COMMUNICATION AVOIDING PARALLEL ALGORITHMS FOR AMORPHOUS PROBLEMS

BY

SAEED MALEKI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor David Padua, Chair, Director of Research
Research Assistant Professor María J. Garzarán
Professor Laxmikant Kale
Professor Keshav Pingali, The University of Texas at Austin
Principal Researcher Madanlal Musuvathi, Microsoft Research

Abstract

Parallelizing large sized problem in parallel systems has always been a challenge for programmer. This difficulty is caused by the complexity of the existing systems as well as the target problems. This is becoming a greater issue as the data sizes are constantly growing and as a result, larger parallel systems are required.

Graph algorithms, machine learning problems and bio-informatics methods are among the many ever-growing problems. These group of problems are amorphous, meaning that memory accesses are unpredictable and the application usually has a poor locality. Therefore, synchronizations in these problems are specially costly since all-to-all communications are required and delivering an efficient parallel algorithm becomes more challenging. Another difficulty with these problems is that the amount of parallelism in them is limited which naturally makes them hard to parallelize. This is due to complicated data-dependences among the data elements in the algorithm.

Writing parallel algorithms for these problems, on the other hand, are specially difficult since an amorphous problem can be expressed in several dramatically different ways. This is because of complex data dependences which are statically unknown and therefore, many unique parallel approaches exist for a single problem. Consequently, programming each single approach requires starting from scratch which is time consuming.

This thesis introduces several ways to avoid costly communications in amorphous problems by compromising from the computation. This means that we can increase the total amount of work done by the processors to avoid synchronizations in an algorithm. This

is specially effective in large clusters since there is a massive computing power with very costly communications. These approaches, clearly, have a trade off between computation and communication and in this thesis, we study these trade offs as well. Also, we propose a new language to express the proposed algorithms to overcome the programming difficulty of the problems by providing tunable parameters for performance.

Acknowledgments

This dissertation was inspired and guided by several colleagues and friends. It has been an absolute honor and privilege to work with each single one of you and I cannot thank you enough for your endless academic support in this long journey.

First, I would like to thank my advisers David Padua and Maria Garzaran for their immense support, guidance with research, and passion for new ideas. I owe my academic wisdom to them.

I would also like to thank my thesis committee members Madanlal Musuvathi, Keshav Pingali and Laxmikant Kale for their invaluable feedback on my research. I especially wish to thank Keshav Pingali with whom I have worked closely; he has given me insight regarding each new project I started and I learned significantly from him. Last but not least, I am honored and very fortunate to have been able to work with two of the sharpest researchers in my field Madanlal Musuvathi and Todd Mytkowicz in multiple projects in my PhD. I truly appreciate my collaborations with them.

Finally, I would like to greatly thank my family and my friends for their moral support. They always praised me for my successes and were there for me in more stressful times.

Table of Contents

List of Figures . . . . . **vii**

List of Tables . . . . . **ix**

Chapter 1 Introduction **1**
1.1 Overview 1
1.2 Contributions 4
1.3 Thesis Organization 5

Chapter 2 Single-Source Shortest Path Problem **6**
2.1 Background 8
2.2 Overview 12
2.3 Parallelizing SSSP 13
2.3.1 Degree-Distance Distributions 14
2.3.2 High Level Idea 15
2.3.3 The Distributor 19
2.3.4 Implementation of DSMR Algorithm 21
2.4 Graph Extraction 22
2.4.1 Graph Characteristics 24
2.4.2 Implementation of Graph Extraction and Fix-Up 26
2.5 Pruning 29
2.5.1 Algorithm 30
2.6 Environmental Setup 31
2.7 Results 33
2.7.1 Shared Memory Results 34
2.7.2 Distributed Memory Results 36
2.8 Conclusions 41

Chapter 3 Parallelizing Dynamic Programming Algorithms **42**
3.1 Background 44
3.2 Linear-Tropical Dynamic Programming 47
3.3 The Parallel Rank-1 Method 50
3.3.1 Breaking Data-Dependences Across Stages 50
3.3.2 Rank Convergence 51

3.3.3 The Parallel Rank-1 Method Overview . . . 52
3.3.4 Parallel Forward Phase for the Rank-1 Method . . . 54
3.3.5 All-Non-Zero Invariance . . . 57
3.3.6 Parallel Backward Phase . . . 58
3.3.7 Rank Convergence and SSSP Discussion . . . 59
3.4 Local Linear-Tropical Dynamic Programming . . . 60
3.4.1 The Delta Method . . . 61
3.4.2 The Parallel Delta Method . . . 63
3.4.3 Algorithmic Comparison between The Parallel Delta Method and The Rank-1 Method . . . 69
3.5 LTDP Examples . . . 69
3.6 Evaluation . . . 74
3.6.1 LTDP Rank Convergence . . . 74
3.6.2 Environmental Setup . . . 75
3.6.3 LTDP Benchmarks and Performance . . . 76
3.6.4 The Rank-1 Method vs The Delta Method . . . 86
3.6.5 Wavefront vs Rank-1 Method . . . 88
3.7 Conclusions . . . 91

Chapter 4 Tiled Linear Algebra . . . 93
4.1 Graph Algorithms using Linear Algebra . . . 94
4.2 Tiled Linear Algebra . . . 97
4.2.1 Delaying Updates . . . 98
4.2.2 Partial Computation . . . 100
4.3 Single Source Shortest Path . . . 101
4.3.1 Algorithms . . . 102
4.3.2 Performance Comparison . . . 107
4.3.3 Impact of D in the DSMR Algorithm . . . 109
4.4 LTDP . . . 111
4.5 Conclusion . . . 114

Chapter 5 Related Work . . . 117

Chapter 6 Future Work . . . 121

Chapter 7 Conclusion . . . 124

Bibliography . . . 126

List of Figures

2.1 An instance of SSSP problem with v_0 as the source vertex. The values next to vertices are the current distances of vertices. 9
2.2 Overview of the engines in our algorithm. 13
2.3 Degree-distance distribution for Co-Author and US Roads Networks. 15
2.4 Overhead distribution of Δ-Stepping (DS) algorithm and DSMR algorithm compared with degree-distance distribution. 18
2.5 Pseudo code for our DSMR algorithm. 23
2.6 HE and LD distributions for an RMAT graph with uniformly random edge weight distribution from $[1 \dots 256]$. 25
2.7 Pseudo code for Fix-Up engine. 28
2.8 Pruning idea. x is the first common ancesstor of v and u. 30
2.9 Pseudo code for Prune engine. 32
2.10 Evaluation of DSMR, Δ-Stepping, Elixir and Oracle algorithms on the shared-memory machine. For readability, we do not show some data points for Oracle US Roads and Pruned Orkut 37
2.11 Performance comparison of Δ-Stepping, DSMR, Graph Extraction Optimization and Pruning. Plots (a), (b) and (c) show strong scaling results and plots (d) and (e) show weak scaling results for RMAT graphs 39

3.1 Dynamic programming examples with dependences between stages. 43
3.2 LTDP implementation that computes the stages sequentially. An implementation can possibly employ wavefront parallelization within a stage. 49
3.3 Rank-1 method parallelization using rank convergence. 53
3.4 Parallel algorithm for the forward Pass of Rank-1 method that relies on rank convergence for efficiency. All inter-processor communication is shown in magenta. 55
3.5 Parallel algorithm for the backward phase of LTDP that relies on rank convergence for efficiency. All inter-processor communication is shown in magenta. 58
3.6 The deltas among the cells. 62
3.7 Sequential Delta method for planar dynamic programming. 63
3.8 The parallel deltas method. 64
3.9 Parallel Delta method for local dynamic programming. 67

3.10 Two ways of grouping the subproblems in LCS into stages such that each stage only depends on one previous stage. 71
3.11 Performance (Mb/S), speed up and efficiency of Voyager and LTE Viterbi decoders with the parallel Rank-1 method. The non-filled data points demonstrates where processors have too few iterations to converge to rank 1 79
3.12 Performance (Mb/S), speed up and efficiency of CDMA and MARS Viterbi decoders with the parallel Rank-1 method. The non-filled data points demonstrates where processors have too few iterations to converge to rank 1 80
3.13 Smith-Waterman performance, speed up and efficiency with the parallel Rank-1 method. 82
3.14 Performance, speed up and efficiency results of Needleman-Wunsch with the parallel Rank-1 method. 84
3.15 Performance, speed up and efficiency results of Longest Common Subsequence with the parallel Rank-1 method. 87
3.16 The speed up of the parallel Delta method over the Rank-1 method using LCS example. 88
3.17 The speed up of the parallel Delta method over the Rank-1 method using NW example. 89
3.18 Performance/speed up results and comparison of the Rank-1 method and Wavefront parallelism for Needleman-Wunsch and LCS 90

4.1 Matrix-vector multiplication for reachability. 95
4.2 Sparse matrix vector multiplication with delaying updates. 99
4.3 Bellman-Ford algorithm main loop using TLA. 103
4.4 Chaotic-Relaxation main loop using TLA. 104
4.5 Dijkstra's algorithm main loop using TLA. 104
4.6 Δ-Stepping main loop using TLA. 106
4.7 DSMR algorithm in TLA. 107
4.8 SSSP algorithms performance comparison 108
4.9 Speed of DSMR over Δ-Stepping. 109
4.10 D impact on the total number of relaxation and performance in the DSMR algorithm. 112
4.11 The Rank-1 method in TLA notation. 115

List of Tables

2.1 Comparison of the overhead and number of synchronizations for Δ-Stepping and DSMR for the configurations in Figure 2.4. **OH:** Overhead, **Syncs:** Number of synchronizations. 20
2.2 Details of the performance evaluation in Figure 2.10 and 2.11. **OH:** Overhead, **Syncs:** Synchronizations, **TH:** Threshold. 36

3.1 Number of steps to converge to rank 1. 75
3.2 Algorithms used for each LTDP problem. 77

Chapter 1

Introduction

1.1 Overview

Exploring parallelism efficiently in different applications on shared or distributed memory systems has always been a challenge for programmers. This difficulty arises from heterogeneous architecture, synchronization and communication costs, ever-growing memory hierarchy complexity, and the restrictions that these systems impose on the programmer in order to achieve an efficient performance. Also, constant changes in underlying systems make the parallel algorithms not portable or efficient on other machines.

Distributed memory systems consist of nodes with multi-core processors in each node sharing memory. Since each node has its own block of memory, the amount of memory on a distributed memory system can scale linearly with the number of nodes. This enables the programmer to utilize a large amount of memory for input data sets that cannot be fit into a single node's memory by distributing the data. However, accessing memory blocks from remote nodes is costly and therefore makes it difficult for certain problems to scale.

One of the main focuses of this thesis proposal is about designing and implementing parallel algorithms for amorphous problems that partially avoid synchronization or commu-

nication [55]. Amorphous problems are the type of problems that have irregular memory access pattern and it is impossible to predict memory accesses statically. Graph problems are among the most important examples of amorphous algorithms. Avoiding communication for amorphous problems is never for free and it is traded with extra computation. The first problem that we studied is Single-Source Shortest Path (SSSP) which is a classical computer science problem. SSSP is the problem of finding the shortest distances in graph from a source vertex to every other vertex. Efficiently parallelizing SSSP for distributed memory systems has not been extensively studied. This is because the problem of massively parallelize the SSSP problem did not become important until some of its applications, such as Betweenness Centrality, had to be applied over immense input graphs in many different area domains [51, 43, 69, 39, 34].

There are multiple algorithms for SSSP problem including Bellman-Ford [10], Chaotic-Relaxation [17], Dijkstra [24] and Δ-Stepping [61]. Each algorithm is suitable for a specific type of graph as they make different balances between communication and computation. Chapter 2 compares these algorithms and presents a new one that is a combination of different algorithms and it avoids unnecessary communication to improve scaling. The focus of our new algorithm is scale-free graphs with non-uniform degree distribution. Non-uniform degree distribution makes parallelization of SSSP more challenging in several aspects including data distribution, load balancing, and communication. However, the skew in degree distribution allows for optimizing parallelization of SSSP in ways which will be discussed in Chapter 2.

Another set of problems that are also instances of SSSP problem are Linear Tropical Dynamic Programming (LTDP) problems. LTDP are dynamic programming problems where the recursive function can be expressed with linear algebra in Tropical semiring. LTDP examples are Longest Common Subsequence (LCS), Smith-Waterman [73], Needleman-Wunsch [65] and Viterbi [78]. These problems can also have very large input data sizes but they are not efficiently parallelizable using the current techniques due to fre-

quent synchronizations (barriers) or communications. In other words, most of the existing parallelizing technique respect the semantic of data dependences in a program and therefore, a limited amount of parallelism is exposed. There are well-known substituting transformations that modify a program semantic in order to enable parallelization such as reduction and parallel prefix sum [47] but there is not any available method that we are aware of for parallelizing these algorithms by changing their semantics. Chapter 3 presents two new methods to parallelize dynamic programming algorithms that breaks data-dependences to avoid communication in the expense of extra computation.

SSSP is the kernel of all the problems we studied in this thesis proposal and our methods avoid communications in different ways. Despite of similarities of these problems, writing parallel code for each one from scratch requires a lot of efforts from the programmer since writing parallel algorithms using low-level languages such as MPI or OpenMP is time-consuming and error prone. This is because reasoning about a parallel program is a lot harder when it is expressed in low-level instructions. Therefore, designing a high-level framework for writing parallel code for graph algorithms with user-controllable communication is a great fit for our problems. Writing parallel programs in high-level languages is not a new idea and there has been many approaches in the form of new libraries or languages that provide new abstractions. Examples of these include HTA [12], PetSc [7], Charm++ [40] and X10 [16]. Also, languages for domain specific problems, such as image processing algorithms [72] and graph problems [45, 13, 52] have been recently proposed. However, none of these languages/libraries has the features that we need to parallelize the set of algorithms we studied for distributed memory systems. Therefore, we designed Tiled Linear Algebra (TLA) [54] notation which is discussed in Chapter 4.

1.2 Contributions

This thesis makes several contributions in the parallelism of amorphous problems. First, we will present Dijkstra Strip Mined Relaxation (DSMR) algorithm for Single Source Shortest Path (SSSP) problem which is specially effective with scale-free networks. In scale-free networks, degree distribution is skewed, meaning that there are a lot of low-degree vertices and a few high-degree vertices. The amount of work associated with each vertex in SSSP algorithms is a factor of the degree of that vertex. Existing algorithms, process vertices completely and as a result, it is likely to have idle processing time. DSMR with equal amount of parallelism in each iteration, on the other hand, avoids this issue. It also provides a tunable parameter, D, which reduces the total number of synchronizations in the expense of extra work. Chapter 2 studies the impact of D and compares its performance with other SSSP algorithms.

Another contribution of this thesis is the set of optimizations that is applied on top of the DSMR's algorithm. These optimizations are associated with the property that heavy edge weights are unlikely to be used in any shortest path in scale-free networks. There are two techniques that we studied to take advantage of this property: subgraph extraction and pruning. Chapter 2 presents these two methods and as we will show, they improve the performance significantly.

In this thesis, we also study parallelization of LTDP problems. LTDP problems usually have very limited parallelism in each iteration but the total amount of computation is significant. Therefore, following the traditional parallelism methods will only provide fair speed up. However, we need new parallelization techniques. Chapter 3 will introduce two techniques to target LTDP problems: the Rank-1 method and the Delta method. These two methods rely on a property associated with LTDP problems which is rank convergence. They expose parallelism by performing extra work as overhead but as we will show these

overheads are very limited and can benefit the overall performance greatly.

The last contribution of this thesis is Tiled Linear Algebra (TLA), a new language for expressing parallel algorithms for amorphous problems. TLA is based on linear algebra and expresses algorithms using matrix and vector operations in different semi-rings. We will show what the necessary features are in TLA in order to express efficient parallel algorithms. These features are tunable parameters that are dependent on the type of problem and the input data as well. Chapter 4 will discuss TLA in details and provide insights about these features.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 discusses Single-Source Shortest Path problem with DSMR algorithm, subgraph extraction and pruning techniques. Chapter 3 presents parallelization of Linear Tropical Dynamic Programming problems with the Rank-1 and the Delta methods. Chapter 4 introduces Tiled Linear Algebra notation. Finally, Chapter 5 explains the related work, Chapter 6 discusses the future works, and Chapter 7 presents the conclusion.

Chapter 2

Single-Source Shortest Path Problem

Parallel graph algorithms are becoming increasingly important in high performance computing, as evidenced by the numerous parallel graph libraries and frameworks in existence today [45, 33, 11, 31, 52]. The reason for this growing interest is that the input graphs are rapidly increasing in size and as a result, their processing requires more computation power and memory space. *Scale-free networks* [8] such as Twitter's tweets graph [46] are among the many examples of today's large graphs.

The first problem studied in this thesis is parallelizing Single-Source Shortest Path (SSSP) problem which is an amorphous problem. Graph problems, in general, where the structure of the input graph is statically unknown is considered an amorphous problem. Amorphous problems, including SSSP, have irregular memory access patterns meaning that it cannot be predicated what memory locations will be accessed next in the algorithm. As a result, the SSSP problem has inherently very small spatial locality. On the other hand, the amount of computation per memory location in SSSP problem is very limited. Therefore, the running time for a parallel SSSP algorithm is mostly dominated by the memory accesses which are mostly remote due to the irregularity of the problem. Therefore, communication avoiding algorithms are required in order to deliver an efficient parallel SSSP algorithm.

This chapter presents DSMR (Dijkstra Strip Mined Relaxation), a new parallel shared and distributed memory algorithm for Single-Source Shortest Path (SSSP) problem that is particularly efficient on scale-free networks. Given a weighted graph G and a source vertex s in G, the SSSP problem computes the shortest distance from s to all vertices of G. SSSP is a classical problem and has many applications, such as transportation, robotics, and the computation of Betweenness Centrality [28], which in turn has multiple applications [51, 43].

Several sequential and parallel algorithms and implementations for SSSP have been proposed, including Dijkstra's algorithm [24], Chaotic-Relaxation [17], Bellman-Ford's algorithm [10] and Δ-Stepping [61]. However, these algorithms target general graphs without any specific property. In this chapter, we study the parallelization of SSSP for scale-free networks which satisfy the *power-law* degree distribution property. This means that scale-free networks have few vertices with high degrees and many vertices with low degrees [8]. Social networks in which celebrities are represented as high degree vertices and commoners as low degree vertices is an example of this property. The skew in degree distribution is seen in many graphs besides social networks, including internet web-graphs, and network of citations in scientific articles.

The skew in degree distribution makes parallelization of SSSP more challenging in terms of data distribution, load balancing, and communication. On the other hand, it is at the same time possible to take advantage of the non-uniform degree distribution to optimize parallelization of SSSP. The contributions of this chapter are:

1. **DSMR algorithm**: a partially asynchronous parallel algorithm for solving SSSP that reduces communication without increasing the computation overhead excessively.

2. **Subgraph Extraction**: given an input graph G, a subgraph G' is extracted form G by considering edges and vertices in the intersection of most shortest paths. SSSP is first solved for G' and then it is solved for $G \backslash G'$.

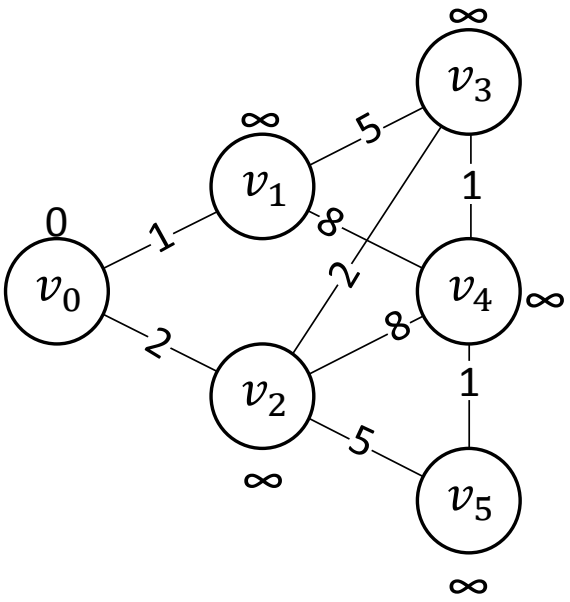
3. **Pruning**: this optimization identifies edges that can be guaranteed to not be used in any shortest path from any source vertex.

Our results show that DSMR is up-to 7.38× faster than one of the best shared-memory implementations of Δ-Stepping algorithms and up-to 2.05× faster than our own implementation of Δ-Stepping on a distributed-memory machine. We also show that our optimization techniques improve the performance by up-to 13×.

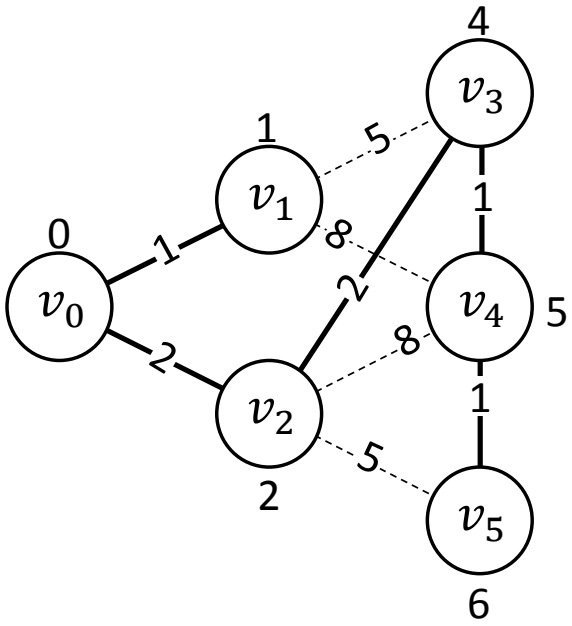
The rest of this chapter is organized as follows: Section 2.1 presents the background, Section 2.2 is an overview of our approach, Section 2.3 introduces DSMR and Sections 2.4 and 2.5 explain the subgraph extraction and pruning techniques, respectively. Section 2.6 describes the environmental setup, Section 2.7 shows the results, and Section 2.8 presents the conclusion.

2.1 Background

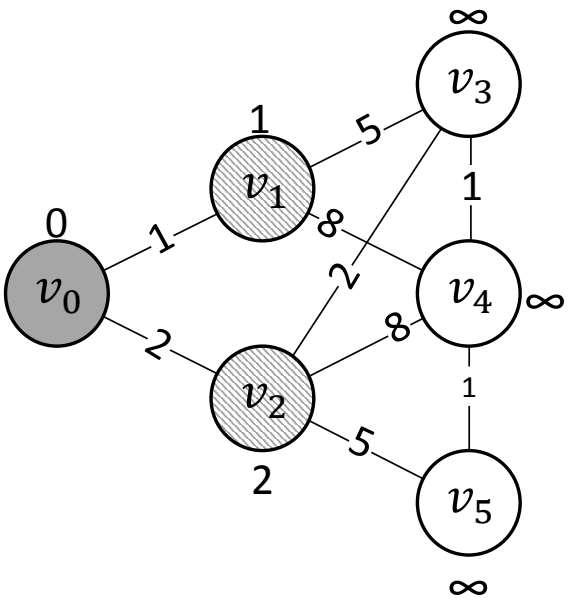
The SSSP problem computes the shortest distances in a weighted graph from a source vertex to every other vertex. Figure 2.1 shows a graph used to illustrate SSSP. v_0 is the source vertex. The values on the edges represent weights, and those on the vertices represent distances. We use the following notation: $w(v_iv_j)$ denotes the weight of edge v_iv_j and $d(v_i)$ denotes the *current* distance of v_i. $d(v_i)$ is a dynamic value that changes as the algorithm advances in the computation. $d_f(v_i)$ denotes the shortest distance computed for v_i which is the final value of $d(v_i)$. Figure 2.1a shows the initialization of the problem: for the source vertex, v_0, $d(v_0)$ is set to 0 and for the other vertices $d(v_i)$ is set to ∞. Given this initial setup, applying a set of *relaxation* operations (explained next) will compute the shortest distances for each vertex. Figure 2.1b shows the final distances with shortest paths in solid bold lines. The shortest paths themselves are not necessarily part of the output, since only the shortest distances may be needed.



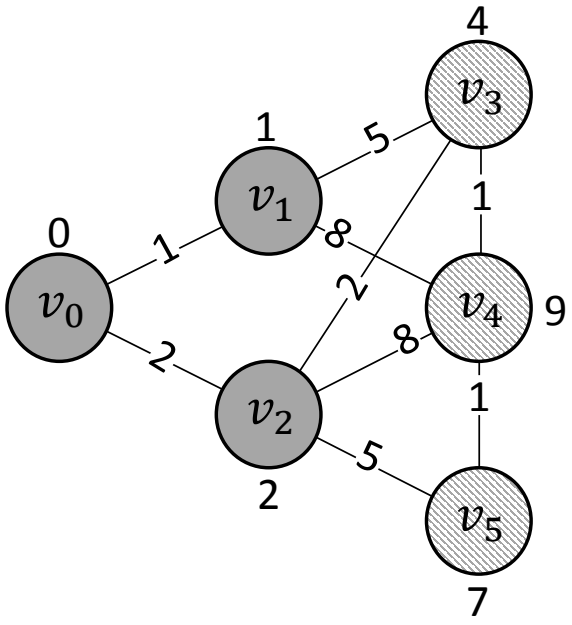
(a) Initial setup. $d(v_1) = \cdots = d(v_5) = \infty$ and $d(v_0) = 0$.



(b) Solution. The bold solid lines show the shortest paths.



(c) Vertex v_0 is relaxed and thus, vertices v_1 and v_2 are activated.



(d) Vertices v_0 and v_1 are relaxed and vertices v_2 and v_3 are active.

Figure 2.1: An instance of SSSP problem with v_0 as the source vertex. The values next to vertices are the current distances of vertices.

Relaxation: Relaxation is the basic operation of every SSSP algorithm. There are two types of relaxations: 1) *Relaxing an edge:* relaxing v_iv_j updates $d(v_j)$ to $\min\{d(v_j), d(v_i) + w(v_iv_j)\}$. 2) *Relaxing a vertex:* relaxing v_i relaxes all of the edges connected to v_i (outgoing edges in directed graphs). Relaxation of a vertex becomes necessary when a vertex becomes *active*, that is, when its distance is updated (updates always lower the distance). In Figure 2.1c, relaxing vertex v_0 activates its neighbors v_1 and v_2. SSSP starts by relaxing the source vertex, followed by the relaxation of the subsequent active vertices, computing the shortest distances for all vertices. However, since there could be multiple active vertices at a time, there are multiple possible orders of relaxation, perhaps carrying out groups of relaxations in parallel. We call the order of relaxation *scheduling*. This order differentiates SSSP algorithms.

Amount of Work: Relaxation of an edge vu requires accessing the distance of the destination vertex. For a parallel algorithm, the information of this destination vertex, most likely, will not be available in the local cache of the processor doing the relaxation due to the size of the graph and the unpredictability of memory accesses. Therefore, edge relaxations typically require a long memory access time, which is the dominating factor in the execution time of the SSSP algorithms. For this reason, we use **number of edge relaxations** as a measurement of the amount of work.

Scheduling: Consider Figure 2.1c again where v_0 is relaxed and vertices v_1 and v_2 are activated by updating their distances. Active vertices v_1 and v_2 can be relaxed in any order or in parallel since they have reached their final shortest distance values. In Figure 2.1d, vertices v_0, v_1 and v_2 have already been relaxed and vertices v_3, v_4 and v_5 are active. If vertex v_4 is relaxed before vertex v_3, value $d(v_4) = 9$ is used to relax v_4. However, when vertex v_3 is relaxed, $d(v_4)$ is updated to 5 and, consequently, v_4 becomes active and needs to be relaxed *again*. Similar scenario applies to v_5. Therefore, relaxing a vertex whose current

distance is not the shortest causes unnecessary work. The order in which active vertices are relaxed is the *scheduling* scheme of an algorithm and is the basic difference among the SSSP algorithms considered in this chapter.

Dijkstra's algorithm, Bellman-Ford's algorithm, Chaotic Relaxation and Δ-Stepping: The Dijkstra's algorithm [24] relaxes active vertices in current distance order meaning that, at each iteration, the active vertex with the minimum current distance is relaxed. For example, in Figure 2.1d v_3 must be relaxed before v_4 because $d(v_3) = 4 < d(v_3) = 9$. The Dijkstra's schedule guarantees that each vertex is relaxed at most once (in non-negative edge-weight graphs) and therefore, it performs the minimum amount of work. However, the only source of parallelism in the Dijkstra's algorithm is that the vertices with the same minimum current distance can be relaxed at the same time. But this parallelism can be limited since not many vertices may have equal distances. We will refer to this algorithm as *the parallel Dijkstra's algorithm.*

The Bellman-Ford's algorithm [10], on the other hand, relaxes all vertices $|V(G)|$ (number of vertices) times regardless of whether or not they are active. Chaotic Relaxation [17] is an optimization of the Bellman-Ford's algorithm and in each iteration, it only relaxes the active vertices from the previous iteration. Both algorithms are inefficient in terms of the amount of work they perform. For example, in Figure 2.1d, these two algorithms allow v_3, v_4 and v_5 to be relaxed at the same time which, as discussed before, results in unnecessary work. On the other hand, they expose more parallelism than the Dijkstra's algorithm. For instance, they allow v_1 and v_2 to be relaxed in parallel in Figure 2.1c.

Δ-Stepping [61] is a SSSP algorithm whose scheduling can be adjusted to fall between Dijkstra's and the Chaotic Relaxation algorithms. In Δ-Stepping, i iterates increasingly in $\{0, 1, 2, \dots\}$. For each i, it allows relaxation of all active vertices v such that $i.\Delta \leq d(v) < (i+1).\Delta$ where Δ is a constant throughout the algorithm. For example, assume $\Delta = 3$ in

Figure 2.1. For $i = 0$, the active vertices with distances between $[0 \dots 3)$ can be relaxed in parallel. That means that for $i = 0$, v_0 is relaxed first and activates v_1 and v_2 as shown in Figure 2.1c. Since $d(v_1), d(v_2) < 3$, they can be relaxed in parallel when $i = 0$ and this order will not introduce any overhead. Then for $i = 1$, vertices with distances between $[3 \dots 6)$ can be relaxed. In Figure 2.1d, at first, only v_3 is included in the range and when it is relaxed, $d(v_4)$ is updated to 5 and then v_4 is relaxed. Similarly, relaxing v_4 updates $d(v_5)$ to 6 and v_6 can be relaxed when $i = 2$. Therefore, Δ-Stepping provides two benefits: performing a close-to minimum amount of work while having a reasonable amount of parallelism. Note that Δ-Stepping with $\Delta = 1$ is equivalent to parallel Dijkstra's (assume that edge weights are integers) and with $\Delta = \infty$ is equivalent to Chaotic-Relaxation. Thus, Δ is adjustable to balance between work efficiency and parallelism. However, as shown later, Δ-Stepping performs poorly when applied to scale-free graphs.

2.2 Overview

Figure 2.2 shows the steps of our SSSP algorithm. Here, the steps in shaded boxes are optional with the third (Subgraph Extraction) and fifth (Fix-Up) boxes representing a single step, broken into two parts. First the input graph is given to the **Distributor** engine which breaks the graph into subgraphs and assigns each to a processor. After the distribution, the graph may be given to the *optional* pre-processing engines: **Pruning** and **Subgraph Extraction**. Pruning is an engine that detects edges that can be guaranteed not to be used for any shortest path from any source vertex. Subgraph Extraction extracts a subgraph of the input such that most shortest paths go through that subgraph. The output graph from the distributor or the pre-processing engines is given to **DSMR** where a random source vertex is chosen and the SSSP algorithms starts. Since the Subgraph Extraction ignores a portion of the graph, it may cause some incorrect computation which are fixed in the **Fix-Up**

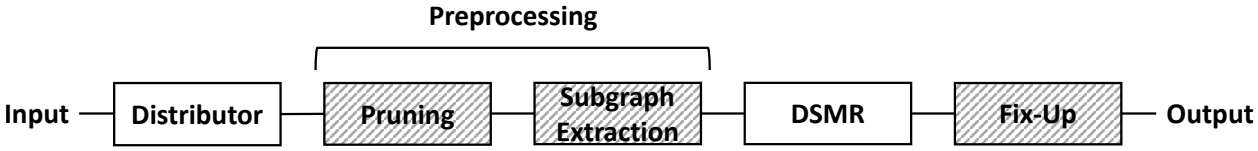


Figure 2.2: Overview of the engines in our algorithm.

stage. Therefore, running the Subgraph Extraction and the Fix-Up engines are codependent. Each engine in our algorithm is inspired by the properties of scale-free graphs which will be discussed in details later. However, our algorithm performs well on other graphs as well.

Focus of the algorithm Our algorithm is a parallel method for SSSP which processes one source node at a time. We can say that the algorithm exploits intra-SSSP parallelism. However, applications of SSSP, such as Betweenness Centrality and routing, make numerous SSSP queries and for those applications multiple SSSP executions could be done in parallel with each other. We did not study this type of parallelism, which could in theory complement the intra-SSSP parallelism. We should, however, point out that the execution of this embarrassingly parallel approach is limited by the need to replicate graph information which would increase memory requirements, perhaps beyond what the target machine can support. Also, when measuring running time for intra-parallelism, we only consider the stages of the algorithm from which the source vertex is known until it finishes computing shortest distances which includes the DSMR and the Fix-Up (if applicable) engines. The time for pre-processing and graph loading will be ignored since they are executed only once while the DSMR and the Fix-Up engine are executed as many as the number of queries.

2.3 Parallelizing SSSP

This section first describes the details of DSMR (Dijkstra Strip Mined Relaxation) algorithm and then the Distributor engine. For now, assume that Distributor breaks the input graph

into P (total number of processors) almost equal-sized vertex-disjoint subgraphs and assigns each subgraph to one of the processors. The owner of each vertex computes the shortest distance to that vertex from the source. Each processor contains information on all edges incident on the vertices it owns. Therefore, the edges joining vertices assigned to different processors will be replicated.

2.3.1 Degree-Distance Distributions

Degree-Distance distribution is a characteristic measured after computing the shortest distances from a source vertex. The distribution function is $y(x) = \sum_{v:d_f(v)=x} degree(v)$ where $d_f(v)$ is the shortest distance of v. In other words, $y(x)$ is the total number of edges that are connected to vertices with shortest distance x. Figure 2.3 shows the degree-distance distribution from a random source vertex for Co-Author and US Roads networks, described in Section 2.6. Co-Author network is a scale-free network while US Roads network is not. For the US Roads network, each point x represents the accumulation of the distribution function for range $[x512 \ldots (x+1)512)$ since the distance values for this network are sparse and understanding it without accumulation is hard.

The obvious difference between the degree-distance distributions of the two networks is that the Coauthor Network's plot has a narrow Gaussian shape with a long tail at the end while the US Roads network's plot has a wide Gaussian shape with short head and tail. The US Road network's degree-distance distribution is natural since there are several locations within each range of distances and the degree of vertices in US Roads network are almost equal. On the other hand, it typically takes few edges to connect any pair of vertices in a scale-free network as the high-degree vertices can serve as hubs [18]. Therefore, the narrow Gaussian shape for the degree-distance distribution of the Co-Author network is because most vertices are reached within few edges (narrow and tall part of the plot) and then, there are few vertices that require several edges to reach (long tail of the plot). Clearly, the

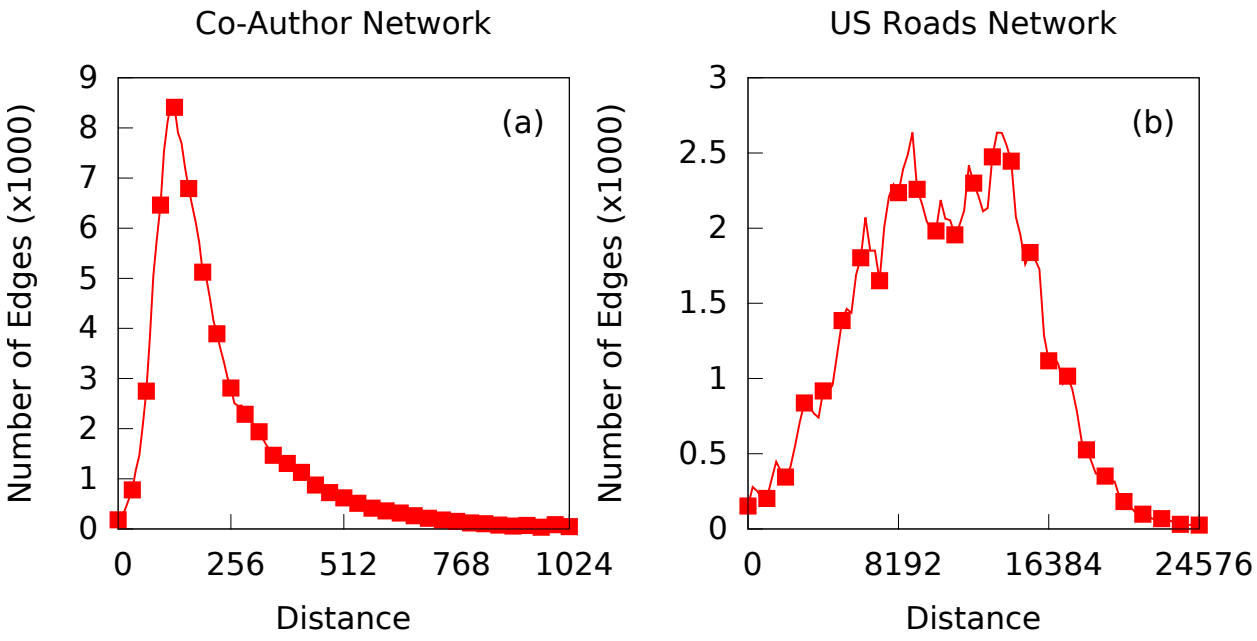


Figure 2.3: Degree-distance distribution for Co-Author and US Roads Networks.

details of the shape highly depends on the edge weights, the source vertex, and the size of a scale-free network, but it is safe to assume that the degree-distance distribution is narrow in scale-free networks.

2.3.2 High Level Idea

In this chapter, the *superstep* notion of the BSP (Bulk Synchronous Parallel) [76] model is used. In every superstep, each processor asynchronously executes its local computation and the remote memory accesses are buffered locally. This continues until a global *synchronization* is reached and the buffers are exchanged.

Parallel Dijkstra's (explained in Section 2.1) could, in each superstep, concurrently relax all active vertices with the same minimum distance. Thus, the degree-distance distribution of Co-Author network shown in Figure 2.3(a) shows the amount of parallelism available in each superstep of the algorithm in terms of the total number of edge relaxations for the parallel Dijkstra's algorithm. Also, since this algorithm introduces no unnecessary relaxations, the

area under the degree-distance distribution curve is the minimum amount of work. As Figure 2.3(a) shows, the amount of parallelism in Co-Author network is high during the first iterations but it drops for longer distances. Note that a synchronization is required after relaxing vertices for each distance value. Because of the long tail, numerous synchronizations are required for longer distances, making this algorithm inefficient. Δ-Stepping algorithm can reduce the number of synchronizations in longer distances by allowing relaxation of vertices in ranges of Δ distances as explained in Section 2.1. This may, however, cause unnecessary edge relaxations.

Figure 2.3(b) shows the amount of parallelism for US Roads network assuming that the computation is organized as for the Δ-Stepping algorithm with $\Delta = 512$. Our experiments show that there are not many unnecessary edge relaxations with this Δ. Therefore, the area under Degree-Distance distribution for this network is almost the minimum amount of work as well. For this Δ, the Degree-Distribution shows that the amount of parallelism for US Roads network, unlike for the Co-Author network, is distributed roughly uniformly, making Δ-Stepping suitable for this graph.

DSMR (Dijkstra Strip Mined Relaxation), our SSSP algorithm, consists of a sequence of supersteps each organized into three stages: 1) Each processor applies Dijkstra's algorithm to the subgraph it owns relaxing its vertices in *distance order* until it has *processed* exactly D edges. D is a parameter that can be set for different input graphs similar to Δ for Δ-Stepping. Processing an edge means that the edges with local destinations are relaxed immediately and the remote edge relaxations that require access to another processor's memory are buffered. This process happens asynchronously and consequently, different processors may work on different distances. 2) After D edges have been processed, the processor rendezvous with all other processors in an all-to-all communication procedure that exchanges the buffered relaxations. 3) After the all-to-all, relaxations update the distances of vertices and activate them. These 3-stage supersteps continue until there are no more

active vertices. Large D values cause late distance updates and work overhead and small D values cause frequent synchronizations increasing communication cost. To study how DSMR's overhead compares with Δ-Stepping's, we studied the Overhead Distribution and the number of synchronizations for both algorithms.

Overhead Distribution and Synchronization: The overhead of different algorithms is measured in terms of additional edge relaxations relative to those that would be performed by Dijkstra's algorithm. The cause of overhead is premature vertex relaxation, i.e. a vertex v is relaxed too soon with a $d(v)$ that is greater than the length of the shortest path. For example, in Figure 2.1d, relaxing vertex v_5 would be premature since the final shortest path to v_5 has not been computed, that is, the current value of $d(v_5)$ is not that of the shortest path (Figure 2.1b). This premature relaxation causes unnecessary relaxations because $d(v_5)$ will be updated again and then, the edges incident on v_5 will have to be relaxed again. The reason for the premature vertex relaxation of v_5 is the order of relaxations. If v_3 and v_4 had been relaxed before v_5, then $d(v_5)$ would be relaxed only once which is not premature. In general, assume that there is a premature vertex relaxation for vertex v at time t. We denote the current shortest distances at time t by $d_t(u)$ for all $u \in V(G)$. If the final shortest path from s to v (that will be eventually computed) is $s, v_1, v_2, \ldots, v_k, v$, we say that the premature vertex relaxation of v at time t is due to the first v_i such that $d_t(v_i)$ equals the final shortest distance ($d_f(v_i)$) and v_i has not been relaxed yet at time t. In other words, v_i is the first vertex that should have been relaxed before v. For a premature vertex relaxation of v at time t, we denote culprit vertex v_i by $B_t(v)$.

Each premature vertex relaxation performs unnecessary edge relaxations on each of the edges incident on the vertex. As discussed in Section 2.1, the number of edge relaxations is a measure of the amount of work. We define *Overhead Distribution* as follows: at point x is $y(x) = \sum_{v:d_f(v)=x} \sum_{u,t:B_t(u)=v} degree(u)$. In other words, for each vertex v, the number

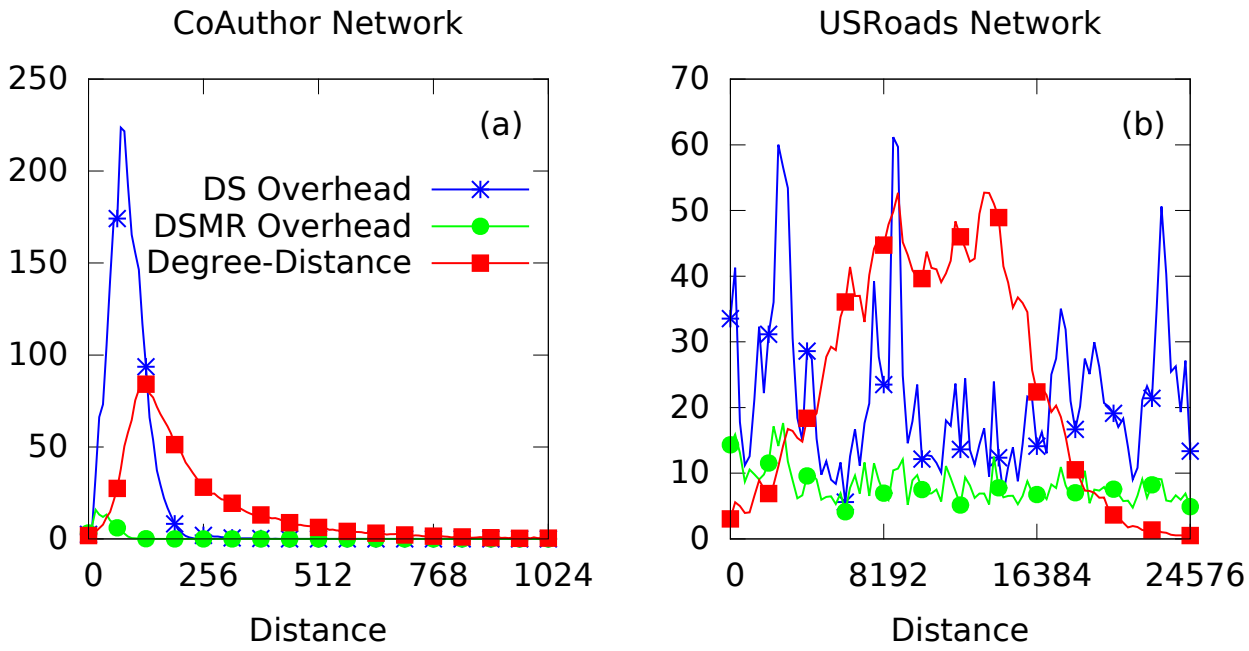


Figure 2.4: Overhead distribution of Δ-Stepping (DS) algorithm and DSMR algorithm compared with degree-distance distribution.

of unnecessary edge relaxations at different times due to v are counted and this number is accumulated to $y(x)$ where x is the final distance of v. Therefore, overhead distribution of an SSSP algorithm shows the amount of overhead associated with each distance.

Figure 2.4 shows the overhead distribution of Δ-Stepping (DS) and DSMR algorithms compared with the degree-distance distributions (minimum amount of work) of Co-Author and US Roads network. The Δ-Stepping algorithm used in this figure and the rest of this chapter is our implementation of the original algorithm [61]. In Δ-Stepping, edges vu with $w(vu) \geq \Delta$ are relaxed at most once (because of a technique explained in [61]) and therefore, they are excluded from the overhead distribution. Our distributor engine distributes the data for both DSMR and the Δ-Stepping algorithm. The values of Δ for Δ-Stepping and D for DSMR algorithms are chosen to facilitate this discussion. Table 2.1 shows the number of synchronizations and the percentage of total amount of overhead relative to the minimum amount of work (the area under the degree-distance distribution) for both algorithms.

As Figure 2.4(a) shows, for the Co-Author network the overhead distribution of Δ-Stepping is skewed towards the shorter distances. Most of the unnecessary relaxations are due to vertices with short distances. Vertices with long distances cause negligible overhead and can be relaxed in parallel. Therefore, Δ-Stepping cannot perform well with scale-free networks or, at least, Δ needs to be dynamically adjusted (small Δs for shorter distances and large Δs for longer distances). On the other hand, in Figure 2.4(a), the DSMR overhead distribution shows very small overhead relative to Δ-Stepping, while the total number of synchronizations are almost equal in both algorithms, as Table 2.1 shows. The total amount of overhead with respect to the minimum amount of work (sum over degree-distance distribution) is 115% and 4.8% for Δ-Stepping and DSMR, respectively. In Δ-Stepping, the constant Δ value causes either too much overhead for early superstar's (short distances) or limited parallelism for later supersteps (longer distances). With DSMR, a constant number D of edges are relaxed in each superstep, avoiding the two problems of Δ-Stepping.

Figure 2.4(b) shows overhead distributions for US Roads network. As it can be seen the overhead distribution for Δ-Stepping is spiky. To explain this behavior, consider range $[\Delta i \dots \Delta(i+1))$. A vertex v with distance close to $\Delta(i+1)$ is unlikely to update any vertices' distance from this range, while vertices with distances close to Δi are. Therefore, the spiky behavior is usually due to premature relaxations of vertices with distances close to Δi. On the other hand, the overhead distribution of DSMR is roughly uniform. This shows that there is more control on the overhead with D in DSMR than with Δ in Δ-Stepping. As the results in Table 2.1 shows, compared to Δ-Stepping, DSMR performs significantly less overhead work and fewer number of synchronizations (30% less).

2.3.3 The Distributor

There are two major challenges in data distribution of a graph on a distributed-memory system: handling high-degree vertices in scale-free networks and assigning vertices to pro-

Graphs	DSMR			Δ-Stepping		
	D	OH	Syncs	Δ	OH	Syncs
Co-Author	2^8	4.8%	62	2^8	115%	64
US Roads	2^5	5.4%	29,932	2^{17}	219%	40,855

Table 2.1: Comparison of the overhead and number of synchronizations for Δ-Stepping and DSMR for the configurations in Figure 2.4. **OH:** Overhead, **Syncs:** Number of synchronizations.

cessors.

1) Handling High-Degree Vertices: As discussed before, scale-free networks have a few high degree vertices and many low degree vertices. Assigning a high-degree vertex to a single processor increases the likelihood of load imbalance, which can be handled by a technique known as *Vertex Splitting* [31]. The Distributor engine specifies a threshold and the vertices with degree higher than that are identified. For each selected vertex, the Distributor copies the vertex on each of the P processors and assigns to each processor $\frac{1}{P}$th of edges of the original vertex. These P copies are connected to a unique copy by P edges with weight 0 which guarantees equal shortest distances for all $P + 1$ copies.

2) Assignment of Vertices: The low degree vertices are shuffled using a random permutation. Then, they are assigned to processors in consecutive chunks such that the number of edges for each processor is roughly the same. Afterwards, the vertices on the boundary of two processors are split such that each processor has equal number of edges.

The output of the distributor engine will be P subgraphs with disjoint vertex sets. However, the edges joining these subgraphs are shared by the processors containing the subgraphs. Each subgraph has an equal number of edges and, consequently, the number of vertices are not necessarily equal. The owner of each vertex is responsible for the computing the shortest distance of that vertex.

2.3.4 Implementation of DSMR Algorithm

Figure 2.5 shows a pseudo code for our DSMR algorithm. The algorithm is written in an SPMD model and uses MPI for communication. Therefore, all of the variables are private. The codes related to constant `D` edge relaxations are shown in magenta.

Array `d` contains the current distance of each vertex. For any vertex `u`, `d(u)` is initially ∞. Variable `relaxed`, declared in line 2, tracks the number of edge relaxations in each superstep and whenever it reaches the threshold `D`, an all-to-all communication is executed. The worklist `wl`, declared in line 5, is a vector of sets where each set corresponds to a distance value and contains all active vertices whose current distance is that of the set. In this algorithm, we are assuming that all the edge values (and consequently distance values) are integers. Therefore going through vertices in distance order locally in each processor is straightforward. Function `RelaxEdge` in line 7 relaxes an edge and updates `d(u)` and `wl` in lines 9-12. `active(u)` specifies if vertex `u` is active which means that it is in the worklist (as checked before erasing in line 9) and needs to be relaxed using the `RelaxVertex` function in line 15. Relaxation of an edge `vu` for which the processor owns both ends occurs immediately in line 20 but the remote relaxations are buffered in line 19. Line 22 enforces to not have more than `D` edge relaxations in a superstep.

The main DSMR algorithm's function is in line 24 which takes a source vertex $\texttt{v}_{\texttt{src}}$ that is non-`NULL` only for the processor which owns it. The initialization of $\texttt{v}_{\texttt{src}}$ occurs in line 25. In line 29, the set with minimum distance in `wl` is found and active vertices from it are relaxed in line 31. Eventually, after `D` edge relaxations, an `MPI_Alltoall` routine exchanges the buffers in line 33 and remote relaxations from the buffers are relaxed in the loop in line 35. At the end, `relaxed` is reset in line 37. The algorithm terminates when `wl`s in all processors are empty.

We are omitting a few parts of the algorithm for the sake of simplicity. This includes the code for when the break in line 22 occurs in the middle of the relaxation of a vertex.

DSMR completes the relaxation of that vertex at the beginning of the next superstep. The other part of the algorithm that we are omitting is determining that the `wls` are empty. This test is performed during the `MPI_Alltoall` communication without requiring additional communication.

Although in Figure 2.5, the Dijkstra's algorithm is used, any other sequential SSSP algorithm could have been applied. While it is true that this implementation of Dijkstra is not the most efficient and only works for integer distances, our experiments show that it works well with most graphs that we have evaluated.

Since each P processor processes D edges and the graph is randomized, it is expected that processor would buffer around $\frac{D}{P}$ remote edge relaxations to be sent to every other processor (line 19). However, we have seen some uneven behavior with a large number of processors. Thus, to maintain a constant amount of work for the loop in line 35, for each $P-1$ buffers in each processor, only a maximum of $1.25 \times \frac{D}{P}$ of the buffered data are sent. As a result, no processor receives more than $1.25 \times D$ edges to relax in that loop.

2.4 Graph Extraction

Graph extraction is a pre-processing technique that **extracts** a subgraph $G' \subseteq G$ from the input graph G, such that most of the shortest paths in G go through G'. This technique runs in two phases. First, DSMR is executed with G' to compute the shortest distances. After this is done, the shortest distances for most vertices would have been computed correctly. Then, for the rest of the graph, $G \backslash G'$, the Fix-Up engine corrects the distances computed incorrectly in the first phase. Edges in $G \backslash G'$ update the distances of only a few vertices and consequently, relaxing them in any order will not cause significant overhead. Therefore, the fix-up phase uses Chaotic Relaxation to minimize the number of synchronizations.

Graph extraction is beneficial only for certain graphs, depending on the characteristics

```
1  // Number of relaxations in each superstep
2  int relaxed = 0;
3  // Worklist for active vertices
4  // Each vector index represents a distance
5  Vector<Set<Vertex> > wl;
6
7  void RelaxEdge(Vertex u, int newDist){
8    if (d(u) > newDist){
9      if (active(u)) // Remove u from old set
10       wl[d(u)].erase(u);
11     d(u) = newDist;
12     wl[d(u)].insert(u); // Insert u to new set
13     active(u) = true; }}
14
15 void RelaxVertex(Vertex v){
16   active(v) = false;
17   foreach Edge vu in edges(v) {
18     relaxed++;
19     if (IsRemote(vu)) Buffer(<u,d(v)+w(vu)>);
20     else RelaxEdge(u, d(v)+w(vu));
21     // When threshold is reached, break
22     if (relaxed >= D) break; }}
23
24 void DSMR(Vertex v_src){
25   if (v_src) RelaxEdge(v_src, 0); // Initialization
26   do {
27     do {
28       // Find the minimum non-empty set
29       int ind = min i: !IsEmpty(wl[i]);
30       while (!IsEmpty(wl[ind]) && relaxed < D){
31         RelaxVertex(wl[ind].pop()); }
32     } while (ind < ∞ || relaxed < D);
33     MPI_Alltoall(buffer); // Exchange buffers
34     // Relax received requests
35     foreach <u,dist> in buffer:
36       RelaxEdge(u, dist);
37     relaxed = 0; // Reset
38   } while (all IsEmpty(wl)); }
```

Figure 2.5: Pseudo code for our DSMR algorithm.

of the graph. When the graph extraction is not beneficial, G' can be set to G and DSMR is executed on the entire graph, bypassing the fix-up phase. Therefore, extraction does not hurt performance, but it can significantly improve the performance of the algorithm for certain graphs. Next, we will discuss the input characteristics that impact the performance with graph extraction.

2.4.1 Graph Characteristics

Artificial or unweighted scale-free networks are typically weighted by distributing edge weights uniformly random from $[1 \dots C)$ where C is a constant. This approach is widely used [14, 30, 60, 53] and also adopted by the DARPA SSCA#2 benchmark [6]. One of the properties of scale-free networks with uniformly random edge weight distribution is that heavy-weight edges are unlikely to be used in shortest paths. There are other edge-weight distribution such as log-uniform [53] but it is even less likely for heavy-edge weights to be used with such a distribution. To understand this property, we studied HE *(Heaviest Edge)* distribution of the vertices. Assume that SSSP is computed for a graph from a random source vertex s and that $sv_0v_1 \dots v_kv$ is the shortest path from s to v. Define $HE(v)$ to be the heaviest edge weight in the shortest path from s to v: $HE(v) = \max\{w(sv_0), w(v_0v_1), \dots, w(v_kv)\}$. The key idea is that if all the edges with weight $> HE(v)$ are ignored, the shortest path for v can still be computed correctly.

Figure 2.6(a) shows two different distributions for a type-2 RMAT graph with $scale = 21$ (graph description in Section 2.6) with edge weight distributed uniformly from $[1 \dots 256]$. X axis shows different weight values and the two distributions are: 1) Accumulative HE distribution of the vertices: for weight x, $y(x)$ shows the percentage of vertices v with $HE(v) \leq x$. 2) Accumulative edge weight distribution: for weight x, $y(x)$ shows the percentage of edges uv with $w(uv) \leq x$. This distribution is linear because of the uniform edge weight distribution. Now, consider the vertical dashed purple line ($x = 48$) of Figure 2.6(a). If subgraph G'

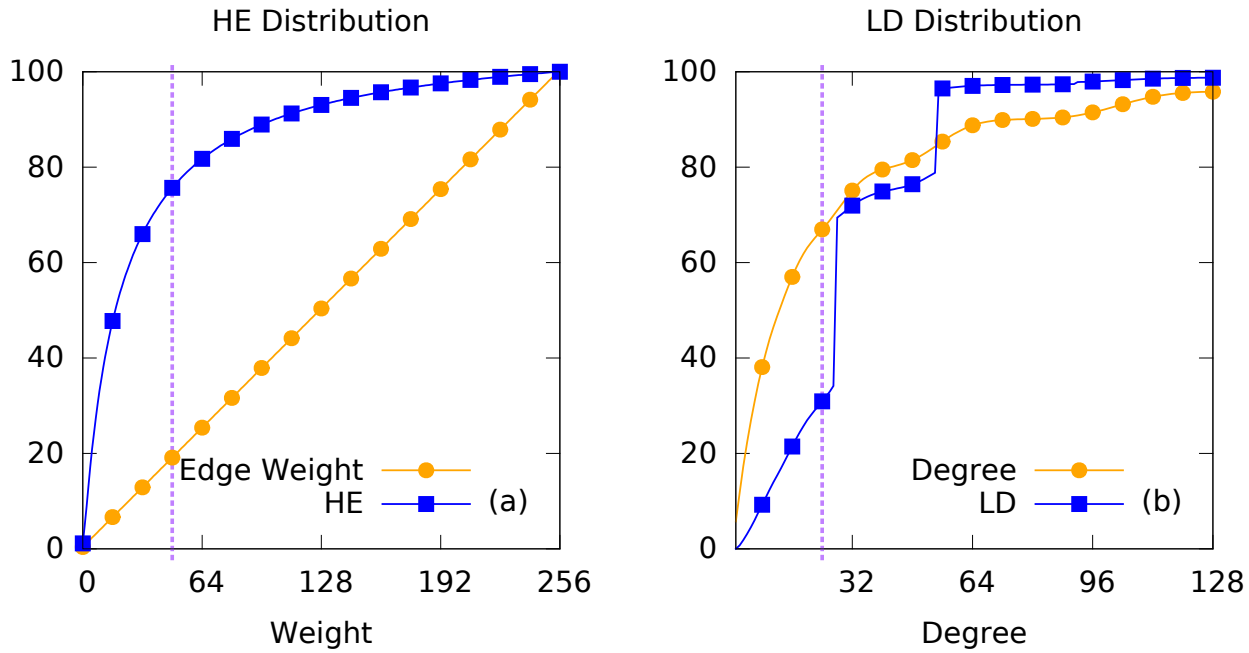


Figure 2.6: HE and LD distributions for an RMAT graph with uniformly random edge weight distribution from $[1 \ldots 256]$.

is extracted from G with all edges uv where $w(uv) \leq 48$, G' contains less than 20% of edges of G (the edge-weight distribution at the vertical dashed line). However, short distances will be computed correctly for almost 80% of the vertices in G' (HE distribution at the dashed vertical line in the figure). This means that by considering a small part of $E(G)$, shortest distance will be computed for a large set of $V(G)$. For the remaining 80% of the edges, fix-up phase will correct the distances for the remaining 20% of the vertices. The subtle difference between the first phase and fix-up phase is that, fix-up phase can have less synchronizations than DSMR while the total amount of work is not increased significantly.

Another property that is associated with scale-free graphs is that any pair of vertices can be reached with a few edges (short diameter) [18]. This is because high-degree vertices can serve as hubs and consequently, most of the shortest paths go through them. To understand this property, we studied LD *(Lowest Degree)* distribution. Assume that the shortest path from source vertex s to vertex v is $sv_1 \ldots v_k v$. Define $LD(v) = \min\{degree(s), degree(v_1), \ldots,$

$degree(v_k)\}$ which is the minimum degree of vertices in the path excluding v. The key idea is that if G' contains all vertices with degree greater than $LD(v)$ and all vertices of G' are relaxed, distance of v will be computed correctly.

Figure 2.6(b), similar to Figure 2.6(a), shows two different distributions for the same RMAT graph. X axis in this plot represents different degree values and the two distributions are: 1) Accumulative distribution of LD: given a degree x, $y(x)$ shows the percentage of vertices v with $LD(v) \leq x$. 2) Accumulative degree distribution: for degree x, $y(x)$ shows the percentage of vertices with degree $\leq x$. The plot is truncated for degrees greater than 128. Now, consider the dashed vertical line ($x = 24$) in Figure 2.6(b). If subgraph G' is extracted with vertices whose degree is greater than 24, G' will contain 33% of the vertices of G since 67% of the vertices have degree ≤ 24 (degree distribution at the dashed in Figure 2.6(a)). Thus, by relaxing the vertices of G' (including their edges in G), the shortest distances will be computed correctly for all vertices v with $LD(v) > 24$. This set of vertices are 70% of the vertices of G since 30% of the vertices have shortest paths with $LD \leq 24$ (LD distribution at the dashed vertical line in the figure). Similar to the HE scenario, setting G' as a small part of G and relaxing $V(G')$ results in correct distances for most vertices in G. As mentioned above, the subsequent fix-up phase relaxes the rest of vertices to fix shortest distances for the remaining 30% vertices.

Note that for the experiments shown in Figure 2.6, the source vertex s was chosen from G'. In the cases where s is not in G', DSMR starts by processing the whole graph, G, and relaxes vertices and edges in G. Once all active vertices are in G', DSMR continues working in G' only. Afterwards, as before, the fix-up phase will take care of the rest of the graph.

2.4.2 Implementation of Graph Extraction and Fix-Up

Extracting subgraph G' from input graph G is done through a function: $f : G \rightarrow \{0, 1\}$ specifying whether a vertex or an edge is in G' or not. While the graph is being loaded to

the system, subgraph G' can be created using f without significantly increasing the loading time. We do not measure the time for extraction since it is a part of the graph loading time.

Figure 2.7 shows the pseudo code for the fix-up engine. This code is sequential (we discuss later how to parallelize it). The algorithm is similar to the one from the DSMR algorithm in Figure 2.5 with a few exceptions. `wl` in line 1 is just a set instead of a vector of sets. This is because the fix-up executes Chaotic-Relaxation, where relaxations can occur in any order whereas in Figure 2.5, DSMR relaxes vertices in distance order. Consequently, `RelaxEdge` in line 2 is simpler than the one in Figure 2.5.

The `FixUp` function has two major loops. The first loop in line 8 goes through all the edges in `G\G'` and relaxes them. In this loop, there is an optional condition in magenta in line 9 whose raison d'etre is discussed below. The second loop in line 11, goes through all vertices of `wl` and relaxes all of their incident edges in `G`. The major difference between these two loops are the graphs: `G\G'` for the loops in line 8 and `G` for the loop in line 11. Before the fix-up engine, DSMR computes shortest distances in `G'`. In the loop in line 8, edges in `G\G'` are relaxed and the incorrect distances are updated. Updating distances of vertices causes activating them and adding them to `wl` in line 5. Later, vertices in `wl` are relaxed in the whole graph, `G`, in the loop in line 11. This loop, itself, may activate other vertices in `G` in line 14.

The parallelization of the fix-up algorithm is straight forward and similar to the parallelization of DSMR. The remote edge relaxations are buffered and after `wl` is empty in all processors, the buffers are exchanged via an `MPI_Alltoall` and then the remote relaxations are performed. The processors continue until no more remote or local relaxations are left.

The magenta condition in line 9 is an optional branch and can be removed. However, it makes a great difference in performance when the graph extraction is based on the weight of the edges. As discussed before, an edge relaxation requires a memory look up of the distance of the target vertex. Now assume that edge `vu` is accessed from the destination vertex `u`

```
1  Set<Vertex> wl; // Set of active vertices
2  void RelaxEdge(Vertex u, int newDist){
3    if (d(u) > newDist){
4        d(u) = newDist;
5        wl.insert(u); }}
6
7  void FixUp(){
8    foreach vu in G\G'
9      if (d(u) > w(vu)) // Optional if
10       RelaxEdge(u, d(v)+w(vu));
11   while (!IsEmpty(wl)){
12     Vertex v = wl.pop();
13     foreach vu in G
14       RelaxEdge(u, d(v)+weight(vu));  }}
```

Figure 2.7: Pseudo code for Fix-Up engine.

where `w(vu)`, `d(u)` are close in memory (spatial locality). Edge `vu` would update `d(u)` only if the magenta condition is true: `d(u)>w(vu)`. Otherwise, it is not required to access `d(v)` which in turn improves the performance. Surprisingly, the condition is seldom true and that comes from the fact that the shortest distances in scale-free networks with uniform edge-weight distribution are even shorter than the length of most heavy-weight edges. Our experiments show that this is independent of the constant C in the uniform distribution $[1 \dots C]$.

The idea behind this condition originated from the pull model in the SSSP algorithm discussed in [14], but it is used in a different way in this chapter. The idea is used in the fix-up engine in this chapter while in [14], it is used within their SSSP algorithm and there is no post fix-up phase. However, the benefit of this condition disappears with the Pruning engine as discussed in Section 2.5. Section 2.7 presents the performance gains with graph extraction and pruning.

2.5 Pruning

Pruning is another pre-processing technique that identifies edges in a graph G which can be guaranteed not to be used in any shortest path from any source vertex. Figure 2.8 shows a pruning scenario where edge vu, shown as a dotted line, is a candidate for pruning. The engine chooses a random source vertex s (not necessarily the one for the DSMR engine) and computes the shortest distances for all vertices. Figure 2.8 shows the shortest paths from the chosen source vertex s to u and v by solid wavy lines. Assume that these two shortest path diverge at vertex x. We call x the first common ancestor of v and u. If the condition $(d(u) - d(x)) + (d(v) - d(x)) < w(vu)$ is true, vu is marked useless. We call this the **the pruning test**. The reason is that the paths from x to v and from x to u are of distance $d(v) - d(x)$ and $d(u) - d(x)$, respectively. Therefore, if the distances of these two paths together are less that $w(vu)$, the edge vu will not be used in any shortest path for any pair of vertices since the path from v to x and x to u is shorter.

A useless edge seems illogical in a road network because a long segment of a road will be useless if there is a faster way around connecting the two points. However, in social networks, edge weights do not represent the distance between vertices, but rather the strength of their connectivity. For example, in the Co-Author network, the weight of an edge between two vertices is a function of the number of articles two authors had together and the number of participants in those publications. Therefore, it is likely to find useless edges in scale-free networks.

The pruning engine can iterative for up-to $|V(G)|$ number of source vertices but, obviously, that would take an excessively long time. Although the pruning algorithm, even for only one source vertex, takes longer than running SSSP itself, it is considered a pre-processing phase and running it for a few iterations is acceptable. Similar to graph extraction engine, the running time for pruning is not measured as discussed in Section 2.2. Also, as discussed

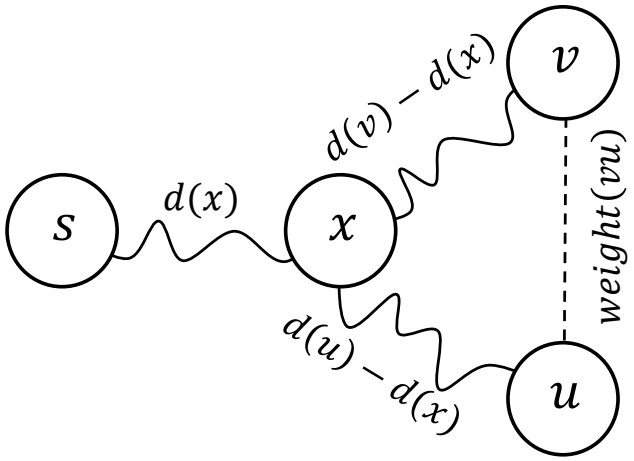


Figure 2.8: Pruning idea. x is the first common ancesstor of v and u.

in Section 2.4.1, edge weights in unweighted graphs are distributed uniformly random and heavy edges are unlikely to be used. As our results in Section 2.7 will show, this makes up-to 90% of the dges prunable in these graphs.

2.5.1 Algorithm

The pruning algorithm is relatively complicated in spite of the simplicity of the idea. There are two approaches to implement the idea: 1) optimized for memory, 2) optimized for running time. The second approach can be implemented by computing shortest distances and storing the intermediate vertices of the shortest path for each vertex. Then, for each edge vu, intermediate vertices of shortest paths to v and u are searched for the first common ancestor and the pruning test can be executed. However this approach requires storing the shortest paths for each vertex, which can be excessive for large networks. Therefore, we implemented the first approach.

Figure 2.9 shows the sequential pseudo code for our pruning algorithm. The algorithm starts by executing DSMR in line 4. The shortest paths in a graph from a source vertex creates a tree which we denote by T. T is computed along with our DSMR algorithm by storing $succ(v)$, the successor list of v for all v in T. Therefore, subtrees of T can be accessed

by their root and following the *succ* lists. We denote the subtree of a vertex v as its root by *subtree*(v). Set `st` in line 2 keeps the root of subtrees for the computation. In line 5, $\texttt{v}_{\texttt{src}}$ is added to the set `st`. At anytime, `st` holds non-overlapping subtrees. The loop in line 7 goes through each root `w` in `st`. `w` is a common ancestor for all vertices in `subtree(w)`. Therefore, all edges among pairs of vertices of `subtree(w)` are traversed in the loops in lines 8 and 9 and the pruning test is executed for them with `w` as their common ancestor in line 12.

After the test is done for all subtrees in `st`, the loop in line 15 goes through the roots in `st` and replaces them with their `succ`s. The new list of subtrees will be used for pruning in loop in line 7. This process continues until `st` is empty (line 17). This is a BFS traversal of the main subtree from $\texttt{v}_{\texttt{src}}$. Note that in our algorithm `subtree(w)` is never stored anywhere but is accessed through `succ` lists as discussed before. Therefore, our algorithm requires at most $O(|V(G)|)$ memory since T has at most $|V(G)|$ edges, which is equal to the sum of the length of the successor lists and the maximum size of set `st` is $|V(G)|$.

Parallelizing pruning is similar to parallelizing DSMR algorithm in Figure 2.5. Operations requiring remote memory accesses are buffered and communicated once there is no more local work.

2.6 Environmental Setup

Machines: Two experimental machines were used for the results: a shared-memory machine with 40 cores (4 10-core Intel Xeon E7-4860) and 128GB of memory; the distributed memory machine Mira, a supercomputer at Argonne. Mira has 49,152 nodes and each node has 16 cores (PowerPC A2) with 16GB of memory. Mira is currently ranked 5^{th} on the TOP500 list.

Graphs: The graphs used in the experiments are:

```
1  // Root vertices of subtrees
2  Set<Vertex> st;
3  void Prune(Vertex v_src){
4    DSMR(v_src); // Run SSSP
5    subtrees.insert(v_src);
6    do {
7      foreach Vertex w in st
8        foreach v in subtree(w)
9          foreach Edge vu in edges(v)
10           if u in subtree(w) && !useless(vu)
11             // Pruning test
12             if d(v)+d(u)-2*d(w)<w(vu)
13               useless(vu) = true;
14     // Go to the subtrees of st
15     foreach Vertex v in sb {
16       sb.remove(v); sb.insert(succ(v)); }
17   } while(!IsEmpty(st)); }
```

Figure 2.9: Pseudo code for Prune engine.

1) Co-Author: Co-Author network represent the connectivity of authors in MathSciNet (American Mathematical Society). It is considered a scale-free network. It has $391,529$ vertices and $873,775$ edges. Vertices represents authors and an article with N authors increases the edge weight between each pairs of authors by $1/(N-1)$ [66]. Consequently, heavier edge weights in this graph represents stronger connectivity. In the experiments, each edge weight w is replaced with $100/w$ so that stronger connections represent shorter distances.

2) US Roads Network: US Roads network is the map of the United States roads. Each edge weight represents the distance between a pair of vertices. It has $23,947,347$ vertices and $58,333,344$ edges and it is not a scale-free network [21].

3) RMAT: RMAT graph model is an artificial scale-free graph generator [15]. An instance of an RMAT graph has the following parameters: 1) *scale*: determines the vertex set size: $|V(G)| = 2^{scale}$, 2) edge factor: determines $|E(G)|/|V(G)|$ ratio, 3) a, b, c and d: determines the skewness of the degree distribution. Edge factor 16 was used as proposed by

Graph500 [32]. For a, b, c and d there are two configurations: type-1 which is Graph500 setup ($a = .57$, $b = c = .19$ and $d = .05$) and type-2 which is SSCA#2 [6] benchmark setup ($a = .55$, $b = c = .1$ and $d = .25$). Edge weights for type-1 and type-2 RMAT graphs are chosen uniformly random from $[0 \dots 256)$ and $[1 \dots 256]$, respectively.

4) Orkut: Orkut is a scale-free social network website and the graph represents its users and their friendship. This network has $3,072,441$ nodes and $117,185,083$ edges. It is originally unweighted and was weighted by distributing edge weights uniformly random from $[1 \dots 256]$.

5) Twitter: Twitter graph represents the follower/following relationship among the users [46]. Each vertex is a user and each edge vu between two users shows v is following u. This graph has 41.7 million users and 1.47 billion edges. This graph is unweighted and edge weights were distributed uniformly random from $[1 \dots 256]$.

2.7 Results

This Section evaluates strong and weak scaling of DSMR with and without the optimizations discussed in Sections 2.4 and 2.5. We also compare our results with the Oracle algorithm (explained next) and the best existing SSSP algorithms.

The Oracle Algorithm: The Oracle Algorithm is an impractical algorithm with minimum amount of work and minimum number of synchronizations. The algorithm works in two phases: 1) An SSSP algorithm such as DSMR is executed for s as the source vertex to compute the final shortest distances for every vertex. The shortest distance for each vertex v, $d_f(v)$, is stored in another array, d', and the values of d are reset to ∞ except that $d(s) = 0$. This phase is not timed. 2) Another SSSP algorithm is executed for the same source vertex s.

A vertex v is scheduled to be relaxed exactly when $d(v) = d'(v)$. By following this schedule, there will be no unnecessary edge relaxations. All-to-all communication takes place not after D relaxations but when all processors have relaxed all vertices they can relax. Therefore, the Oracle algorithm has the minimum number of synchronizations with the schedule just described. The running time for phase 2 is the time that is reported for this algorithm since the time for the first phase is ignored. This algorithm does not necessarily have optimal performance since non-uniform degree distribution may cause load imbalance. We use the Oracle algorithm as a baseline for the evaluation.

2.7.1 Shared Memory Results

Figure 2.10 compares DSMR, our implementation of Δ-Stepping (DS), and the Δ-Stepping from the Elixir collection [70] implemented in the Galois system [45]. This is a strong scaling comparison in which the same input graph across all processor numbers is used. The networks for this evaluation are: Co-Author, US Roads, Orkut and a type-2 RMAT graph with *scale* 22. the experimental machine is the 40-core shared-memory machine. For each algorithm, the best parameters (Δ in Δ-Stepping and D in DSMR) were searched from a random source vertex. These parameters are stable when changing the source vertex and, therefore, we executed the three algorithms with them for 100 other random source vertices. The performance results are presented in TEPS (Traversed Edges Per Second) which is $|E(G)|/T$, where T is the running time in seconds. Note that for computing TEPS, we are not considering the number of edge relaxations but the number of existing edges.

Figure 2.10 shows the result for this evaluation. The X axis represents different number of processors and the Y axis shows the average MTEPS (Mega TEPS) of the 100 random source vertices. As the figure shows, DSMR is faster and scales better than both Δ-Stepping algorithms, except for the US Roads network where DSMR is slower than the Elixir Δ-Stepping with 32 processors. Note that Elixir is a shared memory implementation

while DSMR and our Δ-Stepping algorithms are implemented using MPI. This explains why DSMR is slower than the Elixir Δ-Stepping in Orkut network on less than 16 processors. The speed up of DSMR over Elixir and our Δ-Stepping with 32 processors, respectively, are: for Co-Author 3.59× and 1.64×, for US Roads 0.75× (slow down) and 1.50×, for RMAT22 7.38× and 3.27× and for Orkut 1.74× and 3.19×. Table 2.2 (shared-memory part) shows D, Δ, overhead (with respect to the minimum amount of work) and the number of synchronizations for the experiments in Figure 2.10 with 32 processors. As shown, both DSMR and our Δ-Stepping algorithms have similar overhead but the number of synchronizations are significantly different. This explains the difference in performance.

As it can be seen from Figure 2.10, the Oracle algorithm performs close to DSMR for RMAT22 and Orkut graphs even though, as Table 2.2 indicates, the Oracle algorithm has significantly fewer synchronizations and the overhead is negligible. This is because in each superstep for the Oracle algorithm, the power-law degree distribution causes load imbalance. On the other hand, for US Roads and Co-Author network, the overhead and the number of synchronizations of DSMR are larger. Therefore, the Oracle algorithm performs notably faster than DSMR (1.91× in Co-Author and 4.41× in US Roads with 32 processors).

Now, consider the subgraph extraction results in Figure 2.10. These are shown only for RMAT22 and Orkut since subgraph extraction is not beneficial for the US Roads and Co-Author networks. The HE Extraction column in Table 2.2 shows the thresholds used for the HE property (Section 2.4.1) and the value of $|G'|/|G|$. As it can be seen, graph extraction significantly accelerates DSMR (1.88× in RMAT22 and 2.41× in Orkut with 32 processors). Note that DSMR+Extract is greatly faster than DS+Extract even though, as Table 2.2 shows, G' is a small part of G and the same fix-up code was used for $G \backslash G'$ (a large part of G). Finally, consider the pruned results in Figure 2.10 for Co-Author, RMAT22 and Orkut. Last column of Table 2.2 shows what percentage of each graph was pruned. For Co-Author, it took 84 iterations for the pruning algorithm (Section 2.5) to converge. For

Graph	DSMR			Δ-Stepping			Oracle	*HE* Extraction		Pruned
	D	OH	Syncs	Δ	OH	Syncs	Syncs	TH	$\|G'\|/\|G\|$	
Shared-Memory Results										
Co-Author	2^9	19%	38	2^7	23%	93	24	N/A		21.5%
US Roads	2^5	220%	28831	2^6	221%	47391	12078	N/A		0%
RMAT22	2^{12}	5%	262	2^2	5%	556	35	44	0.17	89.5%
Orkut	2^{14}	4%	120	2^3	4%	187	23	40	0.17	88%
Distributed-Memory Results										
RMAT26	2^{12}	11%	45	2^5	40%	153	37	44	0.17	90.5%
Orkut	2^{12}	40%	19	2^7	101%	33	18	40	0.17	87.3%
Twitter	2^{14}	14%	28	2^6	34%	93	21	26	0.10	91.8%
Weak	2^{16}	11%	27	2^5	15%	79	15	32	0.125	97.1%

Table 2.2: Details of the performance evaluation in Figure 2.10 and 2.11. **OH:** Overhead, **Syncs:** Synchronizations, **TH:** Threshold.

RMAT22 and Orkut, only one iteration was enough. As it can be seen, DSMR+Pruned is better than DSMR+Extract since it removes most of the useless edges. The improvement of DSMR with pruning over DSMR is: 1.22× for Co-Author, 3.12× for RMAT22 and 3.87× for Orkut networks. We excluded DS+Pruned since its difference with DSMR+Pruned is similar to the difference between DS+Extract and DSMR+Extract.

2.7.2 Distributed Memory Results

Figure 2.11 shows results to Figure 2.10 for the distributed-memory machine, Mira, with larger graphs. Plots (a), (b) and (c) show results for three fixed-size graphs (strong scaling): a type-2 RMAT with *scale* 26, Orkut and Twitter, respectively. Plots (d) and (e) show weak scaling result of a type-1 RMAT graph compared with the results reported in [14]. Similarly, the best D and Δ values were searched for DSMR and Δ-Stepping from a random source vertex and used for 100 different random source vertices for our experiments. The distributed-memory part of Table 2.2 for the maximum number of processors that DSMR scales: 4096 for RMAT26, 2048 for Orkut, 4096 for Twitter and 8192 for the weak scaling

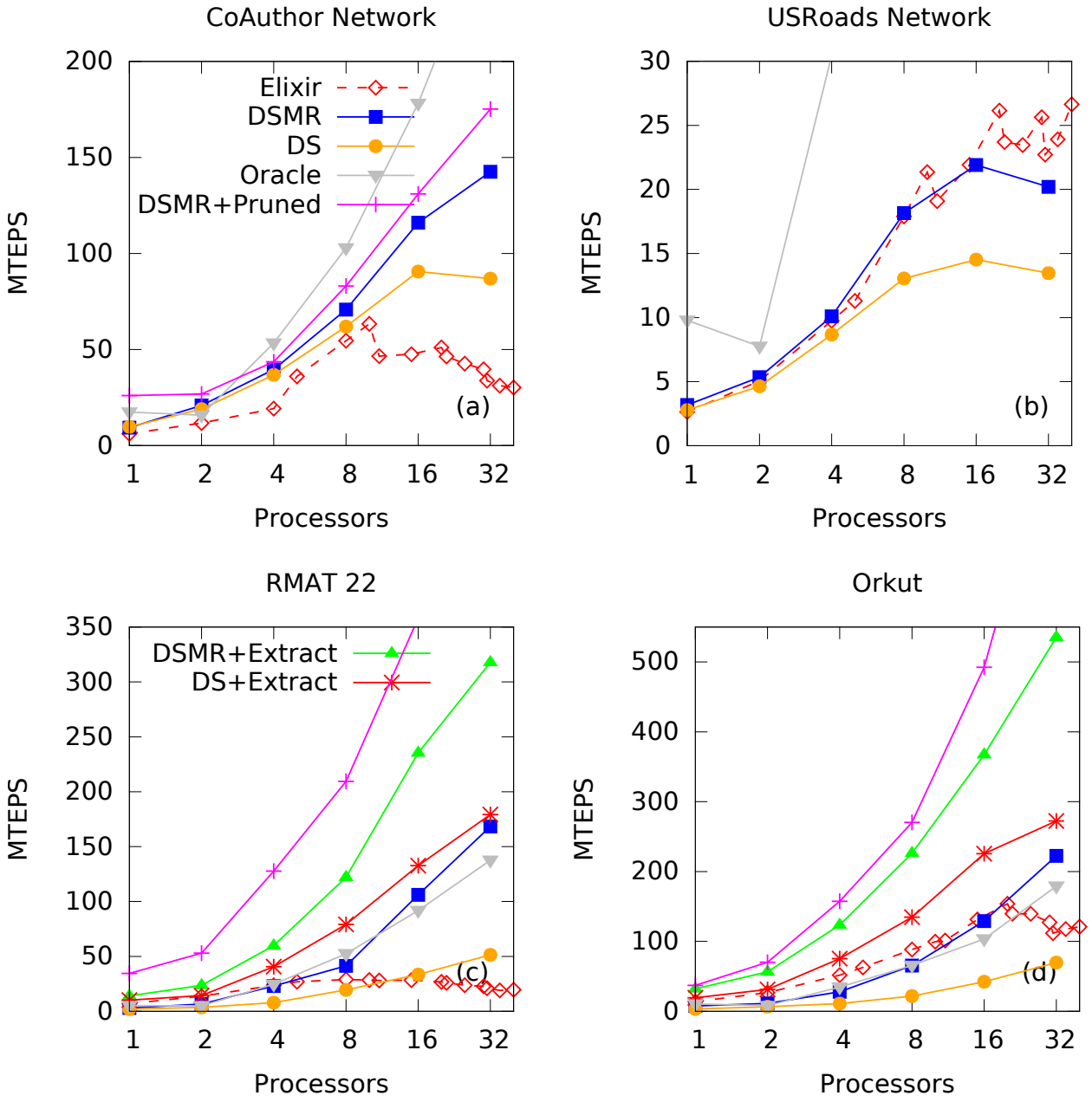


Figure 2.10: Evaluation of DSMR, Δ-Stepping, Elixir and Oracle algorithms on the shared-memory machine. For readability, we do not show some data points for Oracle US Roads and Pruned Orkut

results.

First, consider the strong scaling results in plots (a), (b) and (c) in Figure 2.11. As was the case for shared-memory, DSMR scales better than our Δ-Stepping. It is better by a factor of 2.05 in plot (a), 1.60 in plot (b) and 1.78 in plot (c). Unlike the shared-memory results, as Table 2.2 shows, the overhead of DSMR is noticeably less than that of Δ-Stepping (2.5 times less on average). DSMR also has significantly fewer synchronizations than Δ-Stepping. While the overhead of DSMR and Δ-Stepping are similar in shared memory, in distributed memory the overhead of Δ-Stepping is significantly larger. The reason is that since the communication cost in distributed-memory machines is high, the value of Δ that obtains the best performance reduces the number of synchronizations, but it does that at the expense of doing useless work. This explains why DSMR performs better than Δ-Stepping. On the other hand, as Table 2.2 shows, the number of synchronizations for DSMR is close to that of the Oracle algorithm, as Table 2.2 shows. The overhead is small (except for Orkut). That explains why the Oracle algorithm and DSMR lines are close in the three plots in Figure 2.11.

Now, consider the subgraph extraction optimization results for plots (a), (b) and (c) in Figure 2.11. As in the shared-memory results, this technique improves the performance of DSMR significantly: up-to a factor of 2.90 in RMAT26, 4.03 in Orkut and 3.02 in Twitter. Table 2.2 shows the thresholds used for the subgraph extraction with the HE property. DSMR+Extract scales better than DS+Extract. This shows the impact of DSMR on performance, in spite of the small ratio of G' over G. Finally, consider the pruned results in Figure 2.11. As in the shared-memory results, pruning improves the performance of the algorithm significantly since many edges are identified as useless by the pruning algorithm, as Table 2.2 shows. Only one iteration of the pruning algorithm was executed for all three graphs in Figure 2.11. The speed-ups of DSMR+Pruned over DSMR are up-to : 5.46× for RMAT26, 6.33× for Orkut and 5.59× for Twitter.

Plots (d) and (e) in Figure 2.11 show weak scaling results for type-1 RMAT graphs. The

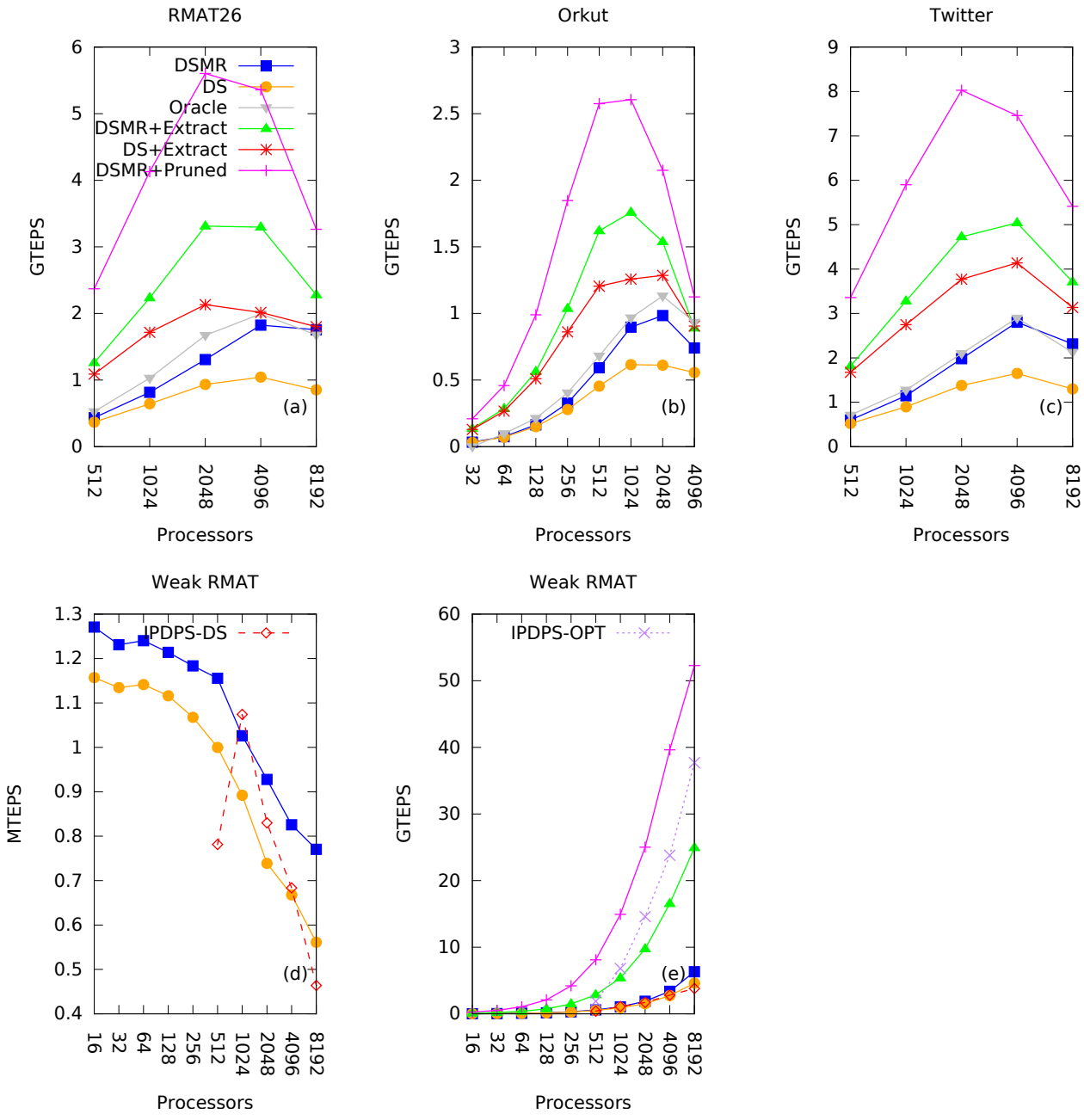


Figure 2.11: Performance comparison of Δ-Stepping, DSMR, Graph Extraction Optimization and Pruning. Plots (a), (b) and (c) show strong scaling results and plots (d) and (e) show weak scaling results for RMAT graphs

RMAT *scale* is $17 + k$ for 2^k processors. Plot (d) compares the performance per processor in MTEPS for DSMR, our Δ-Stepping (DS) and the version of Δ-Stepping described in [14] (IPDPS-DS). This is a descending plot since the communication costs increase with the number of processors. As before, the ratio of DSMR over our Δ-Stepping increases with the number of processors and DSMR runs up-to a factor of 1.37 faster. On the the other hand, IPDPS-DS is faster than DSMR with 1024 processors (1.04×) but the decreasing slope of IPDPS-DS is faster than that of DSMR, which makes it 1.66× slower than DSMR for 8192 processors.

Plot(e) in Figure 2.11 compares the absolute performance of DSMR with and without the optimizations. The subgraph extraction for this plot includes both properties discussed in Section 2.4, HE and LD and it improves the performance of DSMR by up-to 4.76×. The threshold for subgraph extraction with the LD property depends on on the size of the graph and is variable and therefore, we do not report it. Lastly, as it can be seen from the last row of Table 2.2, pruning removes around 97% of the edges from type-1 RMAT graphs. Consequently, DSMR+Pruned in Figure 2.11 provides a speed-up of up-to 13× over DSMR. Authors of [14] applied a set of optimizations to their implementation of Δ-Stepping. IPDPS-OPT in Plot (e) in Figure 2.11 shows their best result. As it can be seen, our pruning results improve upon IPDPS-OPT by factors between $1.38 - 4.26$.

Given the results with shared and distributed memory systems, DSMR always scales better and runs faster than our Δ-Stepping. DSMR is only slower than Elixir Δ-Stepping for the US Roads network but as shown by the results of the Oracle algorithm, there is much room for improvement in this graph and Δ-Stepping is not necessarily the best algorithm. Also, by observing the impact of pruning and subgraph extraction, we can conclude that either these techniques are very effective for real-world graphs or the way edge weights are artificially distributed should be reconsidered.

2.8 Conclusions

This chapter discussed parallelization of SSSP problem which is an amorphous problem. We we introduced DSMR, a new parallel SSSP algorithm and showed how the parameter D can control the balance between the number of communications and the overall overhead. We discussed why we expect DSMR to perform better than Δ-Stepping on scale-free networks. Our results show that, for shared memory, DSMR is faster than our own implementation of Δ-Stepping in all cases and only slower than Elixir Δ-Stepping in the case of US Roads network (25% slower). However, DSMR is faster than Elixir Δ-Stepping on all the other graphs by up-to 7.38×. For distributed-memory systems, DSMR is faster than our Δ-Stepping implementation by up-to 2.05× and by up-to 1.66× faster than the best existing SSSP algorithm for distributed-memory systems. We have also introduced subgraph extraction and pruning techniques, which improved performance by up-to 4.76× and 13×, respectively.

Later in Chapter 4, we will use the findings from this chapter to discuss the necessary features for a parallel framework for amorphous problems. As presented, performing computation asynchronously is the key in delivering an efficient SSSP algorithm. Therefore, having control over the asynchronous computation is an essential feature for a parallel graph framework. Next, we will discuss another set of problems which are surprisingly instances of SSSP except that the graphs have specific structures. We similarly try to find a balance between communication and computation in this set of problems.

Chapter 3

Parallelizing Dynamic Programming Algorithms

The next set of problems that we studied in this thesis are dynamic programming algorithms. Dynamic programming [9] is a method to solve a variety of important optimization problems in computer science, economics, genomics, and finance. Figure 3.1 describes two such examples: Viterbi, which finds the most-likely path through a hidden-Markov model for a sequence of observations and LCS, which finds the longest common subsequence between two input strings. Dynamic programming algorithms proceed by recursively solving a series of subproblems, usually represented as cells in a table as shown in the figure. The solution to a subproblem is constructed from solutions to an appropriate set of subproblems, as shown by the respective recurrence relation in the figure.

These data-dependences naturally group subproblems into *stages* whose solutions do not depend on each other. For example, all subproblems in a column form a stage in Viterbi and all subproblems in an anti-diagonal form a stage in LCS. A predominant method for parallelizing dynamic programming is *wavefront* parallelization [62], which computes all

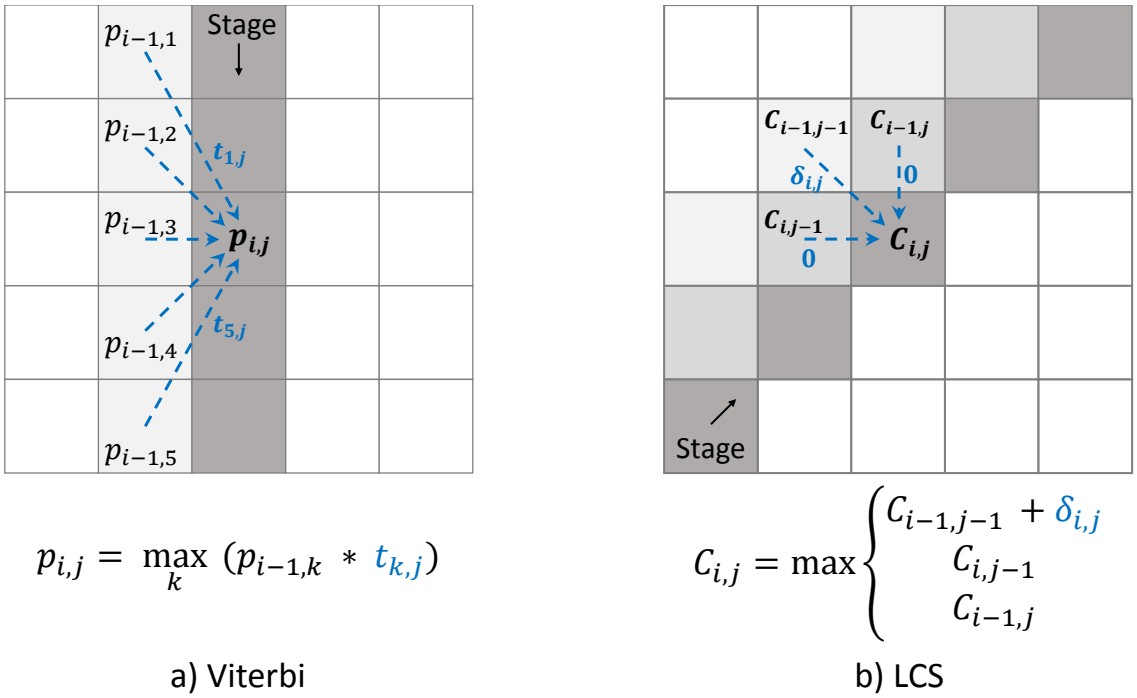


Figure 3.1: Dynamic programming examples with dependences between stages.

subproblems within a stage in parallel.[1]

In contrast, the methods in this chapter break data-dependences *across* stages and fix up incorrect values later in the algorithm. This is similar to the subgraph extraction idea from Chapter 2 where incorrect computations in the first phase are fixed in the fix-up engine. In here, the new approaches expose parallelism for a class of dynamic programming algorithms we call *linear-tropical dynamic programming* (LTDP). A LTDP computation can be viewed as performing a sequence of matrix multiplications in the tropical semiring where the semiring is formed with + as the multiplicative operator and max as the additive operator. This chapter demonstrates that several important optimization problems such as Viterbi, LCS, Smith-Waterman, and Needleman-Wunsch (the latter two are used in bioinformatics for sequence alignment) belong to LTDP. To efficiently break data-dependences across stages,

[1]The definition of wavefront parallelism used here is more general and includes the common usage where a wavefront performs computations across logical iterations as in the LCS example in Figure 3.1(a).

the algorithm uses *rank convergence*, a property by which the rank of a sequence of matrix products in the tropical semiring is likely to decrease quickly.

There are two methods that will be introduced in this chapter: the Rank-1 method and the Delta method. The Rank-1 method is general in the sense that the data-dependences can have arbitrary shapes. In other words, the Rank-1 method can target any amorphous LTDP problem. However, one of the disadvantages of the Rank-1 method is that it only works when the rank of matrix products converges to 1. The Delta method, on the other hand, works with ranks greater than 1 but it only supports specific data-dependence shapes.

A key advantage of our parallel methods is their ability to simultaneously use both the coarse-grained parallelism across stages and the fine-grained wavefront parallelism within a stage. Our implementations achieve multiplicative speed ups over existing implementations. For instance, the parallel Viterbi decoder is up-to 24× faster with 64 cores than a state-of-the-art commercial baseline [71]. This chapter demonstrates similar speed ups for other LTDP instances studied in this chapter.

3.1 Background

In linear algebra, a matrix-vector multiplication maps a vector from an input space to an output space. If the matrix is of low rank, the matrix maps the vector to a subspace of the output space. In particular, if the matrix has rank 1, then it maps all input vectors to a *line* in the output space. These geometric intuitions hold even when one changes the meaning of the sum and multiplication operators (say to max and +, respectively), as long as the new meaning satisfies the following rules.

Semirings A semiring is a five-tuple $(D, \oplus, \otimes, \mathbb{0}, \mathbb{1})$, where D is the domain of the semiring that is closed under the additive operation $\oplus$ and the multiplicative operation $\otimes$. The two

operations satisfy the following properties:

- $(D, \oplus, \mathbb{0})$ forms a commutative monoid with $\mathbb{0}$ as the identity
  - associativity: $\forall x, y, z \in D : (x \oplus y) \oplus z = x \oplus (y \oplus z)$
  - identity: $\forall x \in D : x \oplus \mathbb{0} = x$
  - commutativity: $\forall x, y \in D : x \oplus y = y \oplus x$
- $(D, \otimes, \mathbb{1})$ forms a monoid with $\mathbb{1}$ as the identity
  - associativity: $\forall x, y, z \in D : (x \otimes y) \otimes z = x \otimes (y \otimes z)$
  - identity: $\forall x \in D : x \otimes \mathbb{1} = \mathbb{1} \otimes x = x$
- $\otimes$ left- and right-distributes over $\oplus$
  - $\forall x, y, z \in D : x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$
  - $\forall x, y, z \in D : (y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$
- $\mathbb{0}$ is an annihilator for $\otimes$
  - $\forall x \in D : x \otimes \mathbb{0} = \mathbb{0} \otimes x = \mathbb{0}$

Tropical Semiring The semiring $(\mathbb{R} \cup \{-\infty\}, \max, +, -\infty, 0)$ with the real numbers extended with $-\infty$ as the domain, max as the additive operation $\oplus$, and $+$ as the multiplicative operation $\otimes$ is called the tropical semiring. All properties of a semiring hold with $-\infty$ as the additive identity $\mathbb{0}$ and 0 as the multiplicative identity $\mathbb{1}$. Alternately, one can reverse the sign of every element in the domain and change the additive operation to min.

Matrix Multiplication Let $A_{n \times m}$ denote a matrix with n rows and m columns with elements from the domain of the tropical semiring. Let $A[i, j]$ denote the element of A at

the ith row and jth column. The matrix product of $A_{l\times m}$ and $B_{m\times n}$ is $A \odot B$, a $l \times n$ matrix defined such that

$$\begin{aligned}(A \odot B)[i,j] &= \bigoplus_{1\le k\le m} (A[i,k] \otimes B[k,j]) \\ &= \max_{1\le k\le m} (A[i,k] + B[k,j])\end{aligned}$$

Note, this is the standard matrix product with multiplication replaced by $+$ and addition replaced by max.

The transpose of $A_{n\times m}$ is the matrix $A^{\intercal}_{m\times n}$ such that $\forall i,j : A^{\intercal}[i,j] = A[j,i]$. Using standard terminology, we will denote a $v_{n\times 1}$ matrix as the column vector $\vec{v}$, a $v_{1\times n}$ matrix as the row vector $\vec{v}^{\intercal}$, and $x_{1\times 1}$ matrix simply as the scalar x. This terminology allows us to extend the definition of matrix-matrix multiplication above to matrix-vector, scalar-matrix, and scalar-vector multiplication appropriately. Also, $\vec{v}\,[i]$ is the ith element of a vector $\vec{v}$. The following lemma follows from the associativity, distributivity, and commutativity properties of $\otimes$ and $\oplus$ in a semiring.

Lemma 3.1.1. *Matrix multiplication is associative in semirings*

$$(A \odot B) \odot C = A \odot (B \odot C)$$

Parallel Vectors Two vectors $\vec{u}$ and $\vec{v}$ are parallel in the tropical semiring, denoted as $\vec{u} \parallel \vec{v}$, if there exist scalars x and y such that $\vec{u} \odot x = \vec{u} \odot y$. Intuitively, parallel vectors in tropical semiring $\vec{u}$ and $\vec{v}$ differ by a constant offset. For instance, $[1\,0\,2]^{\intercal}$ and $[3\,2\,4]^{\intercal}$ are parallel vectors differing by an offset 2. Note that the definition above requires two scalars as $-\infty$ does not have a multiplicative inverse in the tropical semiring.

Matrix Rank The rank of a matrix $M_{m\times n}$, denoted by $\text{rank}(M)$, is the smallest number r such that there exist matrices $C_{m\times r}$ and $R_{r\times n}$ whose product is M. In particular, a rank-1 matrix is a product of a column vector and a row vector. There are alternate ways to define the rank of a matrix in semirings, such as the number of linearly independent rows or columns in a matrix. While such definitions coincide in fields (which have inverses for $\oplus$ and $\otimes$), they are not equivalent in semirings [23].

Lemma 3.1.2. *For any vectors $\vec{u}$ and $\vec{v}$ and a matrix A of rank 1, $A\odot\vec{u} \parallel A\odot\vec{v}$*

Intuitively, this lemma states that a rank-1 matrix maps all vectors to a line. If $\text{rank}(A) = 1$ then it is a product of some column vector $\vec{c}$ and a row vector $\vec{r}^\intercal$. For any vectors $\vec{u}$ and $\vec{v}$:

$$A\odot\vec{u} = (\vec{c}\odot\vec{r}^\intercal)\odot\vec{u} = \vec{c}\odot(\vec{r}^\intercal\odot\vec{u}) = \vec{c}\odot x_u$$
$$A\odot\vec{v} = (\vec{c}\odot\vec{r}^\intercal)\odot\vec{v} = \vec{c}\odot(\vec{r}^\intercal\odot\vec{v}) = \vec{c}\odot x_v$$

for appropriate scalars x_u and x_v. As an example, consider

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \qquad \vec{u} = \begin{bmatrix} 1 \\ -\infty \\ 3 \end{bmatrix} \qquad \vec{v} = \begin{bmatrix} -\infty \\ 2 \\ 0 \end{bmatrix}$$

$A = [1\,2\,3]^\intercal \odot [0\,1\,2]$ is rank-1. $A\odot\vec{u} = [6\,7\,8]^\intercal$ and $A\odot\vec{v} = [4\,5\,6]^\intercal$ which are parallel with a constant offset 2. Also note that all rows in a rank-1 matrix are parallel to each other.

3.2 Linear-Tropical Dynamic Programming

Dynamic programming is a method for solving problems that have optimal substructure — the solution to a problem can be obtained from the solutions to a set of its overlapping subproblems. This dependence between subproblems is captured by a recurrence equation.

Classic dynamic programming implementations solve the subproblems iteratively applying the recurrence equation in an order that respects the dependence between subproblems.

LTDP Definition A dynamic programming problem is linear-tropical dynamic programming (LTDP), if (a) the subproblems can be grouped into a sequence of stages such that the solution to a subproblem in a stage only depends on the solutions in the previous stage and (b) this dependence is linear in the tropical semiring. In other words, $s_i[j]$, the solution to subproblem j in stage i of LTDP, is given by the recurrence equation

$$s_i[j] = \max_k(s_{i-1}[k] + A_i[j,k]) \tag{3.1}$$

for appropriate constants $A_i[j,k]$. This linear dependence allows us to view LTDP as computing a sequence of vectors $\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_n$, where

$$\vec{s}_i = A_i \odot \vec{s}_{i-1} \tag{3.2}$$

for an appropriate matrix of constants A_i derived from the recurrence equation. In this equation, we will call $\vec{s}_i$ as the *solution vector* at stage i and call A_i as the *transformation matrix* at stage i. Also, $\vec{s}_0$ is the initial solution vector obtained from the base case of the recurrence equation.

Predecessor Product Once all of the subproblems are solved, finding the solution to the underlying optimization problem of LTDP usually involves tracing the *predecessors* of subproblems. A predecessor of a subproblem is the subproblem for which the maximum in Equation 3.1 is reached. For ease of exposition, we define the *predecessor product* of a matrix

```
1  LTDP_Seq (vector s_0, matrix A_1..A_n) {
2    vector pred[1..n];  vector res;
3    // forward
4    s = s_0;
5    for i in (1..n) {
6      pred[i] = A_i ⋆ s; // pred[i] = p_i
7      s       = A_i ⊙ s; // s = s_i
8    }
9    // backward
10   res[n+1] = 0;  // res = r
11   for i in (n..1) {
12     res[i] = pred[i][res[i+1]];        }
13   return res;                          }
```

Figure 3.2: LTDP implementation that computes the stages sequentially. An implementation can possibly employ wavefront parallelization within a stage.

A and a vector $\vec{s}$ as the vector $A \star \vec{s}$ such that

$$(A \star \vec{s})[j] = \arg\max_k (\vec{s}[k] + A[j,k])$$

Note the similarity between this definition and Equation 3.1. We assume that ties in arg max are broken deterministically. The following lemma shows that predecessor products do not distinguish between parallel vectors, a property that will be useful later.

Lemma 3.2.1. $\vec{u} \parallel \vec{v} \implies \forall A : A \star \vec{u} = A \star \vec{v}$

This follows from the fact that parallel vectors in the tropical semiring differ by a constant and that arg max is invariant when a constant is added to all its arguments.

Sequential LTDP Figure 3.2 shows the *sequential* algorithm for LTDP phrased in terms of matrix multiplications and predecessor products. This algorithm is deemed sequential because it computes the stages one after the other based on the data-dependence in Equation 3.1. However, the algorithm can utilize wavefront parallelism to compute the solutions within a stage in parallel.

The inputs to the sequential algorithm are the initial solution vector $\vec{s}_0$ and transformation matrices $A_1, \ldots, A_n$, which respectively capture the base and inductive case of the LTDP recurrence equation. The algorithm consists of a forward phase and a backward phase. The forward phase computes the solutions in each stage $\vec{s}_i$ iteratively. In addition, it computes the predecessor product $\vec{p}_i$ that determines the predecessor for each solution in a stage. The backward phase iteratively follows the predecessors computed in the forward phase. The algorithm assumes that the first subproblem in the last stage, $\vec{v}_n[0]$, contains the desired solution to the underlying optimization problem. Accordingly, the backward phase starts with 0 in Line 10. The resulting vector `res` is the solution to the optimization problem at hand (e.g., the longest-common-subsequence of the two input strings).

The exposition above consciously hides a lot of details in the $\odot$ and $\star$ operators. An implementation does not need to represent the solutions in a stage as a vector and perform matrix-vector operations. It might statically know that the current solution depends on some of the subproblems in the previous stage (a sparse matrix) and only accesses those. Finally, as mentioned above, an implementation might use wavefront parallelism to compute the solutions in a stage in parallel. All these implementation details are orthogonal to how the parallel algorithms described in this chapter parallelize across stages.

3.3 The Parallel Rank-1 Method

This section describes an efficient algorithm for parallelizing the sequential algorithm in Figure 3.2 across stages.

3.3.1 Breaking Data-Dependences Across Stages

Viewing LTDP computation as matrix multiplication in the tropical semiring provides a way to break data-dependences among stages. Consider the solution vector at the last stage $\vec{s}_n$.

From Equation 3.2, we have

$$\vec{s}_n = A_n \odot A_{n-1} \ldots A_2 \odot A_1 \odot \vec{s}_0$$

Standard techniques [48, 35] can parallelize this computation using the associativity of matrix multiplication. For instance, two processors can compute the partial products $A_n \odot \ldots \odot A_{n/2+1}$ and $A_{n/2} \odot \ldots \odot A_1$ in parallel, and multiply their results with $\vec{s}_0$ to obtain $\vec{s}_n$.

However, doing so converts a sequential computation that performs matrix-vector multiplications to a parallel computation that performs matrix-matrix multiplications. This results in a parallelization overhead linear in the size of the stages and thus requires linear number of processors to observe constant speed ups. In practice, the size of stages can easily be hundreds or larger and thus is not practical on real problems and hardware.

The key contributions of this chapter are parallel algorithms that avoid he overhead of matrix-matrix multiplications. These algorithms rely on the convergence of matrix rank in the tropical semiring as discussed below. Its exposition requires the following definition.

Partial Product For a given LTDP instance, the partial product $M_{i \to j}$, defined for stages $j \geq i$, is given by

$$M_{i \to j} = A_j \odot \ldots A_{i+1} \odot A_i$$

Partial product determines how a later stage j depends on stage i as $\vec{s}_j = M_{i \to j} \odot \vec{s}_i$.

3.3.2 Rank Convergence

Rank of the product of two matrices is not greater than the rank of the individual matrices.

$$\text{rank}(A \odot B) \leq \min(\text{rank}(A), \text{rank}(B)) \tag{3.3}$$

This is because, if $\text{rank}(A) = r$, then $A = C \odot R$ for some matrix C with r columns. Thus, $A \odot B = (C \odot R) \odot B = C \odot (R \odot B)$ implying that $\text{rank}(A \odot B) \leq \text{rank}(A)$. Similar argument shows that $\text{rank}(A \odot B) \leq \text{rank}(B)$.

Equation 3.3 implies that for stages $k \geq j \geq i$

$$\text{rank}(M_{i \to k}) \leq \text{rank}(M_{i \to j}) \leq \text{rank}(A_i)$$

In effect, as the LTDP computation proceeds, the rank of the partial products will never increase. Theoretically, there is a possibility that the ranks do not decrease. However, we have only observed this for carefully crafted problem instances that are unlikely to occur in practice. On the contrary, the rank of these partial products is likely to converge to 1, as will be demonstrated in Section 3.6.1.

Consider a partial product $M_{i \to j}$ whose rank is 1. Intuitively, this implies a *weak* dependence between stages i and j. Instead of the actual solution vector, $\vec{s_i}$, say the LTDP computation starts with a different vector $\vec{t_i}$ at stage i. From Lemma 3.1.2, the new solution vector at stage j, $\vec{t_j} = M_{i \to j} \odot \vec{t_i}$, is parallel to the actual solution vector $\vec{s_j} = M_{i \to j} \odot \vec{s_i}$. Essentially, the direction of the solution vector at stage j is independent of stage i. The latter stage only determines its magnitude. In the tropical semiring, where the multiplicative operator is $+$, this means that the solution vector at stage j will be off by a constant if one starts stage i with an arbitrary vector.

3.3.3 The Parallel Rank-1 Method Overview

The parallel Rank-1 method uses this insight to break dependences between stages as shown pictorially in Figure 3.3. The figure uses three processors as an example. Figure 3.3(a) represents the forward phase of the sequential algorithm described in Figure 3.2. Each stage is represented as a vertical column of cells and an arrow between stages represents

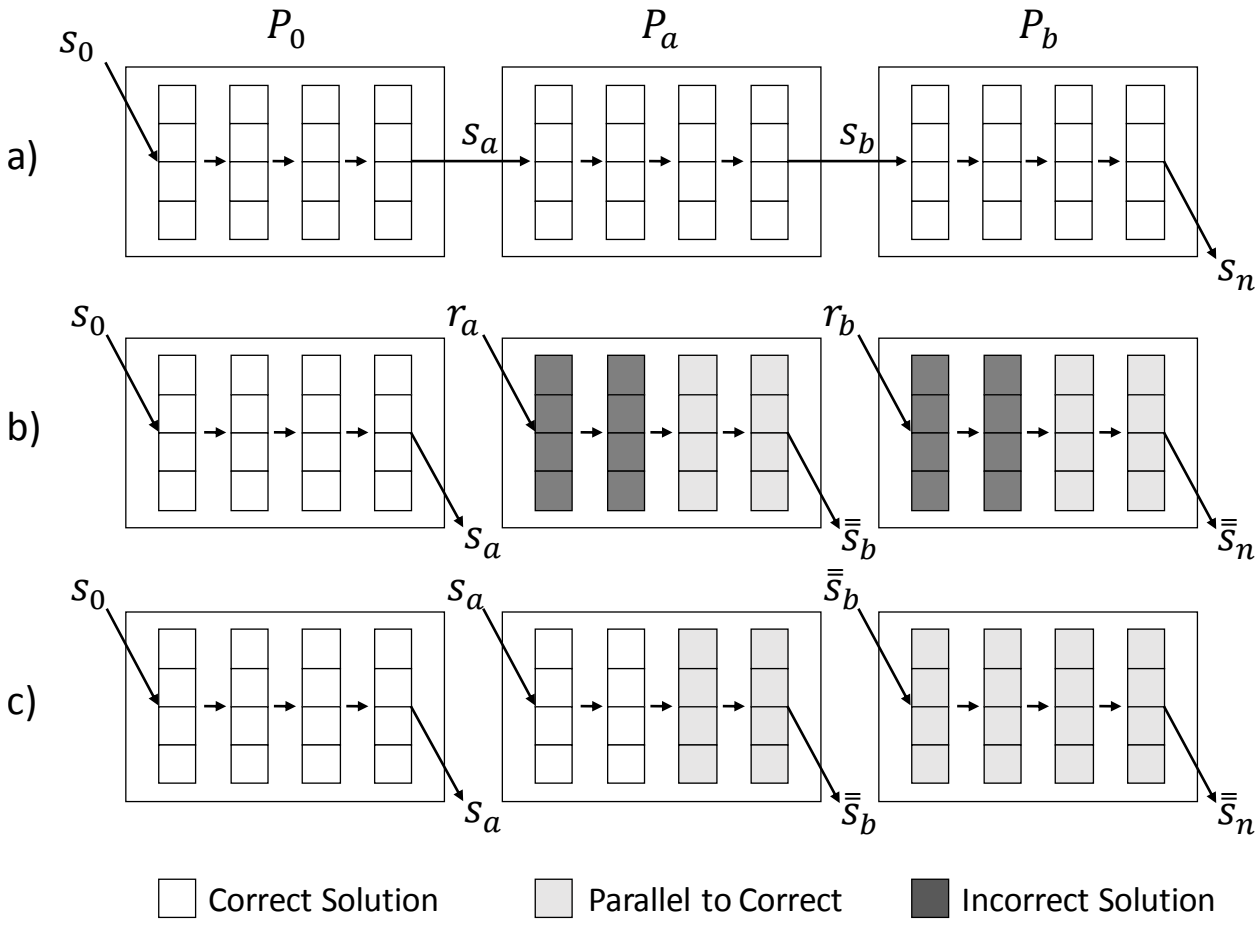


Figure 3.3: Rank-1 method parallelization using rank convergence.

a multiplication with an appropriate transformation matrix. Processor P_0 starts from the initial solution vector s_0 and computes all its stages. Processor P_a waits for s_a, the solution vector in the final stage of P_0, in order to start its computation. Similarly, processor P_b waits for s_b the solution vector at the final stage of P_a.

In Figure 3.3(b), processors P_a and P_b start from *arbitrary* solutions r_a and r_b respectively in parallel with P_0. Of course, the solutions for the stages computed by P_a and P_b will start out as completely wrong (shaded dark in the figure). However, if rank convergence occurs then these erroneous solution vectors will eventually become parallel to the actual solution vectors (shaded gray in the figure). Thus, P_a will generate some solution vector $\bar{\bar{s}}_b$ parallel to s_b and P_b will generate some solution vector $\bar{\bar{s}}_n$ parallel to s_n.

In a subsequent *fix up phase*, shown in Figure 3.3(c), P_a uses s_a computed by P_0 and P_b uses $\bar{\bar{s}}_b$ computed by P_1 to fix stages that are not parallel to the actual solution vector at that stage. After the fix up, the solution vectors at each stage are either the same as or parallel to the actual solution vector at those respective stages.

For LTDP, it is not necessary to compute the actual solution vectors. As parallel vectors generate the same predecessor products (Lemma 3.2.1), following the predecessors in Figure 3.3(c) will generate the same solution as the following the predecessors in Figure 3.3(a).

The next sections describe the parallel algorithm in more detail.

3.3.4 Parallel Forward Phase for the Rank-1 Method

The goal of the parallel forward phase in Figure 3.4 is to compute a solution vector `s[i]` at stage i that is *parallel* to the actual solution vector $\vec{s}_i$, as shown in Figure 3.3. During the execution of the algorithm, we say that a stage i has *converged* if `s[i]` computed by the algorithm is parallel to its actual solution vector $\vec{s}_i$.

The parallel Rank-1 method splits the stages equally among P processors such that a processor p owns stages between $\texttt{l}_\texttt{p}$ (exclusive) and $\texttt{r}_\texttt{p}$ (inclusive), as shown in line 5. While

```
1  Rank1_Method(vector s_0, matrix A_1..A_n) {
2    vector s[1..n];  vector pred[1..n];
3    vector conv;
4    // proc p owns stages (l_p..r_p]
5    ∀p: l_p = n/P*(p-1); r_p = n/P*p;
6    // parallel forward phase
7    parallel.for p in (1..P) {
8      local s = (p == 1 ? s_0 : nz);
9      for i in (l_p+1..r_p) {
10       pred[i]  = A_i ⋆ s;
11       s = s[i] = A_i ⊙ s;     }}
12   --- barrier ---
13   do {  // till convergence (fix up loop)
14     parallel.for p in (2..P) {
15       conv[p] = false;
16       // obtain final soln from prev proc
17       s = s[l_p];
18       for i in (l_p+1..r_p) {
19         pred[i] = A_i ⋆ s;
20         s       = A_i ⊙ s;
21         if( s is parallel to s[i] ) {
22           conv[p] = true;
23           break;                    }
24         s[i] = s;                   }}
25     --- barrier ---
26     conv = ⋀_p conv[p];
27   } while (!conv);
28
29   //parallel backward phase is in Figure 3.5
30   return Backward_Par(pred);
```

Figure 3.4: Parallel algorithm for the forward Pass of Rank-1 method that relies on rank convergence for efficiency. All inter-processor communication is shown in magenta.

processor 1 starts its computation from $\vec{s}_0$, other processors start from some vector `nz` (line 8). This initial vector can be arbitrary except none of its entries can be $\mathbb{0} = -\infty$. Section 3.3.5 explains the importance of this constraint.

The loop starting in line 9 is similar to the sequential forward phase (Figure 3.2) except that the parallel version additionally stores the computed `s[i]` needed in the convergence loop below.

Consider a processor $p \neq 1$ that owns stages $(\mathtt{l}_\mathrm{p} = l \ldots r = \mathtt{r}_\mathrm{p}]$. If there exists a stage

k in $(l \ldots r]$ such that rank$(M_{l \to k})$ is 1, then stage k converges, irrespective of the initial vector `nz` (Lemma 3.1.2). Moreover, by Equation 3.3, rank$(M_{l \to j})$ is 1 for all stages j in $[k \ldots r]$, implying that these stages converge as well (Figure 3.3(b)). However, processor p is not cognizant of the actual solution vectors and, thus, does not know the value of k or whether such a k exists.

The fix up loop starting at line 13 (fix up phase in Figure 3.3(c)) fixes stages $i < k$. In this loop, processor p receives the vector at stage l computed by the previous processor $p-1$. (Figure 3.4 shows all such inter-processor communication in magenta.) Processor p then updates `s[i]` for all stages till the new value becomes parallel to the old value of `s[i]` (line 21). This ensures that all stages owned by p have converged, under the assumption that stage l has converged.

In addition, the Boolean variable `conv[p]` indicates whether processor p advertised a converged value for its last stage to processor $p+1$ at the beginning of the iteration. Thus, when `conv` (line 26) is true, all stages have converged. In the ideal case, every processor has a partial product with rank 1, and thus, the fix up loop executes exactly one iteration. Section 3.6 shows that we observe the best case for many practical instances.

Say, however, `conv[p]` is not true for processor p. This indicates that the stages $(\mathtt{l}_\mathrm{p} \ldots \mathtt{r}_\mathrm{p}]$ was not large enough to generate a partial product with rank 1. In the next iteration of the fix up phase, processor $p+1$, in effect, searches for rank convergence in the wider range $(\mathtt{l}_\mathrm{p} \ldots \mathtt{r}_{\mathrm{p}+1}]$. The fix up loop iteratively combines the stages of the processors till all processors converge. In the worst case, the fix up loop executes $P-1$ iterations and the parallel Rank-1 method devolves to the sequential case.

Important to note is that even though the discussion above refers to partial products, the Rank-1 method does not perform any matrix-matrix multiplications. Like the sequential algorithm, the presentation hides many implementation details in the $\star$ and $\odot$ operations (in lines 10,11,19,and 20). In fact, the parallel implementation can reuse efficient imple-

mentations of these operations, including those that use wavefront parallelism, from existing sequential implementations. Also, the computation of `conv` at line 26 is a standard reduce operation that is easily parallelized, if needed.

When compared to the sequential algorithm, the parallel Rank-1 method has to additionally store `s[i]` per stage required to test for convergence in the fix up loop. If space is a constraint, then the fix up loop can be modified to *recompute* `s[i]` in each iteration, trading compute for space.

3.3.5 All-Non-Zero Invariance

A subtle issue with the correctness of the algorithm above is that starting the LTDP computation midway with an arbitrary initial vector `nz` could produce a zero vector (one with all $\mathbb{0} = -\infty$ entries) at some stage. If this happens, all subsequent stages will produce a zero vector resulting in an erroneous result. To avoid this, we ensure that `nz` is *all-non-zero*, i.e. none of its elements are $\mathbb{0} = -\infty$.

A transformation matrix A is non-trivial, if every row of A contains at least one nonzero entry. In Equation 3.1, the j row of matrix A_i captures how the subproblem j in stage i depends on the subproblems in stage $i-1$. If *all* entries in this row are $-\infty$, then the subproblem j is forced to be $-\infty$ for any solution to stage $i-1$. Such trivial subproblems can be removed from a given LTDP instance. So, we can safely assume that transformation matrices in LTDP instances are non-trivial.

Lemma 3.3.1. *For a non-trivial transformation matrix A,*

$$\vec{v} \textit{ is all-non-zero} \implies A \odot \vec{v} \textit{ is all-non-zero}$$

$(A \odot \vec{v})[i] = max_k(A[i,k] + \vec{v}[k])$. But $A[i,k] \neq -\infty$ for some k ensuring that at least one of the arguments to max is not $-\infty$. Here we rely on the fact that no element has an inverse

```
1  Backward_Par(vector pred[1..n]) {
2    vector res;      vector conv;
3    // proc p owns stages (l_p..r_p]
4    ∀p: l_p = n/P*(p-1); r_p = n/P*p;
5    // parallel backward phase
6    parallel.for p in (P..1){
7      // all processors start from 0
8      local x = 0;
9      for i in (r_p..l_p+1) {
10       x = res[i] = pred[i][x]; }}
11   --- barrier ---
12   do {  // till convergence (fix up loop)
13     parallel.for p in (P-1..1) {
14       conv[p] = false;
15       // obtain final result from next proc
16       local x = res[r_p+1];
17       for i in (r_p..l_p+1) {
18         x = pred[i][x];
19         if (res[i] == x)
20           conv[p] = true;
21           break;                }
22         res[i] = x;               }
23     --- barrier ---
24     conv = ⋀_p conv[p];
25   } while (!conv)
26   return res;                     }
```

Figure 3.5: Parallel algorithm for the backward phase of LTDP that relies on rank convergence for efficiency. All inter-processor communication is shown in magenta.

under max, except $-\infty$. As such this lemma is not necessarily true in other semirings.

Thus, starting with a all-non-zero vector ensures that none of the stages result in a zero vector.

3.3.6 Parallel Backward Phase

Once the parallel forward phase of the Rank-1 method is done, performing the sequential backward phase from Figure 3.2 will generate the right result, even though `s[i]` is not exactly the same as the correct solution $\vec{s}_i$. In many applications, the forward phase overwhelmingly dominates the execution time and parallelizing the backward phase is not necessary. If this

is not the case, the backward phase can be parallelized using the same idea as the parallel forward phase as described below.

The backward phase recursively identifies the predecessor at stage i starting from stage n. One way to obtain this predecessor is by iteratively looking up the predecessor products `pred[i]` computed during the forward phase. Another way to obtain this is through repeated matrix multiplication as $M_{i \leftarrow n} \star \vec{s}_i$, where $M_{i \leftarrow n}$ is the backward partial product $A_n \odot \ldots A_{i+1}$. Using the same rank convergence argument, the rank of $M_{i \leftarrow n}$ will converge to 1 for large enough number of matrices (small enough i). Lemma 3.3.2 below shows that the predecessor at stages beyond i do not depend on the initial value used for the backward phase.

Lemma 3.3.2. *For a matrix A of rank 1 and any vector $\vec{v}$, all elements of $A \star \vec{v}$ are equal.*

This lemma follows from the fact that the rows in a rank-1 matrix only differ by a constant and arg max is invariant when an offset is added to all its arguments.

The algorithm in Figure 3.4 uses this insight for a parallel backward phase. Every processor starts the predecessor traversal from 0 (line 8) on the stages it owns. Each processor enters a fix up loop whose description and correctness mirror those of the forward phase above.

3.3.7 Rank Convergence and SSSP Discussion

One can view solving an LTDP problem as computing Single Source Shortest/Longest Path in a graph. In this graph, each subproblem is a node and directed edges represent the dependences between subproblems. The weights on edges represent the constants $A_i[j,k]$ in Equation 3.1. In LCS for instance (Figure 3.1), each subproblem has incoming edges with weight 0 from the subproblem above and to its left, and an incoming edge with weight $\delta_{i,j}$ from its diagonal neighbor. Finding the optimal solution to the LTDP problem amounts to

finding the longest path in this graph from the subproblem 0 in the last stage to subproblems in the first stage, given initial weights to the latter. Alternately, one can negate all the weights and change the max to a min in Equation 3.1 to view this as computing shortest paths.

Entries in the partial product $M_{l \to r}$ represent the cost of the shortest (or longest) path from a node in stage l to a node in stage r. The rank of this product is 1 if these shortest paths go through a single node in some stage between l and r. Road networks have this property. For instance, the fastest path from any city in Washington state to any city in Massachusetts is highly likely to go through Interstate I-90 that connects the two states. Routes that use I-90 are overwhelmingly better than those that do not; choices of the cities at the beginning and at the end do not drastically change how intermediate stages are routed. Similarly, if problem instances have optimal solutions that are overwhelmingly better than other solutions, one should expect rank convergence.

Chapter 2 presented a parallel SSSP algorithm for general scale-free networks. In this chapter, the corresponding graph for LTDP problems can have a specific shape where the edges are only between two consecutive stages. Since the edges can have arbitrary dependences, the LTDP problems are amorphous as well. As we discussed in this chapter, we introduced a method that increases the parallelism by compromising from the total amount of work. This is similar to DSMR where D controls the balance between communication and overhead of the computation. Subgraph extraction was another example where communication could be avoided by paying computation overhead.

3.4 Local Linear-Tropical Dynamic Programming

Section 3.2 described LTDP problems and Section 3.3 introduced the parallel Rank-1 method which relies on rank-1 convergence to be efficient. This section introduces *the Delta method*, an approach which relaxes the reliance of a LTDP problem converging to rank-1.

The Delta method works on a subset of LTDP problems which are *Local*. Figure 3.1b shows an example of a Local LTDP (LLTDP) where the dependences for cell $C_{i,j}$ are the three local neighbors: $C_{i-1,j}$, $C_{i,j-1}$ and $C_{i-1,j-1}$. In general, a LLTDP recursive function is given by:

$$C_{i,j} = \max \begin{cases} C_{i-1,j} + w^1_{i,j} \\ C_{i,j-1} + w^2_{i,j} \\ C_{i-1,j-1} + w^3_{i,j} \end{cases} \tag{3.4}$$

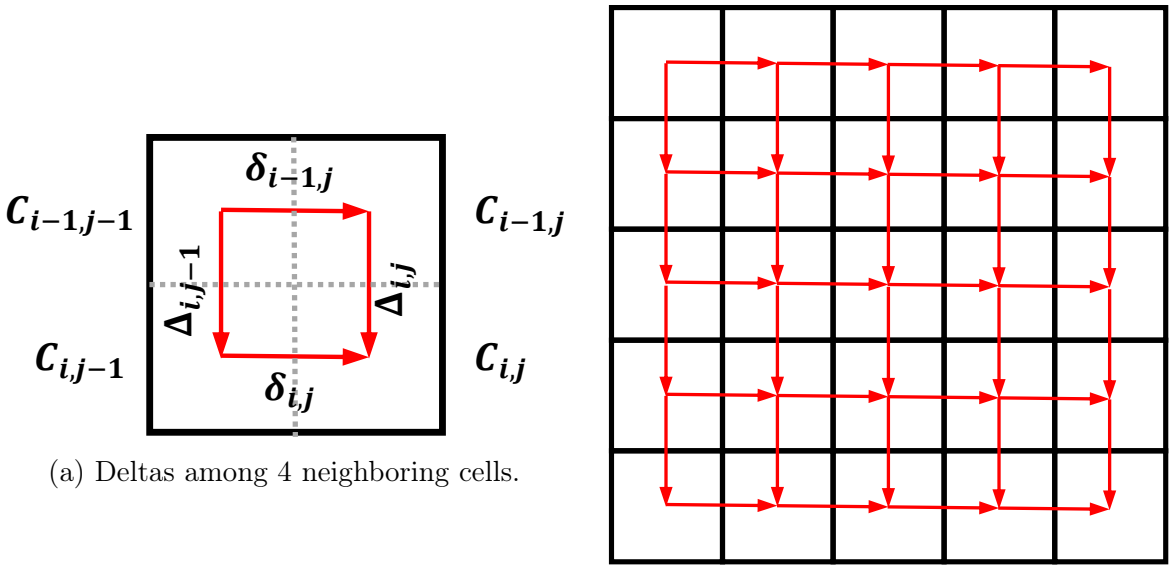
where $w^1_{i,j}$, $w^2_{i,j}$ and $w^3_{i,j}$ are three constant terms that come from the inputs and are independent of the values of the cells. Local LTDP problems are a subset of LTDP problems since Equation 3.4 only restricts Equation 3.1's max terms to only the local neighbors (top, left and top-left cells).

Figure 3.1a shows a LTDP example which is not local and the dependences for a cell come from all the cells of the previous stage. This structure of a general LTDP problem prevents from applying the Delta method which we will discuss in Section 3.4.1.

3.4.1 The Delta Method

The intuition behind the Delta method is that instead of computing a value for each cell in a LLTDP problem, it computes the difference between local cells. For example, Figure 3.6a shows 4 cells: $C_{i-1,j-1}, C_{i-1,j}, C_{i,j-1}, C_{i,j}$. The Delta method computes $\delta_{i,j} = C_{i,j} - C_{i,j-1}$ and $\Delta_{i,j} = C_{i,j} - C_{i-1,j}$ instead of actually computing $C_{i,j}$. Therefore, there are two delta values associated with each cell: the horizontal delta, δ, and the vertical delta, Δ. The whole computation only computes the deltas shown as red arrows in Figure 3.6b. The key contribution of the Delta method is Lemma 3.4.1.

Lemma 3.4.1. *$\delta_{i,j}$, $\Delta_{i,j}$ and $pred_{i,j}$ can be computed given the values of $\delta_{i,j-1}$ and $\Delta_{i-1,j}$ and using recursive function 3.4.*



(a) Deltas among 4 neighboring cells.

(b) All deltas of a table.

Figure 3.6: The deltas among the cells.

Proof. In Figure 3.6a, assume that the value of $C_{i-1,j-1}$ is x and consequently, the values of $C_{i,j-1}$ and $C_{i-1,j}$ will be $x + \delta_{i-1,j}$ and $x + \Delta_{i,j-1}$, respectively. Using the recursive function 3.4, $C_{i,j} = \max\{x + \delta_{i-1,j} + w^1_{i,j}, x + \Delta_{i,j-1} + w^2_{i,j}, x + w^3_{i,j}\}$. Let $K = \max\{\delta_{i-1,j} + w^1_{i,j}, \Delta_{i,j-1} + w^2_{i,j}, w^3_{i,j}\}$ and as a result, $C_{i,j} = x + K$. K is a constant since all of the terms are given. Therefore, $\delta_{i,j} = C_{i,j} - C_{i,j-1} = x + K - (x + \Delta_{i,j-1}) = K - \Delta_{i,j-1}$ which is a constant. $\Delta_{i,j}$ can be similarly proved to be constant. The maximum term in the computation of K is independent of x and so is the maximum term in the computation of $C_{i,j}$ although the value of $C_{i,j}$ itself is not. Therefore, $pred_{i,j}$ is constant given the values of $\delta_{i,j-1}$ and $\Delta_{i-1,j}$. □

Figure 3.6b shows all deltas as red arrows in the table. Given all the initial deltas on the left and top side of the table, all other deltas can be computed using Lemma 3.4.1. Let $f_\delta(i, j, \delta_{i-1,j}, \Delta_{i,j-1})$ and $f_\Delta(i, j, \delta_{i-1,j}, \Delta_{i,j-1})$ be the functions that computes $\delta_{i,j}$ and $\Delta_{i,j}$ according to Lemma 3.4.1, respectively. Figure 3.7 shows the sequential algorithm for the Delta method.

```
1  DeltaSeq(delta δ0,1... δ0,n-1, Delta Δ1,0... Δn-1,0) {
2    for i in (1..n-1) {
3      for j in (1..n-1) {
4        δi,j = fδ(i,j,δi-1,j,Δi,j-1);
5        Δi,j = fΔ(i,j,δi-1,j,Δi,j-1); }}}
```

Figure 3.7: Sequential Delta method for planar dynamic programming.

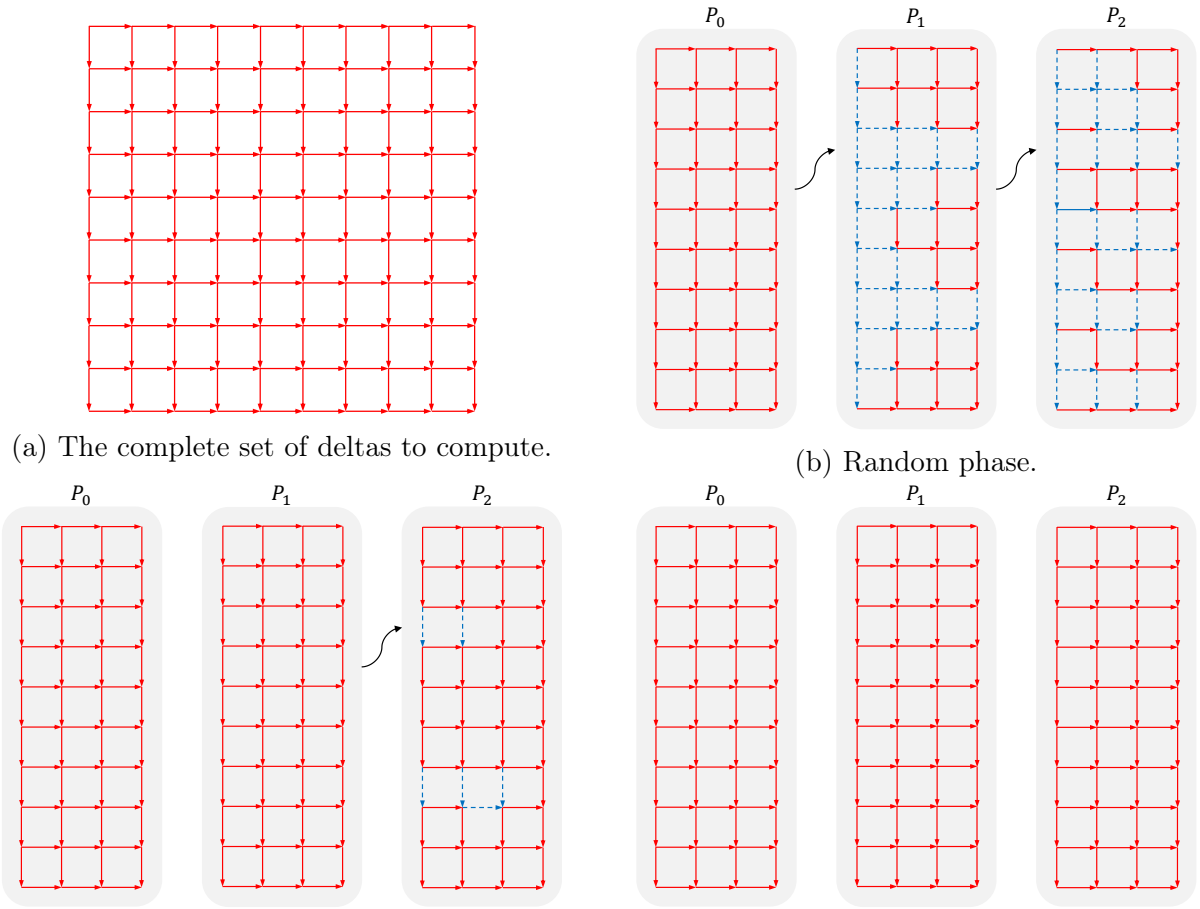
The inputs given to the sequential Delta method in Figure 3.7 are $\delta_{0,1}, \ldots, \delta_{0,n-1}, \Delta_{1,0}, \ldots, \Delta_{n-1,0}$ which are the initial deltas for the top and left side of the table. For the sake of simplicity, we assumed that the table is squared shaped of size $n \times n$. A lot of details are hidden behind the f_δ and f_Δ functions which were described in details in Lemma 3.4.1. Next in Section 3.4.2, we will describe the parallel version of the Delta method.

3.4.2 The Parallel Delta Method

This section describes a parallel version of the Delta method which is depicted in Figure 3.8. Figure 3.8a shows all deltas that are desired for computation. Assume that there are three processors available for the parallel computation: P_0, P_1 and P_2. The set of deltas are distributed column-wise with one column of Δs replicated at the boundaries as shown in Figure 3.8b. The parallel Delta method consists of two phases: i) a random phase, ii) 2 fix-up phases. In the random phase, each processor except P_0 assumes random values for the first column of Δs as shown by dotted blue arrows in Figure 3.8b (Section 3.3.5 details how to pick random initial vectors). The key contribution of parallel Delta method is the following observation.

Observation 3.4.2. *Starting with random (and perhaps incorrect) initial Δs in the first column of a processors leads to many correct deltas on subsequent columns.*

For cell $\{i, j\}$, define *input deltas* to be $\delta_{i-1,j}$ and $\Delta_{i,j-1}$ and *the output deltas* to be $\delta_{i,j}$ and $\Delta_{i,j}$. This naming comes from Lemma 3.4.1 where by taking two deltas, two other deltas



(a) The complete set of deltas to compute.

(b) Random phase.

(c) Fix phase 1.

(d) Fix phase 2.

Figure 3.8: The parallel deltas method.

can be computed. As it can be seen in Figure 3.8b, starting with all incorrect random input deltas in the first column of P_1 and P_2 (shown as blue dotted arrows) leads to many correct output deltas (shows as red solid arrows) as the processors proceed. The deltas that are farther from the random initialization are more likely to be correct.

Denote predecessors and delta values that are computed using the random initialization by $pred'_{i,j}$, $\delta'_{i,j}$, $\Delta'_{i,j}$ and $C'_{i,j}$. The intuition behind Observation 3.4.2 is that $pred_{i,j} = pred'_{i,j}$ for many cells. Assume that in Figure 3.6a both $\delta'_{i-1,j} \neq \delta_{i-1,j}$ and $\Delta'_{i,j-1} \neq \Delta_{i,j-1}$ but $pred_{i,j} = pred'_{i,j} = \{i-1, j\}$. Therefore, according to the formula 3.4, $C'_{i,j} = C'_{i-1,j} + w^1_{i,j}$ and similarly $C_{i,j} = C_{i-1,j} + w^1_{i,j}$ which means that $\Delta'_{i,j} = w^1_{i,j} = \Delta_{i,j}$. Hence, using incorrect input deltas may lead to correct output deltas. Also, correct deltas lead to more correct deltas because of Lemma 3.4.1.

The other observation that comes directly from Lemma 3.4.1 is that a blue dotted incorrect delta in Figure 3.8b has to come from another incorrect input delta. In other words, if there are two correct input deltas the output deltas are also correct. Therefore, we can claim the following lemma.

Lemma 3.4.3. *If there is a path of solid red correct deltas from the top row of the table to the bottom of the table in a processor, the rest of deltas after that path are solid red correct deltas.*

Proof. The top row deltas are always correct since they are a part of initialization. Therefore, because of Lemma 3.4.1, all deltas that come immediately after the path are all correct. We can apply the same reasoning to conclude that the rest of deltas after the path are all correct. □

Using Observation 3.4.2 and Lemma 3.4.3, rest of parallel Delta method can be described. After each processor finishes its computation using a random initialization, P_0 sends to P_1 and P_1 sends to P_2 the column of Δs they share as shown by black arrows in Figure 3.8b.

This concludes the first phase of the parallel Delta method which was called the random phase.

Next, the first fix-up phase starts which is depicted in Figure 3.8c. In this phase, P_0 is idle while P_1 and P_2 fix up the incorrect deltas. Since each incorrect delta only affects its output deltas, each processors needs only recompute $\delta_{i,j}$ and $\Delta_{i,j}$ where $\delta_{i-1,j} \neq \delta'_{i-1,j}$ or $\Delta_{i,j-1} \neq \Delta_{i,j-1}$. Therefore, P_1 and P_2 perform less work in the first fix-up phase than the random phase. At the end of first fix-up phase, only P_1 sends its correct deltas to P_2. In the second fix-up phase, P_0 and P_1 are both idle while P_2 fixes up the incorrect deltas as shows in Figure 3.8d. P_2 performs less work than the first fix-up phase because there are less incorrect deltas after the first fix-up phase.

Figure 3.9 shows the parallel Delta method for local LTDP problems. First, we describe the initialization. The input arguments are given on Line 1. Assume that there are P processors available to the system. Then, each processors $p \in \{0 \dots P-1\}$ on Line 4 computes the first column and the last column of the range of deltas that it is assigned which are specified by S_p and E_p, respectively. Next, each processor p, except the first, initialize its S_p of Δs in the loop at Line 6. In the loop, on Line 10, we store the random values in $\Delta'_{p,j}$ that are going to be used later. Δ' is one array of Δs for the first column of each processor p. The barrier on Line 11 makes sure that every processor is done before continuing the algorithm.

Next, in the random phase, each processor computes the deltas it owns in the loop on Line 14. This loop is similar to the one from the sequential Delta method in Figure 3.7 except that it is parallel. Then, each processor p updates the Δs for the first column of processor $p+1$. This is done by having the barrier on Line 19 and since $S_p = E_{p+1}$, processor $p+1$ will have the correct Δs after the barrier.

After the random phase, there is the fix-up phase and the main loop is on Line 21. First processor does not need any fix-up phase since all of its deltas are correct. p_1 needs one

```
1  ParDeltaSeq(delta δ0,1... δ0,n-1, Delta Δ1,0... Δn-1,0) {
2    //Initialization
3    //The start (Sp) and end (Ep) column of deltas for processor p
4    ∀p: Sp=n/P*p; Ep=n/P*(p+1);
5      //Initializing the first column of deltas
6      parallel for p in (1..P-1) {
7        local s = Sp;
8        for j in (1..n-1) {
9          Δs,j = random();
10         Δ'p,j = Δs,j; }}
11     --- barrier ---
12     //Random phase
13     //Parallel loop to compute deltas in the random phase
14     parallel for p in (0..P-1) {
15       for i in (Sp+1..Ep) {
16         for j in (1..n-1) {
17           δi,j = fδ(i,j,δi-1,j,Δi,j-1);
18           Δi,j = fΔ(i,j,δi-1,j,Δi,j-1); }}}
19     --- barrier ---
20     //Fix-up phase
21     for iter in (1..P-1) {
22       parallel for p in (iter..P-1) {
23         local s = Sp;
24         //The worklists for the current and next iterations to fix
25         local workQueue1, workQueue2;
26         for j in (1..n-1) { // Finding wrong Δs in the first column
27           if Δs,j != Δ'p,j {
28             workQueue1.add(j);
29             Δ'p,j = Δs,j; }}
30         //Iterating through column that processor p owns
31         for i in (Sp+1..Ep) {
32           //Only recompute the ones from the working list
33           while workQueue1.size() {
34             //If output Δ is different,
35             //add it to workQueue2 for the next iteration, i+1
36             local veticalDelta = fΔ(i,j,δi-1,j,Δi,j-1);
37             if Δi,j != verticalDelta {
38               Δi,j = verticalDelta;
39               workQueue2.add(j);  }
40             //If output δ is different,
41             //add it to workQueue1 for the current iteration, i
42             local horizontalDelta = fδ(i,j,δi-1,j,Δi,j-1);
43             if δi,j != horizontalDelta {
44               δi,j = horizontalDelta;
45               if j < n-1 {
46                 workQueue1.add(j+1);  }}}
47           swap(workQueue1,workQueue2);  }}
48       --- barrier ---  }}
```

Figure 3.9: Parallel Delta method for local dynamic programming.

phase of fix-up since it receives the completely correct deltas after the random phase. p_2 needs at most two phases of fix-up since it eventually receives all correct deltas after the first fix-up phase and in general, processor p requires p number of fix-up phases which is specified by the parallel loop on Line 22. The main fix-up loop at Line 21 iterates for $P-1$ times since the last processor has $P-1$ fix-up phases.

For each processor, there are two work lists (Line 22) that keep track of the indices of incorrect delta inputs whose output deltas need to be recomputed: `workQueue1` and `workQueue2` on Line 25. They are used for different iterations of the loop on Line 31 that goes through all columns of processor p. `workQueue1` is the work list for the current iteration of the loop (i) and `workQueue2` is the work list the for the next iteration of the loop ($i+1$). Before the loop on Line 31, the loop on Line 26 initiates `workQueue1` by comparing (Line 27) the newly received Δs and the Δs used previously which were saved in Δ's on Line 10. The ones that do not match are added in `workQueue1` for re-computation of output deltas (Line 28). Also, Δ' stores the new Δs for the next fix-up phase (Line 29).

The loop on Line 31 goes through all columns of p and the while loop on Line 33 goes through the `workQueue1` and recomputes the necessary deltas. If a Δ for the next column is different from the previous iteration, it is added in `workQueue2` for the next iteration of the loop at Line 31 (Line 37). On the other hand, if a δ is different, it affect the current iteration and the necessary index in added to `workQueue1` (Line 43). At the end, the two work lists are swapped in Line 47 for the next iteration of the loop on Line 26. The barrier in Line 48 is the end of a fix-up iteration.

3.4.3 Algorithmic Comparison between The Parallel Delta Method and The Rank-1 Method

The parallel Rank-1 method in Section 3.3 can be explained using deltas for local LTDP. The parallel Rank-1 method only relies on Lemma 3.4.3 and the fix-up phase is more coarse grain than the parallel Delta method. In the Rank-1 method, every processor, similarly, starts with a random initialization but in the fix-up phase the deltas for every column is recomputed until a column with all correct Δs is reached. Because of Lemma 3.4.3, after such a column, rest of the deltas are correct and there is no need for re-computation.

The new parallel Delta method approach recomputes only a subset of the deltas in a column are recomputed where that subset is identified using Lemma 3.4.1. Therefore, the parallel Delta method is an improvement to the Rank-1 method and the expectation is that it is never slower than the parallel Rank-1 method for LLTDP. Specially, in the cases where it takes a lot of columns to converge to a column with all correct Δs, the parallel delta method should be significantly faster. Section 3.6 studies this comparison in practice.

3.5 LTDP Examples

This sections shows four important optimization problems as LTDP — Viterbi, Longest Common Subsequence, Smith-Waterman, and Needleman-Wunsch. Our goal in choosing these particular problems is to provide an intuition on how problems with different structure can be viewed as LTDP. Other problems are LTDP, but not evaluated in this chapter, include dynamic time warping and seam carving.

Viterbi The Viterbi algorithm [78] finds the most likely sequence of states in a (discrete) hidden Markov model (HMM) for a given sequence of n observations. Its recurrence equation is shown in Figure 3.1(a). Here, $p_{i,j}$ represents the probability of the most likely state

sequence ending in state j of the HMM that explains the first i observations. The meaning of the term $t_{k,j}$ is not important here (see [78]). The solution to a Viterbi instance is given by the maximum value of $p_{n,j}$ as we are interested in the most likely sequence ending in *any* HMM state.

The subproblems along a column in Figure 3.1(a) form a stage and they only depend on the subproblems in the previous column. This dependence is not directly in the desired form of Equation 3.1. But applying logarithm on both sides to the recurrence equation brings it to this form. By transforming the Viterbi instance into one that calculates log-probabilities instead of probabilities, we obtain a LTDP instance.

Invoking the LTDP parallel algorithm in Figure 3.4 requires one additional transformation. The algorithm assumes that the solution to LTDP is given by the first subproblem in the last stage n. To account for this, we introduce an additional stage $n + 1$ in which every subproblem is the maximum of all subproblems in stage n. Essentially, stage $n + 1$ is obtained from multiplying a matrix with 0 in all entries with stage n.

The Viterbi algorithm is not a local LTDP since the dependence edges are not local and they cross. Therefore, the parallel Delta method is not applicable.

Longest Common Subsequence LCS finds the longest common subsequence of two input strings A and B [36]. The recurrence equation of LCS is shown in Figure 3.1 (b). Here, $C_{i,j}$ is the length of the longest common subsequence of the first i characters of A and the first j characters of B. Also, $\delta_{i,j}$ is 1 if the ith character of A is the same as the jth character of B and 0 otherwise. The LCS of A and B is obtained by following the predecessors from the bottom-rightmost entry in the table in Figure 3.1(b).

Some applications of LCS, such as the `diff` utility tool, are only interested in solutions that are at most a width w away from main diagonal - ensuring that the LCS is still reasonably similar to the input strings. For these applications, the recurrence relation can be

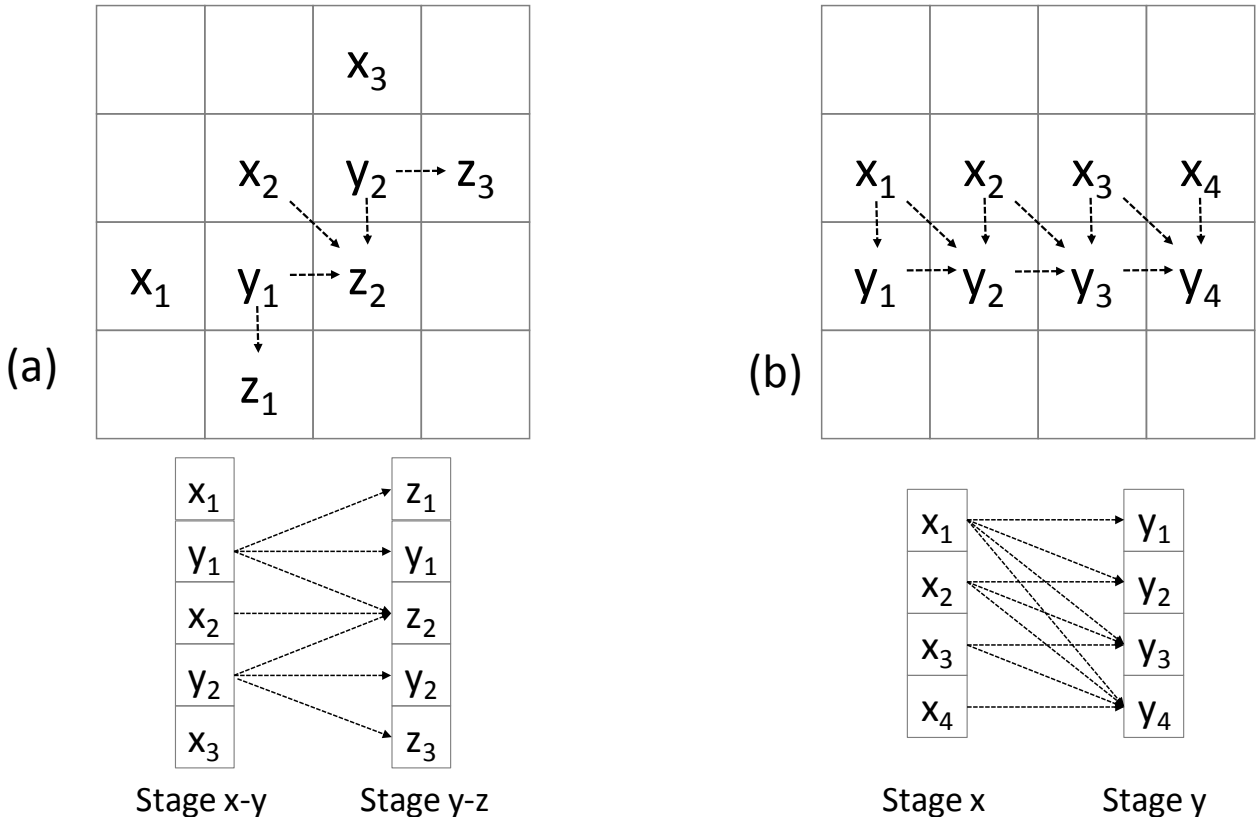


Figure 3.10: Two ways of grouping the subproblems in LCS into stages such that each stage only depends on one previous stage.

modified such that $C_{i,j}$ is set to $-\infty$ whenever $|i-j| > w$. Using a smaller width also reduces the memory requirements of LTDP as the entire table need not be stored in memory. Smaller width limits the scope of wavefront parallelism due to smaller sizes of stages, which emphasizes the need for parallelizing across stages as proposed by this chapter.

LCS is a local LTDP problem as it can be checked from Figure 3.10. Therefore, the parallel Delta method from Section 3.4.2 can be applied. Although using this method instead of the Rank-1 method requires computing two values per cell, one can show that deltas in LCS problem are either 1 or 0. This allows compactly representing the deltas as a sequence of bits [37]).

On the other hand, LCS is also a LTDP problem and the Rank-1 method requires defining

stages such that a stage is only dependent on the previous stage. Grouping the subproblems of LCS into stages can be done in two ways, as shown in Figure 3.10. In the first approach, the stages correspond to anti-diagonals, such as the stage consisting of $\texttt{z}_\texttt{i}$s in Figure 3.10 (a). This stage depends on two previous stages (on $\texttt{x}_\texttt{i}$s and $\texttt{y}_\texttt{i}$s) and does not strictly follow the rules of LTDP. One way to get around this is to define stages as overlapping pairs of anti-diagonals, like stages `x-y` and stage `y-z` in Figure 3.10 (a). Subproblems $\texttt{y}_\texttt{i}$s are replicated in both stages, allowing stage `y-z` to depend only on stage `x-y`.

In the second approach, the stages correspond to the rows (or columns) as shown in Figure 3.10 (b). The recurrence needs to be unrolled to avoid dependences between subproblems within a stage. For instance, $\texttt{y}_\texttt{i}$ depends on all $\texttt{x}_\texttt{j}$ for $\texttt{j} \leq \texttt{i}$. In this approach, since the final solution is obtained from the last entry, the predecessor traversal in Figure 3.2 has to be modified to start from this entry, say by adding an additional matrix at the end to move this solution to the first solution in the added stage.

Needleman-Wunsch This algorithm [65] finds a *global* alignment of two input sequences, commonly used to align protein or DNA sequences. The recurrence equation is very similar to the one in LCS (Section 3.5).

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + m[i,j] \\ s_{i-1,j} - d \\ s_{i,j-1} - d \end{cases}$$

In this equation, $s_{i,j}$ is the score of the best alignment for the prefix of length i of the first input and the prefix of length j of the second input, $m[i,j]$ is the matching score for aligning the last characters of the respective prefixes, and d is the penalty for an insertion or deletion during alignment. The base cases are defined as $s_{i,0} = -i * d$ and $s_{0,j} = -j * d$.

The Needleman-Wunsch algorithm is also a local LTDP problem as it can be checked

from the formula. Therefore, as the LCS case, the parallel Delta method is applicable to this algorithm. Similarly, one can show that deltas are in the range of $[-d \ldots (\max_{i,j} m[i,j] + d)]$ allowing compact representation.

For applying the Rank-1 method, grouping subproblems into stages can done using the same approach as in LCS. Abstractly, one can think of LCS as an instance of Needleman-Wunsch for appropriate values of matching scores and insert/delete penalties. However, the implementation details differ sufficiently enough for us to consider them as two different algorithms.

Smith-Waterman This algorithm [73] performs a *local* sequence alignment, in contrast to Needleman-Wunsch. Given two input strings, Smith-Waterman finds the *substrings* of the input that have the best alignment, where longer substrings have a better alignment. In its simplest form, the recurrence equation is of the form

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j-1} + m[i,j] \\ s_{i-1,j} - d \\ s_{i,j-1} - d \end{cases}$$

Key difference from Needleman-Wunsch is the 0 term in max which ensures that alignments "restart" whenever the score goes to zero. Because of this term, the constants in A_i matrices in equation 3.1 need to be set accordingly. This slight change has significant difference to the convergence properties of Smith-Waterman as we will see later in Section 3.6.1. Our implementation uses a more complex recurrence equation that allows for affine gap penalties when aligning sequences [26].

Also, the solution to Smith-Waterman requires finding the maximum of all subproblems in all stages and performing a predecessor traversal from that subproblem. To account for

this in our LTDP formulation, we add one "running maximum" subproblem per stage that contains the maximum of all subproblems in the current stage and previous stages.

The Smith-Waterman algorithm is not a local LTDP and therefore, the parallel Delta method is not applicable.

3.6 Evaluation

This section evaluates the Rank-1 and Delta methods on four LTDP problems discussed in Section 3.5. Section 3.6.1 empirically evaluates the occurrence of rank convergence in practice. Section 3.6.3 evaluates scalability, speed up and efficiency of our implementation. Finally, Section 3.6.5 compares the parallel algorithm with wavefront parallelization.

3.6.1 LTDP Rank Convergence

Determining whether the parallel Rank-1 and Delta methods benefit a dynamic programming problem requires: 1) the problem to be LTDP (discussed in Section 3.2); 2) rank convergence happens in reasonable number of steps. This section demonstrates how rank convergence can be measured and evaluates it for the 4 LTDP problems discussed in Section 3.5.

Rank Convergence is an empirical property of a sequence of matrix multiplications that depends on both the LTDP recurrence relation in addition to the input. Table 3.1 empirically evaluates the number of steps required for rank convergence across different algorithms and inputs. For a LTDP instance, defined by the algorithm (Column 1) and input (Column 2), we first compute the actual solution vectors at each stage. Then, starting from a random all-non-zero vector at 200 different stages, we measured the number of steps required to generate a vector parallel to the actual solution vector (i.e., convergence). Columns 3,4, and 5 respectively show the minimum, median, and maximum number of steps needed for convergence. For each input, Column 2 specifies the computation width (the size of a stage

Steps to Converge to Rank-1		Min	Median	Max
Viterbi Decoder	Voyager: 2^6	22	40	104
	LTE: 2^6	18	30	62
	CDMA: 2^8	22	38	72
	MARS: 2^{14}	46	112	414
Smith-Waterman	Query-1: 603	2	6	24
	Query-2: 884	4	8	24
	Query-3: 1227	4	8	24
	Query-4: 1576	4	8	24
Needleman-Wunsch	Width: 1024	1,580	19,483	192,747
	Width: 2048	3,045	44,891	378,363
	Width: 4096	5,586	101,085	404,4374
	Width: 8192	12,005	267,391	802,991
LCS	Width: 8192	9,142	79,530	370,927
	Width: 16384	19,718	270,320	–
	Width: 32768	42,597	626,688	–
	Width: 65536	86,393	–	–

Table 3.1: Number of steps to converge to rank 1.

or the size of each A_i matrices). Each algorithm has a specific definition of width: for Viterbi decoder, width is the number of states for each decoder, in Smith-Waterman, it is the size of each query, and in LCS and Needleman-Wunsch, it is a fixed-width around the diagonal of each stage. LCS never converged so we leave those entries blank. The rate of convergence is specific to the algorithm and input (i.e., Smith-Waterman converges fast while LCS sometimes does not converge) and, generally speaking, wider widths require more steps to converge. We will use this table later in Section 3.6.3 to explain scalability of out approach.

3.6.2 Environmental Setup

We conducted experiments on a shared memory machine and on a distributed memory machine. A shared memory machine favors fast communication and is ideal for wavefront approach. Likewise, a distributed machine has a larger number of processors and so we can better understand how our parallel algorithm scales. Next, we describe these two machines.

Distributed-Memory Machine: Stampede [74], a Dell PowerEdge C8220 Cluster with 6,400 nodes. At the time of writing this thesis, Stampede is ranked 8^{th} on the Top500 [75]

list. Each node contains 2 8-core Intel Xeon E5-2600 processor @ 2.70 GHz (16 cores in total) and 32 GB of memory. The interconnect topology is a fat-tree, FDR InfiniBand interconnect. On this cluster, we used the MPI MVAPICH 2 library [63] and the Intel C/C++ compiler version 13.0.1 [38].

Shared-Memory Machine: an unloaded Intel 2.27GHz Xeon (E7) workstation with 40 cores (80 threads with hyper-threading) and 128GB RAM. We use the Intel C/C++ compiler (version 13.0.1) [38] and the Intel MPI library (version 4.1) [59].

We report scalability results on Stampede but results from the shared-memory machine are qualitatively similar. We used the shared-memory machine to compare our parallel algorithm with wavefront parallelization. Unless specified otherwise, the reported results are from Stampede runs.

We use MPI/OMP timer to measure process runtime. We do not measure setup costs — only the time it takes to execute one invocation of a LTDP problem. When we compare against a baseline, we modify that code to take the same measurements.

Finally, to get statistically significant results, we run each experiment multiple times and report the mean and 95% confidence interval of the mean when appropriate. We do not include confidence intervals in the graphs if they are small.

3.6.3 LTDP Benchmarks and Performance

This section evaluates the parallel algorithm on the four LTDP problems discussed in Section 3.5. Table 3.2 overviews what algorithms are used for baseline and parallelization of each LTDP problem. We will discuss in details about each of them in this Section.

To substantiate our scalability results, we evaluate each benchmark across a wide variety of real-world inputs. We break the results down by the LTDP problem.

Problem	Type of LTDP	Baseline	Parallel
Viterbi	Non-Local LTDP	Spiral	The Parallel LTDP
Smith-Waterman	Non-Local LTDP	Farrar	The Parallel LTDP
LCS	Local LTDP	Bit-Parallelism	The Parallel Delta Method
Needleman-Wunsch	Local LTDP	The Delta Method	The Parallel Delta Method

Table 3.2: Algorithms used for each LTDP problem.

Viterbi Decoder

Viterbi decoder uses the Viterbi algorithm (Section 3.5) to communicate over noisy and unreliable channels, such as cell phone communications [78]. Given a potentially corrupted convolution-encoded message [67], Viterbi decoding finds the most likely decoded message.

Baseline We used Spiral's [71] Viterbi decoder: a highly optimized (via auto-tuning) decoder that utilizes SIMD to parallelize decoding within a stage. To the best of our knowledge, there is no efficient multi-processor algorithm for Viterbi decoders since the amount of parallelism in each stage is limited.

Our Implementation Spiral code is heavily optimized and even small changes negatively affect performance. Therefore, the performance-critical internal loop of the Spiral code is used as a black box. Each processor starts from an arbitrary all-non-zero vector (except the first, which uses the initial vector) and uses Spiral to execute its set of stages. Each processor (except the last) then communicates its result to the next processor.

Data We use four real-world convolution codes; Voyager, the convolution codes used on NASA's deep space Voyager. Mars, the convolution codes used to communicate with NASA's mars rovers, and both CDMA and LTE, two convolution codes commonly used in modern cell-phone networks. For each of these 4 convolution codes, we investigate the impact of 4 network packet sizes (2048, 4096, 8192, and 16384), which determine the number of stages in the computation. For each size, we used Spiral's input generator to create 50 network

packets.

Performance and Scalability Figure 3.11 and 3.12 shows the performance, the speed up, and the efficiency of each 4 decoders for *the parallel Rank-1 method*. To evaluate the impact of different decoder sizes, each plot has four lines (one per network packet size). A point (x, y) in a performance/speed up plot with the primary y-axis on left, gives the throughput y (the number of bits processed in a second) in megabits per second (Mb/S) as a function of the number of processors x used to perform the Viterbi Decoding. The same point with the secondary y-axis on right shows the speed up y with x number of processors over the sequential performance. Note that Spiral sequential performance at $x = 1$ is almost the same for different packet sizes. The filled data points in the plots show that convergence occurred in the first iteration of the fix-up loop in Figure 3.4 algorithm (i.e. each processor's stage is large enough for convergence). The non-filled data points show multiple iterations of the fix-up loop were required. Similarly, a point in an efficiency plot provides the speed up of our parallel implementation over the sequential performance of Spiral generated code divided by the number of processors. Each point is the mean of 50 random packets.

Figure 3.11 demonstrates (i) our approach provides significant speed ups over the sequential baseline and (ii) different convolution codes and network packet sizes have different performance characteristics. For example, with 64 processors, our CDMA Viterbi Decoder processing packets of size 16384 decodes at a rate of 434 Mb/S which is 24× faster than the sequential algorithm. Note that for the same network packet size and number of processors, our MARS decoder only processes at 4.4 Mb/S because the amount of computation per bit (size of each stage) is significantly greater than CDMA.

The performance of our approach — and thus our speed up numbers — depend on the rate of rank convergence for each pair of convolution codes and network size as shown in Table 3.1. Larger network packet size provide better performance across all convolution

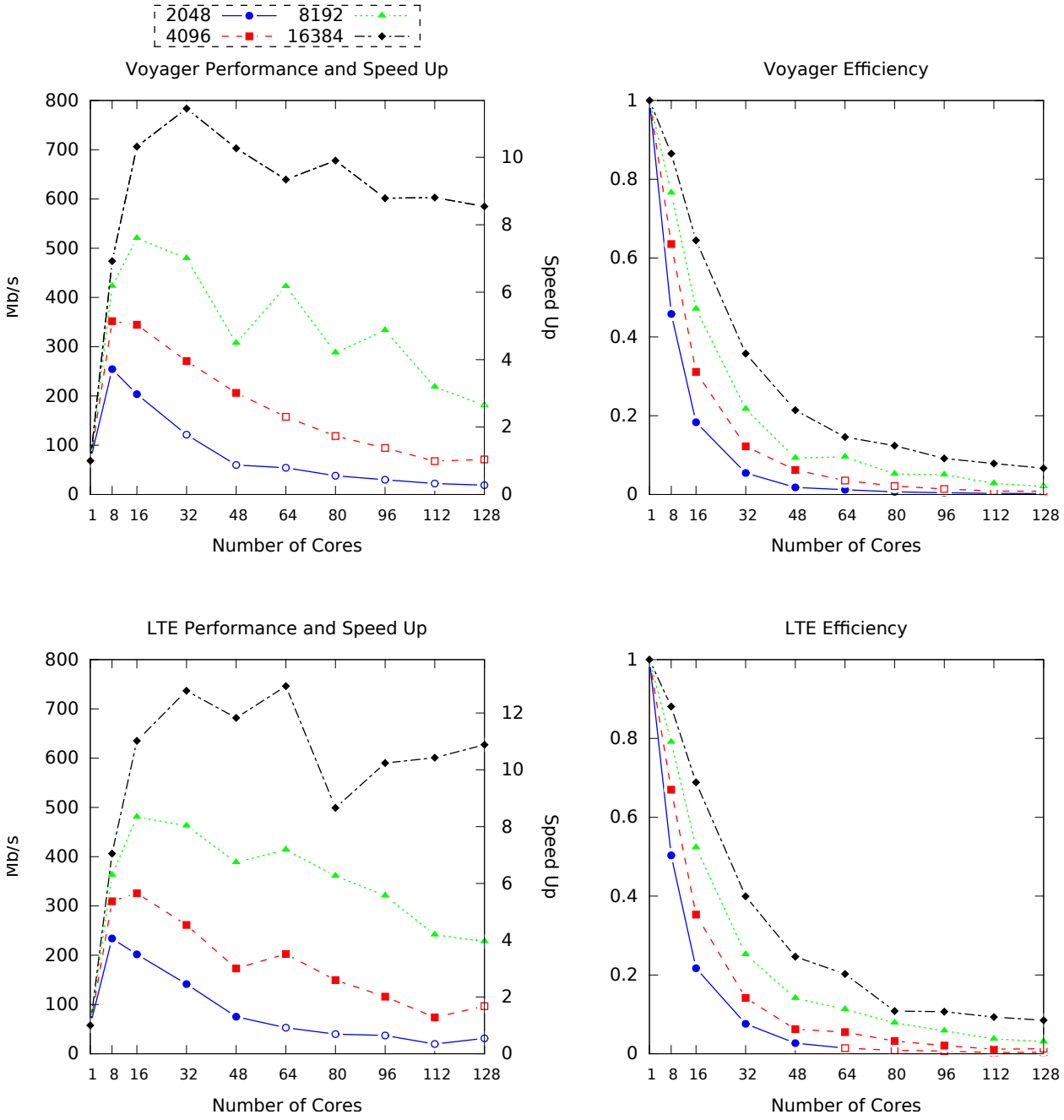


Figure 3.11: Performance (Mb/S), speed up and efficiency of Voyager and LTE Viterbi decoders with the parallel Rank-1 method. The non-filled data points demonstrates where processors have too few iterations to converge to rank 1

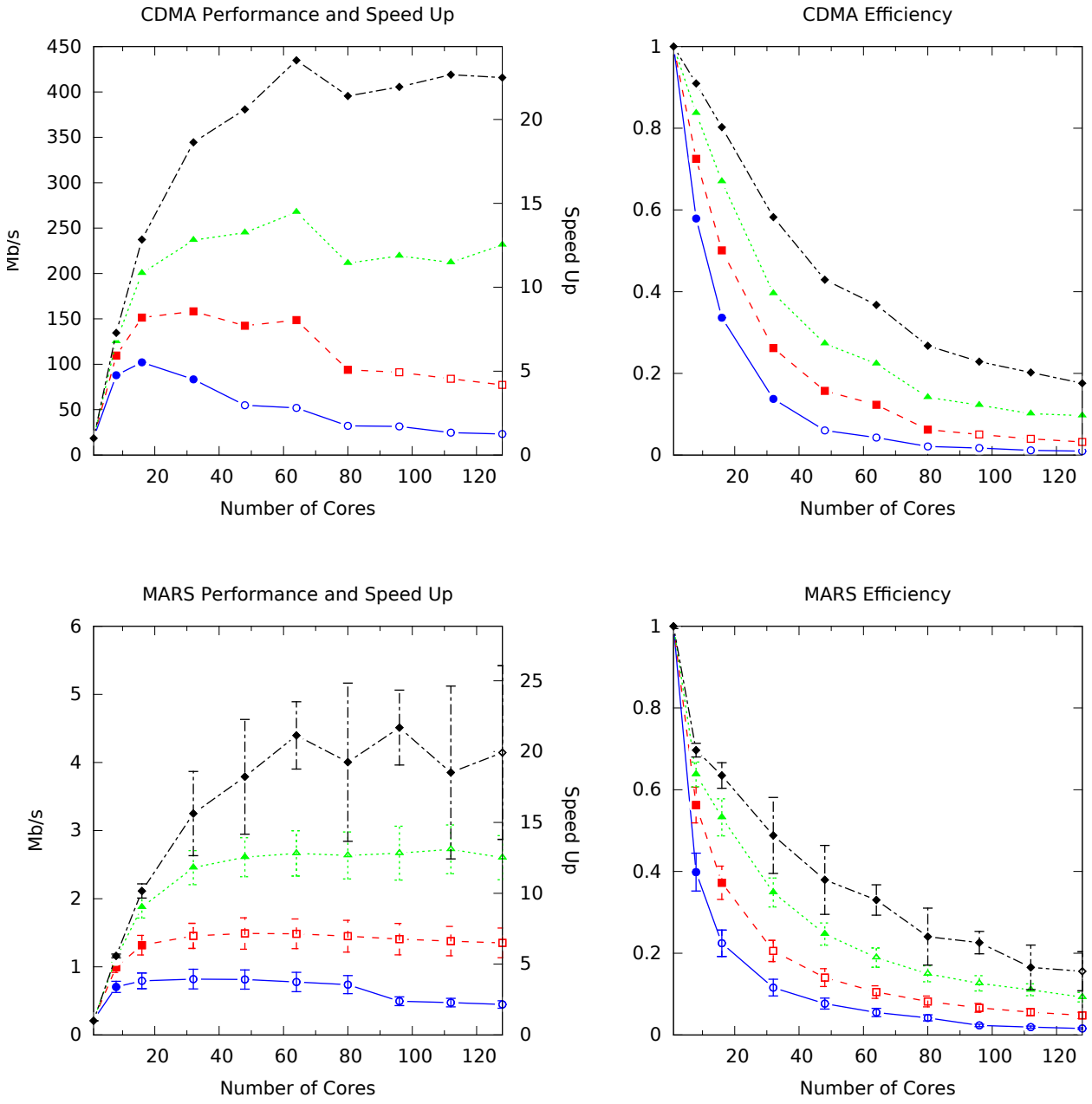


Figure 3.12: Performance (Mb/S), speed up and efficiency of CDMA and MARS Viterbi decoders with the parallel Rank-1 method. The non-filled data points demonstrates where processors have too few iterations to converge to rank 1

codes (i.e., a network packet size of 16384 is *always* the fastest implementation, regardless of convolution code) because the amount of re-computation (i.e., the part of the computation that has not converged), as a proportion of the overall computation decreases with larger network packet size.

Also, as it can be seen in Figure 3.11, efficiency plots drop as the packet sizes decrease and this is again because the ratio of the amount of re-computation to the whole computation decreases. Note that with 48 processors, our algorithm for CDMA can reach efficiency of more than 0.4.

Smith-Waterman

As described in Section 3.5, Smith-Waterman is an algorithm for local sequence alignment [73] often used to align DNA/protein sequences.

Baseline We implemented the fastest known CPU version, Farrar's algorithm, which utilizes SIMD to parallelize within a stage [26].

Our Implementation Our parallel implementation of Smith-Waterman uses Farrar's algorithm as a black-box.

Data We aligned chromosomes 1, 2, 3 and 4 from the human reference genome hg19 as databases and four randomly selected expressed sequence tags as queries. All the inputs are publicly available to download from [64]. We report the average of performance across all combinations of DNA and query (16 in total).

Performance and Scalability As before, there are the results for *the parallel Rank-1 method*. A point (x, y) in the performance/speed up plot in Figure 3.13 with the primary y-axis on left, gives the performance y in Giga cell updates per second, or (GigaCUPS) as

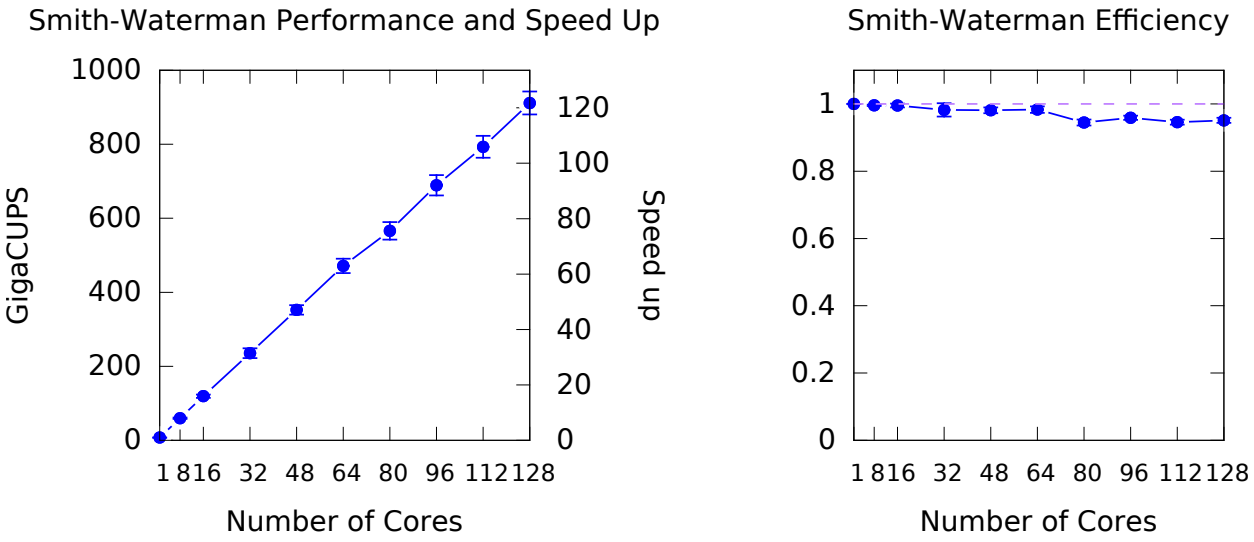


Figure 3.13: Smith-Waterman performance, speed up and efficiency with the parallel Rank-1 method.

a function of the number of processors used to perform the Smith-Waterman alignment. GigaCUPS is a standard metric used in bioinformatics to measure the performance of DNA based sequence alignment problems and refers to the number of cells (in a dynamic programming table) updated per second. Similar to the Viterbi decoder plots, the secondary y-axis on the left show the speed up for each number of processors. We run Smith-Waterman on all combinations of 4 DNA databases and 4 DNA queries (we run each combination 5 times). Unlike the prior Viterbi results, we do not see large variability in performance as a function of the problem data. In other words, the DNA database and query pairs do not significantly impact our performance numbers. This can also be confirmed from Table 3.1 where the number of steps to converge to rank 1 is significantly smaller than a DNA database size which is more than 100 million long. Thus, we plot the average, across all combinations of DNA databases and queries.

The performance gain of our approach for this algorithm is significant: the efficiency plot in Figure 3.13 demonstrates that our approach has efficiency ~ 1 for any number of processors which means almost linear speed up with up-to 128 processors. This can be also

confirmed from the performance/speed up plot. Our algorithm would scale more with more number of processors but we only report up-to 128 processors to keep Figure 3.13 consistent with the others.

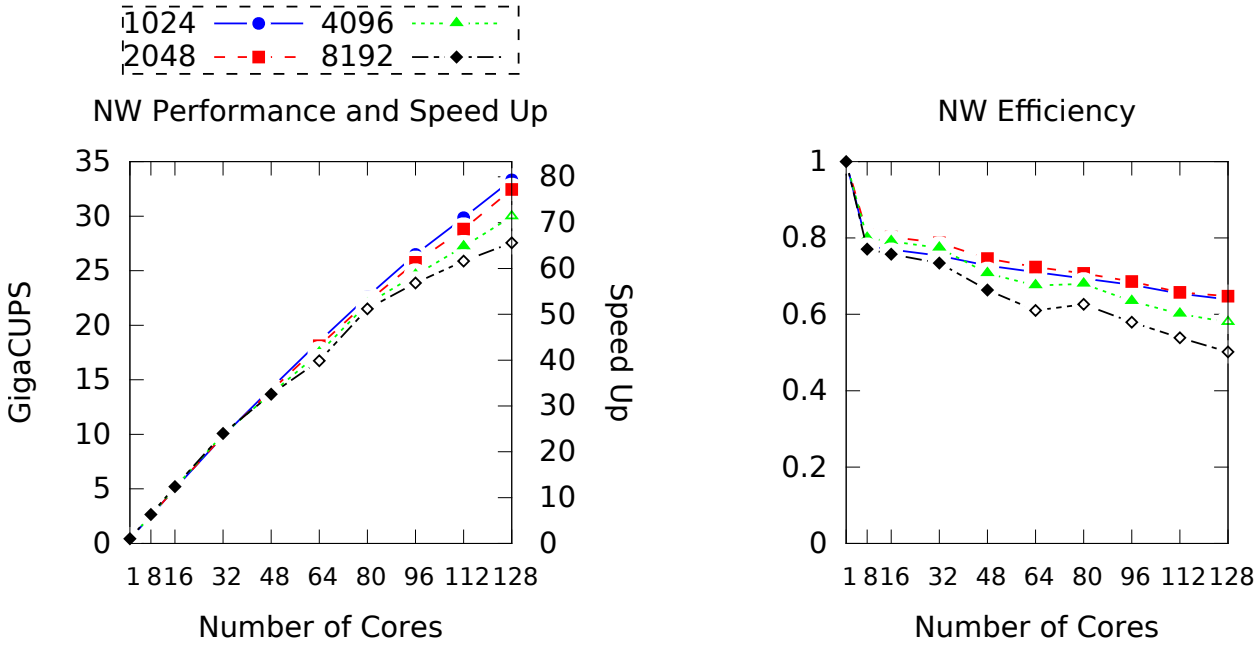
Needleman-Wunsch

In contrast to Smith-Waterman, which performs a *local* alignment between two sequences, Needleman-Wunsch *globally* aligns two sequences and is often used in bioinformatics to align protein or DNA sequences [65].

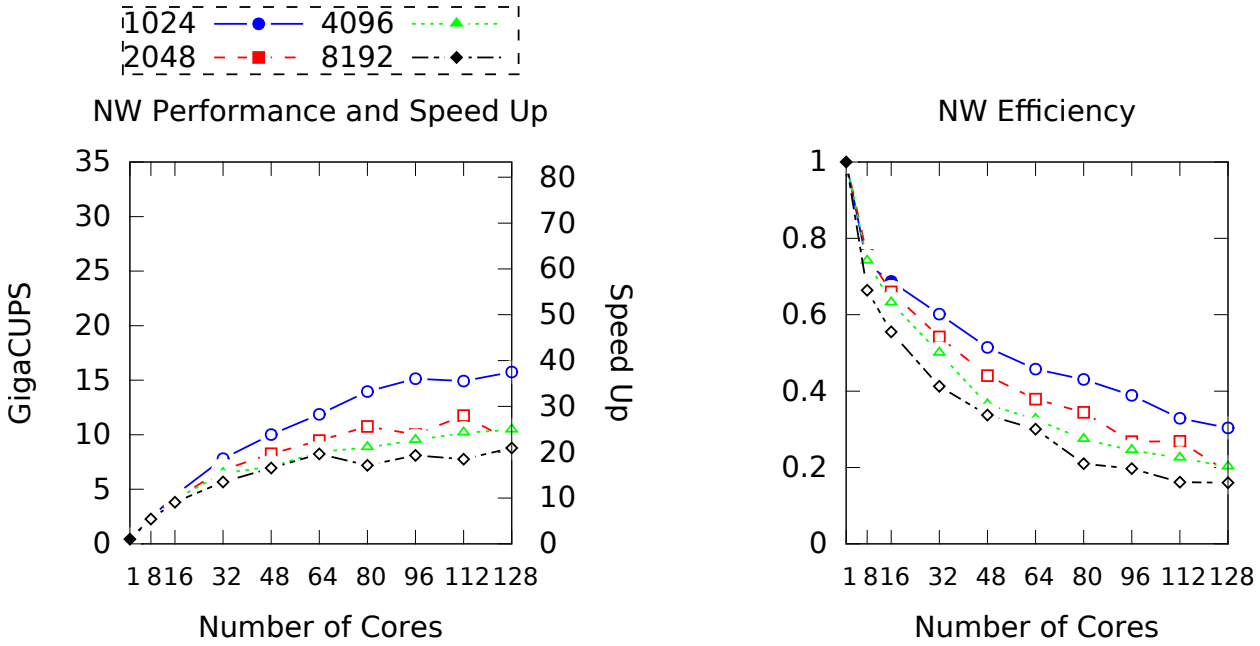
Baseline We utilized SIMD parallelization of the Delta method discussed in Section 3.4.1 *within* a stage for this benchmark. In other words, consecutive input deltas that are independent in a stage are used to compute output deltas for the next stage. Although computing deltas instead of the cell values requires more work (2 deltas per cell), the smaller range of possible values for deltas (as described in Section 3.5) enables the algorithm to use smaller data types for deltas and consequently, fits more deltas in a single SIMD vector unit. Our experiments show that the Delta method is faster than computing cell values.

Our Implementation We implemented the parallel Delta method described in Section 3.4.2.

Data We used 4 pairs of DNA sequences as inputs: Human Chromosomes $(17, 18)$, $(19, 20)$, $(21, 22)$ and (X, Y) from the human reference genome hg19. We only used the first 1 million elements of the sequences since Stampede does not have enough memory on a single node to store the cell values for the complete chromosomes. We also tried 4 different width sizes: 1024, 2048, 4096 and 8192 since we found that widths larger than 8192 do not affect the final alignment score.



(a) Chromosome pair (X, Y): the best performing



(b) Chromosome pair $(21, 22)$: the worst performing

Figure 3.14: Performance, speed up and efficiency results of Needleman-Wunsch with the parallel Rank-1 method.

Performance and Scalability Again, there are the results for *the parallel Rank-1 method*. Figure 3.14 shows the performance, speed up and efficiency of Needleman-Wunsch algorithm parallelized using our approach for two pairs of chromosomes: (X, Y) and $(21, 22)$. Instead of averaging performance numbers over all 4 pairs, we separated them and reported the best performing pair ((X, Y) in Figure 3.14a) and the worst performing pair ($(21, 22)$ in Figure 3.14b). This is because the performance varies significantly between different pairs as can be seen in Figures 3.14a and 3.14b. The figures show results for each of the width sizes: 1024, 2048, 4096 and 8192. Similar to the Viterbi decoder benchmark, filled/non-filled data points show whether convergence occurred in the first iteration of the fix up phase.

The figures show great variability in performance for different inputs based on the variability in convergence. Also, as it can be seen from non-filled data points and Table 3.1, rank convergence in this benchmark is not as fast as in Viterbi decoder or Smith-Waterman.

In Figure 3.14, larger widths perform poorer than smaller ones since the convergence rate depends on the size of each stage in a LTDP instance. Note that we used the same sequence size (1 million element) for all plots.

LCS

Longest Common Subsequence is a method to find the largest subsequence common to two candidate sequences [36] (See Section 3.5 for description).

Baseline We adapted the fastest known single-core algorithm for LCS that exploits bit-parallelism to parallelize the computation *within* a row [22, 37]. This approach is similar to the Delta method from Section 3.4.1 and uses the the grouping technique shown in Figure 3.10 b but only computes the horizontal deltas.

Our Implementation Similar to Needleman-Wunsch, we implemented the parallel Delta method described in Section 3.4.2 using the bit-parallel baseline code.

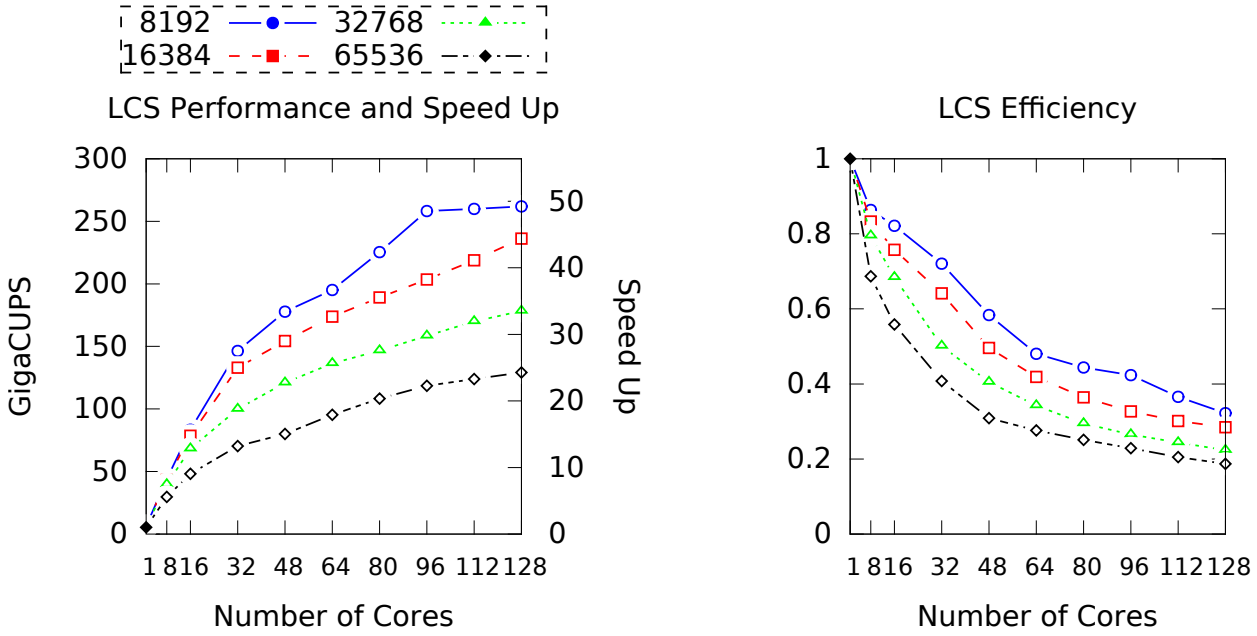
Data We used the same input data as with Needleman-Wunsch except that we used the following width range: 8192, 16384, 32768 and 65536. We report performance numbers in the same way as in Needleman-Wunsch.

Performance and Scalability As before, these are the results for *the parallel Rank-1 method*. The performance, speed up and efficiency plots in Figure 3.15 are very similar to Figure 3.14. We used the same two pairs of chromosomes: (X, Y) and $(21, 22)$ as they are the best and worst performing pairs respectively. The 4 lines in each plot corresponds to one of following width sizes: 1024, 2048, 4096 and 8192. Likewise, the input pair has a great impact on rank convergence as it can be seen in Figure 3.15a and Figure 3.15b.

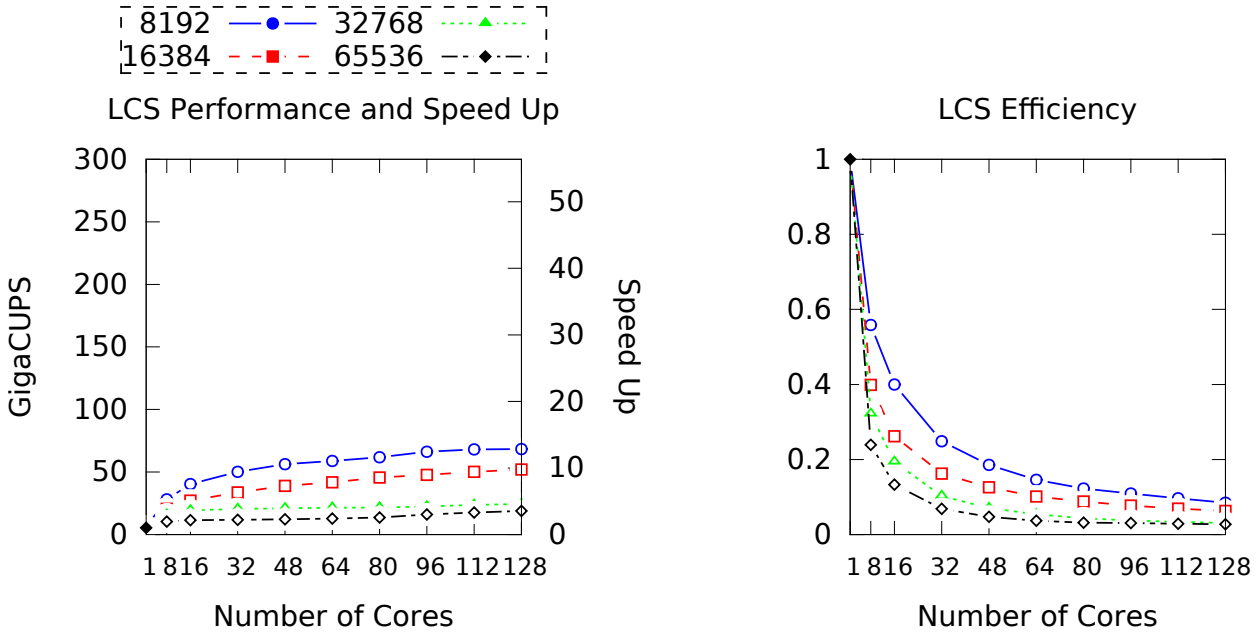
3.6.4 The Rank-1 Method vs The Delta Method

This section shows the effectiveness of the Delta method in compare to the Rank-1 method. For this purpose, we will compare the amount of extra work each algorithm performs along with the timings.

The Comparison for LCS: Figure 3.16 compares the two methods: The left plot shows the performance of the parallel Delta method over the Rank-1 method for different number of cores using different widths. The right plot shows the ratio of the total amount of work done in the last processor by the LTDP algorithm over by the parallel Delta method. We used the amount of work for the last processor because it needs the most number of fix-up phases and consequently, does the most amount of work among all. As it can be seen, the parallel Delta method is between $\sim 5\times$ to $\sim 15\times$ faster than the parallel Rank-1 method with 128 cores which is compatible with the ratio of amount of work for the last processor. Also, as it can be expected, the parallel Delta method has a greater impact with wider widths since the Rank-1 method recomputes all deltas even if only one of them is incorrect.



(a) Chromosome pair (X, Y): the best performing



(b) Chromosome pair $(21, 22)$: the worst performing

Figure 3.15: Performance, speed up and efficiency results of Longest Common Subsequence with the parallel Rank-1 method.

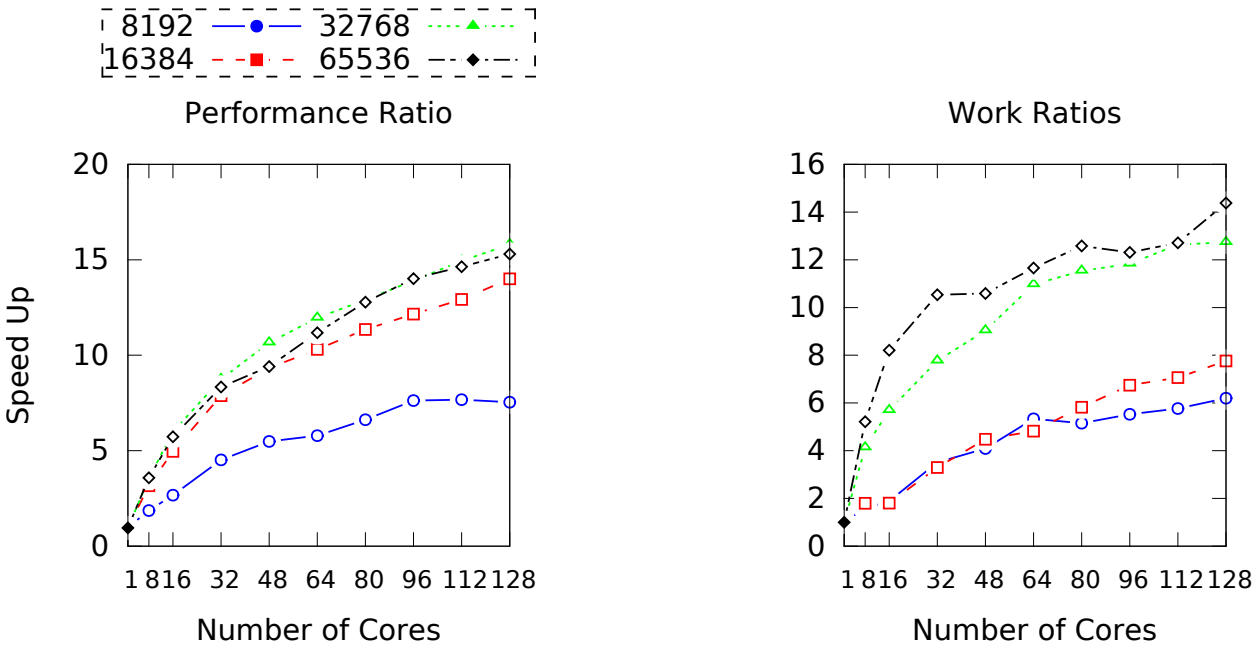


Figure 3.16: The speed up of the parallel Delta method over the Rank-1 method using LCS example.

The Comparison for Needleman-Wunsch: Figure 3.17 similarly compares the two methods for the Needleman-Wunsch algorithm and the plots show the performance speed up and the last processor work ratio. As it can be seen, the parallel Delta method is between $\sim 2\times$ to $\sim 10\times$ faster than the Rank-1 method with 128 cores. This again proves the effectiveness of the parallel Delta method.

3.6.5 Wavefront vs Rank-1 Method

Our goal in this section is to directly compare across-stage parallelism with wavefront parallelism. We focus on Needleman-Wunsch and LCS as the size of the stages in Viterbi and Smith-Waterman is very small for wavefront parallelism to be viable. We should note that the two approaches are complementary. Exploring the optimal way to distribute a given budget of processors to simultaneously use across-stage parallelism and within-stage parallelism is left for future work. Furthermore, note that we implemented the best known wavefront

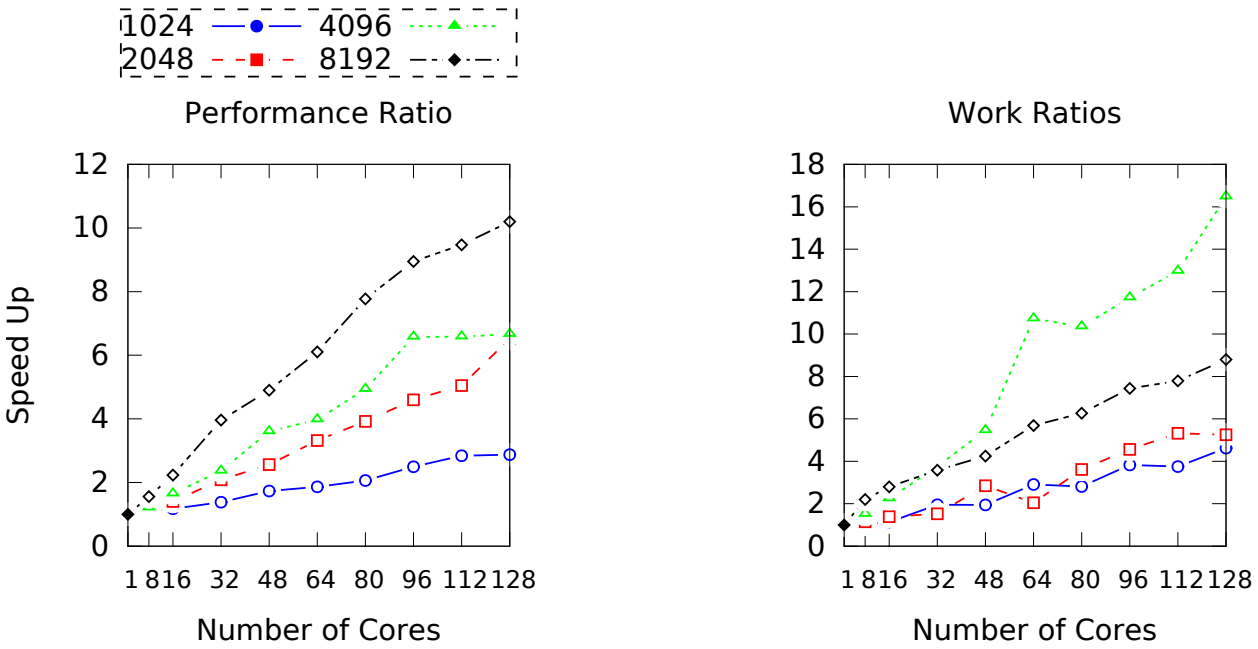


Figure 3.17: The speed up of the parallel Delta method over the Rank-1 method using NW example.

algorithm for each of our benchmarks.

We used OpenMP for wavefront implementations and compared it with our MPI implementation of the parallel Rank-1 method used in our Stampede experiments above, but running on our shared-memory machine. This difference in implementation choice should at the worst bias the results against our parallel algorithm.

Wavefront for Needleman-Wunsch: We used tiling to group cells of the computation table and used SIMD in each tile. Wavefronts proceed along the anti-diagonal of these tiles. Tiling greatly reduces the number of barriers involved [57]. On the other hand, processing cells in a tile by utilizing SIMD has computation overhead over the baseline that we used for our parallel approach (without tiling). Therefore, the sequential performance of the baseline with tiling is slower than the baseline without tiling. We investigated different tiling parameters and chose the best performing configuration.

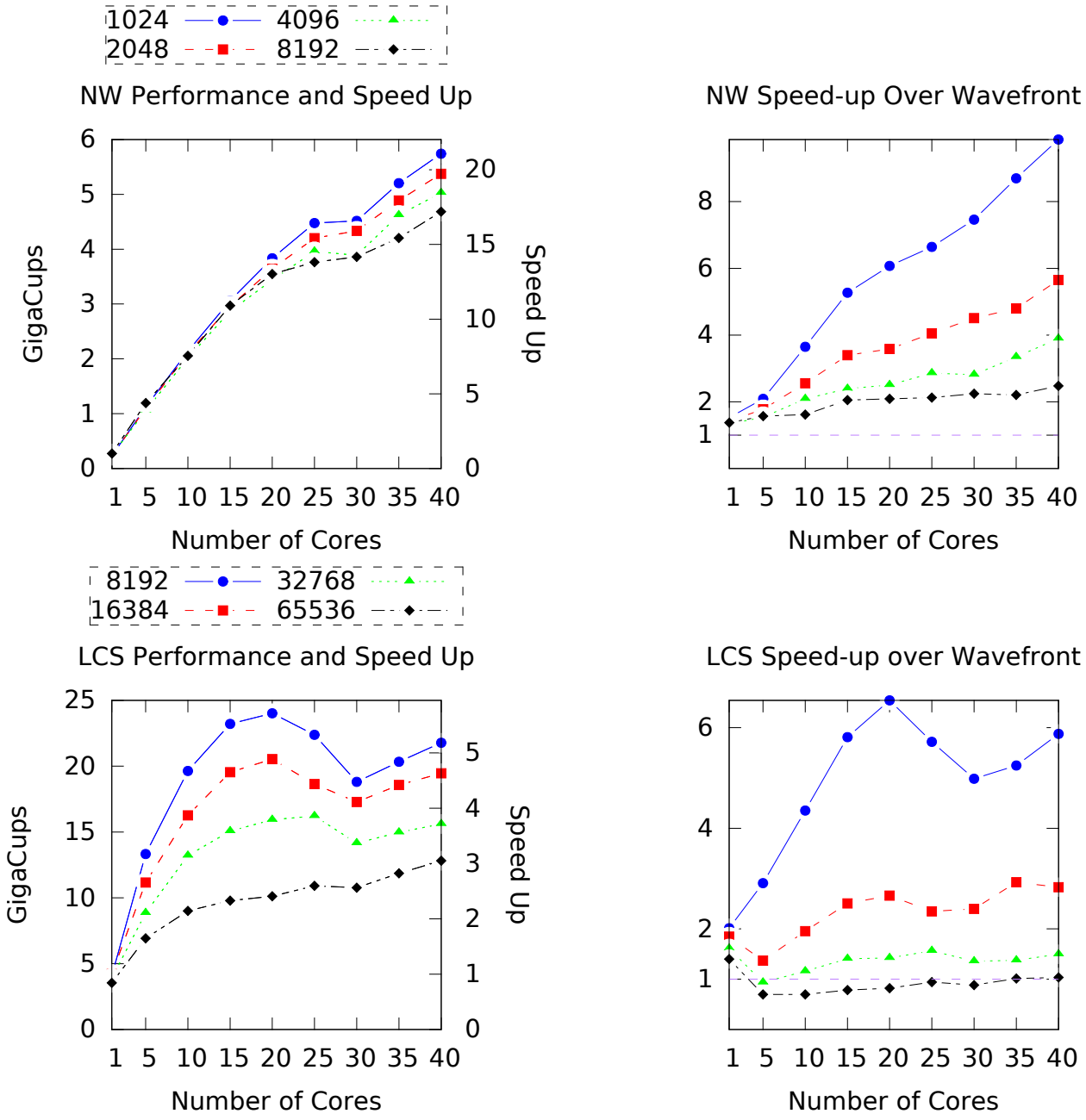


Figure 3.18: Performance/speed up results and comparison of the Rank-1 method and Wavefront parallelism for Needleman-Wunsch and LCS

Wavefront for LCS: Similar to the baseline of Needleman-Wunsch, we tiled the cells. For computation in each cell, we used the same bit-parallelism to parallelize the computation within a column of each tile. Likewise, we parallelized computation of tiles that are in the same anti-diagonal.

Figure 3.18 compares the performance of our parallel Rank-1 method approach with an optimized wavefront based approach for both LCS and Needleman-Wunsch. The plots on the left in Figure 3.18 show the performance and speed up (over sequential non-tiled baseline) of our approach for Needleman-Wunsch and LCS. Plots on the right, a point (x, y) gives the speed up (y as runtime of wavefront divided by runtime of our approach) as we change the number of processors allocated to each approach (x). We plot 4 lines, one for each of four *widths*. Small widths are better for our approach (as wavefront approach incurs more barriers per unit of compute) while large widths are better for wavefront approach (as our approach is less likely to reach rank 1). As we add more processors, our approach utilizes each additional processor more efficiently than a wavefront based approach, particularly so when the width is small (i.e., our approach is $\sim 9\times$ faster than wavefront approach with 40 processors for Needleman-Wunsch and $\sim 6\times$ faster than wavefront approach for LCS with width size 8192).

3.7 Conclusions

This chapter introduces two novel methods for parallelizing a class of dynamic programming problems called linear-tropical dynamic programming problems, which includes important optimization problems such as Viterbi and longest-common subsequence. The algorithm uses algebraic properties of the tropical semiring to break data dependence efficiently.

Our implementations show significant speed ups over optimized sequential implementations. In particular, the parallel Viterbi decoding is up-to $24\times$ faster (with 64 cores) than a

highly optimized commercial baseline.

While we evaluate our approach on a large shared memory multi-core machine, we expect equally impressive results on a wide variety of parallel hardware platforms (clusters, GPUs and even FPGAs).

As discussed in this chapter, LTDP problems are instances of SSSP problem where the structure of the graph between stages is the same but the weights are dynamically changing. Since the graph can have any shape, the LTDP problems are considered amorphous problems. The Rank-1 method avoids the overwhelming synchronizations of the wavefront parallelism by performing extra work as overhead. This idea is similar to that of DSMR and subgraph extraction technique from Chapter 2. Next, we will discuss a new notation to express these problems.

Chapter 4

Tiled Linear Algebra

Chapter 3 showed that how linear algebra and LTDP problems are tightly connected. In this chapter, we show that other amorphous problems can be expressed with linear algebra operators in a concise and clear manner as discussed in [13, 58]. This includes LTDP and graph problems such as reachability and SSSP (whose parallelization was discussed in details in Chapter 2). The use of this notation allows for rapid development of complex algorithms. Today, the power and flexibility of using linear algebra primitives comes with drawbacks. Standard sparse linear algebra kernels do not always take advantage of the structure of the sparsity or parameters of the target machine and, as a result, fall short of the performance of custom implementations. To address this, we propose *Tiled Linear Algebra (TLA)* that is a multi-level parallel system with high-level linear algebra structure. In TLA, linear algebra primitives are used to construct a correct program and then performance features controlled by "knobs" are used to tune the kernels. These features include controlling distribution and communication frequency.

We organize this chapter as follows. Section 4.1 describes the use of linear algebra for graph algorithms and Section 4.2 describes our extensions. We describe the algorithms to solve the Single Source Shortest Path (SSSP) problem and our experiments in Section 4.3.

Section 4.4 describes how LTDP problems can be expressed in TLA. We wrap up with our conclusions in Section 4.5.

4.1 Graph Algorithms using Linear Algebra

Using linear algebra as an abstraction for programming parallel graph algorithms is not new. CombBLAS [13] is perhaps the best known system using this approach. This section overviews linear algebra for representing graph algorithms.

There is a correspondence between a graph and a matrix. A graph $G = (V, E)$ with n vertices (set V) and m edges (set E) can be represented by its adjacency matrix A which is an $n \times n$ matrix such that $A(i, j) = 1$ if there is an edge e_{ij} from vertex v_i to vertex v_j and $= 0$ otherwise. This allows for directed and undirected graphs and can be extended to weighted graphs by using the weight rather than 1 to represent an edge. This representation is at the core of using linear algebra to describe graph algorithms. It should be noted that typically this matrix will be sparse and performance efficiency will depend, just as in conventional linear algebra, on how sparsity is handled.

Reachability Example: Reachability is the problem of finding all the reachable vertices in a directed graph $G = (V, E)$ from a source vertex $s \in V$. More formally, $Reach(G, s) = \{v \in V | \exists v_1, v_2, \ldots, v_k \in V, v_i v_{i+1} \in E, v_1 = s, v_k = v\}$. There exists a duality between reachability and matrix vector multiplication. Consider a vector r with $|V|$ elements (i.e. one element per vertex of the graph) with values $r(s) = 1$ and 0 everywhere else and the adjacency matrix A of graph G. All neighbors of s reachable in 1 step correspond to the non-zero entries of the vector $A^T \cdot r$. Consider Figure 4.1 which shows a graph with its transposed adjacency matrix, A^T, with non-zeros shown as dots. Vertex 7 is the source vertex, s, of the reachability problem and as just mentioned, r has only one non-zero element (represented

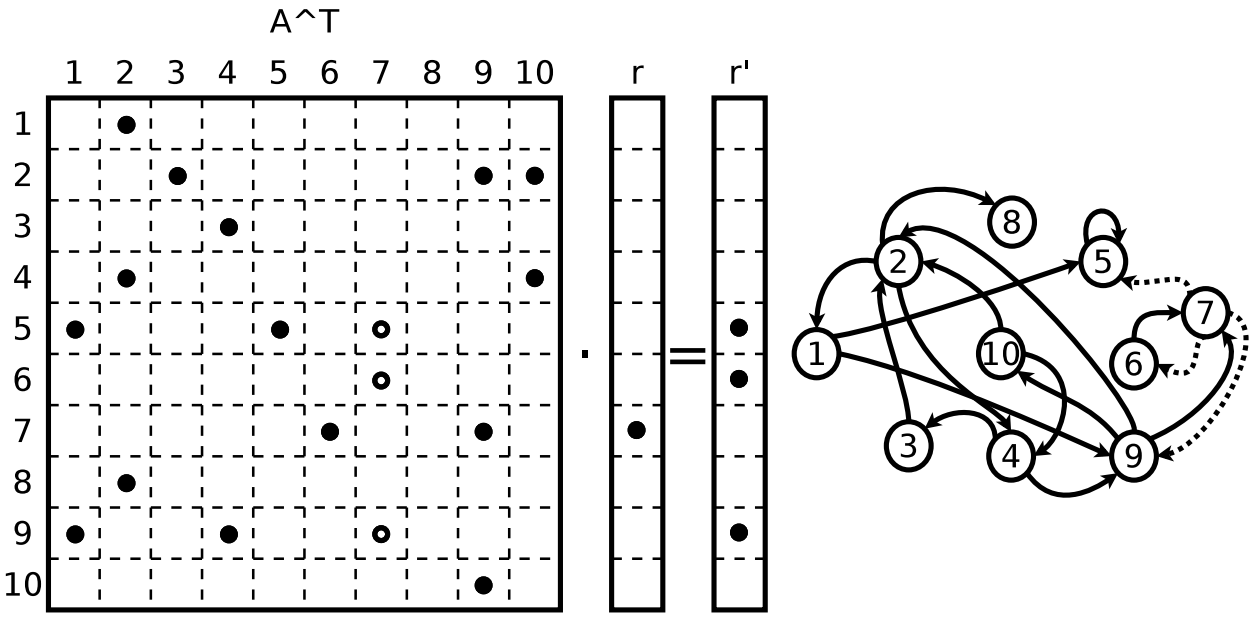


Figure 4.1: Matrix-vector multiplication for reachability.

by the dot in position 7 for s). The result of the matrix vector multiplication will produce a vector r' with non-zeros represented by dots which corresponds to the non-zeros in the matrix shown by unfilled dots in the figure. $r' = A^T \cdot r$ computes the vertices that can be reached from 7 by traversing one edge (shown by dotted arrows in the figure). $r'' = r + r' = r + A^T \cdot r$ represents all vertices that can be reached in 1 or fewer (0) edge traversals. In general, if r_0 includes 7 as a reachable vertex, an iterative matrix-vector multiplication $r_{i+1} = r_i + A \cdot r_i$ will find all of the vertices which are in $i + 1$ or fewer edge traversals. The algorithm would terminate when a fixed point has been reached which in this case means that r_{i+1} and r_i has non-zeros in the same positions. These non-zeros in the final vector corresponds to all vertices reachable in any number of edge traversals. Note that, the elements of r_i could have different (positive) values and these values do not have a clear meaning. However, if we had a different algebra and replaced 1 and 0 by *true* and $false$, regular multiplication by $\wedge$, and regular addition by $\vee$, the result would have been the same except that all the non-zeros would all have been *true*.

In Section 3.1 we explained semirings and the tropical semiring. We review these concepts here one more time. A **semiring** is a five-tuple $(D, \oplus, \otimes, \mathbb{0}, \mathbb{1})$, where D is the set of elements of the semiring and D is closed under $\oplus$ and $\otimes$. $\mathbb{1} \in D$ is the identity for $\otimes$ which means that $\forall x \in D : x \otimes \mathbb{1} = \mathbb{1} \otimes x = x$ and $\mathbb{0} \in D$ is the identity for $\oplus$ which means that $\forall x \in D : x \oplus \mathbb{0} = \mathbb{0} \oplus x = x$. Also, $\mathbb{0}$ nullifies any elements of D with $\otimes$: $\forall x \in D : x \otimes \mathbb{0} = \mathbb{0} \otimes x = \mathbb{0}$. Given two matrices, $A_{l,m}$ and $B_{m,n}$, with elements from a semiring, D, their product is denoted $A \odot B$ and results in an $l \times n$ matrix defined such that

$$(A \odot B)(i, j) = A(i, 1) \times B(1, j) + A(i, 2) \times B(2, j) + ... + A(i, m) \times B(m, j)$$

In this notation, the semiring $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$ with the real numbers extended with ∞ as the domain, min as the additive operation $\oplus$, and $+$ as the multiplicative operation $\otimes$ is called the **tropical** semiring. Tropical semiring is specifically useful for computing single source shortest path from a source vertex to every other vertex in a graph which is discussed in more details below. The other useful semirings for other algorithms are the real field $(\mathbb{R}, +, \times, 0, 1)$ for page-rank computation or the boolean semiring $(\{0, 1\}, \vee, \wedge, 0, 1)$ which, as mentioned in the example above, is a natural algebra for reachability problems.

A directed and weighted graph $G = (V, E)$ can be represented by its adjacency matrix A_G of size $n \times n$ where $|V| = n$ and whose values are $A_G(i, j) = weight(v_i v_j)$ if $v_i v_j \in E$ and $A_G(i, j) = \infty$ otherwise. That is, the elements of A_G are from the tropical semiring. A_G is a sparse matrix where the sparsity comes from the $\mathbb{0} = \infty$ elements in the matrix which represent the non-existent edges in the graph. Let d be an $n \times 1$ vector where its i^{th} element, $d(i)$ is the distance to v_i, then $d' = A_G^T \odot d$ would also be a distance vector where $d'(i) = \min_{v_j v_i \in E(G)}(d(j) + weight(v_j v_i))$. $d'' = d \oplus A_G^T \odot d = d \oplus d'$ is another distance vector where in the computation of $d''(i)$, $\forall j : d(j) + w(v_j v_i)$ are considered as well $d(i)$ itself. This is equivalent to first, computing a new distance $d'(i)$ for each vertex v_i considering distance

d of its incoming neighbors and the weight of corresponding edge ($\forall j : d(v_j) + w(v_j v_i)$) and second, comparing this new $d'(i)$ distance with $d(i)$ and setting $d''(i)$ to the smaller one.

Using adjacency matrix representation, one can express algorithms to find the shortest path using matrix-vector product in tropical semiring. For example, assuming that d_0 is a distance vector where $d_0(s) = 0$ and $\forall v \in V(G) \backslash \{s\} : d_0(v) = \infty$. At step i, the well known Bellman-Ford algorithm computes $d_{i+1} = d_i \oplus A_G^T \odot d_i$ and it iterates for $|V(G)| - 1$ times. In Section 4.2.2, we will explain how linear algebra can be used to express other algorithms.

4.2 Tiled Linear Algebra

One of the most important aspects of linear algebra for graph algorithms is that the adjacency matrix of a graph G, A_G, is sparse and the system needs to use sparse algorithms. Otherwise, for example, a matrix-vector multiplication will require $O(|V|^2)$ operations instead of $O(|E|)$. Furthermore, different ways of representing a sparse matrix can impact the performance. Therefore, we believe that it is important for the programmer to have control over the representation.

Papers [42] and [13] discuss how graph algorithms can be represented in terms of matrix operations on different semirings. However, there are many ways to parallelize a matrix operation. In particular, the parallelization of matrix operations can be represented using *tiling* which partitions an array into sub-arrays by dividing each of the dimensions of the original array into segments. Each tile is assigned to a processor which will be responsible for the values of that tile. This assignment is done by a mapping function from tiles to processors.

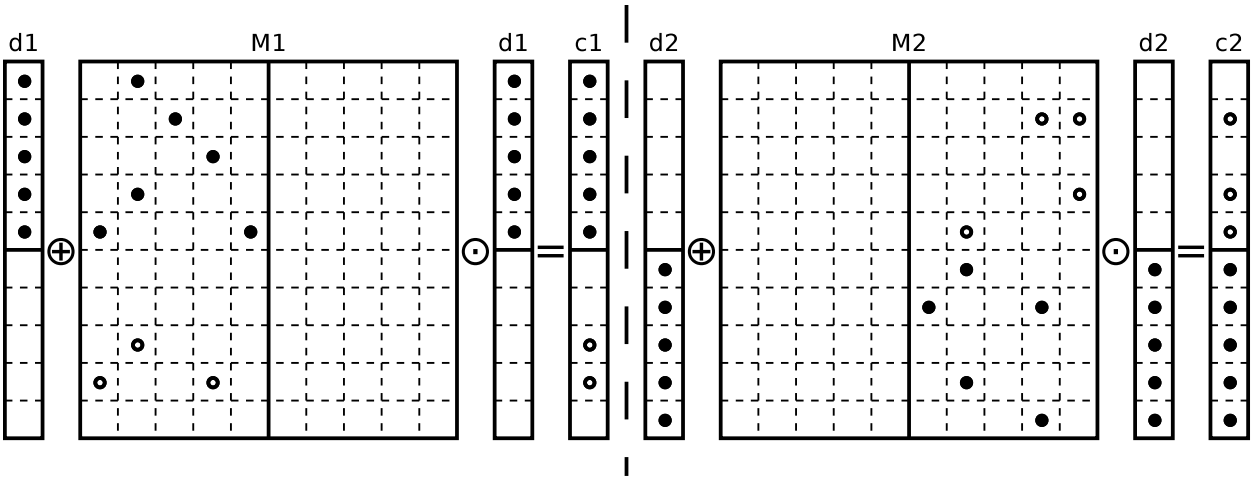
In our notation, we tile and assign tiles to processors at the same time. Let's say matrix A is read from file `input.txt` (which, for example, contains the edge list of a graph) and we want to divide the rows and columns into two segments. We will use command `A =`

`ReadAndTile('input.txt', 2, 2, f);` to read from the file and tile it accordingly. `f` is a function that assigns each tile to a processor. Therefore, there are four tiles which we denote by $A_{1,1}$, $A_{1,2}$, $A_{2,1}$ and $A_{2,2}$. Each tile is conceived as having the size equal to the whole matrix, but contain non-$\mathbb{0}$ only in the regions associated with the tile. Therefore, $A_{1,1} \oplus A_{1,2} \oplus A_{2,1} \oplus A_{2,2} = A$. To create a vector of size $n \times 1$ and tile it into 2 segments, we will use command `v = CreateArray(n,2,f2);` where `f2` is another mapping function. Even though we explicitly ask the user for a tiling pattern, we do not explicitly use the tiling when representing operations.

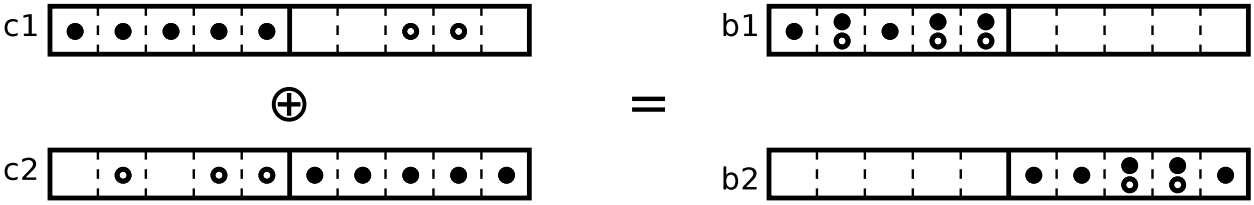
4.2.1 Delaying Updates

As is well known, we can use the tiling to control where each component of the computation occurs and how the processors communicate. In regular parallel linear algebra, updates need to be visible by all the processors as soon as they occur. However, the updates can be postponed as, for example, is done in asynchronous algorithms [5, 44].

Let A_G be the adjacency matrix of G and $M = A_G^T$. As discussed above, if d is a distance vector for the vertices of G, $c = d \oplus M \odot d$ will be another distance vector with distances updated by traversing 0 or 1 edges. Figure 4.2a represents the computation of $c = d \oplus M \odot d$. Let M be tiled 1×2 and $f(x, y) = y$ be the tile to processor mapping function such that tile $(1, 1)$ is assigned to processor p_1 and tile $(1, 2)$ is assigned to processor p_2. Also assume that d and c are vectors of size $n \times 1$ ($n = |V(G)|$ which is 10 in Figure 4.2a) and they are tiled 2×1 and $f_2(x, y) = x$ is their mapping function. Figure 4.2a shows these tilings and distributions of M, d and c. The dots in the Figure represent the non-zeros. Processor 1 stores `M1`, `d1` and `c1` and processor 2 stores `M2`, `d2` and `c2`. Notice that `M1` and `M2` are each the size of the original matrix `M`, but with zeros outside the tile each of them represents. `d1` and `d2` have the same property. On the other hand, `c1` and `c2` are the size of the original vector `c` but there are non-zeros outside of the tiles that they are representing shown by



(a) Saving off-tile values locally to be communicated later.



(b) Communicating off-tile values.

Figure 4.2: Sparse matrix vector multiplication with delaying updates.

unfilled dots.

It is easy to see that $c = d \oplus M \odot d = (d_1 \oplus M_{1,1} \odot d_1) \oplus (d_2 \oplus M_{1,2} \odot d_2)$. Therefore, without any communication, p_1 can compute $d_1 \oplus M_{1,1} \odot d_1$ and p_2 can compute $d_2 \oplus M_{2,1} \odot d_2$. However, since c_1 and c_2 have non-zeros everywhere, it is necessary to do a global computation to prepare for the next iteration. The value of d in the next iteration is $c = (d_1 \oplus M_{1,1} \odot d_1) \oplus (d_2 \oplus M_{1,2} \odot d_2)$ which requires adding c_1 and c_2.

To save communication time, we assign c_1 to d_1 and c_2 to d_2 before going to the next iteration. In this way, we do not carry out a global reduction. In other words, we do not add c_1 to c_2 to get c. Instead, the values in the second tile of c_1 and the first tile of c_2 continue accumulating separately. At some point in time a global reduction is performed. As discussed below, by avoiding numerous global reductions, the performance of the algorithm improves. Postponing the reduction as just described is useful in the cases where it is not necessary to

communicate the computed values immediately. This is the case of the reachability problem in which a processor can compute multiple iterations locally and find more reachable vertices before it communicates the remote reachable vertices.

To be able to postpone updates using our notation, we introduce $\leftarrow$ as an assignment operand which computes the linear algebra operation using only local values on the processor. In the case where an element on the left hand side of $\leftarrow$ is assigned to a different process, that value is only updated locally. For example, `c <- d+M*d` will perform the local computation and will produce values that go to another processor. These values are the unfilled dots shown in Figure 4.2a for both processors involved.

For communicating the values saved locally to the processor which owns the value, we introduce the $\oplus =$ operation which communicates all of the saved values to the owner processor where they are accumulated to the local copies of the elements using $\oplus$ as Figure 4.2b depicts it. In the figure, `b1` and `b2` are the values assumed by `c1` and `c2` in the next iteration. After the owner processors update their values, communicated values become $\mathbb{0}$.

4.2.2 Partial Computation

In the simple version of reachability described in Section 4.1, all the reached vertices are processed in each iteration. Processing a vertex v in this problem means that marking all neighbors of v as reachable. This is clearly sub-optimal since after one iteration all neighbors of a vertex have been reached. To improve upon this, we limit the processing to only vertices who were reached for the first time in the previous iteration. To be able to support this feature, we propose using mask vectors with $\mathbb{0}$ representing false and $\mathbb{1}$ representing true from the boolean semiring. Mask vectors are not different from other vectors except for the purpose they are used. A mask vector is used with *element-wise multiplication* represented by the operator "$\otimes.$". If a and b are two vectors in a semiring, $a \otimes. b$ is another vector where $(a \otimes. b)(i) = a(i) \otimes b(i)$. Now if b is a mask vector, $a \otimes. b$ will be a sparse sub-vector of a

with some elements set to $\mathbb{0}$.

Note that mask vectors are used to avoid unnecessary computation. Therefore, the system should be aware of the fact that a vector can be sparse. For example, in the case of $A \odot v$ where A is a sparse matrix and v is a sparse vector, only a corresponding columns of non-zero elements of v should be considered.

Partial computation is important for many algorithms where an update to a vertex will only affect a few neighboring vertices. For example, in the case of SSSP, if a vertex is updated, an algorithm needs only to update its outgoing neighbors. Another example is the PageRank problem where if an update to a vertex is higher than the threshold, it only affects the neighbors of that vertex.

4.3 Single Source Shortest Path

The Single-Source Shortest Path (SSSP) problem finds the shortest distance from a source vertex to every other vertex in a graph. An instance of the problem is denoted by (G, w, s) where $G = (V, E)$ is a graph with the set of vertices, V, and the set of edges, E, and a source vertex, $s \in V$. Each edge $vu \in E$ has a tail, $v \in V$, and a head, $u \in V$. The map $w : E \to \mathbb{R}$ associates a weight for each edge $vu \in E$. Vertex $s \in V$ is the source whose distances to all other vertices is desired. This section assumes that all the weights are positive. The shortest distance from s to v is denoted by $d(s, v)$.

As discussed in detail in Chapter 2, there are several algorithms to solve SSSP and we will discuss about how some of them can be expressed in TLA. In spite of their differences, the main operation in these algorithms is matrix-vector multiplication in tropical semiring.

4.3.1 Algorithms

The four best-known algorithms to solve SSSP problem are: Dijkstra [24], Bellman-Ford [10], Chaotic-Relaxation [17], and Δ-Stepping [61]. Bellman-Ford is the only algorithm that is capable of solving SSSP with negative edge weights but this aspect will not be discussed further in this chapter and we assume all the graphs have positive edge weights. As discussed in Chapter 2, the basic operation that all four algorithm use is **edge relaxation** which takes an edge vu and checks if $d(s,v)+w(vu) < d(s,u)$ where d is not necessarily the final minimum distance but the shortest path "so-far" in the computation. If the check condition is true, $d(s,u)$ is updated with $d(s,v) + w(vu)$. The difference between the algorithms mentioned above is in the **order** in which relaxations are applied which directly affects the amount of work each algorithm performs. We measure the amount of work done by each algorithm in terms of the number of edge relaxation (for $d(s,v) + w(vy) < d(s,u)$).

Below, we assume that $G = (V,E)$ is the input graph and that the transpose of its adjacency matrix is M. Initially, in all four algorithm $d(s,v) = \infty$ for all $v \in V - \{s\}$ and $d(s,s) = 0$. The values of d are stored in a tiled vector. Next, we will explain each algorithm and express them in TLA.

Bellman-Ford:

Our implementation of the Bellman-Ford algorithm in TLA (shown in Figure 4.3) relaxes all the vertices during each iteration. The algorithm terminates after $|V| - 1$ iterations. As we discussed in Section 4.1, `d+M*d` which corresponds to the formula $d \oplus M \odot d$ computes a new distance vector for G by relaxing all the edges. Parallelizing this algorithm is straightforward by partitioning the vertices and having each processor relax one or more of the of the resulting subsets. As shown in Figure 4.3, in every iteration of the for loop, each processor relaxes its own portion of edges assignment ("`<-`" in line 2) and then a global communication (operation `+=`) sends remote updates (line 3).

```
1  for (int i = 0; i < n-1; i++){
2    d <- d + M*d;
3    d += d;  }
```

Figure 4.3: Bellman-Ford algorithm main loop using TLA.

Chaotic-Relaxation:

The Chaotic-Relaxation algorithm is the same as the Bellman-Ford algorithm except that at each iteration, it only relaxes those vertices which changed distances in the previous iteration. TLA code for this algorithm is shown in Figure 4.4. This algorithm is a small improvement over Bellman-Ford obtained by avoiding redundant relaxations. To this end, we use the mask vector `r` which has one element for each vertex and is used to keep track of the vertices whose distances did not change in the previous iteration. The vector `r` is initialized so that it is false everywhere except for the position corresponding the vertex s. The element-wise multiplication (`*.`) on line 2 prunes elements which did not change their distance and sparse matrix-sparse vector multiplication (`M*(d*.r)`) takes advantage of it.

In the following algorithms, the scalar `notDone` (replicated across processors) is used to decide when to terminate the algorithm. The last iteration is that in which $d(s, v)$ remain constant for all $v \in V$. In other words, the algorithm is finished when `r` (set on line 4) is all `0` (all false) for each tile. Note that `r <- b != d` sets `r(i)` to `1` if `b(i) != d(i)` and sets it to `0` otherwise. Finding out when `r` is all `0` is done by the local reduction `notDone <- any(r)` on line 6 followed by the global reduction `notDone += notDone` on line 7. If `notDone` is `0`, it means that there was at least one `0` in one of the tiles of `r`.

Dijkstra's Algorithm:

Dijkstra's algorithm is the fastest sequential SSSP algorithm. At each iteration in this algorithm, only one vertex is **processed** which is the vertex with minimum distance among the vertices not processed before. Processing a vertex means relaxing all of its outgoing

```
1 do {
2   c <- d + M*(d*.r);
3   b += c;
4   r <- (b != d);
5   d <- b;
6   notDone <- any(r);
7   notDone += notDone; /* global reduction */
8 } while (notDone != 0);
```

Figure 4.4: Chaotic-Relaxation main loop using TLA.

```
1  m <- 0;
2  for (int i = 0; i < n; i++){
3    d <- d + M*(d*.r);
4    d += d;
5    m <- m+r;
6    ind <- argmin(d*.(!m));
7    minVal <- min(d*.(!m));
8    globalMinVal += minVal;  /* global reduction */
9    r <- 0;
10   if (minVal == globalMinVal)
11     r(ind) <- 1;   }
```

Figure 4.5: Dijkstra's algorithm main loop using TLA.

edges. The TLA code for this algorithm is shown in Figure 4.5.

The major difference between this algorithm and the previous two algorithms is that the matrix-vector multiplication for this algorithm (line 3) is with a vector which has only one non-$\mathbb{0}$ element in it (`d *.  r`) and that element corresponds to the vertex with the minimum distance among the unprocessed vertices. Finding the element with minimum distance is done with the help of vector `m` which keeps track of the processed vertices (line 5). Lines 6 and 7 find the index and the minimum distance vertex in each processor locally and it is communicated globally on line 8. Lines 10 and 11 determines in each processor if the local minimum value is equal to the global minimum value and, if so, sets `r` accordingly. The algorithm terminates after all n vertices are processed.

Δ-Stepping:

Δ-Stepping is another SSSP algorithm which is half way between the Dijkstra's and the Chaotic relaxation algorithms. Δ-Stepping processes a bucket of vertices in each iteration not just one as in the case of Dijkstra's algorithm nor all vertices as in the case of the Chaotic relaxation algorithm. Δ-Stepping distributes vertices into buckets $\{b_0, b_1, b_2, \dots\}$ where bucket $b_i = \{v | \Delta i \leq d(v) < \Delta(i+1)\}$. Note that $d(v)$ is dynamic and as the algorithm advances, it changes value. Therefore, the algorithm should update the bucket for each vertex that is updated. In iteration i, Δ-Stepping only considers vertices from bucket b_i. Since relaxing outgoing edges of a vertex from b_i may add more vertices in it, this process has to continue until there are no more vertices in b_i whose outgoing edges are not relaxed. The algorithm terminates when there are no vertices to process. Note that if b_i is completely processed and the algorithm advances to b_{i+1}, it will never again need to process vertices from b_i since all the weights are positive.

Our version of Δ-Stepping in TLA code is shown in Figure 4.6. It is similar to the code in Figure 4.4 with a few modifications. First, the main matrix-vector multiplication for relaxation is different which is shown on line 2. There are two mask vectors for this algorithm: 1) `r` which is similar to `r` from in Figure 4.4 and it holds the vertices that needs to be processed; 2) `bucket` which is set on line 6 and contains the vertices that belong to bucket b_i. Initially, it contains vertices that are in the range of $[0 \dots \Delta)$. To find out whether there are more vertices in bucket b_i to relax, the scalar `notDoneBucket` is used on line 7. Similar to `notDone` scalar from Figure 4.4, a reduction on array `bucket*.r` determines if `i` should be incremented (line 10) and `bucket` is set accordingly (line 11).

None of these 4 algorithms required delaying updates and all of them could have been done by using local computation with global communication. Next, we will describe our parallel SSSP algorithm which takes advantage of this feature.

```
1  do {
2    c <- d + M*((d*.r)*.bucket);
3    c += c;
4    r <- (c != d);
5    d <- c;
6    bucket <- d >= i*Δ & d < (i+1)*Δ;
7    notDoneBucket <- any(bucket*.r);
8    notDoneBucket += notDoneBucket;
9    if (!notDoneBucket){
10     i++;
11     bucket <- d >= i*Δ & d < (i+1)*Δ;
12   }
13   notDone <- any(r);
14   notDone += notDone;
15 } while (notDone != 0);
```

Figure 4.6: Δ-Stepping main loop using TLA.

Dijkstra Strip Mined Relaxation:

This algorithm is the DSMR algorithm introduced in Chapter 2. As discussed before, local relaxation occurs in a Dijkstra-like order for multiple iterations (for D iterations) and after that a global communication exchanges the distance updates.

The TLA code for this algorithm is shown in Figure 4.7. As described above, the algorithm applies a local Dijkstra for its vertices as shown in Lines 4-9. This part of the code is similar to the one from the Dijkstra's algorithm in Figure 4.5 except the `relaxed` variable and that every computation is local (note that only `<-` is used). The variable `relaxed` keeps track of number of edges relaxed in line 5. Since there is locally only one non-zero in `r` for each processor which corresponds to the vertex with the minimum distance, the number of non-zeros in `M*r` is the number of edges connected to that vertex. Therefore, the computation in Line 5 counts the number of relaxed edges in each iteration of the while loop in Line 3. The while loop in Line 3 exits when `relaxed` reaches the threshold D. As discussed in Chapter 2, D is a parameter that can be set for different graphs. A global reduction in Line 12 updates distances in all processors.

Array `m`, as in the Dijkstra's algorithm in Figure 4.5, is a bit vector for processed vertices.

```
1  do {
2    int relaxed = 0;
3    while (relaxed < D){
4      d <- d + M*(d*.r);
5      relaxed += nnz(M*r);
6      m <- m+r;
7      ind <- argmin(d*.(!m));
8      minVal <- min(d*.(!m));
9      r(ind) <- 1;
10   }
11   old_d <- d;
12   d += d; /* global reduction */
13   m <- m*.(old_d != d);
14   relaxed = 0;
15   notDone <- any(r);
16   notDone += notDone;
17 } while (notDone != 0);
```

Figure 4.7: DSMR algorithm in TLA.

The global reduction on `d` may activate some vertices which should be taken into account for `m`. This is done by copying the values of array `d` to `old_d` before the global reduction in Line 11 and adjusting `m` in Line 13. Finally, `relaxed` is set to `0` after the global reduction. The rest of the code is similar to the other SSSP algorithms.

4.3.2 Performance Comparison

This section compares the parallel performance of each of the SSSP algorithms described in Section 4.3.1. We will also how each feature of TLA affects the performance.

Figure 4.8 shows the parallel performance of Chaotic Relaxation, Dijkstra, Δ-Stepping and DSMR. We excluded Bellman-Ford from this figure since it is significantly slower than the other four algorithms ($\sim 2000\times$). The X axis in this figure represents different number of processors and the Y axis is for running time. Each algorithms is specified by its color: blue for Chaotic Relaxation, gray for Dijkstra, red for Δ-Stepping and purple for DSMR. The orange color is for the communication cost. Therefore, each group of 4 bars represents the running time for each algorithm with one specific number of processors. The input graph

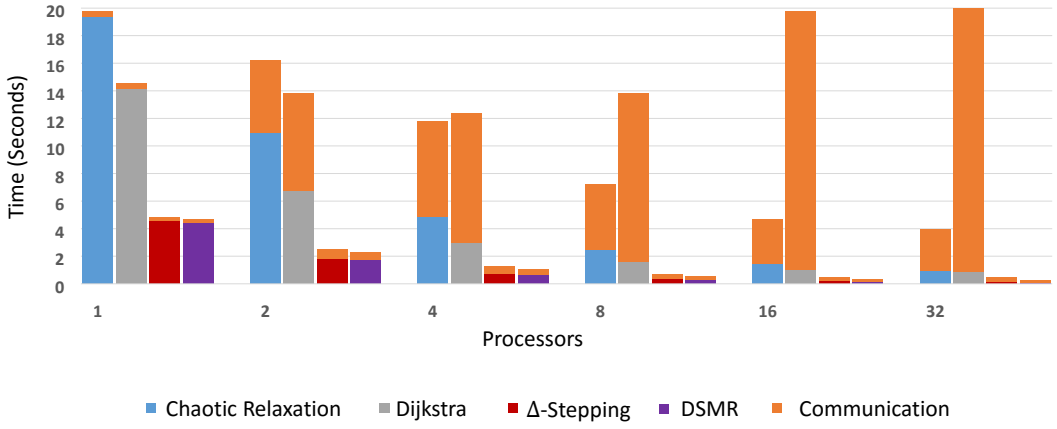


Figure 4.8: SSSP algorithms performance comparison

is an R-MAT graph with $SCALE = 20$.

The fact that Bellman-Ford is significantly slower than Chaotic-Relaxation (order of 2000×) shows that how crucial is partial computation for SSSP algorithms. On the other hand, Chaotic Relaxation algorithm does not scale well because of the communication cost. But as it can be seen, just the blue portion of running time is scaling well. This is because the algorithm is massively parallel and the work is balanced well since each processor owns roughly the same number of edges.

Dijkstra algorithm in Figure 4.8 has a better sequential performance than Chaotic relaxation but since it only processes one vertex at a time, the parallel performance is poor and most of the communication cost is just the idle time. However, Δ-Stepping is performing faster than both Dijkstra and Chaotic Relaxation and it is providing decent speed up. DSMR is almost as fast as Δ-Stepping for 1, 2 and 4 processors. It is hard to see in Figure 4.8 how they compare with higher number of processors. Therefore, Figure 4.9 directly compares

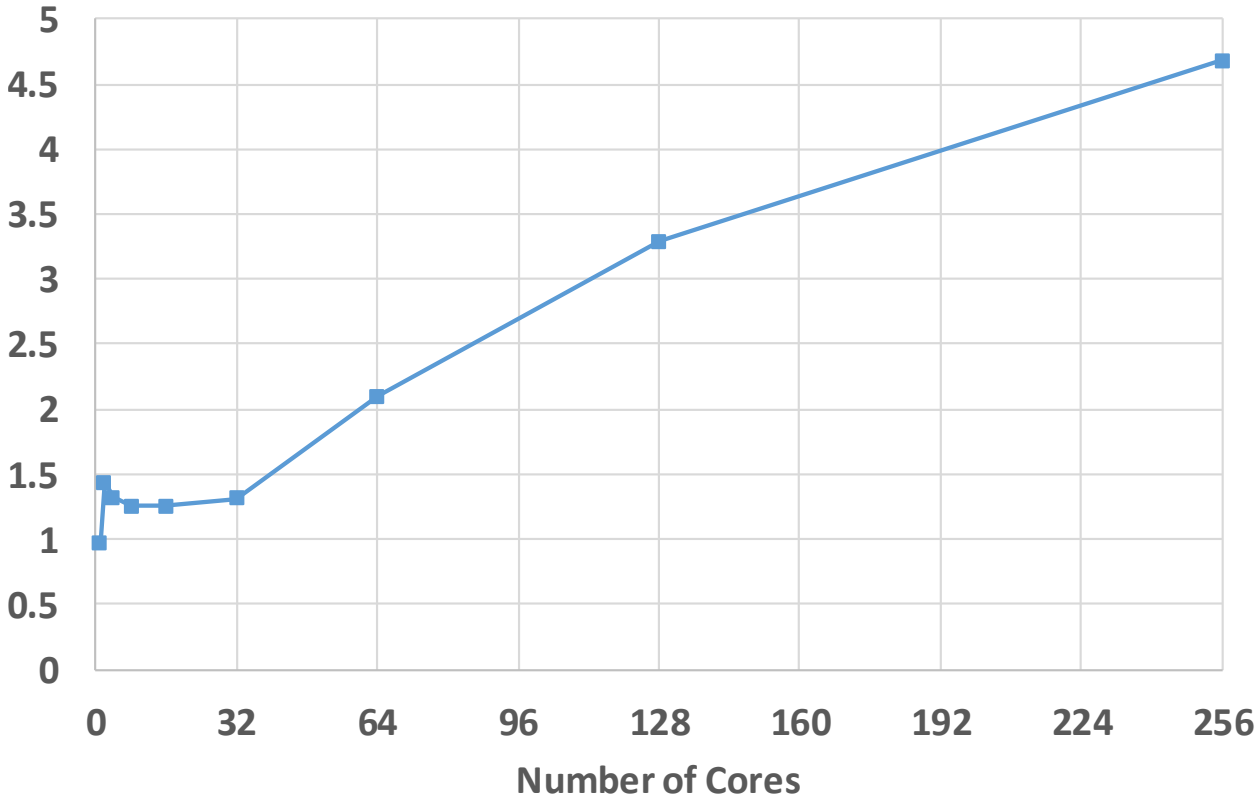


Figure 4.9: Speed of DSMR over Δ-Stepping.

them.

Figure 4.9 shows the speed up of DSMR over Δ-Stepping for different number of processors. As it can be seen, up to 16 processors, it does not provide significant improvement but for larger number of processors, it is certainly effective which in this case is more than 4× faster. This shows the importance of delaying updates feature in TLA. Next section, will study how D itself can control the performance of our algorithm.

4.3.3 Impact of D in the DSMR Algorithm

DSMR, as discussed before, has a tunable parameter, D. D factor which impacts the total number of edge relaxations, has different best values for different number of processors. Again, each edge relaxation is to find whether we can reach a vertex with a shorter distance.

We will use the number of edge relaxations as a measurement for the amount of work the algorithm does. D is a variable, as shown in Figure 4.7, that controls the number of edge relaxation between consecutive global reductions. In other words, it controls the frequency of global communication which impacts the performance in two ways. If the intercommunication interval is too long, a processor almost always computes distances of paths that go through local vertices. This may result in useless edge relaxations for those vertices whose shortest path goes through vertices owned by different processors. Thus, it is better for the exchange of relaxation requests not to be too infrequent so that vertices can reach their final distance sooner. However, updating too frequently may add significant overhead because of the initial cost of each communication.

Figures 4.10a and 4.10b show the execution of the DSMR algorithm of Figure 4.7 with an R-MAT graph [15] with $SCALE = 20, a = 0.55, b = 0.1, c = 0.1, d = 0.25, M = 8 \times N$ and $\Delta = 2^{16}$ on a shared memory machine with 40 cores as the value of D changes. Figure 4.10a shows the number of edge relaxations and Figure 4.10b shows the running times. Each line corresponds to a different number of processor. Numbers are computed by averaging the running times of the algorithm for 16 randomly chosen source vertices. All axes are logarithmic. The error bars represent the standard deviation. The marked points on the plot in Figure 4.10b show the best performing value of D.

The number of edge relaxations, increases with D, however, it is almost constant at first and then increases drastically. The left most points represent frequent global updates. On the other hand, a high D has the same effect as if there were no pipelining at all. With low D the numbers of edge relaxations is the same for all number of processors. As D increases, there is a factor of ~ 8 increase in the number of edge relaxations for all number of processes (except, of course, for the case of 1 processor where it remains constant). As it can be seen in Figure 4.10b, low values of D do not deliver the best performance because for these values, the algorithm sends many short messages. In fact, for low values, the algorithm is $6\times$ slower

than the optimal. On the other hand, high values of D slows down the algorithm because of the large number of edge relaxations. In fact, for high D values the algorithm is $\sim 8\times$ slower than the best execution times. This tracks the factor of 8 increase in the number of edge relaxations. The point at which D delivers the best performance is different for each number of processors. The best D for different graphs are different but the optimum is never at too low or too high values. This suggests that using the idea of delaying updates is effective and it increases the performance by multiple factors.

4.4 LTDP

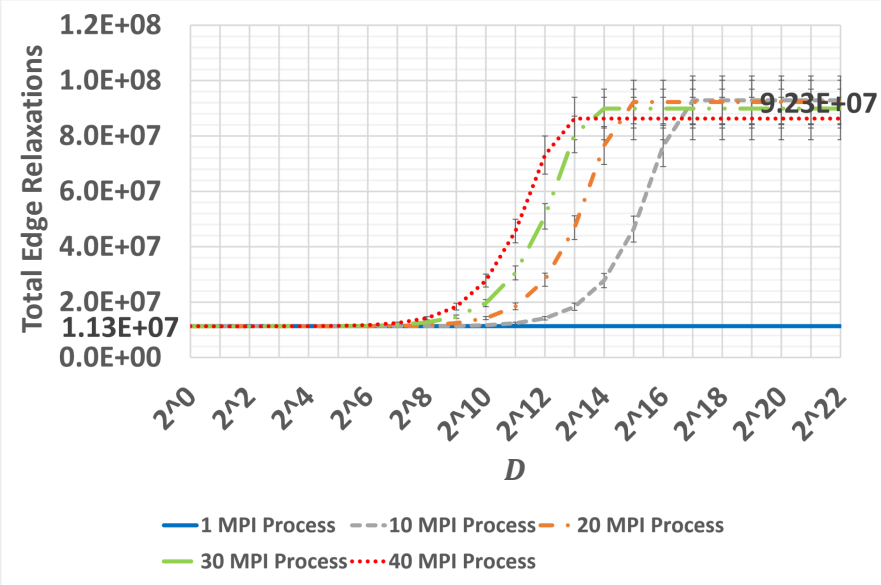
Chapter 3 introduced the Rank-1 method, an approach to parallelize Linear Tropical Dynamic Programming (LTDP) algorithms. In this section we will review the method briefly and show how it can be expressed in TLA.

As discussed in Section 3.2, an LTDP problem consists of a sequence of stages s_i with several values in each stage to be computed. Stage s_i can be computed from stage s_{i-1} by a matrix-vector multiplication in the tropical semiring: $\vec{s_i} = A_i \odot \vec{s}_{i-1}$. As a result, the computation can be represented in the following form:

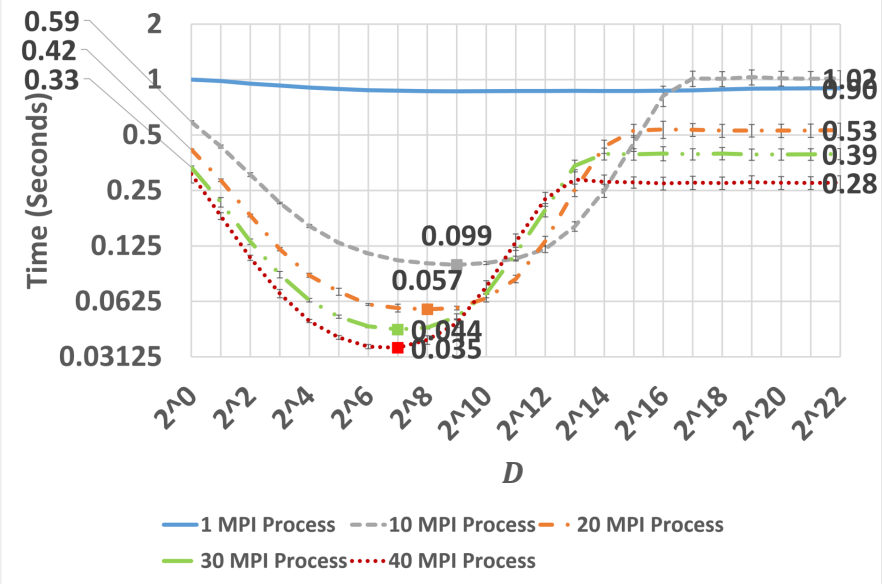
$$\vec{s_n} = A_n \odot A_{n-1} \ldots A_2 \odot A_1 \odot \vec{s_0}$$

where s_0 is the initial setup of the algorithm and A_i are matrices corresponding to the dependences between stages s_i and s_{i-1}. A sequential implementation of an LTDP algorithm works in n iterations as follows: at iteration i, $\vec{s_i} = A_i \odot \vec{s}_{i-1}$ is computed.

In the Rank-1 method, matrices of the computation are distributed in consecutive chunks. Assume that processor p owns matrices A_{l_p+1} to A_{r_p} and stages s_{l_p} to s_{r_p-1}. Note that $r_p = l_{p+1}$ if $p+1$ exists. All processors start computation of $A_{r_p} \ldots A_{l_p+1}\vec{z_p}$ where $\vec{z_p}$ is $\vec{s_0}$



(a) Number of checks



(b) Running time

Figure 4.10: D impact on the total number of relaxation and performance in the DSMR algorithm.

for the first processor and for others, it is a non-zero initialization. After each processor p finished its computation, it sends the locally computed $\vec{s}_{r_p}$, the last stage of its computation, to the processor $p+1$ since $p+1$ owns it (if $p+1$ exists). Then, each processor except the first one starts to fix up the incorrect computation until it reaches a stage where the fixed up stage is parallel to the previously computed stage. At this point, the rank has converged to 1 and the rest of the results in the corresponding processor is correct. Note that the fix-up algorithm either runs for $P-1$ iterations or all processors converge to rank 1 and the algorithm terminates. P is the total number of processors. A pseudo code of the Rank-1 method was presented in Figure 3.4.

Figure 4.11 presents the Rank-1 method in TLA notation. Array `s` in Line 1 is the array of stages and function `f1` distributes `s` block-wise to the processors as described above. Each stage is of size `width` as declared in Line 2. `A` matrices are similarly declared in Lines 3 and 4 and they are distributed block-wise as well. These matrices are of size `width` by `width`. Functions `f1` and `f3` are slightly different at the boundaries of the distribution. Again, processor `p` owns matrices $\mathtt{A}_{\mathtt{l_p+1}}$ to $\mathtt{A}_{\mathtt{r_p}}$ and stages $\mathtt{s}_{\mathtt{l_p}}$ to $\mathtt{s}_{\mathtt{r_p-1}}$. The only exception is that the last processor, `P-1`, owns the last stage $\mathtt{s}_{\mathtt{r_p}}$.

Values of `A` matrices are dependent on the LTDP algorithm and should be initialized accordingly in Line 5. Lines 6 and 7 set $\mathtt{l_p}$ and $\mathtt{r_p}$. Note that `p` is the processor id ranging from `0` to `P-1` where `P` is the total number of processors. Line 10 initializes `s[0]` which is dependent on the LTDP algorithm and only processor `0` uses this initialization. Others processors use a non-zero initialization in Line 12.

The loop in Line 13 is the local computation for each processor in the first phase. Note that `'<-'` is used in this loop and all of the computations are asynchronous. Since processor `p` does not own stage $\mathtt{s[r_p]}$ (which is the same as $\mathtt{s[l_{p+1}]}$), the computed values for the last iteration of the loop in Line 13 are stored locally. Therefore, $\mathtt{s[r_p]}$ should be sent to processor `p+1` if it exists. This is done in Line 15. Note that a `'='` is used which means that

the locally computed values on the right hand side should be communicated to the owner of `s[`r_p`]`. Similarly, in Line 16, `s[`l_p`]` is received from processor `p-1` if it exists. Since the processor running Line 16, `p`, owns the left hand side of the `'='` notation, it is expecting updates from others processors which computed `s[`l_p`]` and stored them locally.

Next, the fix up phase starts in the loop in Line 20. Note that the fix up phase runs for at most `P-1` iterations and processor `p` runs at most `p` iterations of the fix up loop which is controlled by variable `fixupIter` in Line 19 and the condition in Line 21. The computation in the loop in Line 22 is similar to the loop in Line 13 except the condition in Line 25 This condition checks whether the newly computed values of stage `s[i]` are parallel to the previously computed values of the same stage. Since parallel in the tropical semiring means that the corresponding values are different by a constant, the condition checks whether `newS` (new values of the stage) and `s[i]` (old values of the stage) are different by difference of the first elements, `s[i][0]-newS`. If the condition is true, that means that the corresponding processor has converged to rank 1 and can break from the rest of computation of the loop in Line 22 as indicated in Line 27.

The last stage of the computation in the loop in Line 22 is communicated similarly in Lines 29 to 31. However, processor `p = fixupIter+1` is not expecting any updates from processor `p-1` since processor `p` runs at most `p` iterations of the fix-up loop. This is controlled by the condition in Line 30.

The fix up phase in Line 20 continues for `P-1` iterations or until all processors converge to rank 1. The rest of the code is similar to other SSSP algorithms in TLA notation.

4.5 Conclusion

In this chapter, we presented TLA, a system for representing algorithms for amorphous problems using linear algebra. We have demonstrated the express-ability of our library with

```
1  s = CreateArray(n,f1); // Distributed array of stages
2  s[:] <- CreateArray(width,f2);
3  A = CreateArray(n,f3); // Distributed matrices of stages
4  A[:] <- Create2DSparse(width,width,f4);
5  A[:] <- InitializeA();  // Dependent on the LTDP algorithm instance
6  l_p = n/P*p;     // Starting index for p
7  r_p = n/P*(p+1); // Ending index for p
8  if (p == 0)
9    // Dependent on the initialization of the LTDP algorithm
10   s[0] <- IntializeS();
11 else
12   s[l_p] <- NonZeroInitialize(); // Some non-zero initialization
13 for (int i = l_p+1; i <= r_p; i++;)
14   s[i] <- A[i]*s[i-1];
15 s[r_p] = s[r_p]; // Send the last stage to p+1
16 s[l_p] = s[l_p]; // Receive the first stage from p-1
17
18 isDone = false; // Flag to specify if the rank converged to 1
19 fixupIter = 0;
20 do { // till convergence (fix up phase)
21   if (fixupIter < p){
22     for (int i = l_p+1; i <= r_p; i++){
23       newS <- A[i]*s[i-1];
24       // If the newly computed stage is parallel to the previous one
25       if (newS+(s[i][0]-newS[0]) == s[i]) {
26         isDone = true;
27         break; } }
28       s[i] <- newS;
29     s[r_p] = s[r_p]; // Send the last stage to p+1
30     if (p > fixupIter+1) // processor p=fixupIter+1 should not receive
31       s[l_p] = s[l_p]; // Receive the first stage from p-1
32   } else {
33     isDone = true; }
34   allDone <- isDone;
35   allDone += allDone; // Global communication for global convergence
36   fixupIter++;
37 } while (allDone < P);
```

Figure 4.11: The Rank-1 method in TLA notation.

implementations of several SSSP algorithms and the Rank-1 method. Our experiments have shown that by using the extensions in TLA, we achieve performance comparable to custom implementations of the same algorithms.

In the future, we intend to develop TLA in to a full featured library with more included semirings as well as support for user defined ones. We believe that as we implement more algorithms with TLA, we will find more extensions to the underlaying liner algebra. Extensions that we have considered included asynchronous messaging, control over updating, and support for dynamic graphs.

Chapter 5

Related Work

Trading computation for parallelism has been used widely in high-performance computation [5, 44, 79, 47, 61]. In this thesis, we have studied this approach for amorphous problems including SSSP problem and LTDP algorithms.

As for SSSP problem, several algorithms have been developed as discussed in Chapter 2. Bellman-Ford [10], Chaotic Relaxation [17], and Dijkstra's algorithm [24] are among the first algorithms. The problem with parallelizing these algorithms is that Dijkstra's algorithm is inherently sequential but work efficient, while Chaotic Relaxation and Bellman-Ford are parallelizable but work inefficient. Δ-Stepping is a trade-off between Chaotic Relaxation and Dijkstra's algorithm. By setting the right Δ, the algorithm can have enough parallelism without increasing the total amount of work significantly. However, as presented in Chapter 2, Δ-Stepping does not perform well with scale-free networks.

There are multiple implementations of Δ-Stepping available. Chakaravarthy et al. [14] studied parallelization of SSSP on large clusters. In this thesis, we compared our results with theirs using the values reported in [14]. To make an accurate comparison, we used the same graph generation algorithm (through private communication with the authors) they used and the same machine. SSSP from the Elixir [70] benchmark is a shared-memory implementation

of Δ-Stepping that we have run and compared with. SSSP from Parallel Boost Graph Library [25] is an implementation of Dijkstra and Δ-Stepping for distributed-memory systems. We found PBGL slower than the implementations considered in Chapter 2 and because of that we did not show results for it. There is also an implementation of Δ-Stepping on Cray MTA-2 by Madduri and Bader [53]. However, since the code was written for that machine, we could not do a comparison. Finally, there are implementations of Chaotic Relaxation in CombBLAS, GraphLab and PowerGraph [13, 52, 31] but this algorithm performs too much unnecessary relaxation [54]. The DSMR algorithm presented in Chapter 2 is the fastest SSSP algorithm for scale-free networks as our results showed in Section 2.7.

The other set of amorphous problems that we studied are Linear Tropical Dynamic Programming (LTDP) algorithms as discussed in Chapter 3. There has been much prior work in parallelizing dynamic programming. Predominantly, implementations use wavefront parallelism to parallelize within a stage. In contrast, the method in this thesis exploits parallelism across stages in addition to wavefront parallelism. For instance, Martins et al. build a message passing based implementation of sequence alignment dynamic programs (i.e., Smith-Waterman and Needleman-Wunsch) using wavefront parallelism [57]. Our baseline for Needleman-Wunsch builds on this work.

Stivala et al. use an alternate strategy for parallelizing dynamic programming. They use a "top-down" approach that solves the dynamic programming problem by recursively solving the subproblems in parallel. To avoid redundant solutions to the same subproblem, they use a lock-free data structure that memorizes the result of the subproblems. This shared data structure makes it difficult to parallelize across multiple machines.

There is also a large body of theoretical work analyzing the parallel complexity of dynamic programming. Valiant et al. [77] show that straight-line programs that compute polynomials in a field, which includes classical dynamic programming, belong to **NC**, the class of asymptotically efficiently parallelizable problems. Subsequent work [3, 29] has im-

proved both the time complexity and processor complexity of this result. These studies view dynamic programming as finding a shortest path in an appropriate grid graph, computing all-pairs shortest paths in partitions of the graph in parallel, and combining the results from each partition efficiently. They differ in how the structure of the underlying graph is used for efficiency. While it is not clear if these asymptotically efficient algorithms lead to efficient implementation, using the structure of the underlying computation for parallel efficiency was an inspiration for the work in this thesis.

There are many dynamic programming problem-specific implementations. For example, much like we did in this thesis, LCS can exploit bit-parallelism (e.g., [1, 19, 37]). And, Aluru et al. describe a prefix-sum approach to LCS [2] which exploits the fact that LCS only uses binary values in its recurrence equation.

Smith-Waterman has been studied extensively due to its importance to DNA sequencing. The Rank-1 method presented in this thesis uses Farrar's SIMD implementation [26] on multi-core, however, prior work has also investigated other hardware (e.g., GPU [50] and FPGA [49]).

Due to its importance in telecommunications, there has been lots of work on parallel Viterbi decoding. Because this algorithm is often implemented in hardware, one simple approach to increase performance is to pipeline via systolic arrays (i.e. to get good throughput) and increase clock frequency (i.e., to get good latency) [27]. The closest approach to us is Fettweis and Meyr who frame Viterbi as linear operations on the tropical semiring and utilize the associativity of matrix-matrix multiplications. However, they suffer linear overheads of this approach which is hidden by adding more hardware.

Chapter 4 introduced Tiled Linear Algebra (TLA) and explained what are the necessary features to express different amorphous problems. Our work is most similar to the combinatorial BLAS [13] which expresses graph algorithms with BLAS-like routines. The difference between TLA and CombBLAS is that they handle the parallelism entirely under

the abstraction of linear algebra but we make the parallelism and distribution explicit. In TLA, the programmer can write the same type of program that was expressible in the combinatorial BLAS since we are both based on the linear algebra but in TLA the programmer can directly control distribution, communication, and grain size. These features allow an expert programmer to take code and tune it to take advantage of hardware and algorithmic features that are not exposed in a system.

Another model of parallel graph computation is the vertex programming model. This model is used by PowerGraph [31], Pregl [56], and GraphLab [52]. In this model, the programmer thinks of graph algorithms as running in parallel and interaction on the edges between vertices. Also, the very fine grained work is aggregated by the runtime system and not under programmer control. It also lacks the ability to restrict computation when available under programmer control. These limitations would prevent expressing algorithms such as Δ-Stepping or DSMR where the work does flow directly from neighboring vertices.

There are several other frameworks for parallelizing sparse numerical routines. The most widely used one is PETSc [7] which is used for partial differential equations (PDE). Although PETSc provides libraries for some parallel graph partitioning algorithms [4] such as parMETIS [41], it is mainly used for parallelizing numerical PDEs. Charm++ [40] is another parallel framework for amorphous problems which is a dataflow programming model. The model fits very well for programs with load imbalance issue such as NAMD [68].

Chapter 6

Future Work

Interest in parallelizing amorphous problems for large systems is rapidly growing due to the increase in the size of real-world problems, namely, the graph algorithms. Also, future machines are becoming more complex and as a result, writing efficient parallel program is dependent on a careful use of communications. Therefore, we believe communication avoiding algorithms for amorphous problems is not a passing fad and future problems require similar approaches to have scalable algorithms. Below is a list of directions that can be worked on in the future regarding the problems presented in this thesis:

Single Source Shortest Path Problem:

DSMR algorithm presented in Chapter 2 works well for scale-free networks as our result in Section 2.7 showed how close DMSR's performance is to the oracle algorithm's. On the other hand, for non-scale-free networks, neither DSMR nor Δ-Stepping perform well as our results for US Roads network showed. In fact, the oracle algorithm is several times faster than DSMR and Δ-Stepping. The reason for this miss in the performance is that SSSP algorithms for roads network type of graphs have a numerous parallel steps with very limited amount of parallelism within each step. Therefore, there is room for improvement in parallelizing SSSP. In fact, such instances of SSSP fits very well with LTDP model but it is unclear what

the steps are before executing the algorithm. We believe that a similar rank-1 method can be designed to parallelize SSSP for such cases.

Another line of research could focus on understanding how the subgraph extraction and pruning techniques impact large real-world graphs with real weights. Right now it is unclear what the edge weights would represent in social networks and we artificially put weights on the edges. If heavy edge weights are common in real-world graphs, our optimizing techniques would impact them significantly.

Parallelizing Dynamic Programming Algorithms:

Rank convergence is a novel property that is observed in several problems. One of the very important LTDP problems is the voice recognition. Despite of decades of research in this area, this problems remains inherently sequential and difficult to parallelize. We have observed rank convergence in this problem, but it takes a long time to converge to rank-1. As a result, our rank-1 method does not work well for this problem. On the other hand, the Delta method requires the dependences to be local but the voice recognition underlying graph is amorphous which prevents us from using the Delta method. Therefore, we believe that there is room for developing a new parallel method for voice recognition using the rank convergence property.

An important algorithm with the rank convergence property is the Cuthill-McKee [20] algorithm. This graph-based algorithm reduces the bandwidth of a sparse matrix. Because of the tight dependences in the algorithm, its parallelization is challenging. Surprisingly, starting with two different initial orders, after a few steps in the algorithm, the same output will be generated. We believe that by using this property, we can efficiently parallelize this algorithm.

Parallel Framework for Amorphous Problems:

The TLA notation presented in this thesis is a design which requires an implementation. However, before implementing TLA, we need to understand more about the notation and

try to find the necessary abstractions in the language. To do so, we have to study other amorphous problems and express them in linear algebra notation.

On the other hand, there are several challenges in implementing TLA. For example, there are numerous ways of representing a sparse matrix and they differ in performance for different algorithms. The other challenge is performing linear algebra operations on sparse matrices. This problem has been widely studied and many approaches for different purposes are proposed. This variability needs to be controlled carefully in the notation without exposing too much details to the programmer.

Chapter 7

Conclusion

In this thesis, we studied new communication avoiding parallelization methods for amorphous problems. These amorphous problems include Single-Source Shortest Problem (SSSP) and a wide range of dynamic programming algorithms, namely, Linear Tropical Dynamic Programming (LTDP) problems. For each problem, we showed what are the limitation of existing parallelizing approaches and how the new parallelizing approaches overcome these limitations. Finally, we showed that all of the parallel approaches presented in this thesis are closely connected to linear algebra and as a result, we designed Tiled Linear Algebra (TLA) to express all of them.

The key contributions of this thesis are:

- In Chapter 2, we presented Dijkstra Strip Mined Relaxation (DSMR) algorithm for SSSP. In Section 2.3, we introduced degree-distance distribution and overhead distributions and explained why Δ-Stepping algorithm does not scale well with scale-free networks.

- In Section 2.4 and 2.5, we showed that heavy edges are very unlikely to be used in the solution for SSSP in scale-free networks. As a result, we designed Subgraph Extraction and Pruning techniques to improve performance of SSSP algorithms.

- In Chapter 3, we presented a novel approach to parallelize LTDP problems that relies on rank convergence in tropical semiring. In Section 3.3, we proposed the Rank-1 method which parallelizes LTDP algorithms with minimal changes and a promising speed-up.

- In Section 3.4, we introduced the Delta method which similar to the Rank-1 method, uses rank convergence for parallelization but it does not solely rely on rank-1 convergence. The Delta method provides better speed-ups than the Rank-1 method, however, it requires major changes in the LTDP algorithm.

- In Chapter 4, we showed the connection between linear algebra routines and amorphous problems presented in this thesis. Then, we discussed the necessary features for a linear algebra based parallel framework name TLA to enable expressing all of the parallel approaches presented in this thesis.

Bibliography

[1] L. Allison and T. I. Dix. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, 23(6):305–310, Dec. 1986.

[2] S. Aluru, N. Futamura, and K. Mehrotra. Parallel biological sequence comparison using prefix computations. *J. Parallel Distrib. Comput.*, 63(3):264–272, 2003.

[3] A. Apostolico, M. J. Atallah, L. L. Larmore, and S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988, 1990.

[4] Argonne National Laboratory. http://www.mcs.anl.gov/petsc/index.html.

[5] J. Arnal, V. MigallÃşn, and J. PenadÃl's. Synchronous and asynchronous parallel algorithms with overlap for almost linear systems. In V. HernÃąndez, J. Palma, and J. Dongarra, editors, *Vector and Parallel Processing âĂŞ VECPARâĂŹ98*, volume 1573 of *Lecture Notes in Computer Science*, pages 142–155. Springer Berlin Heidelberg, 1999.

[6] D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing*, HiPC'05, pages 465–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[7] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[8] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[9] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[10] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[11] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–14, March 2007.

[12] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM.

[13] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, Nov. 2011.

[14] V. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 889–901, May 2014.

[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *In SDM*, 2004.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[17] D. Chazan and W. Miranker. Chaotic Relaxation. *Linear Algebra and Its Applications*, 2(7):199–222, 1969.

[18] R. Cohen and S. Havlin. Scale-free networks are ultrasmall. *Phys. Rev. Lett.*, 90:058701, Feb 2003.

[19] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6):279 – 285, 2001.

[20] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[21] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. Implementation challenge for shortest paths. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*, pages 1–99. Springer US, 2008.

[22] S. Deorowicz. Bit-parallel algorithm for the constrained longest common subsequence problem. *Fundamenta Informaticae*, 99(4):409–433, 2010.

[23] M. Develin, F. Santos, and B. Sturmfels. On the rank of a tropical matrix. *Combinatorial and computational geometry*, 52:213–242, 2005.

[24] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[25] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-source shortest paths with the parallel boost graph library.

[26] M. Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2007.

[27] G. Fettweis and H. Meyr. Parallel Viterbi algorithm implementation: breaking the ACS-bottleneck. *IEEE Transactions on Communications*, 37(8):785–790, 1989.

[28] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, Mar. 1977.

[29] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than O(1) dependency. *Journal of Parallel and Distributed Computing*, 21(2):213–222, 1994.

[30] A. Goldberg. Shortest path algorithms: Engineering aspects. In *In Proc. ESAAC âĂŹ01, Lecture Notes in Computer Science*, pages 502–513. Springer-Verlag, 2001.

[31] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[32] Graph 500. http://www.graph500.org/.

[33] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[34] R. GuimerÃă, S. Mossa, A. Turtschi, and L. A. N. Amaral. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proceedings of the National Academy of Sciences*, 102(22):7794–7799, 2005.

[35] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, Dec. 1986.

[36] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[37] H. Hyyro. Bit-parallel LCS-length computation revisited. In *In Proc. 15th Australasian Workshop on Combinatorial Algorithms*, pages 16–27, 2004.

[38] Intel C/C++ Compiler. http://software.intel.com/en-us/c-compilers.

[39] H. Jeong, S. P. Mason, A. L. Barabasi, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, May 2001.

[40] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[41] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, Jan. 1998.

[42] J. Kepner and J. Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[43] V. Krebs. Mapping networks of terrorist cells. *CONNECTIONS*, 24(3):43–52, 2002.

[44] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

[45] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[46] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[47] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, Oct. 1980.

[48] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, Oct. 1980.

[49] I. Li, W. Shum, and K. Truong. 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). *BMC Bioinformatics*, 8(1):1–7, 2007.

[50] L. Ligowski and W. Rudnicki. An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–8, 2009.

[51] F. Liljeros, C. R. Edling, L. A. N. Amaral, H. E. Stanley, and Y. Åberg. The Web of Human Sexual Contacts. *Nature*, 411:907–908, 2001.

[52] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[53] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances, 2007.

[54] S. Maleki, G. Evans, and D. Padua. Tiled linear algebra a system for parallel graph algorithms. In J. Brodman and P. Tu, editors, *Languages and Compilers for Parallel Computing*, volume 8967 of *Lecture Notes in Computer Science*, pages 116–130. Springer International Publishing, 2015.

[55] S. Maleki, M. Musuvathi, and T. Mytkowicz. Parallelizing dynamic programming through rank convergence. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 219–232, New York, NY, USA, 2014. ACM.

[56] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[57] W. S. Martins, J. B. D. Cuvillo, F. J. Useche, K. B. Theobald, and G. Gao. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *In Pacific Symposium on Biocomputing*, pages 311–322, 2001.

[58] T. Mattson, D. A. Bader, J. W. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. R. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. E. Leiserson, A. Lumsdaine, D. A. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. Standards for graph algorithm primitives. In *HPEC*, pages 1–2, 2013.

[59] Message Passing Interface Forum. http://www.mpi-forum.org/index.html.

[60] U. Meyer. Average-case complexity of single-source shortest-paths algorithms: Lower and upper bounds. *J. Algorithms*, 48(1):91–134, Aug. 2003.

[61] U. Meyer and P. Sanders. Delta-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, Oct. 2003.

[62] Y. Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1971.

[63] MVAPICH: MPI over InfiniBand, http://mvapich.cse.ohio-state.edu/. 2013.

[64] National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/. 2013.

[65] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[66] G. Palla, I. J. Farkas, P. Pollner, I. DerÃľnyi, and T. Vicsek. Fundamental statistical features and self-similar properties of tagged networks. *New Journal of Physics*, page 123026.

[67] W. W. Peterson and E. J. Weldon. *Error-Correcting Codes*. MIT Press: Cambridge, Mass, 1972.

[68] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten. Scalable molecular dynamics with NAMD. *J Comput Chem*, 26(16):1781–1802, Dec. 2005.

[69] G. Pok, H. S. Shon, K.-A. Kim, and K. H. Ryu. Notice of retraction small-world network properties of protein complexes: Node centrality and community structure. In *Bioinformatics and Biomedical Engineering, (iCBBE) 2011 5th International Conference on*, pages 1–4, May 2011.

[70] D. Prountzos, R. Manevich, and K. Pingali. Elixir: A system for synthesizing concurrent graph programs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '12, 2012.

[71] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, Special issue on "Program Generation, Optimization, and Adaptation"*, 93:232–275, 2005.

[72] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[73] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[74] Texas Advanced Computing Center, http://www.tacc.utexas.edu/resources/hpc. *Stampede: Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors*.

[75] 2013. http://www.top500.org.

[76] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[77] L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM Journal of Computing*, 12(4):641–644, 1983.

[78] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[79] T. Yang. Trading computation for communication: Distributed stochastic dual coordinate ascent. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 629–637. Curran Associates, Inc., 2013.