It can be shown to incur $\mathcal{O}\left(p \cdot n + \sqrt{p} \cdot \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ cache-misses using a recurrence similar to that for Compute-Traceback-Path-3D (in Section 8.2.1).

Therefore, we obtain the following theorem.

Theorem 8.2.3. *There exists a parallel implementation of* Compute-Traceback-Path-3D *that when executed on $p \ge 1$ processors, each with a private cache of size M and block size B, performs $\mathcal{O}\left(n^3\right)$ work, uses $\mathcal{O}\left(n^2\right)$ space, incurs $\mathcal{O}\left(p \cdot n + \sqrt{p} \cdot \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ cache misses and terminates in $\mathcal{O}\left(\frac{n^3}{p} + n\right)$ parallel steps.*

Shared Caches

We consider a parallel machine with p processors sharing a cache of size $M = \Omega\left(p\right)$ and block size B.

Our algorithm is similar to the sequential algorithm given in Section 8.2.1 until we reach a subproblem involving sequences of length r, where $r = \Theta\left(\min(\sqrt{p}, n)\right)$. At that point we solve the subproblem in $\mathcal{O}\left(r\right)$ parallel steps using the parallel algorithm described in Section 8.2.3. Since there are $\mathcal{O}\left(\left(\frac{n}{r}\right)^3\right)$ such subproblems, the parallel running time of the algorithm is $\mathcal{O}\left(\left(\frac{n}{r}\right)^3\right) \times \mathcal{O}\left(r\right) = \mathcal{O}\left(\frac{n^3}{p} + n\right)$. Since $r = \Theta\left(\min(\sqrt{p}, n)\right) = \mathcal{O}\left(\sqrt{M}\right)$, the cache complexity of this algorithm can be shown to be $\mathcal{O}\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ using recurrences similar to those in Section 8.2.1. The algorithm performs $\mathcal{O}\left(n^3\right)$ work and uses $\mathcal{O}\left(n^2\right)$ space. Thus we have the following theorem.

Theorem 8.2.4. *There exists a parallel implementation of* COMPUTE-TRACEBACK-PATH-3D *that when executed on $p \geq 1$ processors with a shared cache of size M and block size B, performs $\mathcal{O}\left(n^3\right)$ work, uses $\mathcal{O}\left(n^2\right)$ space, terminates in $\mathcal{O}\left(\frac{n^3}{p} + n\right)$ parallel steps, and incurs $\mathcal{O}\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ cache-misses provided $p = \mathcal{O}\left(M\right)$.*

8.2.4 Applications of the Cache-oblivious Framework

In this section we describe how to apply the cache-oblivious framework described in Section 8.2 in order to obtain cache-oblivious algorithms for longest common subsequence, pairwise sequence alignment, median of three sequences, and RNA secondary structure prediction with simple pseudoknots. As described in Section 8.2.3 once a problem is mapped to the framework it also immediately implies a parallel cache-oblivious algorithm for the problem.

Longest Common Subsequence (LCS)

We introduced the LCS problem in Section 8.2. The classic iterative dynamic programming solution to the LCS problem is based on recurrence 8.2.2 (or 8.2.1) given in that Section which computes the length of an LCS along with a traceback path in $\Theta\left(n^2\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^2}{B}\right)$ cache misses, where n is the length of each input sequence. Hirschberg's algorithm [70] reduces the space complexity to $\mathcal{O}\left(n\right)$ while keeping the other bounds unchanged.

Cache-oblivious Implementation. Recurrence 8.2.2 is an instance of the general recurrence 8.2.3 for $d = 2$. Therefore, function COMPUTE-BOUNDARY-2D (see Figure D.1 in Appendix D) can be used to compute the length of the LCS cache-obliviously, and COMPUTE-TRACEBACK-PATH-2D (see Figure D.2 in Appendix D) can be used to extract an LCS. Thus the following claim follows from Theorem D.0.1 in Appendix D.

Claim 8.2.1. *The longest common subsequence of two sequences of length n each can be computed cache-obliviously in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space and $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache misses.*

Cache-efficient parallel implementations of this algorithm can be obtained using the techniques in Section 8.2.3.

Pairwise Global Sequence Alignment with Affine Gap Penalty

Sequence alignment plays a central role in biological sequence comparison, and can reveal important relationships among organisms. Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ over a finite alphabet Σ, an *alignment* of X and Y is a matching M of sets $\{1, 2, \dots, m\}$ and $\{1, 2, \dots, n\}$ such that if $(i, j), (i', j') \in M$ and $i < i'$ hold then $j < j'$ must also hold [78]. The i-th letter of X or Y is said to be in a *gap* if it does not appear in any pair in M. Given a *gap penalty* g and a mismatch cost $s(a, b)$ for each pair $a, b \in \Sigma$, the *basic (global) pairwise sequence alignment problem* asks for a matching M_{opt} for which $(m + n - |M_{opt}|) \times g + \sum_{(a,b) \in M_{opt}} s(a, b)$ is minimized [78].

For simplicity of exposition we will assume $m = n$ for the rest of this section.

The formulation of the basic sequence alignment problem favors a large number of small gaps while real biological processes favor the opposite. The alignment can be made more realistic by using an *affine gap penalty* [61, 8] which has two parameters: a *gap introduction cost* g_i and a *gap extension cost* g_e. A run of k gaps incurs a total cost of $g_i + g_e \times k$.

In [61] Gotoh presented an $\mathcal{O}\left(n^2\right)$ time and $\mathcal{O}\left(n^2\right)$ space DP algorithm for solving the global pairwise alignment problem with affine gap costs. The algorithm incurs $\mathcal{O}\left(\frac{n^2}{B}\right)$ cache misses. The space complexity of the algorithm can be reduced to $\mathcal{O}\left(n\right)$ using Hirschberg's space-reduction technique [93] or the diagonal checkpointing technique described in [63]. However, the time and cache complexities remain unchanged. Gotoh's algorithm solves the following DP recurrences.

$$D(i, j) = \begin{cases} G(0, j) + g_e & \text{if } i = 0 \wedge j > 0 \\ \min\{D(i-1, j), G(i-1, j) + g_i\} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases} \tag{8.2.4}$$

$$I(i, j) = \begin{cases} G(i, 0) + g_e & \text{if } i > 0 \wedge j = 0 \\ \min\{I(i, j-1), G(i, j-1) + g_i\} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases} \tag{8.2.5}$$

$$G(i, j) = \begin{cases} 0 & \text{if } i = 0 \wedge j = 0 \\ g_i + g_e \times j & \text{if } i = 0 \wedge j > 0 \\ g_i + g_e \times i & \text{if } i > 0 \wedge j = 0 \\ \min\{D(i, j), I(i, j), G(i-1, j-1) + s(x_i, y_j)\} & \text{if } i > 0 \wedge j > 0. \end{cases} \tag{8.2.6}$$

The optimal alignment cost is $\min\{G(n, n), D(n, n), I(n, n)\}$ and an optimal alignment can be traced back from the smallest of $G(n, n)$, $D(n, n)$ and $I(n, n)$.

Cache-oblivious Implementation. Recurrences 8.2.4 - 8.2.6 can be viewed as a single recurrence evaluating a single matrix $c[\ 0 : n,\ 0 : n\ ]$ with three fields: D, I and G. When $i = 0$ or $j = 0$ each field of $c[i, j]$ depends only on the indices i and j, and constants g_i and g_e, and hence each such entry can be computed using a function similar to h in the general recurrence 8.2.3 in Section 8.2. When both i and

j are positive, $c[i,j]$ depends on x_i, y_j, the entries in $c[\ i-1:i,\ j-1:j\ ] \setminus c[i,j]$ (i.e., in $D(\ i-1:i,\ j-1:j\ ) \setminus D(i,j)$, $I(\ i-1:i,\ j-1:j\ ) \setminus I(i,j)$ and $G(\ i-1:i,\ j-1:j\ ) \setminus G(i,j),$), and constants g_i and g_e. Hence, in this case $c[i,j]$ can be computed using a function similar to function f in recurrence 8.2.3. Thus recurrences 8.2.4 - 8.2.6 completely match the general recurrence 8.2.3 for $d = 2$ in Section 8.2. Therefore, function COMPUTE-BOUNDARY-2D (see Figure D.1 in Appendix D) can be used to compute the optimal alignment cost cache-obliviously, and function COMPUTE-TRACEBACK-PATH-2D (see Figure D.2 in Appendix D) can be used to extract the optimal alignment. Thus the following claim follows from Theorem D.0.1 in Appendix D.

Claim 8.2.2. *Optimal global alignment of two sequences of length n each can be performed cache-obliviously using an affine gap cost in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space and $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache misses.*

The cache-complexity of the our cache-oblivious algorithm is a factor of M improvement over previous implementations [61, 93].

Efficient parallel implementations of this cache-oblivious algorithm for both distributed and shared caches can be obtained using techniques described in Section 8.2.3. The resulting algorithms perform $\mathcal{O}\left(n^3\right)$ work and terminate in $\mathcal{O}\left(\frac{n^2}{p}+n\right)$ parallel steps, where p is the number of processors.

Median of Three Sequences

Given three sequences X, Y and Z, the *median problem* asks for a sequence W such that the sum of the pairwise alignment costs of W with X, Y and Z is minimized. The sequence W is called the *median* of the three given sequences. In this section we will assume affine gap costs for the alignments.

In [80] Knudsen presented a dynamic programming algorithm for optimal multiple alignment of any number of sequences related by a tree under affine gap costs. The input sequences are assumed to be at the leaves of the tree, and the optimal alignment cost is the minimum sum of pairwise alignment costs of the sequence pairs at the ends of each edge of the tree over all possible ancestral sequences (i.e., the unknown sequences at the internal nodes of the tree). For N sequence of length n each, the algorithm runs in $\mathcal{O}\left(16.81^N n^N\right)$ time and uses $\mathcal{O}\left(7.442^N n^N\right)$ space. For $N = 3$, Knudsen's algorithm solves the median problem in $\mathcal{O}\left(n^3\right)$ time and space, and incurs $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses. An Ukkonen-based algorithm for the median problem is presented in [101], which performs well especially for sequences whose (3-way) edit distance δ is small. On average, it requires $\mathcal{O}\left(n+\delta^3\right)$ time and space [101].

Knudsen's algorithm [80] for three sequences (say, $X = x_1x_2 \dots x_n$, $Y = y_1y_2 \dots y_n$ and $Z = z_1z_2 \dots, z_n$) is a dynamic program over a three-dimensional matrix K. Each entry $K(i, j, k)$ is composed of 23 fields. Each field corresponds to an indel configuration q, which describes how the last characters x_i, y_j and z_k are matched. A residue configuration defines how the next three characters of the sequences will be matched. Each configuration is a vector $e = (e_1, e_2, e_3, e_4)$, where $e_i \in \{0, 1\}$, $1 \le i \le 4$. The entry e_i, $1 \le i \le 3$ indicates if the aligned character of sequence i is a gap or a residue, while e_4 corresponds to the aligned character of the median sequence. There are 10 residue configurations out of 16 possible ones. The recursive step calculates the value of the next entry by applying residue configurations to each indel configuration. We define $\nu(e, q) = q'$ if applying the residue configuration e to the indel configuration q gives the indel configuration q'. The recurrence relation used by Knudsen's algorithm is:

$$K(i,j,k)_q = \begin{cases} 0 & \text{if } i = j = k = 0 \wedge q = q_o \\ \infty & \text{if } i = j = k = 0 \wedge q \neq q_o \\ \min_{e,q':q=\nu(e,q')} \left\{ \begin{array}{c} K(i',j',k')_{q'} + G_{e,q} \\ +M_{(i',j',k') \to (i,j,k)} \end{array} \right\} & \text{otherwise.} \end{cases} \tag{8.2.7}$$

where q_o is the configuration where all characters match, $i' = i - e_1$, $j' = j - e_2$ and $k' = k - e_3$, $M_{(i',j',k')=(i,j,k)}$ is the matching cost between characters of the sequences, and $G_{e,q}$ is the cost of introducing or extending the gap.

The M and G matrices can be pre-computed. Therefore, Knudsen's algorithm runs in $\mathcal{O}\left(n^3\right)$ time and space with $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses.

Cache-oblivious Algorithm. In order to make recurrence 8.2.7 match the general recurrence 8.2.3 for $d = 3$ given in Section 8.2, we shift all characters of X, Y and Z one position to the right, introduce a dummy character in front of each of those three sequences, and obtain the following recurrence by modifying recurrence 8.2.7.

$$c[i,j,k]_q = \begin{cases} \infty & \text{if } i = 0 \ \vee \ j = 0 \ \vee \ k = 0 \\ 0 & \text{if } i = j = k = 1 \ \wedge \ q = q_o \\ \infty & \text{if } i = j = k = 1 \ \wedge \ q \neq q_o \\ \min_{e,q':q=\nu(e,q')} \left\{ \begin{array}{c} c[i',j',k']_{q'} + G_{e,q} \\ +M_{(i',j',k') \to (i,j,k)} \end{array} \right\} & \text{otherwise.} \end{cases}$$

It is easy to see that $K(i, j, k)_q = c[i + 1, j + 1, k + 1]_q$ for $0 \le i, j, k \le n$ and any q. If $i = 0$ or $j = 0$ or $k = 0$ then $c[i, j, k]_q$ can be evaluated using a function $h(\langle \ i, \ j, \ k \ \rangle) = \infty$ as in the general recurrence 8.2.3. Otherwise the value of $c[i, j, k]_q$ depends on the values of i, j, and k, values in some constant size arrays (G and M), and on the cells to its left, back and top. Hence, in this case, $c[i, j, k]_q$ can be evaluated using a function similar to f in recurrence 8.2.3 for $d = 3$. Therefore, the above recurrence matches the 3-dimensional version of the general recurrence 8.2.3, and function COMPUTE-BOUNDARY-3D (see Figure 8.1) can be used to compute the matrix c and function COMPUTE-TRACEBACK-PATH-3D (see Figure 8.2) to retrieve an optimal alignment. Hence we claim the following using Theorem 8.2.1.

Claim 8.2.3. *Optimal alignment of three sequences of length n each can be performed and the median sequence under the optimal alignment can be computed cache-obliviously using an affine gap cost in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses.*

Using the techniques in Section 8.2.3 we obtain parallel implementations of this cache-oblivious algorithm for both distributed and shared caches that on a p-processor machine perform $\mathcal{O}\left(n^2\right)$ work and terminate in $\mathcal{O}\left(\frac{n^3}{p}+n\right)$ parallel steps.

RNA Secondary Structure Prediction with Pseudoknots

A single-stranded RNA can be viewed as a string $X = x_1x_2\ldots x_n$ over the alphabet $\{A, U, G, C\}$ of bases. An RNA strand tends to give rise to interesting structures by forming *complementary base pairs* with itself. An *RNA secondary structure* (w/o pseudoknots) is a planar graph with the nesting condition: if $\{x_i, x_j\}$ and $\{x_k, x_l\}$ form base pairs and $i < j$, $k < l$ and $i < k$ hold then either $i < k < l < j$ or $i < j < k < l$ [129, 106, 6]. An *RNA secondary structure with pseudoknots* is a structure where this nesting condition is violated [106, 6].

In [6] Akutsu presented a DP to compute RNA secondary structures with maximum number of base pairs in the presence of *simple pseudoknots* (see [6] for definition) which runs in $\mathcal{O}\left(n^4\right)$ time, $\mathcal{O}\left(n^3\right)$ space and $\mathcal{O}\left(\frac{n^4}{B}\right)$ cache-misses. In this Section we improve its space and cache complexities to $\mathcal{O}\left(n^2\right)$ and $\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$, respectively, without changing its time complexity.

We list below the DP recurrences used in Akutsu's algorithm [6]. For every pair (i_0, k_0) with $1 \leq i_0 \leq k_0 - 2 \leq n - 2$, recurrences 8.2.8 - 8.2.12 compute the maximum number of base pairs in a pseudoknot with endpoints at the i_0-th and k_0-th residues. The value computed by recurrence 8.2.12, i.e., $S_{pseudo}(i_0, k_0)$, is the desired value. In recurrences 8.2.8 and 8.2.9, $v(x, y) = 1$ if (x, y) form a base pair, otherwise $v(x, y) = -\infty$. All uninitialized entries are assumed to have value 0.

$$S_L(i,j,k) = \begin{cases} v(a_i,a_j) & \text{if } i_0 \leq i < j \geq k, \\ v(a_i,a_j) + S_{MAX}(i-1,j+1,k) & \text{if } i_0 \leq i < j < k. \end{cases} \tag{8.2.8}$$

$$S_R(i,j,k) = \begin{cases} v(a_j,a_k) & \text{if } i_0 = i+1 < j = k-1, \\ v(a_j,a_k) + S_{MAX}(i,j+1,k-1) & \text{if } i_0 \leq i < j < k. \end{cases} \tag{8.2.9}$$

$$S_M(i,j,k) = \max \left\{ \begin{array}{c} S_L(i-1,j,k),\ S_M(i-1,j,k), \\ S_{MAX}(i,j+1,k), \\ S_M(i,j,k-1),\ S_R(i,j,k-1) \end{array} \right\} \text{ if } i_0 \le i < j < k. \quad (8.2.10)$$

$$S_{MAX}(i,j,k) = \max \{\ S_L(i,j,k),\ S_M(i,j,k),\ S_R(i,j,k)\ \} \quad (8.2.11)$$

$$S_{pseudo}(i_0,k_0) = \max_{i_0 \le i < j < k \le k_0} \{\ S_{MAX}(i,j,k)\ \} \quad (8.2.12)$$

After computing all entries of S_{MAX} for a fixed i_0, all $S_{pseudo}(i_0,k_0)$ values for $k_0 \ge i_0+2$ can be computed using equation 8.2.12 in $\mathcal{O}\left(n^3\right)$ time and space and $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses. Since there are $n-2$ possible values for i_0, all $S_{pseudo}(i_0,k_0)$ can be computed in $\mathcal{O}\left(n^4\right)$ time, $\mathcal{O}\left(n^3\right)$ space and $\mathcal{O}\left(\frac{n^4}{B}\right)$ cache-misses.

Finally, the following recurrence computes the optimal score $S(1,n)$ for the entire structure in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses [6].

$$S(i,j) = \max \left\{ \begin{array}{c} S_{pseudo}(i,j),\ S(i+1,j-1)+v(a_i,a_j), \\ \max_{i<k\le j}\left\{S(i,k-1),S(k,j)\right\} \end{array} \right\} \quad (8.2.13)$$

Recurrence 8.2.13 can be evaluated in only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses and $\mathcal{O}\left(n^2\right)$ space without changing the other bounds using our GEP framework in Chapter 6.

Space Reduction. We now describe our space reduction result. Observe that evaluating recurrence 8.2.12 requires retaining all $\mathcal{O}\left(n^3\right)$ values computed by recurrence 8.2.11. We avoid using this extra space by computing all required $S_{pseudo}(i_0,k_0)$ values on the fly while evaluating recurrence 8.2.11. We achieve this goal by introducing recurrence 8.2.14, replacing recurrence 8.2.12 with recurrence 8.2.15 for S'_{pseudo}, and using S'_{pseudo} instead of S_{pseudo} for evaluating recurrence 8.2.13. All uninitialized entries in recurrences 8.2.14 and 8.2.15 are assumed to have value $-\infty$.

$$S_P(i,j,k) = \left\{ \begin{array}{ll} \max\{\ S_{MAX}(i,j,k),\ S_P(i,j+1,k)\ \} & \text{if } i_0 \le i < j < k, \\ S_P(i,j+1,k) & \text{if } i_0 \le i \ge j < k. \end{array} \right. \quad (8.2.14)$$

$$S'_{pseudo}(i_0,k_0) = \max \left\{ \begin{array}{c} S'_{pseudo}(i_0,k_0-1), \\ \max_{i_0 \le i < k_0-1}\left\{S_P(i,i_0+1,k_0)\right\} \end{array} \right\} \text{ if } k_0 \ge i_0+2. \quad (8.2.15)$$

We claim that recurrence 8.2.15 computes exactly the same values as recurrence 8.2.12.

Claim 8.2.4. *For* $1 \le i_0 \le k_0 - 2 \le n-2$, $S'_{pseudo}(i_0,k_0) = S_{pseudo}(i_0,k_0)$.

Proof. (sketch) We obtain the following by simplifying recurrence 8.2.14.

$$S_P(i,j,k) = \begin{cases} \max_{\max\{i+1,j\} \le j' < k} \{ \ S_{MAX}(i,j',k) \ \} & \text{if } i_0 \le i \ \wedge \ j < k, \\ -\infty & \text{otherwise.} \end{cases}$$

Therefore, $\max_{i_0 \le i < k_0 - 1} \{S_P(i, i_0+1, k_0)\} = \max_{i_0 \le i < j < k_0} \{ \ S_{MAX}(i,j,k_0) \ \}$. We can now evaluate $S'_{pseudo}(i_0,k_0)$ by induction on k_0. For $k_0 \ge i_0 + 2$,

$$\begin{aligned} S'_{pseudo}(i_0,k_0) &= \max\left\{ S'_{pseudo}(i_0,k_0-1), \max_{i_0 \le i < k_0 - 1} \{S_P(i,i_0+1,k_0)\} \right\} \\ &= \max\left\{ \max_{i_0 \le i < j < k \le k_0 - 1} \{S_{MAX}(i,j,k)\}, \max_{i_0 \le i < j < k_0} \{S_{MAX}(i,j,k_0)\} \right\} \\ &= \max_{i_0 \le i < j < k \le k_0} \{S_{MAX}(i,j,k)\} \ = \ S_{pseudo}(i_0,k_0) \end{aligned}$$

■

Now observe that in order to evaluate recurrence 8.2.15 we only need the values $S_P(i,j,k)$ for $j = i_0 + 1$, and each entry (i,j,k) in recurrences 8.2.8 - 8.2.11 and 8.2.14 depends only on entries $(\cdot, j, \cdot)$ and $(\cdot, j+1, \cdot)$. Therefore, we will evaluate the recurrences for $j = n$ first, then for $j = n-1$, and continue down to $j = i_0 + 1$. Observe that in order to evaluate for $j = j'$ we only need to retain the $\mathcal{O}\left(n^2\right)$ entries computed for $j = j' + 1$. Thus for a fixed i_0 all $S_P(i, i_0+1, k)$ and consequently all relevant $S'_{pseudo}(i_0,k_0)$ can be computed using only $\mathcal{O}\left(n^2\right)$ space, and the same space can be reused for all n values of i_0.

The time and cache complexities of the algorithm remain unchanged from [6].

Cache-oblivious Algorithm. The evaluation of recurrences 8.2.8 - 8.2.11 and 8.2.14 can be viewed as evaluating a single $n \times n \times n$ matrix c with five fields: S_L, S_R, S_M, S_{MAX} and S_P. If we replace all j with $n - j + 1$ in the resulting recurrence it conforms to recurrence 8.2.3 for $d = 3$. Therefore, for any fixed i_0 we can use the Compute-Boundary-3D function in Figure 8.1 to compute all entries $S_P(i, i_0+1, k)$ and consequently all relevant $S'_{pseudo}(i_0,k_0)$ values. All $S'_{pseudo}(i_0,k_0)$ values can be computed by n applications (i.e., once for each i_0) of Compute-Boundary-3D (see Figure 8.1).

For any given pair (i_0, k_0) the pseudoknot with the optimal score can be traced back cache-obliviously by calling Compute-Traceback-Path-3D (see Figure 8.2). Thus from Theorem 8.2.1 we obtain the following claim.

Claim 8.2.5. *Given an RNA sequence of length n, a secondary structure that has the maximum number of base pairs in the presence of simple pseudoknots can be computed cache-obliviously in $\mathcal{O}\left(n^4\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$ cache misses.*

Parallel Cache-oblivious Implementation. As mentioned before, all $S'_{pseudo}(i_0, k_0)$ values can be computed by n applications of COMPUTE-BOUNDARY-3D, and we observe that all these n function calls can be made in parallel. We know from Section 8.2.3 that PAR-COMPUTE-BOUNDARY-3D executes $\mathcal{O}(n)$ parallel steps when called with an unbounded number of processors on an RNA sequence of length n. Since COMPUTE-BOUNDARY-3D performs $\mathcal{O}(n^3)$ work and used $\mathcal{O}(n^2)$ space, n parallel applications of this function will perform $\mathcal{O}(n^4)$ work, use $\mathcal{O}(n^3)$ space and execute $\mathcal{O}\left(\frac{n^4}{p} + n\right)$ parallel steps when executed with p processors. After computing all $S'_{pseudo}(i_0, k_0)$ values a pseudoknot with the optimal score is determined using recurrence 8.2.13 which can be solved in $\mathcal{O}(n^3)$ work, $\mathcal{O}(n^2)$ pace and $\mathcal{O}\left(\frac{n^3}{p} + n \log^2 n\right)$ parallel steps using our cache-oblivious parallel GEP framework [35]. Finally, the optimal pseudoknot thus determined can be traced back in a single call of PAR-COMPUTE-TRACEBACK-PATH-3D which performs $\mathcal{O}(n^3)$ work, uses $\mathcal{O}(n^2)$ space and executes $\mathcal{O}\left(\frac{n^3}{p} + n\right)$ parallel steps when called with p processors. Therefore, we can claim the following.

Claim 8.2.6. *Given an RNA sequence of length n and p processors, a secondary structure that has the maximum number of base pairs in the presence of simple pseudoknots can be computed cache-obliviously in $\mathcal{O}(n^4)$ work, $\mathcal{O}(n^3)$ space and $\mathcal{O}\left(\frac{n^4}{p} + n \log^2 n\right)$ parallel steps while incurring only $\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$ cache-misses on both distributed and shared caches.*

Extensions. In [6] the basic dynamic program for simple pseudoknots has been extended to handle energy functions based on adjacent base pairs within the same time and space bounds. Our cache-oblivious technique as described above can be adapted to solve this extension within the same improved bounds as for the basic DP. An $\mathcal{O}(n^{4-\delta})$ time approximation algorithm for the basic DP has also been proposed in [6], and our techniques can be used to improve the space (sequential) and cache complexity of the algorithm to $\mathcal{O}(n^2)$ (from $\mathcal{O}(n^3)$) and $\mathcal{O}\left(\frac{n^{4-\delta}}{B\sqrt{M}}\right)$ (from $\mathcal{O}\left(\frac{n^{4-\delta}}{B}\right)$), respectively.

8.3 Cache-oblivious Dynamic Programs with Non-local Dependencies

In this section we consider two dynamic programming problems, namely the *gap problem* and the *basic RNA secondary structure prediction problem*, which unlike the problems in the previous section, compute the value of each cell in the DP table from a non-constant number of cells, not all of which are adjacent to the cell being computed.

FUNCTION 8.3.1. RECURSIVE-GAP(C)

[We assume that C is a square submatrix of D, and the top-left cell of C corresponds to $D[i,j]$ for some $i,j \in [1,n]$. We also assume that the dimension of C is a power of 2. This function recursively computes the entries of C according to recurrence 8.3.16.]

1. ***if*** C is a 1×1 matrix ***then***
2. $\quad D[i,j] \leftarrow \min(\ D[i,j],\ D[i-1,j-1] + S(x_i,\ y_j)\ )$
3. ***else***
4. $\quad$ RECURSIVE-GAP(C_{11}) *{compute top-left quadrant}*
5. $\quad$ APPLY-E(C_{12}, C_{11}), RECURSIVE-GAP(C_{12}) *{compute top-right quadrant}*
6. $\quad$ APPLY-F(C_{21}, C_{11}), RECURSIVE-GAP(C_{21}) *{compute bottom-left quadrant}*
7. $\quad$ APPLY-E(C_{22}, C_{21}), APPLY-F(C_{22}, C_{12})
$\quad$ RECURSIVE-GAP(C_{22}) *{compute bottom-right quadrant}*

RECURSIVE-GAP ENDS

FUNCTION 8.3.2. APPLY-E(A, B)

[A and B are two non-overlapping $2^t \times 2^t$ submatrices of D, where t is a nonnegative integer. $A[1,1]$ corresponds to $D[i,j]$, and $B[1,1]$ corresponds to $D[i,q]$ for some $1 \le i \le n$ and $1 \le q < j \le n$. This function updates the entries of A using the entries of B according to the equation defining $E[i,j]$.]

1. ***if*** $t = 0$ ***then***
2. $\quad D[i,j] \leftarrow \min\begin{pmatrix} D[i,j], \\ D[i,q] + w(q,j) \end{pmatrix}$
3. ***else***
4. $\quad$ APPLY-E(A_{11}, B_{11})
$\quad$ APPLY-E(A_{11}, B_{12})
5. $\quad$ APPLY-E(A_{12}, B_{11})
$\quad$ APPLY-E(A_{12}, B_{12})
6. $\quad$ APPLY-E(A_{21}, B_{21})
$\quad$ APPLY-E(A_{21}, B_{22})
7. $\quad$ APPLY-E(A_{22}, B_{21})
$\quad$ APPLY-E(A_{22}, B_{22})

APPLY-E ENDS

FUNCTION 8.3.3. APPLY-F(A, B)

[A and B are two non-overlapping $2^t \times 2^t$ submatrices of D, where t is a nonnegative integer. $A[1,1]$ corresponds to $D[i,j]$, and $B[1,1]$ corresponds to $D[p,j]$ for some $1 \le p < i \le n$ and $1 \le j \le n$. This function updates the entries of A using the entries of B according to the equation defining $F[i,j]$.]

1. ***if*** $t = 0$ ***then***
2. $\quad D[i,j] \leftarrow \min\begin{pmatrix} D[i,j], \\ D[p,j] + w'(p,i) \end{pmatrix}$
3. ***else***
4. $\quad$ APPLY-F(A_{11}, B_{11})
$\quad$ APPLY-F(A_{11}, B_{21})
5. $\quad$ APPLY-F(A_{12}, B_{12})
$\quad$ APPLY-F(A_{12}, B_{22})
6. $\quad$ APPLY-F(A_{21}, B_{11})
$\quad$ APPLY-F(A_{21}, B_{21})
7. $\quad$ APPLY-F(A_{22}, B_{12})
$\quad$ APPLY-F(A_{22}, B_{22})

APPLY-F ENDS

Figure 8.4: Cache-oblivious algorithm for the gap problem. The initial function call is RECURSIVE-GAP(D) with all $D[i,j]$ for $i,j \in [1,n]$ initialized to $+\infty$ and the remaining entries initialized according to recurrence 8.3.16.

8.3.1 The Gap Problem

The *gap* problem [56, 57, 129] 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 $D[i,j]$ 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

$$D[i,j] = \begin{cases} 0 & \text{if } i = j = 0, \\ w(0,j) & \text{if } i = 0,\ 1 \leq j \leq n, \\ w'(0,i) & \text{if } j = 0,\ 1 \leq i \leq m, \\ \min \left\{ \begin{array}{c} D[i-1, j-1] + S(x_i,\ y_j), \\ E[i,j],\ F[i,j] \end{array} \right\} & \text{if } i, j > 0; \end{cases} \tag{8.3.16}$$

$$\text{where} \quad E[i,j] = \min_{0 \leq q < j} \{ \ D[i,q] + w(q,j) \ \}, \tag{8.3.17}$$

$$\text{and} \quad F[i,j] = \min_{0 \leq p < i} \left\{ \ D[p,j] + w'(p,i) \ \right\}. \tag{8.3.18}$$

Assuming $m = n$, this problem can be solved in $\mathcal{O}\left(n^3\right)$ time using $\mathcal{O}\left(n^2\right)$ space [56]; this algorithm incurs $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses.

Cache-oblivious Algorithm

We observe from recurrence 8.3.16 that for $1 \leq i, j \leq n$ each $D[i,j]$ depends only on the entries in $D[0:i, 0:j] \setminus D[i,j]$. The following cache-oblivious recursive decomposition scheme of the computation space ensures that all entries in $D[0:i, 0:j] \setminus D[i,j]$ are computed before computing $D[i,j]$.

We assume for simplicity that $m = n = 2^t$ for some integer $t \geq 0$. We first initialize all $D[i,j]$ for $i, j \in [1, n]$ to $+\infty$ while the remaining entries are initialized according to recurrence 8.3.16. Then we call Recursive-Gap with $C \equiv D[1:n, 1:n]$ as the input matrix. If C is a 1×1 matrix, we can compute this entry directly using recurrence 8.3.16 (in line 2). Otherwise, we decompose C into four quadrants: top-left (C_{11}), top-right (C_{12}), bottom-left (C_{21}) and bottom-right (C_{22}).

We observe that entries in C_{11} do not depend on the entries in any other quadrant, and hence can be computed recursively by calling Recursive-Gap on it (see line 4). The entries in C_{11} contribute in computing the entries in C_{12} through equation 8.3.17, and the entries in C_{21} through equation 8.3.18. We first recursively update C_{12} using equation 8.3.17 with the entries in C_{11} (by calling Apply-E in line 5). After these updates C_{12} no longer depends on the entries in any other quadrant, and is solved recursively by calling Recursive-Gap (see line 5). In line 6, we first call a recursive function Apply-F which updates the entries in C_{21} using equation 8.3.18 with the entries in C_{11}, and then complete the update of C_{21} by calling Recursive-Gap. The entries in C_{22} depend on C_{21} through equation 8.3.17, and on C_{12} through equation 8.3.18. Therefore, we first update C_{22} by calling Apply-E with C_{21} and Apply-F with C_{12}, and finally call Recursive-Gap on C_{22} to complete updating the quadrant.

Cache Complexity. Let $I(n)$ and $I'(n)$ be the cache complexities of Recursive-Gap and Apply-E/Apply-F, respectively, on an input of size $n \times n$. Then

$$I'(n) = \begin{cases} \mathcal{O}\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma' M, \\ 8I'\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases} \quad I(n) = \begin{cases} \mathcal{O}\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma M, \\ 4I\left(\frac{n}{2}\right) + 4I'\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

where, γ' and γ are suitable constants. Solving the recurrences we obtain $I'(n) = \mathcal{O}\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ and $I(n) = \mathcal{O}\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ for all values of n.

All three functions run in $\mathcal{O}\left(n^3\right)$ time and use $\mathcal{O}\left(n^2\right)$ space. Hence, we have the following theorem.

Theorem 8.3.1. *The generalized* gap *problem (i.e., recurrence 8.3.16) on a pair of sequences of length n each, can be solved cache-obliviously in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$ cache misses.*

Parallel Cache-oblivious Implementation

We observe that steps 5 and 6 of Recursive-Gap can be executed in parallel while the first function calls in steps 4 – 7 of both Apply-E and Apply-F can also be executed in parallel followed by the parallel execution of the second function calls in those four steps. Let $T_p(n, m)$ denote the parallel running time of Recursive-Gap when called with p processors, but recursive calls on inputs of size $m \times m$ $(1 \leq m \leq n)$ are executed entirely on a single processor. Let $T'_p(n, m)$ denote the same for Apply-E/Apply-F. Then clearly, for all $p \geq 1$, $T_p(n, n) = \mathcal{O}\left(n^3\right)$ and $T'_p(n, n) = \mathcal{O}\left(n^3\right)$, and also for all $m \in [1, n]$, $T_1(n, m) = \mathcal{O}\left(n^3\right)$ and $T'_1(n, m) = \mathcal{O}\left(n^3\right)$. We also have,

$$T'_\infty(n, m) \leq 2 \cdot T'_\infty\left(\frac{n}{2}, m\right) + 8 \text{ and } T_\infty(n, m) \leq 3 \cdot T_\infty\left(\frac{n}{2}, m\right) + 3 \cdot T'_\infty\left(\frac{n}{2}, m\right) + 8.$$

Solving the recurrences we obtain $T'_\infty(n) = \mathcal{O}\left(nm^2\right)$ and $T_\infty(n) = \mathcal{O}\left(\left(\frac{n}{m}\right)^{\log_2 3} \cdot m^3\right)$. Therefore, using "Brent's theorem" [20],

$$T_p(n,m) = \mathcal{O}\left(\frac{T_1(n,m)}{p} + T_\infty(n,m)\right) = \mathcal{O}\left(\frac{n^3}{p} + \left(\frac{n}{m}\right)^{\log_2 3} \cdot m^3\right).$$

Putting $m = 1$, we get $T_p(n,1) = \mathcal{O}\left(\frac{n^3}{p} + n^{\log_2 3}\right)$.

We first consider cache-efficient execution of this parallel algorithm on distributed caches, where each processor has its own private cache of size M, and then on a shared cache, where all processors share the same cache of size M.

Distributed Caches. For good performance on distributed caches, we want m as large as possible (to reduce data transfer between caches), but at the same time do not want $T_p(n,m)$ to degrade too much. The value of m that keeps $T_p(n,m)$ at $\mathcal{O}\left(\frac{n^3}{p}\right)$ can be calculated by equating $\frac{n^3}{p}$ and $\left(\frac{n}{m}\right)^{\log_2 3} \cdot m^3$, which gives $m = n/p^{\frac{1}{\log_2 \frac{8}{3}}}$. Let $\alpha = \frac{1}{\log_2 \frac{8}{3}}$. There are $\left(\frac{n}{m}\right)^3$ recursive calls on inputs of size $m \times m$ each incurring $\mathcal{O}\left(m + \frac{m^2}{B} + \frac{m^3}{B\sqrt{M}}\right)$ cache-misses. Thus the total number of cache-misses incurred is $\left(\frac{n}{m}\right)^3 \times \mathcal{O}\left(m + \frac{m^2}{B} + \frac{m^3}{M} + \frac{m^3}{B\sqrt{M}}\right) = \mathcal{O}\left(p^{2\alpha} \cdot n + p^{\alpha} \cdot \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$. Therefore, we have the following theorem.

Theorem 8.3.2. *There exists a parallel implementation of* Recursive-Gap *that when executed on $p \geq 1$ processors, each with a private cache of size M and block size B, terminates in $\mathcal{O}\left(\frac{n^3}{p} + n^{\log_2 3}\right)$ parallel steps and incurs $\mathcal{O}\Big(p^{2\alpha} \cdot n + p^{\alpha} \cdot \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\Big)$ cache misses, where $\alpha = \frac{1}{\log_2 \frac{8}{3}} \approx 0.7067$.*

Shared Caches. For good performance on shared caches we want the processors share as much data as possible. We achieve this goal as follows.

Let $\mathcal{G}$ denote the computational DAG of Recursive-Gap, and let $\mathcal{C}(\mathcal{G})$ be the DAG obtained by contracting each subDAG of $\mathcal{G}$ corresponding to a recursive function call (Recursive-Gap or Apply-E or Apply-F) on inputs of size $m \times m$ to a supernode. For any supernode $v \in \mathcal{C}(\mathcal{G})$, the subDAG of $\mathcal{G}$ corresponding to v is denoted by $\mathcal{S}(v)$.

Now we use the following hybrid scheduling scheme. The nodes of $\mathcal{C}(\mathcal{G})$ are executed under 1DF-schedule [18]. However, for each supernode v, the nodes in the subDAG $\mathcal{S}(v)$ are executed under PDF-schedule [18] with all p processors before moving to the next supernode.

The parallel execution time of Recursive-Gap under the scheduling scheme described above can be computed as follows. Let $N_G(n,m)$, $N_E(n,m)$ and $N_F(n,m)$

denote the number of recursive calls to Recursive-Gap, Apply-E and Apply-F, respectively, on inputs of size $m \times m$ when the initial call to Recursive-Gap was on inputs of size $n \times n$. It is not difficult to see that $N_G(n,m) = \left(\frac{n}{m}\right)^2$ and $N_E(n,m) = N_F(n,m) = \frac{1}{2} \cdot \left(\left(\frac{n}{m}\right)^3 - \left(\frac{n}{m}\right)^2\right)$. The parallel running time of Recursive-Gap is then $N_G(n,m) \times T_p(m,1) + N_E(n,m) \times T'_p(m,1) + N_F(n,m) \times T'_p(m,1) = \mathcal{O}\left(\frac{n^3}{p} + \frac{n^3}{m^2} + \frac{n^2}{m^{\log_2 \frac{4}{3}}}\right)$.

A 1DF-schedule schedule on the original DAG $\mathcal{G}$ incurs $I(n) = \mathcal{O}\Big(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\Big)$ cache-misses. Both the 1DF-schedule and the hybrid scheduling scheme execute the supernodes of $\mathcal{G}$ in the same order though the order of execution of the nodes withing any given supernode may differ. Since the subproblem corresponding to any supernode uses $\Theta\left(m^2\right)$ space, the hybrid schedule will incur no more cache-misses than the 1DF-schedule provided we have $\Theta\left(m^2\right)$ additional cache space. Therefore, if we set aside $\Theta\left(m^2\right)$ locations from the cache and execute Recursive-Gap under the hybrid scheduling scheme on a reduced cache of size $M - \Theta\left(m^2\right)$, it will incur $\mathcal{O}\left(I(n)\right)$ cache-misses provided $M - \Theta\left(m^2\right) = \Theta\left(M\right)$.

Therefore, setting $m = \sqrt{p}$ we have the following theorem.

Theorem 8.3.3. *There exists a parallel implementation of* Recursive-Gap *that when executed on $p \geq 1$ processors with a shared cache of size M and block size B, terminates in $\mathcal{O}\left(\frac{n^3}{p} + \frac{n^2}{p^{\log_2 \frac{2}{\sqrt{3}}}} + n^{\log_2 3}\right)$ parallel steps and incurs $\mathcal{O}\Big(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\Big)$ cache misses, provided $p = \mathcal{O}\left(M\right)$.*

The Least Weight Subsequence Problem

The *least weight subsequence* problem [71, 57] which is defined by the following dynamic programming recurrence, can be viewed as a one dimensional version of the gap problem:

$$D[j] = \begin{cases} 0 & \text{if } j = 0, \\ \min_{0 \leq i < j}\left(D[i] + w(i,j)\right) & \text{otherwise;} \end{cases} \qquad (8.3.19)$$

where w is a real-valued function. This problem can be solved cache-obliviously in $\mathcal{O}\left(n^2\right)$ time, using $\mathcal{O}\left(n\right)$ space and incurring $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache misses using a one dimensional version of Recursive-Gap, provided either $w(i,j)$ is a function of $j - i$, or it can be generated on the fly in constant time without incurring any additional cache misses.

This problem arises in optimum paragraph formation and in finding a minimum height B-tree [57].

8.3.2 RNA Secondary Structure Prediction without Pseudoknots

As mentioned in Section 8.2.4, an RNA string $X = x_1 x_2 \ldots x_n$ forms complementary base pairs with itself, and often gives rise to interesting shapes with the following nesting condition: if $\{x_i, x_j\}$ and $\{x_k, x_l\}$ form base pairs and $i < j$, $k < l$ and $i < k$ hold then either $i < k < l < j$ or $i < j < k < l$ holds [129, 106, 6]. These shapes are known as RNA secondary structures (w/o pseudoknots). Given an RNA string, the *basic RNA secondary structure prediction* problem asks for an RNA secondary structure (w/o pseudoknots) with the maximum number of base pairs. Let $R[i, j]$ denote the maximum number of base pairs in a secondary structure formed by the RNA substrand $x_i x_{i+1} \ldots x_j$. Then the following recurrence can be used to compute $R[i, j]$ for all $i, j \in [1, n]$ [78].

$$R[i,j] = \begin{cases} 0 & \text{if } i \geq j-4, \\ \max\{\ R[i,j-1],\ P[i,j]\ \} & \text{otherwise;} \end{cases} \tag{8.3.20}$$

$$\text{where} \quad P[i,j] \quad = \max_{\substack{i<k<j-1, \\ \{x_k, x_j\} \text{ is a base pair}}} \left\{ \begin{array}{c} 1 \ + \ R[i,k-1] \\ + \ R[k+1,j-1] \end{array} \right\} \tag{8.3.21}$$

The above recurrence has the same structure as the recurrence for the *parenthesis problem* [57]. In Section 6.7.1 of Chapter 6 we show how to transform the dynamic program for the parenthesis problem into a sequence of dynamic programs in the *Gaussian Elimination Paradigm* (GEP), and thus solve the problem in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses. The resulting algorithm uses $\mathcal{O}\left(n^{1.75}\right)$ space in addition to the $\mathcal{O}\left(n^2\right)$ space for the input matrix. Therefore, we have the following claim.

Claim 8.3.1. *Given an RNA sequence of length n, a secondary structure (without pseudoknots) that has the maximum number of base pairs can be computed cache-obliviously in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses.*

8.4 Conclusion

In this chapter we presented a general cache-oblivious framework for a class of widely encountered dynamic programming problems with local dependencies, and applied it to obtain efficient cache-oblivious algorithms for several important string problems in bioinformatics, namely the longest common subsequence problem, global pairwise sequence alignment and median (both with affine gap costs), and RNA secondary structure prediction with simple pseudoknots. We show that our algorithms are theoretically more cache-efficient than the previous algorithms for these problems.

Our algorithms can be parallelized with very little effort and they show good parallel performance in practice. Our framework can also be applied to several other dynamic programming problems in bioinformatics including local alignment, generalized global alignment with intermittent similarities, multiple sequence alignment under several scoring functions such as 'sum-of-pairs' objective function and RNA secondary structure prediction with simple pseudoknots using energy functions based on adjacent base pairs.

We also presented sequential and parallel cache-oblivious algorithms for pairwise sequence alignment with general gap costs, and a sequential cache-oblivious algorithm for solving the basic RNA secondary structure (w/o pseudoknots) prediction problem.

Chapter 9

Experimental Results: Cache-oblivious DP for Bioinformatics

It doesn't matter how beautiful your theory is,
it doesn't matter how smart you are.
If it does not agree with experiment, it's wrong.
(Richard Feynman)

In this chapter we present experimental results on our cache-oblivious algorithms for pairwise sequence alignment, the median problem and RNA secondary structure prediction with simple pseudoknots presented in Chapter 8.

All our cache-oblivious code run faster on current desktop machines than the best previous implementations for these problems. Our empirical results also show good performance for the parallel implementations of our algorithms for the first two problems.

9.1 Introduction

In Chapter 8 we presented cache-oblivious sequential and parallel algorithms for several important dynamic programming problems in bioinformatics, and proved theoretical bounds on their performance. In this chapter we present experimental results on three of our cache-oblivious algorithms presented in Chapter 8, namely, pairwise sequence alignment, the median problem and RNA secondary structure prediction with simple pseudoknots. Our experimental results show that all our algorithms run faster on current desktop machines than the best previous algorithm for the problem. For the first two problems we compare our code to publicly available software written by others, and for the last problem our comparison is to our implementation of the best previous algorithm (due to Akutsu [6]). We also include empirical results showing good performance for the parallel implementations of our algorithms for the first two problems.

The experiments on the median problem were performed in collaboration with undergraduate student Hai-Son Le [87].

9.1.1 Organization of the Chapter

In Section 9.2 we describe our experimental setup, and in Section 9.3 we present our experimental results. In Sections 9.3.1, 9.3.2 and 9.3.3 we describe our experimental results on pairwise sequence alignment, the median problem and RNA secondary structure prediction with simple pseudoknots, respectively. Finally, in Section 9.4 we present some general remarks on our findings.

9.2 Experimental Setup

Machine	Processors	Speed	L1 Cache	L2 Cache	RAM
Intel P4 Xeon	2	3.06 GHz	8 KB (4-way, $B = 64$ B)	512 KB (8-way, $B = 64$ B)	4 GB
AMD Opteron 250	2	2.4 GHz	64 KB (2-way, $B = 64$ B)	1 MB (8-way, $B = 64$ B)	4 GB
AMD Opteron 850	8 (4 dual-core)	2.2 GHz	64 KB (2-way, $B = 64$ B)	1 MB (8-way, $B = 64$ B)	32 GB

Table 9.1: Machines used for experiments.

We ran our experiments on the machines listed in Table 9.1. All machines ran Ubuntu Linux 5.10. All our algorithms were implemented in C++ (compiled with *g++* 3.3.4), while some software packages we used for comparison were written in C (compiled with *gcc* 3.3.4). Optimization parameter -O3 was used in all cases. Each machine was exclusively used for experiments (i.e., no other programs were running on them). The *Cachegrind* profiler [112] was used for simulating cache effects.

9.3 Experimental Results

We describe our experimental results below.

9.3.1 Pairwise Global Sequence Alignment with Affine Gap Penalty

We performed experimental evaluation of the implementations listed in Table 9.2: PA-CO is our implementation of our linear-space cache-oblivious algorithm given in

Algorithm	Comments	Time	Space	Cache Misses
PA-CO	our cache-oblivious algorithm (see Section 8.2.4 of Chapter 8)	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n\right)$	$\mathcal{O}\left(\frac{n^2}{BM}\right)$
PA-FASTA	linear-space implementation of Gotoh's algorithm [93] available in *fasta2* [96]	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n\right)$	$\mathcal{O}\left(\frac{n^2}{B}\right)$

Table 9.2: Pairwise sequence alignment algorithms used in our experiments.

Section 8.2.4 of Chapter 8, and PA-FASTA is the implementation of the linear-space version of Gotoh's algorithm [93] available in the *fasta2* package [96].

In order to reduce the overhead of recursion in PA-CO, instead of stopping the recursion at $r = 1$ in Compute-Boundary-2D and Compute-Traceback-Path-2D, we stopped at $r = 256$, and solved the subproblem using the traditional iterative method.

Sequential Performance. We performed our experiments on AMD Opteron 250 and Intel P4 Xeon. On both machines only one processor was used.

In most cases PA-FASTA ran about 20%-30% slower than PA-CO on AMD Opteron and up to 10% slower on Intel Xeon. We summarize our results below.

Random Sequences. We ran both implementations on randomly generated equal-length string-pairs over $\{ A, C, G, T \}$ both on AMD Opteron 250 (see Figure 9.1(a)) and Intel P4 Xeon (see Figure 9.1(b)). We varied string lengths from 1 K to 512 K. In our experiments, PA-FASTA always ran slower than PA-CO on AMD Opteron (around 27% slower for sequences of length 512 K) and generally the relative performance of PA-CO improved over PA-FASTA as the length of the sequences increased. The trend was almost similar on Intel Xeon except that the improvement of PA-CO over PA-FASTA was more modest. We also obtained some anomalous results for $n \approx 10,000$ which we believe is due to architectural affects.

Real-World Sequences. We ran PA-CO and PA-FASTA on CFTR DNA sequences of lengths between 1.3 million to 1.8 million [119] on AMD Opteron, and PA-FASTA ran 20%-30% slower than PA-CO on these sequences (see Table 9.3). Though proper alignment of these genomic sequences require more sophisticated cost functions, running times of PA-CO and PA-FASTA on these sequences give us some idea on the relative performance of these implementations on very long sequences.

Cache Performance. We measured the number of L1 and L2 cache-misses incurred by both PA-FASTA and PA-CO on random sequences (see Figure 9.2). Though PA-FASTA causes fewer cache-misses than PA-CO when the input fits into the cache, it incurs significantly more misses than PA-CO as the input size grows beyond cache size. On AMD Opteron PA-FASTA incurs up to 300 times more L1 misses and 2500 times more L2 misses than PA-CO, while on Intel Xeon the figures are 10 and 1000, respectively. Observe that the larger the cache the larger the cache-miss

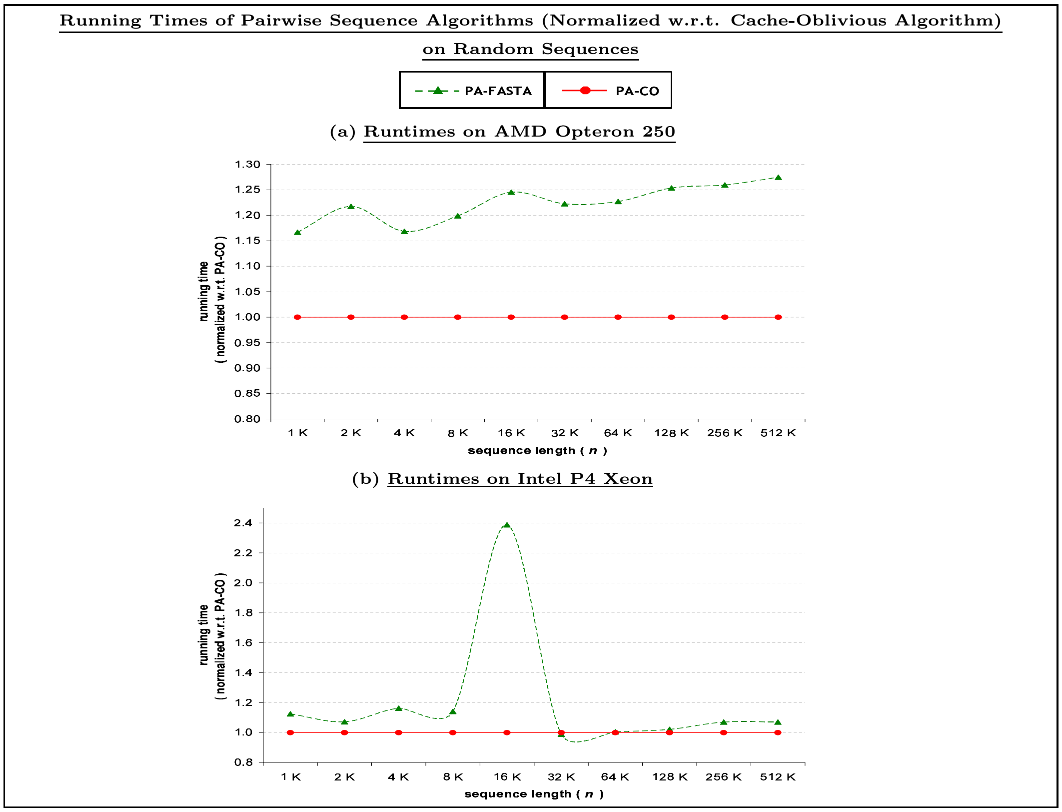

Figure 9.1: Comparison of running times of our cache-oblivious pairwise alignment algorithm (PA-CO) with the linear-space implementation of Gotoh's algorithm available in FASTA [96] (PA-FASTA). All running times are normalized w.r.t. that of PA-CO. Figure (a) plots running times on AMD Opteron 250 and Figure (b) on Intel P4 Xeon. Each data point is the average of 3 independent runs on randomly generated strings over $\{\ A, T, G, C\ \}$.

ratio which follows theoretical predictions since we know that PA-CO should incur fewer cache-misses on larger caches while cache performance of PA-FASTA should be independent of cache size.

Parallel Performance. In Figure 9.3 we plot the speed-up factors achieved by the parallel (multithreaded) implementation of PA-CO w.r.t. its sequential (unthreaded) implementation on an 8-processor AMD opteron 850. This is the straight-forward $\mathcal{O}\left(\frac{n^2}{p} + n^{\log_2 3}\log n\right)$ time parallel implementation described in the first paragraph of Section 8.2.3 in Chapter 8. The number of concurrent threads (i.e., the number of available processors p) was varied from 1 to 8, and the length of the sequences was varied from 8 K to 1024 K. We used the standard `pthreads` library for multi-threading. Threads were created recursively as the computation progressed, and the simple scheduling policy described in Section 7.3.3 of Chapter 7 was used.

The experimental results show that PA-CO achieves reasonable speed-up as

Sequence pairs with lengths (10^6)	Running times of pairwise alignment algorithms on *CFTR DNA Sequences* [119] (on AMD Opteron)		
	PA-FASTA (t_1)	PA-CO (t_2)	ratio $(\frac{t_1}{t_2})$
human/baboon (1.80/1.51)	$20h\ 34m$	$17h\ 23m$	1.18
human/chimp (1.80/1.32)	$19h\ 51m$	$15h\ 25m$	1.29
baboon/chimp (1.51/1.32)	$16h\ 43m$	$12h\ 43m$	1.31
human/rat (1.80/1.50)	$24h\ \ 1m$	$18h\ 16m$	1.31
rat/mouse (1.50/1.49)	$16h\ 49m$	$13h\ 55m$	1.21

Table 9.3: Comparison of running times (on AMD Opteron 250) of our cache-oblivious pairwise alignment algorithm (PA-CO in col 3) with the linear-space implementation of Gotoh's algorithm available in FASTA [96] (PA-FASTA in col 2) on CFTR DNA sequences [119]. Column 4 gives the ratio of the running time of PA-FASTA to that of PA-CO. Each number in columns 2 and 3 is the time for a single run.

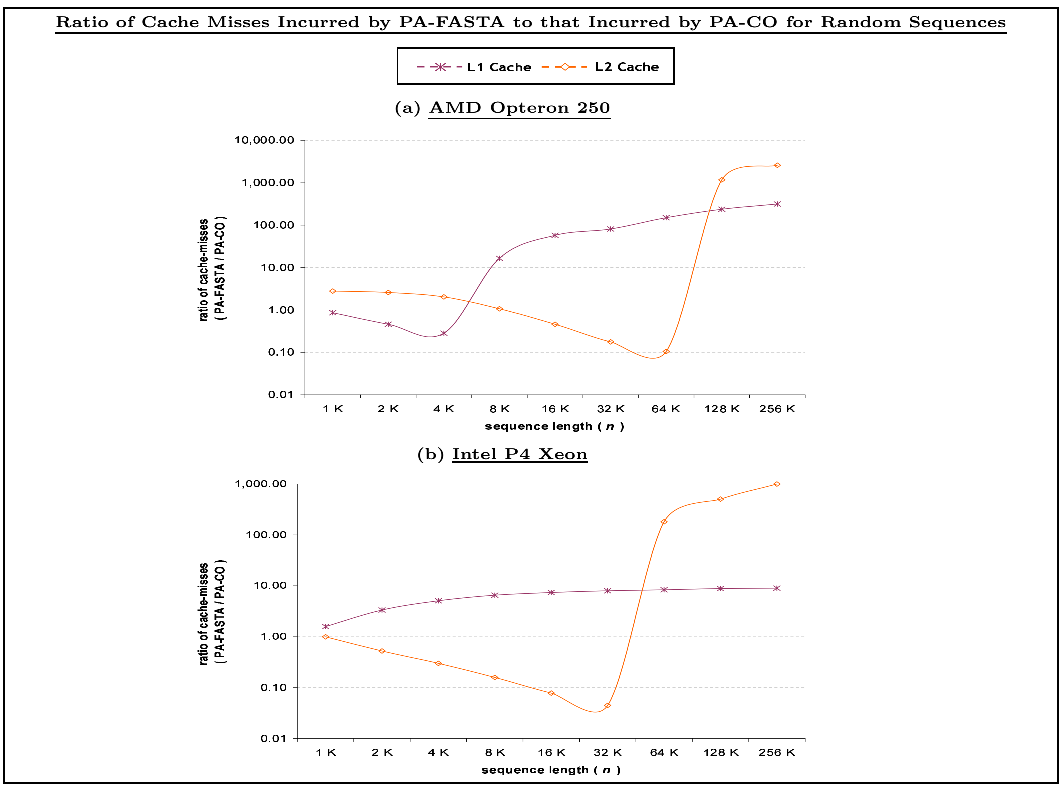

Figure 9.2: Ratio of cache-misses incurred by PA-FASTA to that incurred by PA-CO (see Table 9.2) on random sequences for both L1 and L2 caches. Data was obtained using Cachegrind [112].

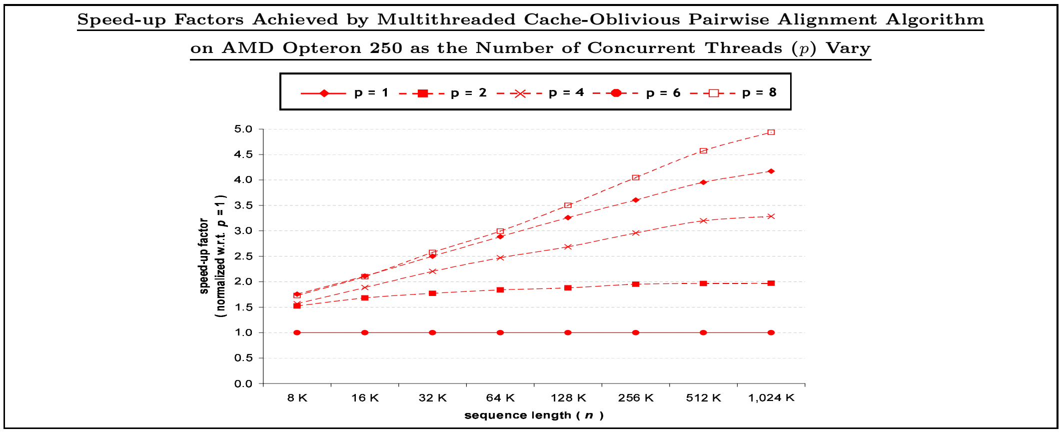

Figure 9.3: Speed-up factors (w.r.t. unthreaded code) achieved by multithreaded cache-oblivious pairwise alignment algorithm as number of threads (p) vary. The results were obtained on an 8-processor Opteron 850 with randomly generated strings over $\{ A, T, G, C \}$.

the number of processors increases, and for a fixed number of processors the speed-up factor improves as the length of the sequence grows. For example, with 8 processors the algorithm achieves a speed-up factor of 1.7 when $n = 8$ K, but achieves a speed-up factor of about 5 when $n = 1024$ K.

9.3.2 Median of Three Sequences

We performed experimental evaluation of the implementations listed in Table 9.4. Among the five implementations MED-CO implements our quadratic-space cache-oblivious median algorithm described in Section 8.2.4 of Chapter 8, MED-Knudsen is Knudsen's cubic-space median algorithm [80] implemented by Knudsen himself [79], and MED-H is our quadratic-space implementation (see Hai-Son Le's undergraduate honors thesis [87]) of Knudsen's algorithm based on Hirschberg's space-reduction technique. Between the remaining two MED-ukk.alloc is the $\mathcal{O}\left(n + \delta^3\right)$-space (where δ is the 3-way edit distance of sequences) Ukkonen-based median algorithm described in [101], and MED-ukk.checkp is the space-reduced version of MED-ukk.alloc based on checkpointing technique. Both algorithms were implemented by Powell [100],

In order to reduce the overhead of recursion in MED-CO, instead of stopping the recursion at $r = 1$ in COMPUTE-BOUNDARY-3D and COMPUTE-TRACEBACK-PATH-3D, we stopped the recursion at $r = 64$ on both Intel Xeon and AMD Opteron, and solved the problem by calling a sub-routine similar to Knudsen's algorithm at that point. Implementing MED-H, the quadratic-space version of Knudsen's algorithm based on Hirschberg's space-reduction technique, was substantially difficult compared to our quadratic-space algorithm MED-CO. Detailed discussion of the difficulties in implementing MED-H can be found in Hai-Son Le's undergraduate

Algorithm	Comments	Time	Space	Cache Misses
MED-CO	our cache-oblivious algorithm (see Section 8.2.4 of Chapter 8)	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$
MED-Knudsen	Knudsen's implementation of his algorithm [79]	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(\frac{n^3}{B}\right)$
MED-H	our implementation [87] of MED-Knudsen with Hirschberg's space-reduction	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^3}{B}\right)$
MED-ukk.alloc	Powell's implementation [100] of an $\mathcal{O}\left(\delta^3\right)$-space algorithm ($\delta$ = 3-way edit dist. of sequences)	$\mathcal{O}\left(n+\delta^3\right)$ (avg.)	$\mathcal{O}\left(n+\delta^3\right)$	$\mathcal{O}\left(\frac{\delta^3}{B}\right)$
MED-ukk.checkp	Powell's implementation [100] of an $\mathcal{O}\left(\delta^2\right)$-space algorithm ($\delta$ = 3-way edit dist. of sequences)	$\mathcal{O}\begin{pmatrix} n\log\delta \\ +\delta^3 \end{pmatrix}$ (avg.)	$\mathcal{O}\left(n+\delta^2\right)$	$\mathcal{O}\left(\frac{\delta^3}{B}\right)$

Table 9.4: Median algorithms used in our experiments.

honors thesis [87].

We will first compare the sequential performance of our cache-oblivious algorithm MED-CO with the three publicly available code: MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp. Then we will compare the performance MED-CO and MED-Knudsen with our reduced-space implementation of Knudsen's algorithm (MED-H) in order to study the effects of space-reduction and cache-efficiency on the performance of MED-Knudsen.

Sequential Performance. We performed our experiments on AMD Opteron 250 and Intel P4 Xeon. Only a single processor was used on each machine.

We used a gap insertion cost of 3 (i.e., $g_i = 3$), a gap extension cost of 1 (i.e., $g_e = 1$) and a mismatch cost of 1 in all experiments.

Overall, MED-Knudsen ran about 1.5-2.5 times slower than MED-CO on both machines, and MED-ukk.alloc and MED-ukk.checkp were even slower. Furthermore, due to their high space overhead MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp could not be run on any machine for sequences longer than 640. We summarize our results below.

Random Sequences. We ran all implementations on random (equal-length) sequences of length $64i$, $1 \le i \le 16$ on AMD Opteron (see Figure 9.4(a)) and Intel Xeon (see Figure 9.4(b)). On both machines MED-CO ran the fastest, and MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp crashed as they ran out of memory for sequences longer than 384, 256 and 640, respectively.

On Intel Xeon, MED-CO was the fastest. It ran at least 1.45 times faster than MED-Knudsen. Both MED-ukk.alloc and MED-ukk.checkp ran at least 2 times slower than MED-CO for length 64, and continued to slow down even further with increasing sequence length. They ran up to 3.3 times (for length 256) and 4.8 times

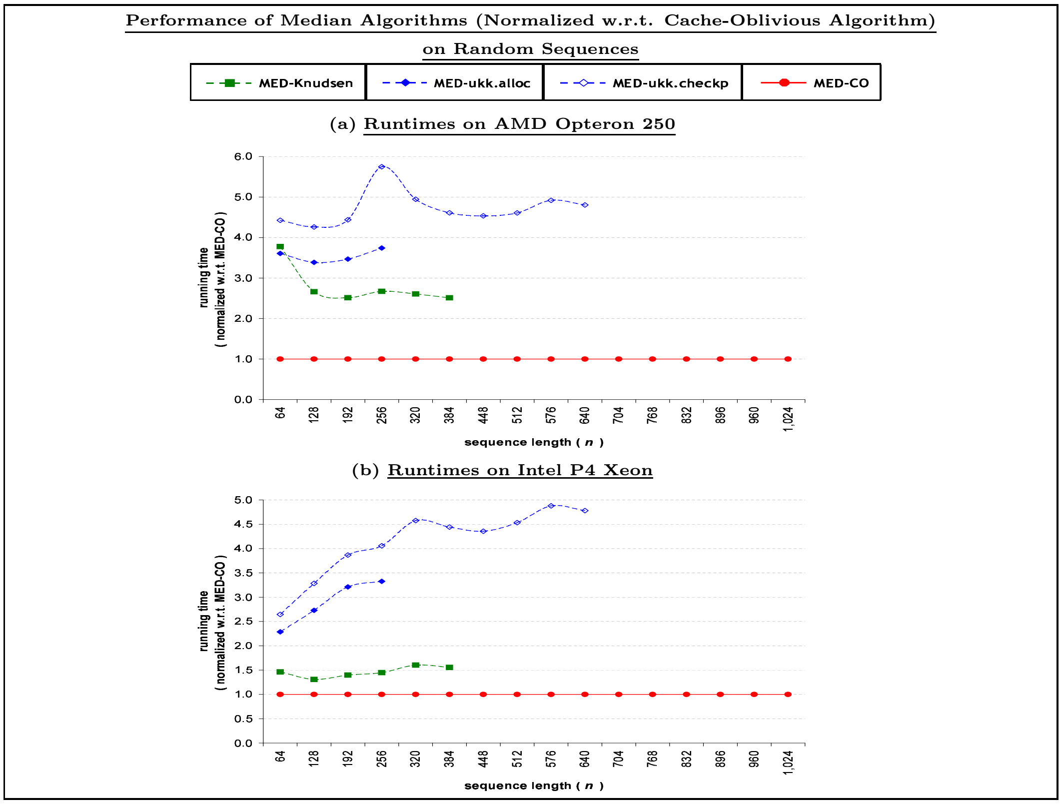

Figure 9.4: Comparison of performance of our cache-oblivious median algorithm (MED-CO) with Knudsen's implementation of his algorithm [80] available in [79] (MED-Knudsen), and Powell's implementation of two algorithms [101] available in [100] (MED-ukk.alloc and MED-ukk.checkp). Figures (a) and (b) plot running times of the algorithms on AMD Opteron 250 and Intel P4 Xeon, respectively. figure for MED-CO. Due to their high space overhead MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp could not be run for sequences longer than 640 on any machine. Each data point is the average of 3 independent runs on random strings over $\{ A, T, G, C \}$.

(for length 640) slower than MED-CO, respectively. The trends were similar on AMD Opteron and MED-CO ran at least 2.5, 3.4 and 4.2 times faster than MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp, respectively.

Real-World Sequences. We ran all algorithms (except MED-H) on triplets of 16S bacterial rDNA sequences from the *Pseudanabaena group* [42] (see Table 9.5). All experiments were run on Intel Xeon.

Triplets 1–5 in Table 9.5 were formed by choosing three sequences of length less than 500 from the group at random, while triplet 6 was formed manually for reasons to be explained in the next paragraph. On triplets 1–5, MED-Knudsen ran around 35–50% slower and MED-ukk.checkp up to 3.2 times slower than MED-CO.

	Running times (in sec) on Intel Xeon for random triples of *16S Bacterial rDNA Sequences* from the *Pseudanabaena Group* [42] (runtime w.r.t. MED-CO)					
No.	Lengths	Alignment Cost	MED-Knudsen	MED-ukk.alloc	MED-ukk.checkp	MED-CO
1	367, 387, 388	299	722 (1.48)	512 (1.05)	601 (1.23)	487 (1.00)
2	378, 388, 403	324	752 (1.42)	− (−)	769 (1.45)	529 (1.00)
3	342, 367, 389	339	611 (1.35)	− (−)	863 (1.91)	451 (1.00)
4	342, 370, 474	432	764 (1.44)	− (−)	1,701 (3.20)	531 (1.00)
5	370, 388, 447	336	− (−)	− (−)	824 (1.49)	553 (1.00)
6	367, 388, 389	260	695 (1.42)	330 (0.67)	380 (0.77)	491 (1.00)

Table 9.5: Comparison of running times (on Intel Xeon) of four algorithms on 16S bacterial rDNA sequences from the Pseudanabaena group [42]: our cache-oblivious median algorithm (MED-CO in col 7), Knudsen's implementation of his algorithm [79] (MED-Knudsen in col 4), and Powell's implementation of two Ukkonen-based algorithms [100] (MED-ukk.alloc and MED-ukk.checkp in cols 5 and 6, respectively). Triplets 1–5 were formed by choosing random sequences of length less than 500 from the group while triplet 6 was formed by choosing sequences manually in order to keep the alignment cost small. Each number outside parentheses in columns 4–7 is the time for a single run, and the ratio of that running time to the corresponding running time for MED-CO is given within parentheses. A '−' in a column denotes that the corresponding algorithm could not be run due to high space overhead.

Running time of MED-ukk.checkp w.r.t. MED-CO degraded as the alignment cost increased. MED-ukk.alloc which requires space cubic in the alignment cost could not be run on triplets with alignment cost larger than 299, that is, on triplets 2–5. On triplet 5 MED-Knudsen also crashed due to its high space requirement which is cubic in the sequence length.

The sequences in triplet 6 were chosen manually in order to keep their alignment cost small. We used this triplet in order to verify the theoretical prediction that MED-ukk.alloc and MED-ukk.checkp would run faster on triplets with small alignment costs since unlike MED-CO whose running time is cubic in the sequence length, running times of those two algorithms are cubic in the 3-way edit distance of the input sequences (see Table 9.4). The alignment cost of triplet 6 is 260, and as Table 9.4 shows both MED-ukk.alloc and MED-ukk.checkp, indeed, ran faster than MED-CO on this triplet.

Overall, our experimental results suggest that MED-CO is always a better choice than MED-Knudsen, and a better choice than the two Ukkonen-based algorithms (MED-ukk.alloc and MED-ukk.checkp) when the alignment cost is moderately large.

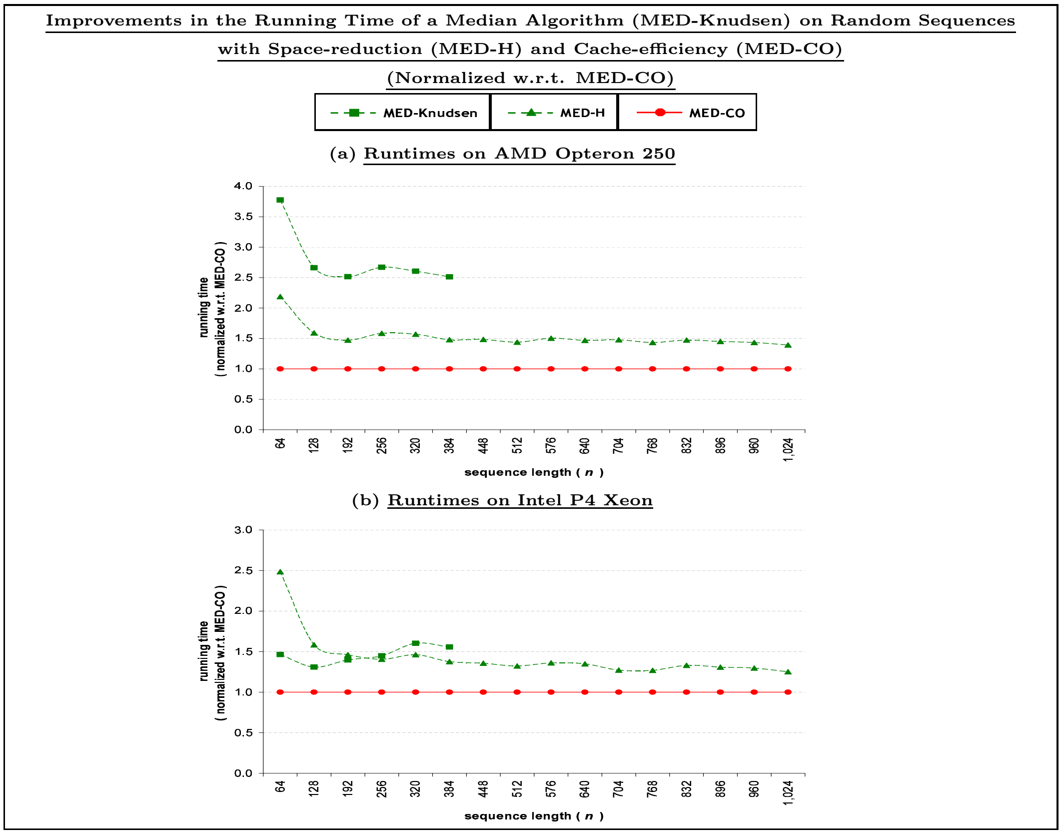

Figure 9.5: Improvements in the performance of a median algorithm (i.e., MED-Knudsen: Knudsen's implementation of his algorithm [80]) as its space requirement is reduced (with a Hirschberg-style implementation of Knudsen's algorithm (MED-H)), and as both its space usage and cache performance are improved using our cache-oblivious median algorithm (MED-CO). Figures (a) and (b) plot running times of the algorithms on AMD Opteron 250 and Intel P4 Xeon, respectively. Each data point is the average of 3 independent runs on random strings over $\{\ A, T, G, C\ \}$.

Effects of Space-reduction and Cache-efficiency. In Figures 9.5(a) and 9.5(b) we plot the running times of Knudsen's algorithm (MED-Knudsen), a Hirschberg-style space-reduced version of the same algorithm (MED-H), and our space-efficient cache-oblivious algorithm (MED-CO) on random sequences. As the plots show, after simply reducing the space usage from $\mathcal{O}\left(n^3\right)$ (MED-Knudsen) to $\mathcal{O}\left(n^2\right)$ (MED-H), the median algorithm runs faster and can handle much longer sequences. For space-intensive algorithms reducing space usage can improve its cache performance significantly since now the data fits in lower cache levels and thus incurs fewer cache-misses. The plots also show that we can improve the running time of the algorithm even further by improving its cache performance even more, that is, using our space-efficient cache-oblivious algorithm. On AMD Opteron running time of Knudsen's algorithm improves by 40% after space-reduction (MED-H), and a further 30% after improving the cache-efficiency using our cache-oblivious implementation (MED-CO). Similar trend is also observed on Intel P4 Xeon.

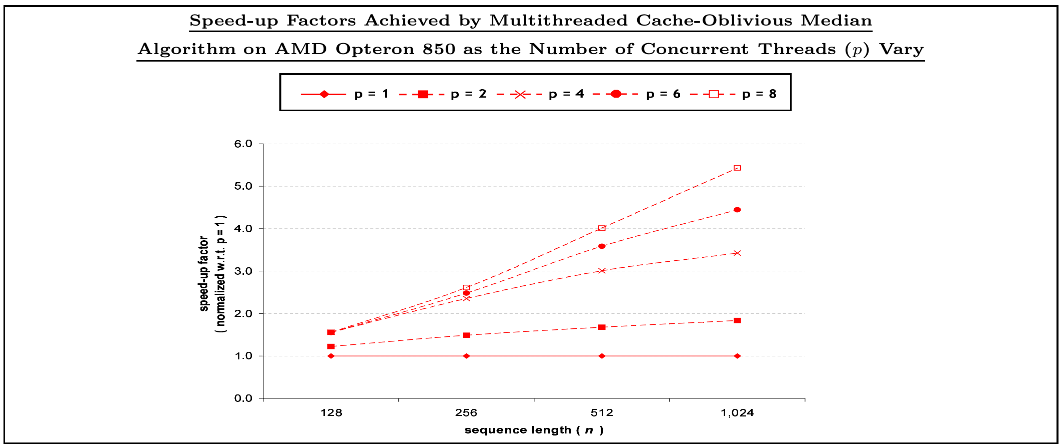

Figure 9.6: Speed-up factors (w.r.t. unthreaded code) achieved by multithreaded cache-oblivious median algorithm on 8-processor Opteron 850 as number of threads (p) vary. Sequences were randomly generated strings over $\{ A, T, G, C \}$.

Parallel Performance. Figure 9.6 plots the speed-up factors achieved by multi-threaded (parallel) MED-CO w.r.t. its sequential implementation on an 8-processor AMD opteron 850. We implemented the simple $\mathcal{O}\left(\frac{n^3}{p} + n^2 \log n\right)$ time parallel algorithm described in the first paragraph of Section 8.2.3 in Chapter 8. The number of concurrent threads (i.e., the number of available processors p) was varied from 1 to 8, and the sequences lengths from 2^7 (128) to 2^{10} (1024). The multithreaded code was implemented using the standard `pthreads` library. We used the strategy described in Section 7.3.3 of Chapter 7 for creating and scheduling threads.

Experimental results show that MED-CO speeds up better than the cache-oblivious pairwise alignment algorithm PA-CO in Section 9.3.1 as the number of processors grows. For example, with 8 processors MED-CO achieves a speed-up factor of around 5.5 when the sequence length is only 1024, while PA-CO requires sequences of length 1024 K in order to reach a speed-up factor of 5 (see Figure 9.3).

9.3.3 RNA Secondary Structure Prediction with Pseudoknots

We implemented the algorithms in Table 9.6 (all are sequential) for computing all values of S_{pseudo} or S'_{pseudo} (i.e., we compute the optimal scores only, we do not traceback the pseudoknots). We ran all experiments on Intel Xeon using a single processor.

Algorithm	Comments	Time	Space	Cache Misses
RNA-CO	our cache-oblivious algorithm (see Section 8.2.4 of Chapter 8)	$\mathcal{O}\left(n^4\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$
RNA-CS	Akutsu's original cubic-space algorithm [6]	$\mathcal{O}\left(n^4\right)$	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(\frac{n^4}{B}\right)$
RBA-QS	Our iterative quadratic-space version of Akutsu's algorithm (see Section 8.2.4 of Chapter 8)	$\mathcal{O}\left(n^4\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^4}{B}\right)$

Table 9.6: RNA secondary structure prediction algorithms used in our experiments.

In order to reduce the overhead of recursion in RNA-CO, instead of executing line 1 of the algorithm for $r = 1$ (see Figure 8.2 in Chapter 8), we stopped as soon as we reached $r \leq 64$ and solved the problem directly using our iterative quadratic-space variant RNA-QS.

Overall RNA-QS ran about 50% slower than RNA-CO and RNA-CS ran up to 7 times slower than RNA-CO for sequence lengths it could handle. For sequences longer than 512 RNA-CS could not be run due to lack of memory space. We summarize our results below.

Random Sequences. We ran all three algorithms on randomly generated string-pairs over $\{\ A, U, G, C\ \}$ (see Figure 9.7). The lengths of the strings were varied from 64 to 2048. However, due to lack of space RNA-CS could not be run for strings longer than 512. In our experiments RNA-CO ran the fastest while RNA-CS was the slowest. Though both RNA-QS and RNA-CS have the same time and cache-complexity, RNA-QS ran significantly faster than RNA-CS (e.g., ≈ 4.5 times faster for length 512). We believe this happened because even for small sequence lengths RNA-CS overflows the L2 cache, and most of its data reside in the slower RAM, while both RNA-QS and RNA-CO still work completely inside the faster L2 cache. For strings of length 512 RNA-CO ran about 35% faster than RNA-QS and about 7 times faster than RNA-CS. The performance of RNA-CO improved over that of both RNA-CS and RNA-QS as the length increased.

Real-World Sequences. We ran all three implementations on a set of 24 bacterial 5S rRNA sequences obtained from [27]. The average length of the sequences was 118, and the average running times of RNA-CS, RNA-QS and RNA-CO on each sequence were 1.46 sec, 0.45 sec and 0.35 sec, respectively. We also ran RNA-QS and RNA-CO on a set of 10 bacterial (spirochaetes) 16S rRNA sequences of average

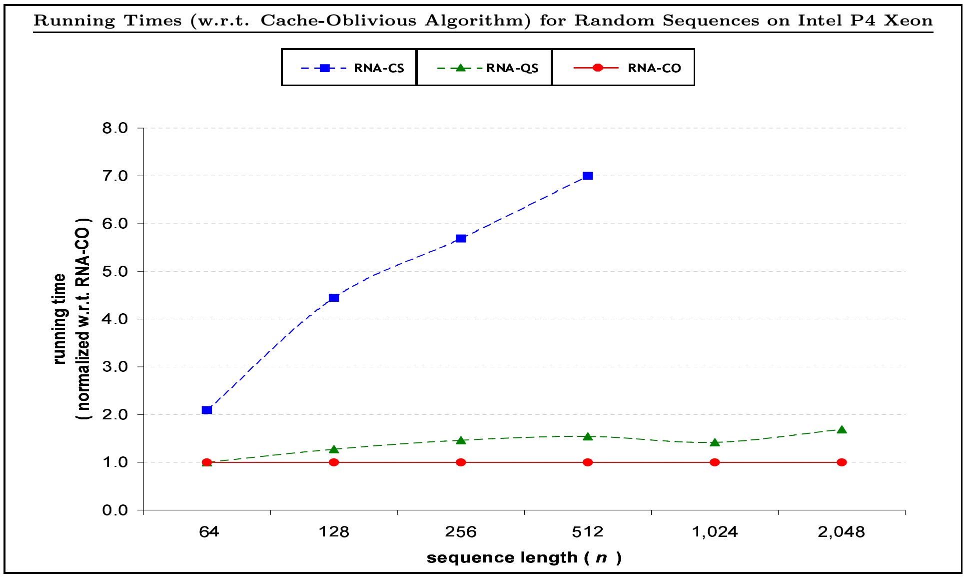

Figure 9.7: Comparison of running times of three algorithms for RNA secondary structure prediction with simple pseudoknots on Intel P4 Xeon: our cache-oblivious algorithm (RNA-CO), Akutsu's algorithm [6] (RNA-CS) and our quadratic-space version of Akutsu's algorithm (RNA-QS). All running times are normalized w.r.t. RNA-CO. Due to its high space overhead RNA-CS could not be run for sequences longer than 512. Each data point is the average of 3 independent runs on random strings over $\{ A, U, G, C \}$.

length 1509. The RNA-CS implementation could not be run on these sequences due to space limitations. On these sequences RNA-CO took 1 hour 38 minutes while RNA-QS took 2 hours 38 minutes on the average (see Table 9.3.3).

9.4 Discussion

We observed that the method used to implement the base case of a cache oblivious algorithm affects its performance. In order to reduce the overhead of recursion the base case is implemented using a nonrecursive algorithm (typically the traditional iterative dynamic programming algorithm). If the traceback path is not required the traditional iterative DP can be easily implemented to use a factor of n less space than the case when the traceback path is required (see [37]). That's why the base case of RNA-CO (an implementation of our RNA secondary structure prediction algorithm which does not compute a traceback path) was implemented using a quadratic-space algorithm while that of MED-CO (our median algorithm that computes a traceback path) was implemented using a cubic-space algorithm. Though the size of the base case is a constant and thus does not affect asymptotic running times, execution of

Runtimes of algorithms for RNA secondary structure prediction with simple pseudoknots on Intel Xeon for *Bacterial (Spirochaetes) 16S rRNA Sequences* [27]				
Organism	Length (n)	Quadratic Space (RNA-QS)	Cache-oblivious (RNA-CO)	$\frac{\text{RNA-QS}}{\text{RNA-CO}}$
Brevinema andersonii	1443	$2h$ $14m$	$1h$ $22m$	1.64
Borrelia burgdorferi	1530	$2h$ $48m$	$1h$ $44m$	1.62
Borrelia burgdorferi	1537	$2h$ $48m$	$1h$ $45m$	1.60
Borrelia hermsii	1523	$2h$ $43m$	$1h$ $41m$	1.61
Brachyspira hyodysenteriae	1463	$2h$ $21m$	$1h$ $27m$	1.61
Cristispira CP1	1491	$2h$ $30m$	$1h$ $33m$	1.62
Leptonema illini	1526	$2h$ $45m$	$1h$ $42m$	1.61
Leptospira interrogans	1508	$2h$ $37m$	$1h$ $38m$	1.61
Spirochaeta aurantia	1520	$2h$ $42m$	$1h$ $41m$	1.60
Treponema pallidum (rRNA A)	1549	$2h$ $54m$	$1h$ $48m$	1.60
Average	1509	$2h$ $38m$	$1h$ $38m$	1.61

Table 9.7: Comparison of running times of two algorithms for RNA secondary structure prediction with simple pseudoknots on Intel P4 Xeon: our cache-oblivious algorithm (RNA-CO), and our quadratic-space version of Akutsu's algorithm (RNA-QS). Inputs were Bacterial (Spirochaetes) 16S rRNA sequences [27] with an average length of 1509. Akutsu's cubic space algorithm could not be run because these sequences are too long for it. Each number in columns 2 and 3 represents time for a single run.

the base case can flood the smaller cache levels if it uses a space-intensive algorithm and thus can increase the running time of the algorithm by degrading its cache performance. We believe this is one of the reasons why the performance MED-CO relative to its counterparts was not as impressive as that of RNA-CO. However, we also implemented the base case of MED-CO using a Hirschberg-style quadratic-space implementation of the standard cubic-space DP, and our experimental results (not included in this dissertation) show that the space-reduced base case does not improve the running time of MED-CO. We believe this happens because for small problem sizes such as the base case the overhead of the Hirschberg-based implementation dominates its running time and consequently it fails to improve over the running time of the simple iterative DP.

We believe that in addition to being faster and more cache-efficient our cache-oblivious algorithms are simpler to implement than Hirschberg's space-reduction technique. As explained in Section 8.2.1 of Chapter 8, complicated recurrence relations and recurrence relations with multiple fields increase the difficulties of implementing Hirschberg's algorithm, but they do not complicate the implementation of our algorithms.

One of the major advantages of our cache-oblivious algorithms is that they are parallelizable with very little extra effort. In contrast, iterative algorithms such as MED-Knudsen and MED-ukk.alloc (see Section 9.3.2) are not parallelizable without substantial modifications. Though algorithms obtained using Hirschberg-style space-reduction techniques (e.g., PA-FASTA in Section 9.3.1, and MED-H and MED-ukk.checkp in Section 9.3.2) can be parallelized, the computation-space is divided only into two subproblems and substantial amount of sequential computation is performed before the next level of recursive subdivision. The sequential computation along with the fact that the decomposition is often unbalanced drastically reduces the amount of parallelism. Moreover, as described above, it is not always easy to apply Hirschberg's technique.

Chapter 10

Conclusion

The Lord of the path benignly smiles: Silly boy, my path does not end at the bamboo groves of your village, or below the banyan tree of Biru Roy the bandit, or at the crossing of the Dhalchite. Beyond the farmlands of Sonadanga, across the Ichamati, skirting around the lagoons of Modhukhali – filled with lotuses – crossing the Betravati ferry – my pathway spreads on and on – from land to land, from sunrise to sunset, from the familiar to the obscure. Let's move on.

(Bibhutibhusan Banerjee in *Pather Panchali*, adapted from a translation by Naresh Guha)

In this final chapter we summarize the major ideas and results of this thesis, and also discuss some future research directions.

10.1 Summary

In this dissertation we focussed on developing cache-efficient and cache-oblivious algorithms and data structures for problems in three separate domains: graph problems, problems in the Gaussian Elimination Paradigm (GEP), and problems with dynamic programming algorithms. We concentrated on shortest path problems in the first domain, and for computation-intensive problems in the last two domains we also concentrated on obtaining efficient parallel cache-oblivious algorithms. For problems in each domain, we performed extensive experimental evaluation of many of our algorithms against best known existing algorithms.

One of the key problems in cache-oblivious shortest paths computation is the lack of cache-oblivious priority queues supporting *Decrease-Key* operations. We introduced the first cache-oblivious priority queue with *Decrease-Keys*, and used it to obtain the first non-trivial cache-oblivious single-source shortest path algorithms for both directed and undirected weighted graphs. Experimental results suggest that shortest path algorithms based on a variant of our cache-oblivious priority queue run faster than algorithms based on highly optimized traditional priority queues.

We considered the all-pairs shortest path problem on both weighted and unweighted graphs. We used various techniques to reduce unstructured accesses to adjacency lists, and consequently obtained all-pairs shortest path algorithms with improved cache performance.

For the Gaussian Elimination Paradigm (GEP), we presented a general framework for efficient cache-oblivious execution of any problem in GEP including Gaussian elimination w/o pivoting, Floyd-Warshall's APSP and matrix multiplication. We also presented a parallel implementation of our framework, and provided scheduling schemes for its cache-efficient execution on both distributed and shared caches separately. We performed empirical comparison of several cache-oblivious algorithms obtained using our framework with high-performance industrial-strength cache-aware code. The results suggest that our cache-oblivious framework offers very good performance along with simplicity and portability.

In the dynamic programming domain, we presented a general cache-oblivious framework that gives efficient cache-oblivious sequential and parallel algorithms for a number of important dynamic programming problems in bioinformatics including *optimal pairwise global sequence alignment* and *median of three sequences* (both with affine gap costs), and *RNA secondary structure prediction with simple pseudoknots*. We also developed cache-oblivious sequential and parallel algorithms for *optimal pairwise alignment with general gap costs*. All our algorithms improve significantly over the cache-efficiency of earlier algorithms. We empirically compared most of our algorithms with the best publicly available code written by others, and observed that our algorithms run faster than these software.

In our experimental study of all three problem domains we considered, our cache-oblivious algorithms and data structures consistently outperformed their traditional flat-memory counterparts. In the first two domains we also compared our implementations with highly optimized cache-aware sequence heap and BLAS (Basic Linear Algebra Subprograms) routines, respectively, while no such high performance cache-aware softwares were available for the problems in the third domain. Though these two cache-aware implementations performed better than our cache-oblivious implementations, the performance gaps were always within reasonable limits, and additionally our algorithms and data structures were easier to implement and more portable. Thus our cache-oblivious results offer a very useful tradeoff between efficiency on one hand, and simplicity and portability on the other.

10.2 Future Work

In this section we discuss several directions for further research to extend our work presented in this thesis.

Cache-efficient Dynamic Graph Algorithms.

A *dynamic graph algorithm* maintains a data structure on a graph supporting two types of operations: *updates* and *queries*. An update is a local change to the graph (for example: edge insertion/deletion, or change of an edge-weight), and a query is a question about a certain property of the current graph (for example: "what is the shortest distance from node u to node n in the current graph?"). The objective is to maintain structural information about the current graph in order to handle updates and queries faster than recomputing from scratch.

Many real-world massive graphs (e.g., the web graph) are continuously changing in nature. Therefore, I/O-efficient *fully dynamic* (i.m., supporting both insertion and deletion of nodes/edges) algorithms are needed to handle these graphs. At this time, no external-memory algorithms are known for fundamental dynamic graph problems except for a fully-dynamic undirected MSF algorithm given in [64, 65] that requires either $\mathcal{O}\left(\sqrt{\frac{n}{B}}\right)$ or $\mathcal{O}\left(sort\left(\sqrt{n\log_2 B}\right)\right)$ I/Os per update in the worst case. No results are known for the cache-oblivious model.

If a dynamic algorithm must answer a query as soon as it receives it, an adversary can always choose a sufficiently large sequence of updates and queries that will force the algorithm to incur $\Omega(1)$ (even in amortized sense) cache-misses per query. Therefore, unlike some external-memory data structures used by static graph algorithms, such as external-memory priority queues, an external-memory dynamic graph data structure cannot support queries in $o(1)$ amortized cache-misses. An open question is whether this $\Omega(1)$ I/O barrier can be overcome if we allow only batched queries, or only allow query sequences that can be decomposed into sufficiently large subsequences of queries with high locality.

Automatic Generation of Cache-oblivious DP Algorithms.

Dynamic programs occur so frequently in practice that instead of trying to obtain an efficient cache-oblivious implementation for each of them individually, it is natural to ask the following question:

Does there exist a general method that given the recurrence relation associated with any dynamic programming problem, can produce an efficient cache-oblivious version of it?

This method must be able to

(*a*) decompose the given problem into smaller subproblems,

(*b*) determine any inter-dependence among the subproblems, and the order in which the subproblems must be solved respecting their inter-dependence, and

(*c*) ensure that there is a separation between the space complexity and the time complexity of the algorithm, i.e., the space complexity is asymptotically smaller than the time complexity (since otherwise there will be no temporal locality in memory accesses).

Any such method is likely to have enormous practical value.

Automating Cache-obliviousness - Compiler Optimizations for the Memory Hierarchy.

We are used to writing programs for the *flat memory model* which is arguably simpler than writing programs for the cache-oblivious model, but is often very inefficient in its cache usage. One way of having the best from both worlds is to allow the users write code assuming a flat memory, and then use a compiler to port this code to the cache-oblivious model. Unfortunately, however, though modern optimizing compilers employ optimization techniques for improved cache-efficiency, they often fail to produce portable system-independent (i.e., cache-oblivious) code [91, 133]. In contrast, cache-oblivious compiler optimizations will not only produce portable code, the resulting code will also use all levels of a memory hierarchy efficiently.

Towards Resource-obliviousness - Parallel Cache-oblivious Model.

For increased efficiency through increased level of parallelism, computation is often performed on multi-processor, multi-disk systems. Very recently, the cache-oblivious model has been extended to analyze multithreaded cache-oblivious algorithms run on multi-processor parallel machines both with shared caches [18] and with distributed caches [55]. In Chapters 6 and 8 we presented efficient parallel cache-oblivious algorithms for shared and distributed caches separately. But modern multi-core architectures include both distributed (L1) and shared (L2) caches. Therefore, an important open problem is to combine these two extreme cache models into a single unified model, and thereby formulate a robust model for cache-oblivious computation on multi-processor, multi-disk systems.

Appendix A

The Cache-oblivious Tournament Tree

In Section 3.4.2 of Chapter 3 we used the cache-oblivious buffer heap as a priority queue in order to solve the directed SSSP problem cache-obliviously. Here, we present the *cache-oblivious tournament tree* (COTT) which is a priority queue supporting the same set of operations (*Delete*, *Delete-Min* and *Decrease-Key*) as the buffer heap. Although our bounds for COTT are weaker than those for buffer heap, COTT is a simpler data structure, and may be more amenable to practical implementation. Our directed SSSP algorithm runs just as efficiently with COTT as with buffer heap.

COTT is a cache-oblivious version of the cache-aware tournament tree introduced by Kumar and Schwabe in [83]. This structure can contain only a predetermined set of elements which are initially inserted into fixed positions in the structure with $+\infty$ key value. While the cache-aware tournament tree of [83] with N elements supports a sequence of k *Delete*, *Delete-Min* and *Decrease-Key* operations in at most $\mathcal{O}\left(\frac{k}{B}\log_2\frac{N}{B}\right)$ cache-misses, our cache-oblivious version supports *Delete* and *Delete-Min* in $\mathcal{O}\left(\log_2\frac{N}{B}\right)$, and *Decrease-Key* operations in $\mathcal{O}\left(\frac{1}{B}\log_2\frac{N}{B}\right)$ amortized cache-misses, respectively, under the tall cache assumption.

A COTT with N elements is a static binary tree with N leaves numbered 1 through N from left to right. The root of the tree is denoted by R, and T_v denotes the subtree rooted at any node v. Each node v stores an ordered pair (x_v, k_v), where x_v is an element corresponding to a leaf in T_v and k_v is a key of x_v. Each internal node v has an associated stack S_v.

The supported operations are propagated lazily from the root to appropriate leaves and the following two invariants are maintained:

Invariant A.0.1. *For each internal node v, each entry in S_v is a* Decrease-Key *operation to be performed on a leaf of T_v. Thus the stacks associated with the nodes on the path from a leaf to the root together contain all* Decrease-Keys *to be performed*

Function **A.0.1.** Delete(x)

[Delete element x from the COTT if exists.]

1. (*Distribution Step*) Follow the path from the root to the leaf corresponding to x. At each internal node v with children v_1 and v_2 on this path pop all *Decrease-Key* operations from S_v and distribute them to S_{v_1} and S_{v_2}, and also update (x_{v_1}, k_{v_1}) and (x_{v_2}, k_{v_2}) if necessary. However, if a child node is a leaf, discard any *Decrease-Key* operation it receives after updating the ordered pair stored in that leaf.
2. (*Fixing Step*) Set the key value of element x at the leaf to $+\infty$ and propagate this change along the leaf to root path from x. At each internal node v with children v_1 and v_2 on this path, set (x_v, k_v) to (x_{v_1}, k_{v_1}) if $k_{v_1} \leq k_{v_2}$, otherwise set it to (x_{v_2}, k_{v_2}).

Delete Ends

on that leaf.

Invariant A.0.2. *The ordered pair (x_v, k_v) stored in node v refers to the element x_v corresponding to a leaf in T_v, having the minimum key k_v taking into account only the operations (*Delete, Delete-Min *and* Decrease-Key*) seen by v so far. Thus, at any time, the root of the tree stores the element with the minimum key value in the whole tree.*

Initially, the elements in all leaves are assigned a key value of $+\infty$. All stacks are empty and the ordered pair in each internal node refers to the leftmost leaf in the corresponding subtree. Thus both invariants hold initially.

Decrease-Keys operations are performed lazily. Whenever a *Decrease-Key* operation arrives, it is pushed on to the stack S_R of the root node and the minimum value at the root is updated if necessary. Thus, invariants A.0.1 and A.0.2 are maintained.

A *Delete-Min* operation is a special case of the *Delete* operation: it reads the ordered pair (x_R, k_R) from the root of the tree and performs a *Delete*(x_R) operation. A *Delete*(x) operation is performed as follows.

Lemma A.0.1. *A COTT with N elements supports* Delete/Delete-Min *and* Decrease-Key *operations.*

(a) cache-obliviously in $\mathcal{O}(\log_2 N)$ and $\mathcal{O}\left(\frac{1}{B}\log_2 N\right)$ amortized cache-misses, respectively, without a tall cache.

(b) cache-obliviously in $\mathcal{O}\left(\log_2 \frac{N}{B}\right)$ and $\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{B}\right)$ amortized cache-misses, respectively, with a tall cache.

(c) in $\mathcal{O}\left(\log_2 \frac{NB}{M}\right)$ and $\mathcal{O}\left(\frac{1}{B}\log_2 \frac{NB}{M}\right)$ amortized cache-misses, respectively, in the two-level I/O (cache-aware) model.

Proof.

(a) During a *Delete* or *Delete-Min* operation 2 nodes are accessed at each of the $\mathcal{O}(\log_2 N)$ levels during the root to leaf path traversal in step 1 (distribution step) and the same holds for step 2 (fixing step). We charge the $\mathcal{O}(1)$ cache-misses for retrieving the node information and the stacks in each level to the current *Delete*/*Delete-Min* operation. Thus each *Delete*/*Delete-Min* is charged for $\mathcal{O}(\log_2 N)$ cache-misses in total. On the other hand, each *Decrease-Key* operation is pushed and popped to a stack only once in each level which incurs $\mathcal{O}\left(\frac{1}{B}\right)$ cache-misses per level. Each *Decrease-Key* operation also incurs $\mathcal{O}\left(\frac{1}{B}\right)$ amortized cache-misses during its insertion into S_R. Thus each *Decrease-Key* operation is charged for $\mathcal{O}\left(\frac{1}{B}\log_2 N\right)$ amortized cache-misses in total.

(b) The claim follows from the observation that since $N \gg M$, the tall cache assumption implies $\log_2 N = \mathcal{O}\left(\log_2 \frac{N}{B}\right)$.

(c) We modify COTT so that, for some $q = \log_2 \beta \frac{M}{B}$, where $\beta < 1$ is a constant, nodes in the first $q-1$ levels do not have any stacks, and the topmost block of each of the stacks in level q are kept in the cache.

■

The following lemma can be proved by replacing the buffer heap with a COTT in the proof of lemma 3.4.1 in Chapter 3.

Lemma A.0.2. *Single source shortest paths in a directed graph can be computed cache-obliviously in $\mathcal{O}\left(\left(n + \frac{m}{B}\right) \cdot \log_2 \frac{m}{B}\right)$ cache-misses using a COTT under the tall cache assumption.*

Appendix B

Implementations of Dijkstra's SSSP Algorithm

In our shortest path experiments in Chapter 4, we considered the following three implementations of Dijkstra's SSSP algorithm.

Dijkstra's SSSP Algorithm with *Decrease-Keys* (Dijkstra-Dec).

Dijkstra's SSSP algorithm for directed graphs [43] works by maintaining an upper bound on the shortest distance (a *tentative distance*) to every vertex from the source vertex s and visiting the vertices one by one in non-decreasing order of tentative distances (see function Dijkstra-Dec in Figure B.1). The next vertex to be visited is the one with the smallest tentative distance extracted from the set of unvisited vertices (with finite tentative distance) kept in a priority queue Q. After a vertex has been extracted from Q it is considered *settled*, and each of its unvisited neighbors is either inserted into Q with a finite tentative distance or has its tentative distance updated if it already resides in Q. Dijkstra's algorithm performs n *Insert* and *Delete-Min* operations each and $\mathcal{O}(m-n)$ *Decrease-Key* operations on Q.

Using a binary heap that supports *Insert*, *Decrease-Key* and *Delete-Min* operations in $\mathcal{O}(\log n)$ time and I/O operations each Dijkstra's algorithm can be implemented to run in $\mathcal{O}((n+m)\cdot\log n)$ time and perform $\mathcal{O}((n+m)\cdot\log n)$ I/Os. If a Fibonacci heap is used that supports *Insert*/*Decrease-Key* operations in constant amortized time and I/Os, both the time and the I/O complexity of the algorithm reduces to $\mathcal{O}(m+n\cdot\log n)$. However, if a buffer heap is used as the priority queue Dijkstra's algorithm runs in $\mathcal{O}((n+m)\cdot\log n)$ time, but performs $\mathcal{O}\left(m+\frac{n+m}{B}\cdot\log n\right)$ I/O operations which is a factor of $\Theta(\log n)$ improvement over the I/O complexity of Dijkstra's algorithm with Fibonacci heap provided the graph is very sparse (i.e., $m=\mathcal{O}(n)$), and $B \gg \log n$ which typically holds for memory levels deeper in the hierarchy such as the disk. See Table 4.3 for the I/O complexity of this algorithm using the remaining priority queues in our experiment.

Dijkstra's Algorithm without *Decrease-keys* (Dijkstra-NoDec).

It is straight-forward to implement Dijkstra's algorithm using a priority queue that supports only *Insert* and *Delete-Min* operations (see function Dijkstra-NoDec in Figure B.1). The implementation performs $\mathcal{O}(m)$ *Insert* and *Delete-Min* operations and runs in $\mathcal{O}(m \cdot \log n)$ time and performs $\mathcal{O}(m \cdot \log n)$ I/O operations using an internal-memory priority queue. However, if a buffer heap or an auxiliary buffer heap is used, the algorithm continues to run in $\mathcal{O}(m \cdot \log n)$ time but performs $\mathcal{O}\left(m + \frac{m}{B} \cdot \log m\right)$ I/O operations which a $\Theta(\log n)$ factor improvement over the I/O bound using an internal-memory priority queue if the graph is very sparse (i.e., $m = \mathcal{O}(n)$) and $B \gg \log n$.

External-Memory Implementation of Dijkstra's Algorithm for Undirected Graphs (Dijkstra-Ext).

The traditional implementations of Dijkstra's algorithm (see Figure B.1) do not perform *Decrease-Key* operations on vertices that are already settled, but in the process they incur $\Theta(1)$ cache misses per edge. If one allows such *Decrease-Key* operations then either one must be able to identify after each *Delete-Min* operation whether the deleted vertex has been settled before which again causes $\Theta(m)$ additional cache misses, or be able to remove those extra *Decrease-Key* operations from the priority queue before they are extracted by a *Delete-Min* operation.

In [83] (see also [77]) Kumar & Schwabe presented an external-memory implementation of Dijkstra's algorithm for undirected graphs that allows spurious *Decrease-Key* operations to be performed on the primary priority queue Q but uses a mechanism to remove those operations from Q using an auxiliary priority queue Q' (see Function 3.4.1 in Chapter 3). This mechanism eliminates the need for identifying settled vertices directly and thus saves $\Theta(m)$ cache misses. The auxiliary priority queue only needs to support *Insert* and *Delete-Min* operations. The algorithm performs m *Decrease-Key* operations and about $n + m$ *Delete* operations on Q, and about m *Insert* and *Delete-Min* operations each on Q'. This algorithm is the same as Function 3.4.1 (i.e., Kumar & Schwabe's algorithm with both Q and Q' as buffer heaps) in Chapter 3 with *Decrease-Key*$_{(Q')}$ replaced with *Insert*$_{(Q')}$, and *Delete*$_{(Q')}$ replaced with *Delete-Min*$_{(Q')}$. We refer to the resulting function as Dijkstra-Ext.

The algorithm can be used to solve the SSSP problem on undirected graphs cache-obliviously in $\mathcal{O}\left(n + \frac{m}{B} \log m\right)$ I/O operations by replacing Q with a buffer heap and Q' with an auxiliary buffer heap.

In Chapter 3 we show how to implement Dijkstra's SSSP algorithm for directed graphs cache-obliviously in $\mathcal{O}\left(\left(n + \frac{m}{B}\right) \cdot \log \frac{n}{B}\right)$ I/Os under the tall cache assumption. The implementation requires one additional data structure called the

FUNCTION **B.0.2.** DIJKSTRA-DEC($G,\ w,\ s,\ d$)
{*Dijkstra's SSSP algorithm [43] with a priority queue that supports Decrease-Keys*}

1. perform the following initializations:

- (i) $Q \leftarrow \emptyset$
- (ii) ***for*** each $v \in V[G]$ ***do*** $d[v] \leftarrow +\infty$
- (iii) $\text{INSERT}_{(Q)}(\ s,\ 0\ )$

2. ***while*** $Q \neq \emptyset$ ***do***

- (i) $(u, k) \leftarrow \text{DELETE-MIN}_{(Q)}(\ )$, $d[u] \leftarrow k$
- (ii) ***for*** each $(u, v) \in E[G]$ ***do***
  - ***if*** $d[u] + w(u, v) < d[v]$ ***then***
    - ***if*** $d[v] = +\infty$ ***then*** $\text{INSERT}_{(Q)}(\ v,\ d[u] + w(u, v)\ )$
    - ***else*** $\text{DECREASE-KEY}_{(Q)}(\ v,\ d[u] + w(u, v)\ )$
    - $d[v] \leftarrow d[u] + w(u, v)$

DIJKSTRA-DEC ENDS

FUNCTION **B.0.3.** DIJKSTRA-NODEC($G,\ w,\ s,\ d$)
{*Dijkstra's SSSP algorithm [43] with a priority queue that does not support Decrease-Keys*}

1. perform the following initializations:

- (i) $Q \leftarrow \emptyset$
- (ii) ***for*** each $v \in V[G]$ ***do*** $d[v] \leftarrow +\infty$
- (iii) $\text{INSERT}_{(Q)}(\ s,\ 0\ )$

2. ***while*** $Q \neq \emptyset$ ***do***

- (i) $(u, k) \leftarrow \text{DELETE-MIN}_{(Q)}(\ )$
- (ii) ***if*** $k < d[u]$ ***then***
  - $d[u] \leftarrow k$
  - ***for*** each $(u, v) \in E[G]$ ***do***
    - ***if*** $d[u] + w(u, v) < d[v]$ ***then*** $\text{INSERT}_{(Q)}(\ v,\ d[u] + w(u, v)\ )$, $d[v] \leftarrow d[u] + w(u, v)$

DIJKSTRA-NODEC ENDS

Figure B.1: Given a directed graph G with vertex set $V[G]$ (each vertex is identified with a unique integer in $[1, |V[G]|]$), edge set $E[G]$, a weight function $w : E[G] \rightarrow \Re$ and a source vertex $s \in V[G]$, both functions compute the shortest distance from s to each vertex $v \in V[G]$ and stores it in $d[v]$.

buffered repository tree [25] for remembering settled vertices. Since we performed our experiments mainly on sparse graphs for which implementations in Figure B.1 give better bounds we have not considered this implementation.

I/O Cost of Accessing the Graph Data Structure Only			
Implementation	Accessing/updating tentative distances	Accessing adjacency lists	Total
DIJKSTRA-DEC	$n+m$	$n+\frac{m}{B}$	$2n+m+\frac{m}{B}$
DIJKSTRA-NODEC	$n+m+D$	$n+\frac{m}{B}$	$2n+m+\frac{m}{B}+D$
DIJKSTRA-EXT	*none*	$n+\frac{m}{B}$	$n+\frac{m}{B}$

Table B.1: I/O complexity of different implementations of Dijkstra's algorithm for accessing the graph data structure only, where D ($\leq m$) is the number of *Decrease-Key*s performed by DIJKSTRA-DEC and B is the block size.

Number of Priority Queue Operations Performed				
Implementation	*Insert*	*Decrease-Key*	*Delete/Delete-Min*	Total
DIJKSTRA-DEC	n	D	n	$2n+D$
DIJKSTRA-NODEC	$n+D$	*none*	$n+D$	$2n+2D$
DIJKSTRA-EXT	Primary (Q)			
	none	$2m$	$n+2m$	$n+4m$
	Auxiliary (Q')			
	$2m$	*none*	$2m$	$4m$
	Total (Q & Q')			
	$2m$	$2m$	$n+4m$	$n+8m$

Table B.2: Number of priority queue operations performed by different implementations of Dijkstra's algorithm, where D ($\leq m$) is the number of *Decrease-Key*s performed by DIJKSTRA-DEC.

Discussion on the Relative Performance of the Three Implementations.

Table B.1 lists the I/O cost of accessing the graph data structure by the three implementations while Table B.2 lists the number of different priority queue operations performed by them.

We observe that DIJKSTRA-NODEC performs slightly more I/O operations for accessing the graph data structure as well as more priority queue operations compared to DIJKSTRA-DEC. However, DIJKSTRA-NODEC can sometimes use a more efficient priority queue than DIJKSTRA-DEC since unlike DIJKSTRA-DEC it does not require the priority queue to support *Decrease-Key* operations. As a result DIJKSTRA-NODEC is likely to run faster than DIJKSTRA-DEC for in-core computations on very sparse graphs. For example, consider running the two implementations in-core on a large graph with $m = \Theta(n)$. In that case, $D \leq kn$ for some constant k and $\log n > B$, and DIJKSTRA-NODEC will run faster than DIJKSTRA-DEC provided it uses a priority queue that runs at least $1 + k$ times faster than the priority queue used by DIJKSTRA-DEC.

The external-memory implementation DIJKSTRA-EXT performs the smallest number of I/O operations for accessing the graph data structure than the other two implementations. However, this reduction in graph operations comes at the cost of considerably increasing the number of priority queue operations performed. For example, for $\mathcal{G}_{n,m}$ with average degree 8 for which $D \leq n$ typically holds, DIJKSTRA-EXT performs at least 11 times more priority queue operations than DIJKSTRA-DEC. However, if DIJKSTRA-EXT uses I/O-efficient priority queues such as the buffer heap and the auxiliary buffer heap, and the block size B is sufficiently large then the I/O cost of performing the priority queue operations will no longer dominate its running time. In such a scenario (e.g., out-of-core computations) DIJKSTRA-EXT will outperform the other two implementations because of the relatively smaller number of graph operations it performs.

Appendix C

Formal Definitions of δ and π

In Section 6.2.1 of Chapter 6 we defined functions π and δ (see Definition 6.2.2) based on the notions of aligned subintervals and aligned subsquares. Here we define these two functions more formally in closed form.

Recall from Definition 6.2.2(a) that for $x, y, z \in [1, n]$, $\delta(x, y, z)$ is defined as follows.

- If $x = y = z$, then $\delta(x, y, z) = z - 1$.
- If $x \neq z$ or $y \neq z$, then $\delta(x, y, z) = b$ for the largest aligned subsquare $[a, b], [a, b]$ of $c[1 \ldots n, 1 \ldots n]$ that contains (z, z), but not (x, y), and this subsquare is denoted by $S(x, y, z)$. Now consider the initial function call $\mathrm{F}(X, k_1, k_2)$ on c with $X \equiv c$, $k_1 = 1$ and $k_2 = n$, where $n = 2^q$ for some integer $q \geq 0$. We know from Lemma 6.2.1(a) that if $S(x, y, z)$ is one of the quadrants of X then it must be either X_{11} or X_{22}, otherwise $S(x, y, z)$ must be entirely contained in one of those two quadrants. Hence, in order to locate $S(x, y, z)$ in X and thus to calculate the value of $\delta(x, y, z)$ we need to consider the following four cases:

  (*i*) $\underline{(z, z) \in X_{11} \text{ and } (x, y) \notin X_{11}}$: $X_{11} \equiv S(x, y, z)$ and $\delta(x, y, z) = 2^{q-1}$ by definition.

  (*ii*) $\underline{(z, z) \in X_{22} \text{ and } (x, y) \notin X_{22}}$: $X_{22} \equiv S(x, y, z)$ and $\delta(x, y, z) = 2^q$ by definition.

  (*iii*) $\underline{(z, z) \in X_{11} \text{ and } (x, y) \in X_{11}}$: $S(x, y, z) \in X_{11}$, and compute $\delta(x, y, z)$ recursively from X_{11}.

  (*iv*) $\underline{(z, z) \in X_{22} \text{ and } (x, y) \in X_{22}}$: $S(x, y, z) \in X_{22}$, and compute $\delta(x, y, z)$ recursively from X_{22}.

  Now for each integer $u \in [1, 2^q]$, define $u' = u - 1$ which is a q-bit binary number $u'_q u'_{q-1} \ldots u'_2 u'_1$. Then it is easy to verify that the following recursive function

$\rho(x,y,z,q)$ captures the recursive method of computing $\delta(x,y,z)$ described above, i.e., $\delta(x,y,z) = \rho(x,y,z,q)$ if $x \neq z$ or $y \neq z$.

$$\rho\left(x,y,z,q\right) = \begin{cases} 2^{q-1} & \text{if } \left(x'_q = 1 \vee y'_q = 1\right) \wedge z'_q = 0 \\ 2^{q} & \text{if } \left(x'_q = 0 \vee y'_q = 0\right) \wedge z'_q = 1 \\ \rho\left(x,\ y,\ z,\ q-1\right) & \text{if } x'_q = y'_q = z'_q = 0, \\ 2^{q-1} + \rho\left(\begin{array}{c} x - 2^{q-1}, y - 2^{q-1}, \\ z - 2^{q-1}, q-1 \end{array}\right) & \text{if } x'_q = y'_q = z'_q = 1. \end{cases}$$

We can derive a closed form for $\rho\left(x,y,z,q\right)$ from its recursive definition given above. Let $\boxdot$, $\boxplus$, and $\boxtimes$ denote the bitwise AND, OR and XOR operators, respectively, and define

(a) $\alpha(x,y,z) = 2^{\lfloor \log_2 \{((x-1)\ \boxtimes\ (z-1))\ \boxplus\ ((y-1)\ \boxtimes\ (z-1))\}\rfloor}$,

(b) $\overline{u} = 2^r - 1 - u$ (bitwise NOT), and

(c) $\beta(x,y,z) = \left(\overline{x-1}\ \boxplus\ \overline{y-1}\right)\ \boxdot\ (z-1)$.

Then

$$\rho\left(x,y,z,q\right) = \left\lfloor \frac{z-1}{2\alpha(x,y,z)} \right\rfloor \cdot 2\alpha(x,y,z) + \alpha(x,y,z) + \alpha(x,y,z) \boxdot \beta(x,y,z) \tag{C.0.1}$$

Now we can formally define function $\delta : [1, 2^q] \times [1, 2^q] \times [1, 2^q] \to [0, 2^q]$ as follows.

$$\delta(x,y,z) = \begin{cases} z - 1 & \text{if } x = y = z, \\ \rho(x,y,z,q) & \text{otherwise (i.e., } x \neq z\ \vee\ y \neq z). \end{cases}$$

The explicit (nonrecursive) definition of δ is the following, based on C.0.1.

$$\delta(x,y,z) = \begin{cases} z - 1 & \text{if } x = y = z, \\ \left\lfloor \frac{z-1}{2\alpha(x,y,z)} \right\rfloor \cdot 2\alpha(x,y,z) + \alpha(x,y,z) + \alpha(x,y,z) \boxdot \beta(x,y,z) & \text{otherwise.} \end{cases}$$

From Definition 6.2.2(b), we have that function $\pi : [1, 2^q] \times [1, 2^q] \to [0, 2^q]$ is the specialization of δ to one dimension, hence we obtain:

$$\pi(x,z) = \delta(x,x,z) = \begin{cases} z-1 & \text{if } x=z, \\ \rho(x,x,z,q) & \text{otherwise (i.e., } x \neq z). \end{cases}$$

Using the closed form for ρ, we can write π in a closed form as follows:

$$\pi(x,z) = \begin{cases} z-1 & \text{if } x=z, \\ \left\lfloor \frac{z-1}{2\alpha'(x,z)} \right\rfloor \cdot 2\alpha'(x,z) \ + \ \alpha'(x,z) \ + \ \overline{x-1} \ \boxdot \ (z-1) \ \boxdot \ \alpha'(x,z) & \text{otherwise;} \end{cases}$$

where $\alpha'(x,z) = \alpha(x,x,z) = 2^{\lfloor \log_2 \{((x-1) \ \boxtimes \ (z-1)\rfloor}$.

Appendix D

Cache-oblivious Algorithm for Recurrence 8.2.3 in 2D

In Section 8.2.1 of Chapter 8 we described and analyzed an algorithm that solves recurrence 8.2.3 in 3D. Here, in Figure D.2 we provide a cache-oblivious algorithm similar to the one given in Section 8.2.1 which solves recurrence 8.2.3 in 2D along with a traceback path.

FUNCTION **D.0.4.** COMPUTE-BOUNDARY-2D($X,\ Y,\ L,\ T$)

Input. Same as the input description of COMPUTE-TRACEBACK-PATH-2D in Figure D.2.

Output. Returns an ordered tuple $\langle R, D\rangle$, where R ($\equiv Q[\ r,\ 1:r\ ]$) and D ($\equiv Q[\ 1:r,\ r\ ]$) are the right and bottom boundaries of $Q[\ 1:r,\ 1:r\ ]$, respectively.

1. ***if*** $r = 1$ ***then*** $R = D \leftarrow f(\ \langle\ u,\ v\ \rangle,\ \langle\ X,\ Y\ \rangle,\ L \cup T\ )$
2. ***else***
3. Extract $L_{1,j}$ from L, and $T_{i,1}$ from T, respectively, where $i, j \in [1, 2]$
4. $quadrant[\ 1:4\ ] \leftarrow \langle\ \langle\ 1,1\ \rangle,\ \langle\ 1,2\ \rangle,\ \langle\ 2,1\ \rangle,\ \langle\ 2,2\ \rangle\ \rangle$
5. ***for*** $l \leftarrow 1$ ***to*** 4 ***do***
6. $\langle\ i,\ j\ \rangle \leftarrow quadrant[\ l\ ]$,
   $\langle\ R_{ij},\ D_{ij}\ \rangle \leftarrow$ COMPUTE-BOUNDARY-2D($X_i,\ Y_j,\ L'_{ij},\ T'_{ij}$)
7. Compose R from $R_{2,j}$, and D from $D_{i,2}$, respectively, where $i, j \in [1, 2]$
8. ***return*** $\langle\ R,\ D\ \rangle$

COMPUTE-BOUNDARY-2D ENDS

Figure D.1: Evaluating recurrence 8.2.3 cache-obliviously for $d = 2$ without a traceback path. We assume for simplicity that $n = 2^q$ for some integer $q \geq 0$. In initial call to COMPUTE-TRACEBACK-PATH-2D, $X = x_1x_2\ldots x_n$, $Y = y_1y_2\ldots y_n$, $L \equiv c[\ 0,\ 0:n\ ]$ and $T \equiv c[\ 0:n,\ 0\ ]$.

Analysis. The following theorem can be proved by analyzing the time, space and cache-complexities of the functions given in Figures D.1 and D.2. The analyses are simpler than those given in Section 8.2.1 for our algorithm for solving the 3-dimensional version of the general recurrence 8.2.3, and hence are omitted.

Theorem D.0.1. *Given two sequences X and Y of length n each the two dimensional version (i.e., $d = 2$) of recurrence 8.2.3 can be solved and a traceback path can be computed in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space and $\mathcal{O}\left(1 + \frac{n}{B} + \frac{n^2}{BM}\right)$ cache misses.*

FUNCTION **D.0.5.** COMPUTE-TRACEBACK-PATH-2D($X,\ Y,\ L,\ T,\ P$)

Input. Here $r = |X| = |Y| = 2^t$ for some integer $t \in [1, p]$, and $Q[\ 0 : r,\ 0 : r\ ] \equiv c[\ u-1 : u+r-1,\ v-1 : v+r-1\ ]$, $X = x_u x_{u+1} \ldots x_{u+r-1}$ and $Y = y_v y_{v+1} \ldots y_{v+r-1}$ for some u and v $(1 \le u, v \le n-r+1)$. The left and top boundaries of $Q[\ 1 : r,\ 1 : r\ ]$ are in L ($\equiv Q[\ 0,\ 0 : r\ ]$) and T ($\equiv Q[\ 0 : r,\ 0\ ]$), respectively. The current traceback path is given in P.

Output. Returns the updated traceback path.

1. ***if*** $P \cap Q = \emptyset$ ***return*** P

2. ***if*** $r = 1$ ***then*** update P using recurrence 8.2.3 for $d = 2$

3. ***else*** { *For $i, j \in [1, 2]$, the left, right, top and bottom boundaries of quadrant Q_{ij} are denoted by L_{ijk}, R_{ijk}, T_{ijk} and D_{ijk}, respectively. X_1 and X_2 denote the 1st and the 2nd half of X, respectively (similarly for Y).* }

4. Extract $L_{1,j}$ from L, and $T_{i,1}$ from T, where $i, j \in [1, 2]$
{ *$L_{2,j} \equiv R_{1,j}$ and $T_{i,2} \equiv D_{i,1}$ for $i, j \in [1, 2]$* }

5. $quadrant[\ 1 : 4\ ] \leftarrow \langle\ \langle\ 1, 1\ \rangle,\ \langle\ 1, 2\ \rangle,\ \langle\ 2, 1\ \rangle,\ \langle\ 2, 2\ \rangle\ \rangle$

6. ***for*** $l \leftarrow 1$ ***to*** 3 ***do*** {*forward pass (compute boundaries)*}

7. $\langle\ i,\ j\ \rangle \leftarrow quadrant[\ l\ ]$,
$\langle\ R_{ijk},\ D_{ijk}\ \rangle \leftarrow$ COMPUTE-BOUNDARY-2D($X_i,\ Y_j,\ L'_{ijk},\ T'_{ijk}$)
{ *if $L_{ij} \equiv Q[\ i',\ j_1 : j_2\ ]$ then $L'_{ij} \equiv Q[\ i',\ j_1 - 1 : j_2\ ]$; similarly for B_{ij} and T_{ij}.* }

8. ***for*** $l \leftarrow 4$ ***downto*** 1 ***do*** {*backward pass (compute traceback path)*}

9. $\langle\ i,\ j\ \rangle \leftarrow quadrant[\ l\ ]$,
$P \leftarrow$ COMPUTE-TRACEBACK-PATH-2D($X_i,\ Y_j,\ L'_{ij},\ T'_{ijk},\ P$)

10. ***return*** P

COMPUTE-TRACEBACK-PATH-2D ENDS

Figure D.2: Evaluating recurrence 8.2.3 cache-obliviously for $d = 2$ along with a traceback path. We assume for simplicity that $n = 2^q$ for some integer $q \ge 0$. In initial call to COMPUTE-TRACEBACK-PATH-2D, $X = x_1 x_2 \ldots x_n$, $Y = y_1 y_2 \ldots y_n$, $L \equiv c[\ 0,\ 0 : n\ ]$, $T \equiv c[\ 0 : n,\ 0\ ]$ and $P = \langle (n, n) \rangle$.

Bibliography

[1] 9th DIMACS implementation challenge - shortest paths. url: http://www.dis.uniroma1.it/~challenge9/.

[2] Fujitsu MAP3147NC/NP MAP3735NC/NP MAP3367NC/NP disk drives product/maintenance manual.

[3] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*.

[4] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[5] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation pf diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28:1167–1181, 1999.

[6] T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104:45–62, 2000.

[7] L. Allulli, P. Lichodzijewski, and N. Zeh. A faster cache-oblivious shortest-path algorithms for undirected graphs with bounded edge lengths. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 910–919, New Orleans, Louisiana, 2007.

[8] S. Altschul and B. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology*, 48:603–616, 1986.

[9] ARC/INFO. *Understanding GIS – the ARC/INFO method*. ARC/INFO, 1993. Rev. 6 for workstations.

[10] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, LNCS 955, pages 334–345. Springer-Verlag, 1995.

[11] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 24th ACM Symposium on Theory of Computing*.

[12] L. Arge, G. Brodal, and L. Toma. On external-memory MST, SSSP, and multi-way planar graph separation. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, LNCS 1851, pages 433–447. Springer-Verlag, 2000.

[13] L. Arge, U. Meyer, and L. Toma. External-memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. In *Proceedings of the 31st International Colloquium on Automata, Languages, and Programming*, pages 146–157, Turku, Finland, 2004.

[14] P. Ashar and M. Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 622–627, San Jose, California, 1994.

[15] D. Bader and K. Madduri. GTgraph: A suite of synthetic graph generators. url: http://www-static.cc.gatech.edu/~kamesh/GTgraph/.

[16] V. Bafna and N. Edwards. On de novo interpretation of tandem mass spectra for peptide identification. In *Proceedings of the 7th Annual International Conference on Research in Computational Molecular Biology*, pages 9–18, Berlin, Germany, 2003.

[17] R. Bellman. *Dynamic Programming*. The Princeton University Press, Princeton, New Jersey, 1957.

[18] G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244, Barcelona, Spain, 2004.

[19] R. Blumofe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall. An analysis of DAG-consistent distributed shared-memory algorithms. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, 1996.

[20] R. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, 1974.

[21] G. Brodal. Cache-oblivious algorithms and data structures. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, LNCS 3111, pages 3–13, Humlebæk, Denmark, 2004. Springer-Verlag.

[22] G. Brodal and R. Fagerberg. Funnel heap – a cache oblivious priority queue. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation*, LNCS 2518, Vancouver, BC, Canada. Springer-Verlag.

[23] G. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 307–315, San Diego, California, 2003.

[24] G. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory*, pages 480–492, Humlebæk, Denmark, July 2004.

[25] A. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[26] A. Buchsbaum and J. Westbrook. Maintaining hierarchical graph views. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 566–575, 2000.

[27] J. Cannone, S. Subramanian, M. Schnare, J. Collett, L. D'Souza, Y. Du, B. Feng, N. Lin, L. Madabusi, K. Muller, N. Pande, Z. Shang, N. Yu, and R. Gutell. The comparative RNA web (CRW) site: An online database of comparative sequence and structure information for ribosomal, intron, and other RNAs. *BioMed Central Bioinformatics*, 3:2, 2002. url: http://www.rna.icmb.utexas.edu/.

[28] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 4th SIAM International Conference on Data Mining*, Orlando, Florida, 2004.

[29] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thotethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, 1999.

[30] C. Cherng and R. Ladner. Cache efficient simple dynamic programming. In *Proceedings of the International Conference on the Analysis of Algorithms*, pages 49–58, Barcelona, Spain, 2005.

[31] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, and J. Vitter. External-memory graph algorithms. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[32] R. Chowdhury and V. Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 245–254, Barcelona, Spain, June 2004.

[33] R. Chowdhury and V. Ramachandran. External-memory exact and approximate all-pairs shortest paths in undirected graphs. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 735–744, Vancouver, BC, Canada, 2005. More details can be found in the technical report with the same title, TR-04-38, CS Dept., UT Austin, August 2004.

[34] R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pages 591–600, Miami, Florida, 2006.

[35] R. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 71–80, San Diego, California, 2007.

[36] T. Cormen. *Virtual Memory for Data Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[37] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[38] R. Cromp. An intelligent information fusion system for handling the archiving and querying of terabyte–sized spatial databases. In S. Tate, editor, *Report on the Workshop on Data and Image Compression Needs and Uses in the Scientific Community*, pages 75–84. CESDIS Technical Report Series, 1993.

[39] P. D'Alberto and A. Nicolau. R-Kleene: a high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. *Algorithmica*, 47(2):203–213, 2007.

[40] R. Dementiev. STXXL homepage, documentation and tutorial. url: http://stxxl.sourceforge.net/.

[41] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard template library for XXL data sets. In *Proceedings of the 13th Annual European Symposium on Algorithms*, LNCS 1004, pages 640–651. Springer-Verlag, 2005.

[42] T. DeSantis, I. Dubosarskiy, S. Murray, and G. Andersen. Comprehensive aligned sequence construction for automated design of effective probes

(cascade-p) using 16S rDNA. *Bioinformatics*, 19:1461–1468, 2003. url: http://greengenes.llnl.gov/16S/.

[43] E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[44] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *SIAM Journal on Computing*, 29:1740–1759, 2000.

[45] S. Dreyfus and A. Law. *The Art and Theory of Dynamic Programming*. Academic Press Inc., 1977.

[46] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[47] P. Erdös and A. Rényi. On the evolution of random graphs. *Mat. Kuttató. Int. Közl.*, 5:17–60, 1960.

[48] R. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[49] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles & Practice*. Addison-Wesley, 1999.

[50] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.

[51] M. Fredman and R. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[52] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[53] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Canada, 1998.

[54] M. Frigo and V. Strumpen. Cache-oblivious stencil computations. In *Proceedings of the 19th ACM International Conference on Supercomputing*, pages 361–366, Cambridge, Massachusetts, 2005.

[55] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 271–280, Cambridge, Massachusetts, 2006.

[56] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

[57] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than $o(1)$ dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[58] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[59] G. Golub and C. Van Loan. *Matrix Computations*. The John Hopkins University Press, third edition, 1996.

[60] K. Goto. GotoBLAS, 2005. url: http://www.tacc.utexas.edu/resources/software.

[61] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[62] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, second edition, 1994.

[63] J. Grice, R. Hughey, and D. Speck. Reduced space sequence alignment. *Computer Applications in the Biosciences*, 13(1):45–53, 1997.

[64] R. Grossi and G. Italiano. Efficient cross-trees for external memory. In J. Abello and J. Vitter, editors, *External Memory Algorithms and Visualization*, pages 87–106. American Mathematical Society Press, Providence, RI, 1999.

[65] R. Grossi and G. Italiano. Revised version of "Efficient cross-trees for external memory". Technical Report TR-00-16, Dipartimento di Informatica, Università de Pisa, Pisa, Italy, 2000.

[66] J. Gunnels, F. Gustavson, G. Henry, and R. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.

[67] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, New York, 1997.

[68] L. Haas and W. Cody. Exploiting extensible DBMS in integrated geographic information systems. In *Proceedings of the 2nd International Symposium on Advances in Spatial Databases*, LNCS 525, pages 423–450. Springer-Verlag, 1991.

[69] P. Hayes, D. Joyce, and P. Pathak. Ubiquitous learning – an application of mobile technology in education. In *Proceedings of the World Conference on Educational Multimedia, Hypermedia and Telecommunications*, volume 1, Lugano, Switzerland.

[70] D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[71] D. Hirschberg and L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.

[72] C. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961.

[73] C. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[74] J. Hong and H. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[75] K. Iversion. *A Programming Language*. Wiley, 1962.

[76] P. Kanellakis, S. Ramaswamy, D. Vengroff, and J. Vitter. Indexing for data models with constraints and classes. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems*, pages 233–243, 1993.

[77] I. Katriel and U. Meyer. Elementary graph algorithms in external memory. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, LNCS 2625. Springer-Verlag.

[78] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesely, 2005.

[79] B. Knudsen. Multiple parsimony alignment with "affalign". Software package `multalign.tar`.

[80] B. Knudsen. Optimal multiple parsimony alignment with affine gap cost using a phylogenetic tree. In *Proceedings of Workshop on Algorithms in Bioinformatics*, pages 433–446, 2003.

[81] D. Knuth. *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[82] D. Knuth. Two notes on notation. *American Mathematical Monthly*, 99:403–422, 1992.

[83] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 169–177, 1996.

[84] A. LaMarca and L. R. The influence of caches on the performance of heaps. *Journal of Experimental Algorithmics*, 1:4, 1996.

[85] D. Lan Roche. Experimental study of high performance priority queues, 2007. Undergraduate Honors Thesis, CS-TR-07-34, The University of Texas at Austin, Department of Computer Sciences.

[86] R. Laurini and A. Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, 1992.

[87] H. Le. Algorithms for identification of patterns in biogeography and median alignment of three sequences in bioinformatics, 2006. Undergraduate Honors Thesis, CS-TR-06-29, The University of Texas at Austin, Department of Computer Sciences.

[88] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th European Symposium on Algorithms*, LNCS 2461, pages 723–735. Springer-Verlag, 2002.

[89] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the 11th European Symposium on Algorithms*, LNCS 2832, pages 434–445. Springer-Verlag, 2003.

[90] B. Moret and H. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. In *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*. 1994.

[91] S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[92] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.

[93] E. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[94] S. Pan, C. Cherng, K. Dick, and R. Ladner. Algorithms to take advantage of hardware prefetching. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, pages 91–98, 2007.

[95] J. Park, M. Penner, and V. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[96] W. Pearson and D. Lipman. Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Sciences of the USA*, volume 85, pages 2444–2448, 1988.

[97] S. Pettie. Towards a final analysis for pairing heaps. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 174–183, 2005.

[98] S. Pettie and V. Ramachandran. Command line tools generating various families of random graphs. url: http://www.dis.uniroma1.it/~challenge9/code/Randgraph.tar.gz.

[99] S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 713–722, San Francisco, CA, 2002.

[100] D. Powell. Software package `align3str_checkp.tar.gz`.

[101] D. Powell, L. Allison, and T. Dix. Fast, optimal alignment of three sequences using linear gap cost. *Journal of Theoretical Biology*, 207(3):325–336, 2000.

[102] D. Powell, L. Allison, and T. Dix. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. url: http://math-atlas.sourceforge.net.

[103] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1986.

[104] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, June 1999.

[105] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proceedings of the 13th ACM Symposium on Principles of Database Systems*, pages 25–35, Vancouver, BC, Canada, 1994.

[106] E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. 285(5):2053–2068, 1999.

[107] W. Rytter. On efficient parallel computations for some dynamic programming problems. *Theoretical Computer Science*, 59:297–307, 1988.

[108] H. Samet. *The Design and Analyses of Spatial Data Structures*. Addison-Wesley, 1989.

[109] P. Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithmics*, 5:1–25, 2000.

[110] P. Sanders. Memory hierarchies – models and lower bounds. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, LNCS 2625. Springer-Verlag, 2003.

[111] P. Sanders and D. Schultes. United states road networks (tiger/line). Data Source: U.S. Census Bureau, Washington, DC, url: http://www.dis.uniroma1.it/~challenge9/data/tiger/.

[112] J. Seward and N. Nethercote. Valgrind (debugging and profiling tool for x86-Linux programs). url: http://valgrind.kde.org/index.html.

[113] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[114] M. Sniedovich. *Dynamic Programming*. The Marcel Dekker, Inc., New York, NY, 1992.

[115] J. Stasko and J. Vitter. Pairing heaps: experiments and analysis. *Communications of the ACM*, 30:234–249, 1987.

[116] G. Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, third edition, 1988.

[117] G. Tan, S. Feng, and S. Ninghui. Cache oblivious algorithms for nonserial polyadic programming. *The Journal of Supercomputing*, 39(2):227–249, 2007.

[118] G. Tan, S. Ninghui, and G. Gao. A parallel dynamic programming algorithm on a multi-core architecture. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 135–144, San Diego, California, 2007.

[119] J. Thomas, J. Touchman, R. Blakesley, G. Bouffard, S. Beckstrom-Sternberg, E. Margulies, M. Blanchette, A. Siepel, P. Thomas, J. McDowell, B. Maskeri, N. Hansen, M. Schwartz, R. Weber, W. Kent, D. Karolchik, T. Bruen, R. Bevan, D. Cutler, S. Schwartz, L. Elnitski, J. Idol, A. Prasad, S. Lee-Lin, V. Maduro, T. Summers, M. Portnoy, N. Dietrich, N. Akhter, K. Ayele, B. Benjamin, K. Cariaga, C. Brinkley, S. Brooks, S. Granite, X. Guan, J. Gupta, P. Haghihi, S. Ho, M. Huang, E. Karlins, P. Laric, R. Legaspi, M. Lim, Q. Maduro, C. Masiello, S. Mastrian, J. McCloskey, R. Pearson, S. Stantripop,

E. Tiongson, J. Tran, C. Tsurgeon, J. Vogt, M. Walker, K. Wetherby, L. Wiggins, A. Young, L. Zhang, K. Osoegawa, B. Zhu, B. Zhao, C. Shu, P. De Jong, C. Lawrence, A. Smit, A. Chakravarti, D. Haussler, P. Green, W. Miller, and E. Green. Comparative analyses of multi-species sequences from targeted genomic regions. *Nature*, 424:788–793, 2003.

[120] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[121] L. Toma and N. Zeh. I/O-efficient algorithms for sparse graphs. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, LNCS 2625. Springer-Verlag, 2003.

[122] L. Tong. Implementation and experimental evaluation of the cache-oblivious buffer heap, 2006. Undergraduate Honors Thesis, CS-TR-06-21, The University of Texas at Austin, Department of Computer Sciences.

[123] J. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence*, 3:331–360.

[124] L. Valiant. General context-free recognition in less than cubic time. *Journal of Compute and System Sciences*, 10:308–315, 1975.

[125] D. Vengroff and J. Vitter. I/O–efficient scientific computation using TPIE. In *Proceedings of IEEE Symposium on Parallel and Distributed Computing*, pages 74–77, 1995.

[126] V. Viswanathan, S. Huang, and H. Liu. Parallel dynamic programming. In *Proceedings of IEEE Conference on Parallel Processing*, pages 497–500, 1990.

[127] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[128] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[129] M. Waterman. *Introduction to Computational Biology.* Chapman & Hall, London, UK, 1995.

[130] J. Watson. The human genome project: Past, present and future. *Science*, 248:44–49, 1990.

[131] I. Wegner. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT, beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 118(1):81–98, 1993.

[132] J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7:347–348, 1964.

[133] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[134] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, 1993.

[135] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-aware programs. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 93–104, San Diego, California, 2007.

[136] U. Zwick. Exact and approximate distances in graphs – a survey. updated version at http://www.cs.tau.ac.il/~zwick. In *Proceedings of the 9th European Symposium on Algorithms*, LNCS 2161, pages 33–48. Springer-Verlag, 2001.

Vita

Rezaul Alam Chowdhury, the eldest son of Azam Chowdhury and Shajeda Kohinoor, was born in Chittagong, a beautiful hilly city in Bangladesh lying on the banks of the river Karnafuli near the shore of the Bay of Bengal. He graduated from Bangladesh University of Engineering and Technology (BUET) in 1999 with a degree in Computer Science & Engineering. After working for a couple of years in BUET he entered the University of Texas at Austin to pursue a Ph.D. in Computer Sciences.

Permanent Address: 11/B NAEM Road
Flat 2a
Dhanmondi, Dhaka-1205
Bangladesh

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.