The Pennsylvania State University
The Graduate School
Department of Computer Science and Engineering

SCALABLE GRAPH AND MESH ALGORITHMS ON DISTRIBUTED-MEMORY SYSTEMS

A Dissertation in
Computer Science and Engineering
by
Thap Panitanarak

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2017

The dissertation of Thap Panitanarak was reviewed and approved* by the following:

Kamesh Madduri
Assistant Professor of Computer Science and Engineering
Dissertation Co-Advisor, Co-Chair of Committee

Suzanne M. Shontz
Adjunct Associate Professor of Computer Science and Engineering
Dissertation Co-Advisor, Co-Chair of Committee

Mahmut T. Kandemir
Professor of Computer Science and Engineering

Keefe B. Manning
Associate Professor of Biomedical Engineering and Surgery

Chita R. Das
Distinguished Professor and Department Head of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Big datasets are now becoming a standard quantity in large-scale data analysis; they involve social and information network, and scientific mesh computations. These datasets are commonly stored and processed across multiple machines due to limited capabilities (such as memory and CPU) of single machines. However, many available analysis tools are still lacking in terms of an ability to fully utilize existing distributed-memory architectures. As these datasets are usually processed and analyzed in the form of graphs or meshes, we propose scalable and efficient approaches for graph and mesh computations for distributed-memory systems in this dissertation. Although graph and mesh computations are closely related regarding their parallelization approaches, some of their unique characteristics still need to be addressed separately. Thus, we organize the dissertation into two parts. The first part is for distributed graph computations, and the second part is for distributed mesh computations.

In the first part of the dissertation, we focus on graph computations. First, we study a problem of Single-Source Shortest Path (SSSP) by analyzing and evaluating three well-known SSSP algorithms, i.e, Dijkstra's, Bellman-Ford, and Δ-stepping algorithms. We implement these algorithms to run on distributed-memory systems based on a bulk synchronous parallel model. Their performances are evaluated and compared. Next, we propose our SSSP algorithm by combining advantages of these SSSP algorithms and utilizing a two-dimensional (2D) graph layout for our graph data structures. Then, we extend our study of the 2D graph data structures and optimization approaches to other well-known graph algorithms including breadth-first search, approximate diameter, connected components, and PageRank on various real-world graphs. Our objective is to implement an efficient graph framework for distributed-memory systems that works efficiently for many graph algorithms on various graph types. Finally, we propose graph coloring algorithms that are scalable and can be efficiently used for both graph and mesh applications.

In the second part of the dissertation, we focus on parallel mesh computations

on distributed-memory systems. First, we propose a domain decomposition method for 2D parallel mesh generation based on the MeTis partitioner with angle improvements. Our method is fast and gives good subdomain quality in terms of subdomain angles and mesh quality. Next, we propose a general-purpose parallel mesh warping method based on a parallel formulation of a sequential, log barrier-based mesh warping algorithm called LBWARP. Our parallel algorithm utilizes a modified distributed graph data structure with a vertex ghosting technique resulting in an efficient mesh warping algorithm which employs minimal communication. Since the algorithm needs to solve a sparse linear system with three right-hand sides (for 3D meshes), i.e., are each for the final x-, y- and z-coordinates in the deformed meshes, we also provide three parallel sparse linear solvers that support multiple right-hand sides for users to choose from based on the size of the problem and the number of available cores. These solvers further improve the overall performance of the algorithm, especially when a sequence of multiple deformations is required.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my sincere appreciation to my co-advisors Dr. Kamesh Madduri and Dr. Suzanne Shontz. When I first started my Ph.D studies, I was very new to conducting research. I was really fortunate to work with my two advisors. They helped me a lot with their guidance and encouragement. Without their help, I could not have completed my dissertation. I appreciate the advice and guidance of my committee members, Drs. Mahmut Kandemir and Keefe Manning, which helped strengthen my dissertation. I would like to thank my friends: Shankar Prasad Sastry, Nicholas Voshell, Jibum Kim, Jeonghyung Park, and George Slota. They helped me greatly in the beginning of my research, and my Ph.D work benefited considerably from discussions with them and their feedback. My gratitude also goes to our Scalable Scientific Computing Lab members: Shankar Prasad Sastry, Nicholas Voshell, Joshua Booth, Jibum Kim, Jeonghyung Park, George Slota, Humayan Kabir, and Paul Philip. I also would like to thank my family for their patience, love and seamless support during my Ph.D study. Finally, I would like to acknowledge my sponsors for their support during my Ph.D. My research was funded by the Royal Thai Government and through instrumentation funded by the NSF through grant OCI-0821527.

Chapter 1
Introduction

Big datasets are now increasingly common in large-scale data analysis due to the rapid growth of research related to social and informational networks (such as online social networks, electronic communications, and e-commerce) and scientific mesh computations that often involve large-scale and/or high-accuracy simulations. Examples of large data relating online and web services includes the reports of over 1.71 billion monthly active Facebook users (June 2016) [124], 188 million monthly visitors on Amazon's websites (September 2015) [123], and 75 million subscribers for Netflix (January 2016) [125]. For high-accuracy scientific simulations, there have been recent reports and research on large-scale mesh simulations, such as the M8 earthquake simulation involving 436 billion mesh vertices [33], aerodynamic wing design simulations involving 92 billion mesh elements [101], and computational fluid dynamic simulations involving 78 billion mesh elements [126].

As most large data is heterogeneous in nature, it is commonly represented in the form of a graph with relations depicted among entities. Facebook's friend graph, for example, is a graph in which each vertex represents a user, and each edge represents a friend relation between two vertices. These graph representations help one to visualize and understand the structures of large data. Furthermore, more detailed information and knowledge can be extracted using graph analytics, such as recommendations (e.g., friend recommendations in Facebook and product/movie recommendations in Amazon), prediction and decision (e.g., threat determination in CyberSecurity and medicine), and optimization (e.g., optimal routes for traffic network and optimal layouts for electronics).

As for meshes, they have been commonly used in computer modeling and simulation in order to discretize the geometric domain of interest. In some applications,

especially for engineering applications for which meshes are required to accurately represent the geometric domain, they can become very complex in terms of the number of elements and the element types. For example, a simulation of the aerodynamics of a car or an airplane requires very detailed and good quality meshes especially on/near the boundary surfaces, interesting features, high curvature areas and time-dependent areas of the model. Normally not only is the vehicle represented by a mesh, but the air around it also needs to be represented by a mesh to capture the dynamic flow during the simulation when a finite element method is employed.

With tremendous size of the data, relating graphs and meshes also becomes more complicated, and overwhelms the capacity of single processor machines. Thus, processing and storing this information is more practical with high performance computing systems, such as multicore and multithreaded architectures and distributed-memory systems. However, distributed-memory systems are more appealing due to higher potential for the scalability, i.e., they can provide very large number of cores and memory. Furthermore, the emerging cloud computing in the past few years provided by Amazon, Google, and Salesforce.com gives an opportunity to the public to be able to access abundant computing resources and services. Some services provided by the cloud can be viewed as a specialized form of distributed-memory systems, such as Amazon EC2. Although the hardware and networking are not up to the standard of many distributed-memory HPC systems, Amazon EC2 is more cost efficient for many users and researchers.

For distributed memory systems, each processor has its own local memory which is not shared among other processors. Thus, each processor can only process the data on its own memory. Accessing non-local data requires explicit communication, usually in the form of messages. Since each processor has its own memory, when the number of processors increases, the size of the memory also increases. Furthermore, memory accessing is less likely a bottleneck, as the memory space can only be accessed by the owning processor. This architecture is also widely used because it is very cost effective, as it can be built from commodity processors and networking.

There are several difficulties and challenges associated with utilizing such systems to their full potential. First, algorithms need to be re-implemented specifically for distributed-memory machines since sequential algorithms do not efficiently run on distributed setups, as they do not utilize all available processors. One of the main differences between the two environments is the presence of interprocessor

communication and data sharing. While single machines do not require any communication in processing their data, it is one of major considerations for distributed-memory systems. Furthermore, real-world, large-scale data is difficult to process. Even though computational techniques are important for parallel graph computations, data structures are also equally important as they affect the overall performance. For example, different graph types can lead to totally different results on the same algorithm, specifically for the real-world graphs due to their highly irregular structures as most graph algorithms are data-driven. Furthermore, many real-world graphs usually have highly skewed power-law degree distributions that are challenging to process efficiently in large-scale parallel environments. These lead to difficulties in parallelizing graph computations since these graphs are hard to partition efficiently [1] and distribute evenly, and thus, result in performance degradation from poor load-balancing and data locality issues. In contrast to parallel mesh processing which is usually computationally-intensive, one needs to determine the tradeoff between data duplication and communication needed to achieve the optimal performance.

Several researchers have focused on efficiently implementing parallel graph and mesh frameworks for distributed-memory systems. Parallel Boost Graph [55] is among the early attempts that extend the well-known (sequential) Boost Graph Library. However, the focus of the implementation is on generic programming that provides high flexibility and customization to support various graph data structures and algorithms. As a result, the framework does not scale well for large distributed systems. Distributed GraphLab [79] is a distributed graph framework based on an asynchronous parallel shared-memory framework, i.e., GraphLab [80]. Distributed GraphLab provides a graph abstraction similar to GraphLab by introducing ghost vertices, which are nonlocal vertices at the partition boundary, and distributed read/write lock systems. Another variant of Distributed GraphLab, PowerGraph, is an extension framework that is more efficient when processing power law graphs, (i.e., graphs that contain few vertices with very high degree while the majority of vertices have very low degree, which leads to a severe load imbalance). The idea of PowerGraph is to utilize a vertex-cut technique to handle high degree vertices.

In this dissertation, we focus on large-scale graph and mesh computations for distributed-memory systems. We parallelize and optimize not only the algorithms, but also the underlying data structures and representations which can easily extend

to other similar algorithms. Distributed computations for graphs and meshes share several key considerations, such as the concept of data distribution, inter-processor communication, and efficient concurrent computation. An overview of these considerations is given in Chapter 2. Even though graphs and meshes are related to some extent, they are still quite different in terms of structures and characteristics. Thus, the dissertation is organized in two major parts, one for graph computations (Chapters 3-6), and the other for mesh computations (Chapter 7-8).

In Chapter 3, performance analysis and evaluation of three well-known single-source shortest path (SSSP) algorithms are discussed. These algorithms include Dial's algorithm (i.e., a Dijkstra variation that is more suitable for parallel implementation), Bellman-Ford, and Δ-stepping, on distributed-memory systems. We implement the distributed versions of these three algorithms based on the bulk synchronous parallel model. The analysis results include weak and strong scaling and the effect of weight distributions of a graph. Furthermore, we give more detailed results of the Δ-stepping algorithm since it performs the best among those three. We show the impact of Δ values on the algorithm performance. Finally, we provide some performance comparisons of Δ-stepping to some well-known graph libraries on single-core and single-node (16 core) environments.

In Chapter 4, we generalize the concept of the 2D layout for graph computations and apply to other graph algorithms such as breadth-first search, approximate diameter, connected components, Bellman-Ford and PageRank. We show performance results and analysis of algorithms on various large-scale graphs. The performance results between the algorithms with the 1D and 2D graph layouts are given. Finally, we compare the performance results of our implementations with those in other graph frameworks.

In Chapter 5, we implement novel SSSP algorithms based on the analysis from Chapter 3 combined with the 2D graph layout from Chapter 4. Even though SSSP algorithms with the 2D graph layout involves more communication steps than SSSP algorithms that use the common vertex distribution approach (or the 1D graph layout), the communication dimension is decreased, and additional communication usually has low overhead. Furthermore, adjacencies of high-degree vertices are now distributed across multiple processors. Thus, overall load balancing is improved. More detailed discussion and analysis are provided in this chapter.

In Chapter 6, we combine some advantages of some recent graph coloring

algorithms with the 2D graph layout to further improve the overall performance of the algorithms. We implement distributed graph coloring algorithms using similar approaches as the graph algorithms implemented in Chapter 5. Finally, we evaluate the effect of 2D graph layout on various types of graphs and meshes.

In Chapter 7, we introduce a domain decomposition technique for 2D parallel mesh generation. Our algorithm is based on mesh partitioning using MeTis. It is very fast and gives good subdomain quality in terms of the subdomain angles which are guaranteed to be greater than 60°. Meshes generated on our subdomains provide good mesh quality, i.e., comparable to a static geometric medial axis domain decomposition algorithm called MADD which is more computationally-intensive.

In Chapter 8, we propose a general-purpose parallel mesh warping algorithm for distributed-memory machines. Our algorithm is based on a parallel formulation of a serial, log barrier-based mesh warping algorithm called LBWARP. Most steps of our algorithm can be implemented using an embarrassingly parallel approach that can be achieved from our data distribution combined with a vertex ghosting technique. Thus, our algorithm has a very low communication overhead and demonstrates good strong scalability. We also provide three parallel sparse linear solvers that support multiple right-hand sides for users to choose from based on the size of the problem and the number of available cores. We show that these solvers can further increase the overall performance of the algorithm.

The outline of the dissertation is shown in Fig 1.1. It presents the two parts of the dissertation and their chapters. The arrows in the chart indicate the dependency between chapters.

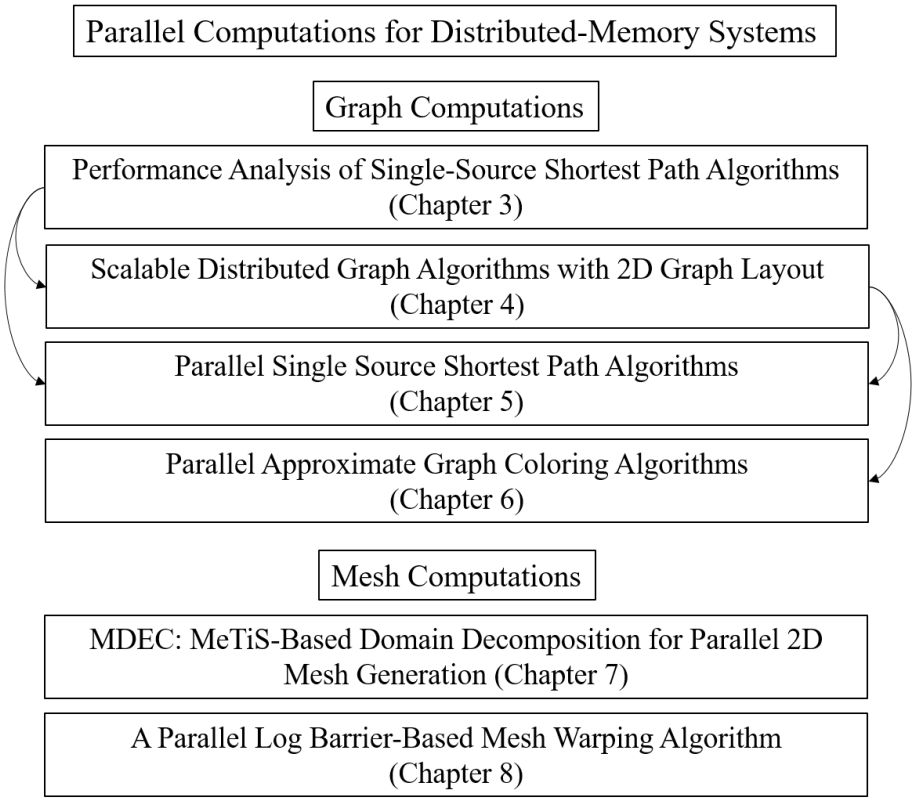


Figure 1.1. The dissertation outline.

Chapter 2
Considerations in Parallel Computations on Distributed-Memory Systems

The key aim for parallel computations is to solve usually large or time-consuming problems as fast and as efficiently as possible in order to minimize the compute cost and to achieve optimal throughput. The parallelization process usually starts with decomposing a large problem into smaller subproblems or tasks, and then processing them simultaneously on multiple processors while making sure that a final result is the same as the result obtained from processing the original problem with a single processor. Ideally, the performance goal of parallel computations is to run a code p times faster on p processors than when running the code in serial. However, due to various factors during the parallelization process, it is nearly impossible to achieve this ideal performance. One of the main reasons is due to the overhead during the parallelization from the decomposition, communication, and synchronization. Moreover, there are also some difficulties as many problems are nontrivial to parallelize due to their irregularity, nonlocality of data, and subtask dependencies. In this chapter, we discuss some considerations that are crucial for performing parallel computations on distributed-memory systems.

2.1 Data Structure Decomposition

One of the most important questions for parallel computations is how the problem is partitioned into multiple subproblems that can then be distributed and processed simultaneously. Domain decomposition is the first step for parallel computations and is very challenging; it is also the most important step. Since the parallel performance is subject to the worst performance among processors, one of the decomposition objectives is to partition the problem into subproblems that require approximately the same amount of computation and communication. However, it is not easy to achieve this goal, as there are numerous factors that can affect the parallel performance.

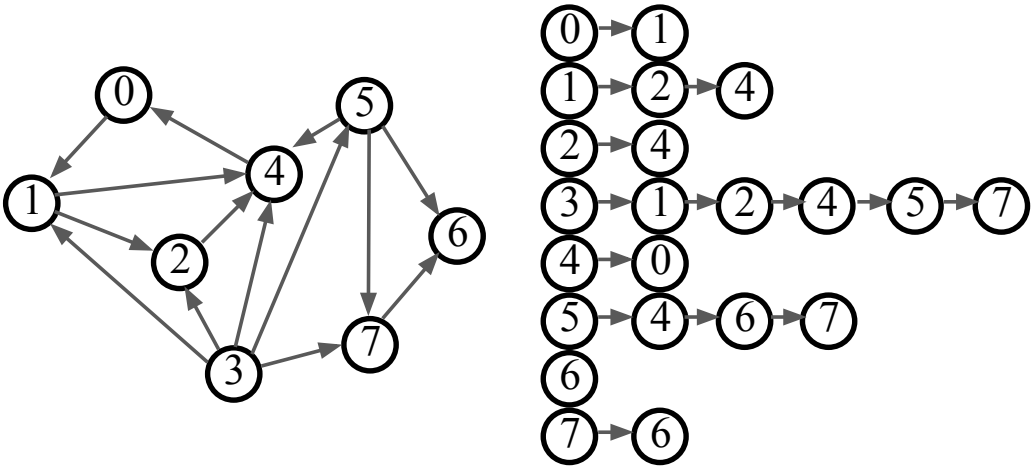


Figure 2.1. (a) A directed graph and (b) its equivalent adjacency list.

A common partitioning approach that is used in most distributed graph algorithms is to partition the graph vertices equally. This approach is usually done by applying methods based on hash partitioning of the vertex ID or other partitioning algorithms [69] that minimize the number of edges between partitions. The partitioned vertices are then distributed along with their outgoing edges to their corresponding partitions. For example, consider a directed graph and its equivalent adjacency list shown in Fig. 2.1 (a) and (b), respectively. To decompose the graph into four partitions, the decomposition starts with partitioning vertices using their ID. Thus, two vertices are assigned for each partition. After that, all outgoing

edges of the vertices are distributed to the same partition. Fig. 2.2 shows the final partition of the same graph and its distributed adjacency list after applying the decomposition. This partitioning approach is widely used in many well-known graph algorithms [10, 24, 84, 93], libraries [45, 55, 115] and frameworks [27, 51, 70, 79, 136].

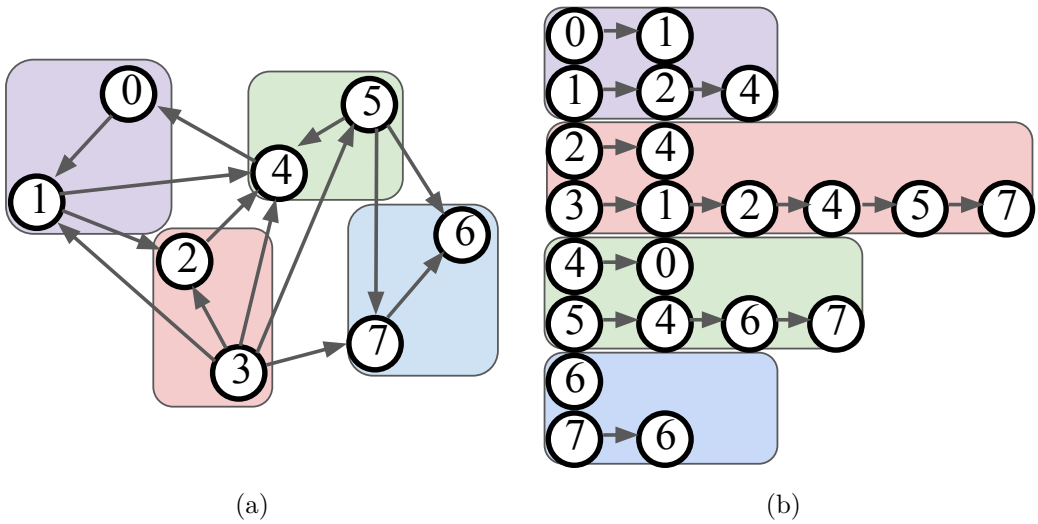


Figure 2.2. (a) A four-partition of the directed graph from Fig. 2.1 and (b) its distributed adjacency list.

2.2 Communication and Synchronization

Communication and synchronization are two of the major considerations in parallel computations, as most parallel programs require processors to share data with others to progress through the computations and to make sure that all processors are in the same state. Within distributed-memory systems, communication among processors requires explicit messages and initialization between the sender and the receiver. This inter-processor communication creates overhead, as machine resources are also utilized for initiating and sending/receiving messages. Synchronization is required at the end of the communication to guarantee the communication succeeds. It also helps determine the global state of the program.

Many graph algorithms [10, 24, 84, 93] often aggregate messages before initiating communication to decrease communication start-up time that can be significant if the number of messages is large. Message aggregation is often done within each

iteration by combining small messages that need to be sent to the same processor into one larger message. For example, in Fig. 2.2, if the algorithm is required to send some vertex information from each vertex to its adjacent vertices, the processor that owns vertices 2 and 3 needs to send three messages to the processor that owns vertices 4 and 5. However, with the aggregation, only one message aggregated from the three separate messages is sent in this step. Furthermore, collective communication (e.g., all-to-all, all-gather, etcetera) is often used in parallel algorithms to simplify the algorithm. Thus, it improves maintainability of the program while also increasing the performance of parallel algorithms.

Normally, there are two synchronization models used by distributed computations, i.e., synchronous and asynchronous processing. With synchronous programming, all processors work on the same (local) task either on computation or communication and wait (or are blocked by a barrier) until all processors finish before they can move onto the next task. This parallel model has been used in many algorithms, not only does it simplify the algorithms, but it also guarantees sequential program execution. However, sometimes, this model can lead to a serious bottleneck from the synchronization overhead, e.g., most processors are idle while waiting for the few others to finish their tasks. To avoid this issue, asynchronous processing is introduced. This model removes the blocking barriers so that each processor can start the next task without waiting for others to finish. However, the performance of algorithms can be varied based on the algorithmic data dependency, architecture and the data structures used in the implementations. This asynchronous model is usually referred to as overlapping the communication with the computation in the context of distributed computations [51, 79, 134].

2.3 Load Balancing and Mapping

Load balancing is also one of the important factors that greatly affect the performance of parallel algorithms since the slowest processor often determines the overall performance. Balancing the workload, communication and memory requirement can reduce the overhead of the parallel computations and increase the algorithm's scalability. Load balancing is closely related to partitioning, as good load balancing can be achieved from an efficient partition of the problem that yields equal workload and/or minimizes the communication among processors. However, some problems

may not be able to be partitioned effectively; thus, alternative techniques which yield good load balancing are required. Normally, mapping of the partitions to processors is statically assigned at the beginning of the computation without further adjustment. Thus, if the partitions of the problem are not well-partitioned, the performance of the algorithm can be affected from load imbalancing. To handle load imbalancing issues, many researchers have proposed various partitioning and/or reordering methods such as randomization, minimum edge-cut [68, 127] and matrix-based reordering [5, 104]. Other techniques that involve dynamically assigning partitions among idle processors have also been introduced [70].

2.4 Other Considerations

There are many other considerations that can affect the overall performance of parallel distributed computations and depend on the characteristics of the problem itself. Usually, the performance of the parallel algorithms depends on the tradeoff between computation and communication. This concept involves the granularity of the parallel algorithms. Fine-grain parallelism which has low computation to communication ratio can give near optimal load balance, but it is also possible to create very high communication and synchronization overhead in this manner. In contrast, coarse-grain parallelism, which has high computation to communication ratio can minimize the communication and synchronization overhead, but good load balancing may be hard to achieve.

Replication is another parallel processing approach that can significantly affect the parallel algorithm's performance. It is an approach that allows some additional duplicated data that is not local to the processor but is required for local computation to be stored in the processor. This approach can be advantageous for algorithms that have high communication cost to exchange information among processors such as algorithms with high data dependency among different partitions. Thus, replicating this nonlocal data can help minimize or avoid a large portion of communication and hence significantly reduce the communication overhead. Fig. 2.3 shows a local view of each partition from Fig. 2.2 when adding replication. The solid circles indicate the actual vertices that belong to the partition, while the dotted circles indicate the nonlocal vertices replicated in the partition. By replicating these nonlocal vertices, the processor that owns vertices 2 and 3 now

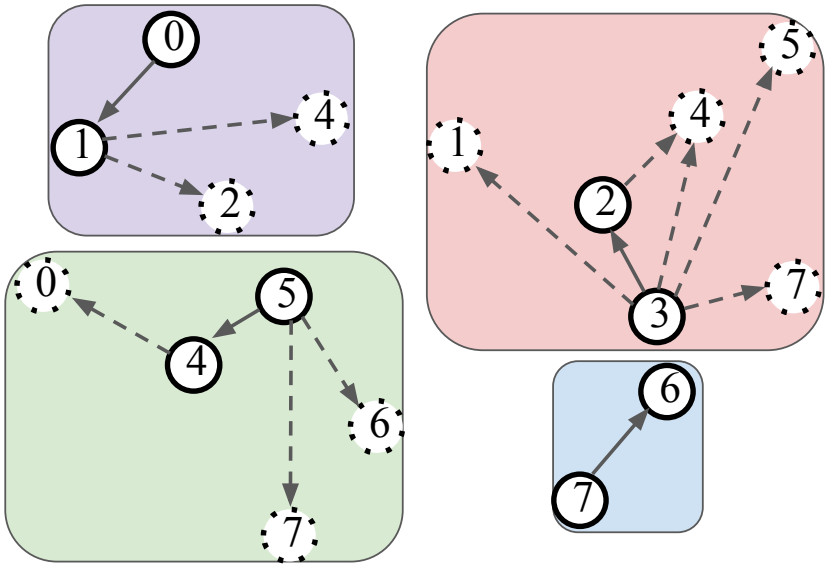


Figure 2.3. A local view of each partition when adding replicated vertices that are adjacent to the local vertices. The solid circles indicate the actual vertices that belong to the partition while the dotted circles indicate the non-local vertices replicated in the partition.

requires three times as much space. This is a tradeoff that can occur when using a replication technique. This approach is well-suited for many parallel mesh optimization algorithms that require accessing positions of neighbor vertices. Since most communication is due to accessing current positions of neighbor vertices residing on different processors, providing local copies of nonlocal information can avoid or reduce such communication. Although overall performance can be considerably improved, it comes with the tradeoff of managing and maintaining redundant data. This technique is sometimes called ghosting or local caching and has been applied in many parallel algorithms [20, 87, 107].

Chapter 3
Performance Analysis of Single-Source Shortest Path Algorithms on Distributed-Memory Systems

Single-source shortest path (SSSP) is a key computation arising in large-scale network analysis, and it is a possible candidate to be included in the Graph500 benchmark [54]. This classical graph analytic problem has been studied for decades because of its widely used applications in many research areas such as communication and transportation, electrical engineering, the World Wide Web and social networks.

Parallel algorithms for SSSP have been studied on various types of architectures. Madduri et al. [84] present an experimental study of parallel algorithms for solving SSSP for large-scale directed graphs with non-negative edge weights on a multithreaded parallel architecture, Cray MTA-2. The algorithms are based on Meyer and Sander's Δ-Stepping algorithm [88] which is a variant of Dijkstra's algorithm [39] that provides more parallelism by introducing a Δ parameter to control a tradeoff between work efficiency and concurrency. Crobak et al. [32] study parallel SSSP on a similar architecture by implementing a parallel version of Thorup's algorithm which is theoretically suited for shared-memory systems. However, their results are only comparable to [84]. Other well-know SSSP implementations that efficiently utilize shared memory multicore processors are as part of Galois

The work of this chapter has been published in:
[94] T. Panitanarak and K. Madduri, "Performance analysis of single-source shortest path algorithms on distributed-memory systems," Proceedings of the 6^{th} SIAM Workshop on Combinatorial Scientific Computing, July 2014, pp. 60-63.

project [71, 90]. They implement parallel versions of modified chaotic iteration and modified Bellman-Ford algorithms on their Galois framework. For distributed-memory systems, Edmonds et al. implement parallel SSSP algorithms that are based on Dijkstra's and Δ-stepping algorithms. However, these implementations are not efficient in terms of execution time as they are parts of Parallel Boost Graph Library [40, 41] which more focuses on a generic programming paradigm. More recent research includes an implementation of SSSP on GPUs and distributed-memory systems. Davidson et al. [35] show good results of their SSSP based on Δ-stepping for GPUs. Chakaravarthy et al. [24] introduce a highly optimized version of Δ-stepping that improve overall scalability of the algorithm on distributed-memory systems.

Our Contributions

We have implemented parallel versions of three well-known SSSP algorithms, Dial's algorithm [38], Bellman-Ford and Δ-stepping [84,88], for graphs with positive integer edge weights. The weights can be either uniformly or normally distributed. The main contributions of this chapter are as follows:

- A performance analysis of three well-known SSSP algorithms, Dial's algorithm, Bellman-Ford and Δ-stepping on distributed-memory systems, running on up to 2048 cores (128 nodes).
- A demonstration of how weight distributions affect the algorithm performance.
- A demonstration of how the values of Δ parameter affect the performance of the Δ-stepping algorithm.

3.1 Single Source Shortest Path Algorithms

Let $G = (V, E, w)$ be a weighted undirected graph with $n = |V|$ vertices, $m = |E|$ edges and integer weights $w(e) > 0$ for all $e \in E$. Define $s \in V$ to be a *source vertex* and $d(v)$ to be a tentative distance from s to v (initially set to ∞) for every $v \in V$. The SSSP problem is to find $\delta(v) \leq d(v)$ for all $v \in V$. Note that $d(s) = 0$ and $d(v) = \infty$ for any v that are not reachable from s.

Relaxation is an operation to update $d(v)$ using in many well-known SSSP algorithms such as Dijkstra's algorithm and Bellman-Ford. The operation updates

$d(v)$ using a previously updated $d(u)$ for each $(u, v) \in E$. An edge relaxation of (u, v) is defined as $d(v) = \min\{d(v), d(u) + w(u, v)\}$. A vertex relaxation of u is a set of edge relaxations of all out-going edges of u. A vertex v is marked as active if it is not settled and its $d(v)$ is previously updated. Thus, a variation of SSSP algorithms is generally based on different relaxation approaches.

The classical Dijkstra's algorithm relaxes vertices in ascending order of all active vertices based on their tentative distances. Any active vertex that has the lowest tentative distance is relaxed first and is marked as settled after one relaxation. (Thus, the first active vertex to be relaxed is s.) To keep track of the relaxation order of all active vertices, the algorithm uses a priority queue that stores and orders the vertices based on their current tentative distances. A vertex is added to the queue only if it is visited for the first time. A vertex that has the lowest distance is always on top of the queue waiting to be processed first before it is removed from the queue. The algorithm terminates when the queue is empty. Although this priority queue provides optimal work efficiency for the algorithm, it also limits the concurrency as only one vertex can be relaxed at a time. Another variant of the Dijkstra's algorithm for integer-weighted graphs that is more suitable for parallel implementation is called Dial's algorithm. It uses a bucket data structure instead of the priority queue to avoid the overhead from maintaining the queue while still giving the same work performance as the Dijkstra's algorithm (i.e., each vertex is relaxed only once). Each bucket has a unit size and holds all active vertices that have the same tentative distance as a bucket number. The bucket k contains all active vertices v with $d(v) = k$. The algorithm processes each bucket in order starting from the lowest to the highest bucket numbers. Any vertex in each bucket has an equal priority and can be processed simultaneously. The present of the buckets is the main parallelism of the algorithm.

Another well-known SSSP algorithm, Bellman-Ford, allows vertices to be relaxed in any order. Thus, there is no guarantee if a vertex is settled after it is relaxed. Generally, the algorithm uses a first-in-first-out (FIFO) queue to maintain the vertex relaxation order since there is no actual priority of vertices. A vertex is added to the queue when its tentative distance is updated and is removed from the queue after it is relaxed. Thus, any vertex can be added to the queue multiple times whenever its tentative distance is updated. The algorithm terminates when the queue is empty. Since the order of relaxations does not affect the correctness

of the algorithm, the algorithm can provide high concurrency from simultaneous relaxations.

While Dijkstra's algorithm yields the best work efficiency since each vertex is relaxed only once, it has very low algorithm concurrency. Only vertices with the shortest distance can be relaxed at a time to preserve the algorithm correctness. In contrast, Bellman-Ford requires more works from (possibly) multiple relaxations of each vertex. However, it gives the best algorithm concurrency since any vertex in the queue can be relaxed at the same time. Thus, the algorithm allows a large number of simultaneous relaxations while the algorithm's correctness is still preserved.

The Δ-stepping algorithm [88] compromises between these two extremes by introducing an integer parameter $\Delta \geq 1$ to control a tradeoff between work efficiency and concurrency. At any iteration $k \geq 0$, the Δ-stepping algorithm relaxes all active vertices that have tentative distances in $[k\Delta, (k+1)\Delta - 1]$. With $1 < \Delta < \infty$, the algorithm yields better concurrency than the Dijkstra's algorithm and lower work redundancy than the Bellman-Ford algorithm. To keep track of active vertices that need to be relaxed in each iteration, the algorithm uses a bucket data structure. All active vertices with the same distant ranges are put in the same bucket. The bucket k contains all vertices that have the tentative distance in range $[k\Delta, (k+1)\Delta - 1]$. The algorithm also provides an additional optimization by using two processing phases in each iteration. The first phase is called *light phase*. In this phase, only edges of active vertices that have edge weights less than Δ (e.g., *light edges*) are relaxed. The reason is that these edge relaxations can result in insertions of some vertices to the current bucket and lead to additional relaxations of those vertices. Thus, this light phase forces these insertions to happen early on so that any redundant work from multiple re-updates of tentative distances is minimized. The relaxations in this phase is similar to those of Bellman-Ford in the sense that some vertices could be relaxed multiple times. The second phase is called *heavy phase*. This phase involves relaxing all edges of all active vertices with edge weights larger than Δ (e.g., *heavy edges*). These edge relaxations guarantee that no vertex is added to the current bucket. Thus, all active vertices are relaxed only once and can be marked as settled (similar to the relaxation in the Dijkstra's algorithm). The Δ-stepping algorithm can be viewed as a general case of SSSP algorithms with the relaxation approach. The work efficiency and concurrency of the algorithm can

be adjusted using Δ. The algorithm with $\Delta = 1$ is equivalent to the Dijkstra's algorithm, while the algorithm with $\Delta = \infty$ yields the Bellman-Ford algorithm.

3.2 Distributed-memory SSSP Implementations

We implement optimized parallel versions of three well-known algorithms, Dial's algorithm, Bellman-Ford and Δ-stepping, for distributed-memory systems. The implementations are based on a bulk synchronous parallel model. This model agglomerates all major computation and communication to occur at the same time in their own separated bulks in each iteration or phase, and all relaxations are done locally and simultaneously. All three algorithms are easily adapted to this model with few modifications.

We use a distributed compressed sparse row (CSR) graph representation for its simplicity and ability to easily access vertex adjacencies which is the main graph query used in the algorithm. The distributed CSR distributes n/p consecutive vertices of the graph to p processors in the same order of the original non-distributed graph where n and p are the total numbers of vertices and processors, respectively. Each processor also stores information of outgoing edges and their corresponding edge weights of all local vertices. The tentative distance array is also partitioned and distributed in a similar manner.

For the Dial's algorithm, its main parallelism is from simultaneous relaxations in each bucket. Without loss of generality, we implement the bucket structure using a regular queue where, in iteration k, the algorithm generates a queue containing vertex u such that $d(u) = k$ where $d(u)$ is a tentative distance from s to u. For all vertices u in the queue, the algorithm looks up for each adjacent vertex v of u, computes $dtv = d(u) + w(u, v)$ where $w(u, v)$ is a weight of an edge uv, and adds a value pair (v, dtv) to a corresponding send buffer. Once finished, these (v, dtv) pairs are distributed to processors that own v by using an *Alltoallv* collective communication. Then, each processor can process on its receive buffer in parallel since all (v, dtv) can be used to update $d(v)$ locally.

Bellman-Ford is a label-correcting algorithm that does not require a priority queue. We implement a modified version that uses a regular queue to keep track all active vertices. In each iteration, only edges of active vertices are relaxed instead of relaxing all graph edges as in a traditional Bellman-Ford algorithm. This

approach also gives a flexibility to terminate the algorithm early when the queue is empty. Thus, the parallel implementation is similar to our Dial's algorithm implementation. However, the algorithm needs to find all u in which their $d(u)$ are previously updated but not settled in each iteration. Note that we can roughly identify whether vertex u is settled by checking if its $d(u)$ is less than the shortest distant of any vertex in the previous iteration.

Algorithm 1 Parallel Δ-stepping

Input: $G = (V, E)$, a source vertex s and the weight function $w : E \to \mathbb{R}$
Output: $\delta(v), v \in V$, the shortest path from $s \to v$

```
1: n_local ← n/p
2: for 0 ≤ v ≤ n_local do
3:    d[v] ← ∞
4: end for
5: curr_bucket ← 0
6: if FindOwner(s) = rank then
7:    s_local ← mod(s, n_local)
8:    AddVertex(s_local, B[curr_bucket])
9:    d[s_local] = 0
10: end if
11: while curr_bucket ≤ num_buckets do
12:    ProcessLightPhase(G, d, B, H)
13:    ProcessHeavyPhase(G, d, B, H)
14:    curr_bucket ← curr_bucket + 1
15: end while
16: Reduce(δ, d)
```

Algorithm 2 *ProcessLightPhase* used in Alg. 1

Input: G, B, d, S

```
1: S ← ∅
2: while B[curr_bucket] ≠ ∅ do
3:    send_buf ← ∅
4:    for each u ∈ B[curr_bucket] do
5:       for each (u, v) ∈ light_edges do
6:          pv ← FindOwner(v)
7:          dtv ← d[u] + w(u, v)
8:          AddRequest((v, dtv), send_buf_pv)
9:       end for
10:      AddVertex(u, S)
11:   end for
12:   B[curr_bucket] ← ∅
13:   recv_buf ← ∅
14:   Alltoallv(recv_buf, send_buf)
15:   Relax(recv_buf)
16: end while
```

Algorithm 3 *ProcessHeavyPhase* used in Alg. 1

Input: G, B, d, S

```
1: send_buf ← ∅
2: for each u ∈ S do
3:    for each (u, v) ∈ heavy_edges do
4:       pv ← FindOwner(v)
5:       dtv ← d[u] + w(u, v)
6:       AddRequest((v, dtv), send_buf_pv)
7:    end for
8: end for
9: recv_buf ← ∅
10: Alltoallv(recv_buf, send_buf)
11: Relax(recv_buf)
```

The Δ-stepping provides a parameter Δ that gives a user an ability to control a bucket size. The unit-size bucket results in an algorithm equivalent to the Dial's

Algorithm 4 *Relax* used in Algs. 2 and 3

Input: $recv\_buf$

1: **for each** $(v, dtv) \in recv\_buf$ **do**
2: $v\_local \leftarrow mod(v, n\_local)$
3: **if** $d[v\_local] > dtv$ **then**
4: **if** $d[v\_local]) \neq \infty$ **then**
5: $RemoveVertex(v\_local, B[\lfloor d[v\_local]/\Delta \rfloor])$
6: **end if**
7: $AddVertex(v\_local, B[\lfloor dtv/\Delta \rfloor])$
8: **end if**
9: **end for**

algorithm which provides the best work efficiency but gives the worst concurrency. The infinity-size bucket results in an algorithm equivalent to the Bellman-Ford algorithm which yields the best concurrency but gives the worst work efficiency. Practically, the Δ-stepping algorithm combines relaxation approaches of both Dial's and Bellman-Ford algorithms. Within each bucket, all relaxations in the light phase are similar to those of the Bellman-Ford algorithm while all relaxations in the heavy phase are similar to those of the Dial's algorithm. Thus, we implement our Δ-stepping by generalizing relaxation approaches of these two algorithms. Similar to [84], there are some optimizations to the data structure and algorithms in our implementations. The following list shows the optimizations that have been introduced in our Δ-stepping implementations.

- An array of buckets B is implemented as a dynamic array. Each bucket is allocated only if there is a vertex being added to the bucket which can be resized when needed. Each bucket can also be de-allocated at the end of its corresponding light phase and the memory can be reused for other bucket allocations since there will be no more insertion into that bucket. Moreover, we implement two auxiliary arrays of size n/p to provide constant time insertions and deletions of vertices for each bucket. One is an array that maps a vertex to its current bucket, and the other is an array that maps a vertex to its current position in its current bucket.

- A timed semi-sort step is introduced as a pre-processing step of the algorithm. It reorders adjacencies of each vertex in an ascending order based on their corresponding edge weights. Then, a heavy edge pointer pointing to the first adjacency v of a vertex u that has $w(u, v) \geq \Delta$ is generated for all adjacency of u. These pointers will be used to determine light and heavy edges in the

algorithm, and give an instant time accessing these two types of edges.

- A unique adjacency array of all local vertices is created. Each element in the array is a tuple of a unique adjacency and its current tentative distance. This array is used to keep track the most recent updates of distances of all (unique) adjacencies. Thus, it helps filtering out unnecessary update requests that have larger distances than the current distances of adjacent vertices.

The complete algorithm for the parallel Δ-stepping algorithm is shown in Alg. 1. The algorithm terminates when all non-empty buckets are processed as shown in line 11. Als. 2 and 3 show the two main processing phases of the algorithm, light and heavy phases, respectively. Note that *num_buckets* is dynamically updated during the relaxation step (Alg. 4) when some vertices are added to B.

3.3 Results and Discussion

3.3.1 Experimental Setup

We collect the experimental performance results of our parallel SSSP implementations on the TACC Stampede cluster [121]. More specifically, all of our experiments were run using a 10 Peta FLOPS Dell Linux Cluster which consists of more than 6,400 Dell PowerEdge server nodes. Each node is equipped with 2 Intel Xeon E5 Sandy Bridge processors (8 cores per processor) and an Intel Xeon Phi Coprocessor. Note that we only used the Intel Xeon E5 processors in the this study. For the inter-node communication, we use the MPI message-passing library (mvapich2 version 1.9a2). Particularly, the collective communication routines, *MPI_Alltoall*, *MPI_Alltoallv* and *MPI_Allreduce*, are the main MPI operations used in the implementations.

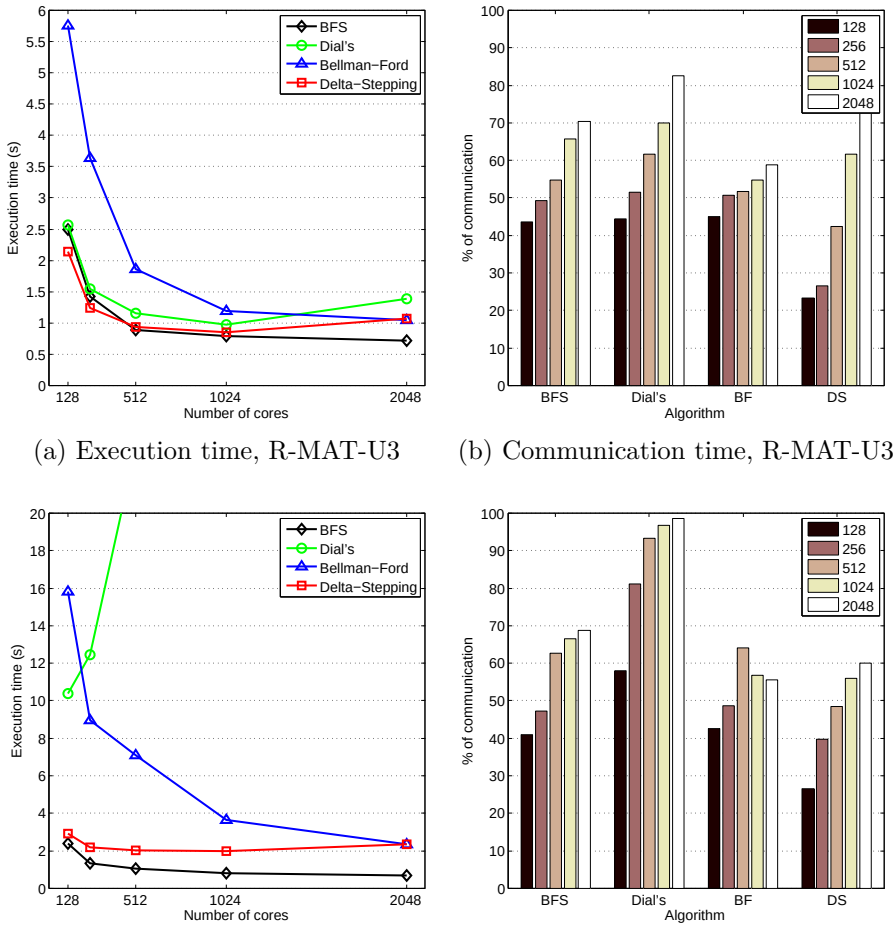
We experiment on synthetic graphs generated from the Graph500 reference implementation v1.2 [54]. The graph generator is based on the R-MAT random graph model [25] with parameters similar to those used in the default Graph500 benchmark, i.e., parameters a, b, c and d are set to 0.59, 0.19, 0.19 and 0.05, respectively, and the edge count to vertex ratio is set to 16. The generated graphs have skewed degree distributions with a very low graph diameter. The corresponding edge weights are generated using the Random123 library [100]. For

ease of references, we categorize the generated graphs used in the experiments into three groups. First, the graphs with uniformly distributed integer weights in the range of $[1, 2^3]$ (R-MAT-U3). Secondly, the graphs with uniformly distributed integer weights in the range of $[1, 2^{10}]$ (R-MAT-U10). Lastly, the graphs with normally distributed integer weights in the range of $[1, 2^{10}]$ (R-MAT-N10).

3.3.2 Strong Scaling

We run the parallel Dial's, Bellman-Ford and Δ-stepping algorithms using 128, 256, 512, 1024 and 2048 cores (one core per MPI task) on three different graph groups. All graphs are generated using the graph scale of 27 (e.g., graphs consisting of 2^{27} vertices). The results are shown in Fig. 3.1. Figures on the left illustrate strong scaling of the three parallel algorithms. Figures on the right show the percentage of time spent in all communication steps. Note that for the Δ-stepping algorithm, we use the best performance from the values of Δ between 16 to 1024.

Among the graph groups, all algorithms give the best performance when running on R-MAT-U3. The Dial's algorithm gives $4\times$ and $5\times$ better performance than it does on R-MAT-U10 and R-MAT-N10 at 128 cores, respectively. It also gives good strong scaling on R-MAT-U3. However, when the number of cores increases, the algorithm's performance significantly decreases on both R-MAT-U10 and R-MAT-N10. On the other hand, the Bellman-Ford algorithm performs slightly better on R-MAT-U3 than it does on R-MAT-N10, but more than $2\times$ faster than the time spending on R-MAT-U10. The Δ-stepping algorithm gives a result similar to the Bellman-Ford algorithm in which, on R-MAT-U3, it runs a little bit faster than it does on R-MAT-N10, but the execution time can be up to $2\times$ faster when running on R-MAT-U10. The reason is that the algorithms require very low number of iterations or phases to settle all reachable vertices in R-MAT-U3 since the numbers of iterations in Dial's and Δ-stepping algorithms are related to edge lengths or distances from s. (Especially, for the Dial's algorithm that the number of iterations and the longest distance are the same. It can run at most DC iterations where D and C are a graph diameter and the maximum weight of all edges in the graph, respectively.) For the Δ-stepping algorithm, the maximum number of iterations is actually the highest number of active buckets which is DC/Δ. Increasing Δ can decrease the number of iterations (or heavy phases) while it also possibly increases

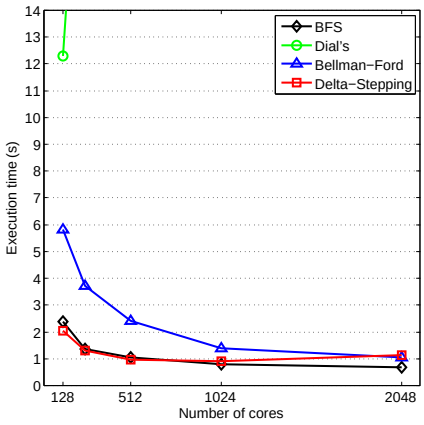


(a) Execution time, R-MAT-U3 (b) Communication time, R-MAT-U3

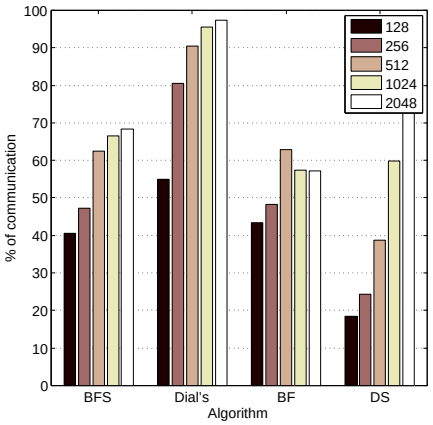
(c) Execution time, R-MAT-U10 (d) Communication time, R-MAT-U10

Figure 3.1. Strong scaling: Overall execution time (on the left column) and percentage of time spent in all communication steps (on the right column) for Dial's, Bellman-Ford and Δ-stepping algorithms on the synthetic graphs with the edge scale of 27, and uniformly and normally distributed weights.

the number of relaxations for each vertex (or the number of light phases). A value of Δ that gives the best performance is a compromise between the number of these light and heavy phases. Furthermore, the results from R-MAT-U3 also show the highest concurrency from a high possibility that many edges have the same weights which increase the number of vertices in the queues or buckets. This also applies to R-MAT-N10. However, only the Bellman-Ford and Δ-stepping algorithms can give the good results on this graph collection because they can get to the mass of the graph very fast by using large-sized buckets (e.g., the infinite size or a large-enough size in the Bellman-Ford and Δ-stepping algorithms, respectively).



(e) Execution time, R-MAT-N10



(f) Communication time, R-MAT-N10

Figure 3.1. Strong scaling: Overall execution time (on the left column) and percentage of time spent in all communication steps (on the right column) for Dial's, Bellman-Ford and Δ-stepping algorithms on the synthetic graphs with the edge scale of 27, and uniformly and normally distributed weights. (cont.).

The Dial's algorithm, on the other hand, has a unit size bucket, and this leads to more number of iterations (and more execution time) before it can get to the mass of the large-weighted graph as we can see in the results on both R-MAT-U10 and R-MAT-N10.

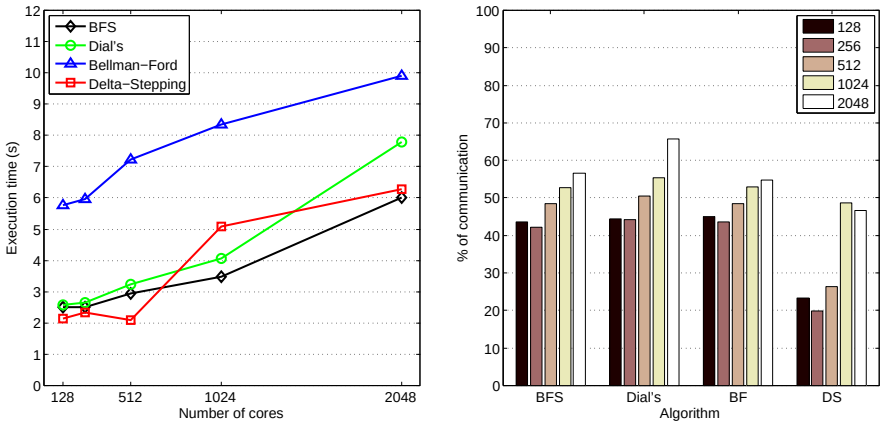
Per the communication overhead, all algorithms usually show more percentages of the communication when increasing the number of cores. However, the Bellman-Ford algorithm shows an interesting trend as the percentage decreases on R-MAT-U10 and R-MAT-N10 when running on 1024 and 2048 cores. It also has low deviation of the communication percentage in which most communication is under 60%. The algorithm also shows very good scaling even if very large number of cores (e.g., 1024 and 2048) is used. The percentage of communication from the Dial's algorithm is the highest among all algorithms in all cases. The communication percentage is 90% or more on R-MAT-U10 and R-MAT-N10 when using more than 256 cores. This characteristic can be expected since the Dial's algorithm has high communication requests from very large number of iterations on the graphs with large weights. For the Δ-stepping algorithm, it has the lowest communication ratio among all algorithms on all graph groups when running on 256 cores or lower. The communication is usually less than 40% except when using 512 cores on R-MAT-U8 and R-MAT-U1024 that the communication percentage can be up to 48%. This

low communication overhead is from the optimized data structure that helps filter out unnecessary requests. The optimization not only reduces the communication overhead but also decreases the computation overhead. However, when the large number of cores (1024 and 2048 cores) is used, the data structure is less efficient as we can see from rapidly increasing of the communication percentage on R-MAT-U3 and R-MAT-N10.

3.3.3 Weak Scaling

We collect a weak scaling performance of Dial's, Bellman-Ford and Δ-stepping algorithms on R-MAT graphs with 2^{24} vertices per computing node (16 cores) range from 7 to 128 nodes or from 128 to 2048 cores with one core per MPI task. We experiment on the same graph collection as in the strong scaling experiment namely R-MAT-U3, R-MAT-U10 and R-MAT-N10. The results are shown in Fig. 3.2. Figures on the left illustrate weak scaling of the three parallel algorithms while figures on the right show the percentage of time spent in all communication steps.

Similar to the strong scaling experiment, all algorithms usually yield better performance on R-MAT-U3. The Dial's algorithm, again, gives 4× and 5× better performance on R-MAT-U3 than it does on R-MAT-U10 and R-MAT-N10 when using 124 cores, respectively. Although the algorithm does not scale well on R-MAT-U10 and R-MAT-N10, it still gives good scaling results on R-MAT-U3. The performance is also close to BFS up to 512 cores, and its running time is even 1.25× faster than the running time of the Δ-stepping algorithm on 1024 cores. When running on 2048 cores, the algorithm execution time is only about 1.25× times slower than the running time of BFS and Δ-stepping algorithms, and around 1.25× faster than the execution time of the Bellman-Ford algorithm. The Bellman-Ford algorithm provides the best performance on R-MAT-U3. Even though the algorithm gives very close results at 128 and 2048 cores when comparing to the results obtained from running on R-MAT-N10, it yields lower execution time between 256 and 1024 cores, especially, at 1024 cores, it runs about 1.5× faster on R-MAT-U3. When comparing the results when running on R-MAT-U3 and R-MAT-U10, the Bellman-Ford algorithm performs more than 2.5× faster on R-MAT-U3 in all cases. Interestingly, for the Δ-stepping algorithm, it gives better execution time on R-MAT-N10 although the performance is not much difference



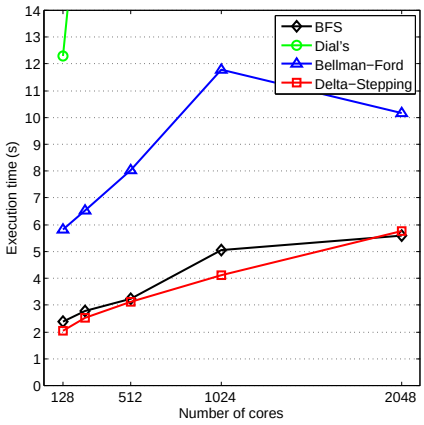
(a) Execution time, R-MAT-U3 (b) Communication time, R-MAT-U3

(c) Execution time, R-MAT-U10 (d) Communication time, R-MAT-U10

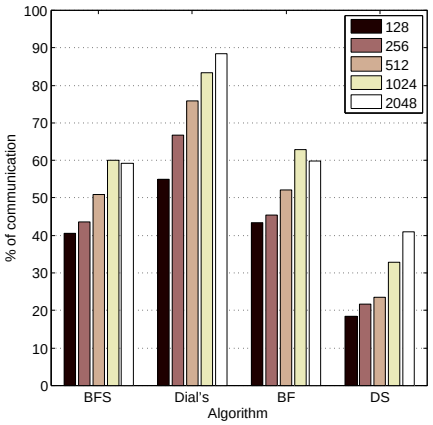
Figure 3.2. Weak scaling: Overall execution time (on the left column) and percentage of time spent in all communication steps (on the right column) for Dial's, Bellman-Ford and Δ-stepping algorithm on synthetic graphs with uniformly and normally distributed weights (with 2^{27} vertices per 128 cores).

when running on R-MAT-U3 where the results are more fluctuated. Note that on R-MAT-N10, the Δ-stepping algorithm performs better than BFS due to the optimized data structure that only presents in the former algorithm.

For the weak scaling experiment, the percentage of time spent in all communication steps of all algorithms is less than the strong scaling results, especially, in Dial's and Δ-stepping algorithms. The Dial's algorithm now shows 60% lower communication percentage on R-MAT-U3 with the exception of 65% when running on 2048 cores. It also gives less than 90% communication percentage in most cases on R-MAT-U10 and R-MAT-N10. For the Bellman-Ford algorithm, the



(e) Execution time, R-MAT-N10



(f) Communication time, R-MAT-N10

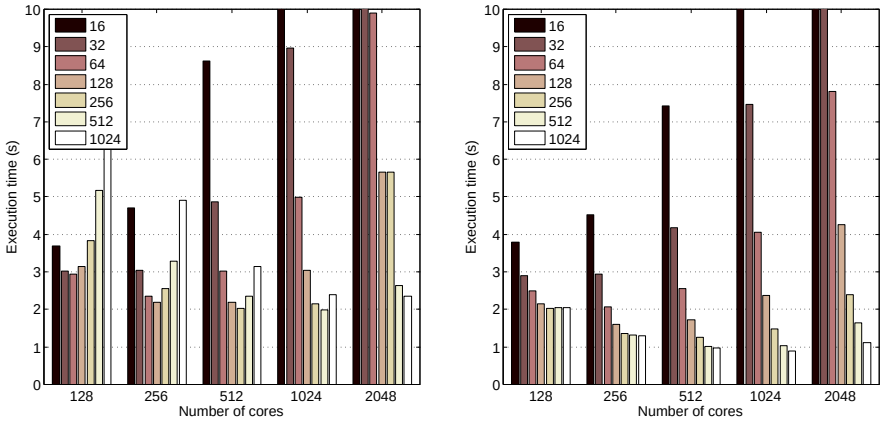
Figure 3.2. Weak scaling: Overall execution time (on the left column) and percentage of time spent in all communication steps (on the right column) for Dial's, Bellman-Ford and Δ-stepping algorithm on synthetic graphs with uniformly and normally distributed weights (with 2^{27} vertices per 128 cores) (cont.).

communication percentage is still less than 60% in most cases, and also shows some decreasing trends as in the strong scaling experiment. These trends can be observed when running on R-MAT-U10 and R-MAT-N10 using 2048 cores. For the Δ-stepping algorithm, the communication shows huge differences between the strong and weak scaling experiments. The algorithm is now has the lowest communication percentage among all algorithms in all cases. All results comprise of less than 50% communication percentage, and show some decreasing trends on R-MAT-U3 and R-MAT-U10 running on 2048 cores. For the Δ-stepping algorithm, it also has less than 40% communication percentage in most cases on R-MAT-N10. This shows that the Δ-stepping algorithm also has very good weak scaling.

3.3.4 Impact of Δ on the Δ-Stepping Algorithm's Performance

The results of strong and weak scaling in the previous sections of the parallel Δ-stepping algorithm are obtained based on the Δ values that give the best execution time. Even though the algorithm yields the best performance among the three algorithms in all cases, it is very sensitive to the choice of Δ. To show the effect of Δ values on various numbers of cores and the problem sizes, we run the parallel Δ-stepping algorithm on the same setup as in strong scaling experiments with both uniform and normal weight distributions using various Δ among 16, 32, 64, 128,

256, 512 and 1024. The results are shown in Fig. 3.3.



(a) Uniform weight distribution (b) Normal weight distribution

Figure 3.3. The impact of Δ on (a) uniformly distributed and (b) normally distributed weight graphs.

From the figures, with different Δ values, the algorithm performance is significantly vary. The difference between shortest and longest times spent in the algorithm can be twice as fast or more than 10× faster. On both uniformly and normally distributed weight graphs (R-MAT-U10 and R-MAT-N10), the timing results can be increasing when the number of cores are increased if Δ is fixed to 16 or 32. In some cases, fixing Δ to some values can decrease the execution time when increasing the number of cores; however, at some points, the running time can increase when the number of cores increases. These can be observed on the algorithm with $\Delta = 256$ running on R-MAT-U10 and R-MAT-N10 that the execution time keeps reducing until 512 cores and then increasing afterward.

As we can see, there is no constant value of Δ that gives most effective results and can be applied in all conditions. However, from the observation, we can conclude that, for uniformly-distributed weighted graphs, a smaller Δ tends to give better performance on a small number of cores while a larger Δ has a tendency to yield better performance on a larger number of cores. For normally-distributed weighted graphs, a larger Δ usually gives better performance since it helps to get to a mass of the graphs faster so that the algorithm can utilize an existing concurrency of the graph data.

The best value of Δ for a given problem is a balance execution between light and heavy phases. As we known, the value of Δ directly affects the number of buckets in

the Δ-stepping algorithm. The larger Δ, the less number of buckets. This variation of a bucket size is also directly affecting the execution in both light and heavy phases. From our experiment, a smaller Δ tends to lead to more communication overhead due to the algorithm requires more heavy phases which usually leads to a large communication ratio in each iteration. Conversely, once a larger Δ is used, a time spent during light phases is dominated, specifically, from handling substantial vertex reinsertions although theses light phases usually have less communication overhead.

3.3.5 Compare Δ-stepping Performance to Previous Works

Graph	#vertices	#edges
R-MAT-UXX^{21}	2,097,152	33,554,432
R-MAT-UXX^{22}	4,194,304	67,108,864
*USA-road-d.W	6,262,104	15,248,146
*USA-road-d.CTR	14,081,816	34,292,496
*USA-road-d.USA	23,947,347	58,333,344

Table 3.1. The number of vertices and edges of the graphs used in the comparison experiments. The graphs marked with (*) are obtained from the challenge benchmarks of The Ninth DIMACS Implementation Challenge.

Graph	BFS(1)	BFS(16)	iGr	Gal(1)	Gal(16)	PB(1)	PB(16)	DS(1)	DS(16)
R-MAT-U1^{21}	1.87	0.14	3.97	5.74	0.21	110.94	*	**2.05**	**0.15**
R-MAT-U10^{21}	1.87	0.14	4.92	5.87	0.74	111.26	*	**2.43**	**0.22**
R-MAT-U1^{22}	4.10	0.28	8.98	8.26	0.48	223.60	*	**4.54**	**0.31**
R-MAT-U10^{22}	4.10	0.28	11.47	9.13	1.75	224.04	*	**5.38**	**0.39**
USA-road-d.W	0.64	0.21	5.13	2.02	**0.67**	36.34	21.07	**1.01**	1.29
USA-road-d.CTR	1.72	0.38	14.31	5.53	**1.53**	84.92	42.55	**3.74**	2.01
USA-road-d.USA	2.32	0.75	24.57	7.91	**2.49**	139.32	67.03	**4.39**	3.19

Table 3.2. The table shows the execution time of our Δ-Stepping (DS) implementation, the Dijkstra's algorithm on iGraph (iGr), the chaotic iteration on Galois (Gal), and the Δ-stepping on PBGL (PB), on various graphs (see Tab. 3.1). For the implementations that can run in both serial and parallel modes, (1) indicates the serial execution time, and (16) indicates the parallel execution time on 16 cores.

We compare the performance of our Δ-Stepping (DS) implementation to some previous SSSP implementations, the Dijkstra's algorithm from iGraph library (iGr), the chaotic iteration algorithm from Galois project (Gal), and the Δ-stepping on Parallel Boost Graph library (PB), on various graphs shown in Tab. 3.1. We also include our simple breath-first search results as a reference timing. The execution

time for each algorithm is given in Tab. 3.2. Note that the result timing of PB on R-MAT graphs with 16 cores is not available due to some memory issue when we tried to run these experiments. The best timing results for serial and parallel execution are shown with bold fonts and highlighted in red and blue, respectively.

For single-core results, DS(1) shows the fastest execution time in all cases. It gives more than 2×, 1.5×, and 40× faster than iGr, Gal(1), PB(1) on R-MAT graphs, respectively. Its performance is also close to BFS(1) execution time on this graph collection. For the road network graphs, DS(1) runs more than 4×, 1.5×, and 25× faster than iGr, Gal(1), PB(1), respectively.

For single-node (16 cores) results, DS(16) performs the best in all cases of the R-MAT graphs with more than 1.4× faster than Gal(16). On the other hand, the shared-memory Gal(16) implementation shows the best results in all cases on the road network graphs, and around 1.3× better than the distributed-memory DS(16) implementation. Although the road network graphs have less vertices and edges than the R-MAT graphs, they usually have much larger graph diameters and edge weights. On these graphs, the Δ-stepping algorithm does not perform well since the algorithm suffers from computational load balancing and low graph concurrency.

3.4 Conclusions and Future Work

In this chapter, we present the experimental study of three parallel SSSP algorithms, Dial's, Bellman-Ford and Δ-stepping algorithms, running on the Stampede cluster. The experiments are conducted using synthetic graphs generated using Graph500 with uniformly and normally distributed edge weights. We report both strong and weak scaling including the communication percentages spent in the algorithms. In most cases, the Δ-stepping algorithm yields the best performance, and gives the results that are relatively close to the results from our simple BFS implementation. On the other hand, the Dial's and Bellman-Ford algorithm can give good results depending on the input graphs. The Bellman-Ford algorithm usually performs better than the Dial's algorithm on graphs with large edge weights. The algorithm also gives the lowest increment of the communication percentages which results in very good scaling of the algorithm. For graphs with large edge weights, both Bellman-Ford and Δ-stepping algortihms run faster the the Dial's algorithm on the same size problems with normally distributed edge weights while the Dial's

algorithm gives better results on the graphs with uniformly distributed edge weights.

We are planning on introducing more optimizations for our SSSP implementations to make the algorithms more efficient and scalable. One approach is to alternately run multiple algorithms on the same problem. For example, new SSSP algorithm starts with the Δ-Stepping algorithm following by the Bellman-Ford algorithm when some criteria is met. This is from our observation that the Δ-Stepping algorithm usually performs better in early iterations when most of vertices are not settled, while the Bellman-Ford algorithm gives better results in later iterations when more than half of all vertices are settled. Another optimization approach is to implement an efficient, automatic Δ predetermination that always yields the optimal performance for any combination of problem sizes and number of cores used. Furthermore, we also plan to implement a hybrid parallel algorithm that can efficiently utilize both shared- and distributed-memory systems to further reduce the communication cost in the algorithm which is currently a major bottleneck when running on the systems with very large number of cores.

Chapter 4
Scalable Distributed Graph Algorithms

Many graph processing frameworks have been actively developed and updated over the past few years to support large-scale graph data which is commonly in a size of billions of vertices and trillions of edges. Examples of these frameworks include GraphLab/PowerGraph [51, 79], GPS [106], GraphX [136], Trinity [110] and many others [85, 130] to name a few. However, they are usually outperformed by standalone implementations of specific graph algorithms such as the distributed breadth-first search and single-source shortest paths [10, 19, 24, 26] since by focusing on one specific algorithm, ones can fine-tune and optimize both algorithm's communication and computation in more detail.

Normally, it is difficult to optimize a graph framework to perform as close as being done for a specific algorithm since a framework optimization is usually about improving primitive graph operations and inter-processor communication. Most optimizations are done at the level of underlying graph models or data structures, and usually do not directly reflect the performance of any specific graph algorithm since different graph algorithms have different communication and computation patterns. Moreover, many real-world graphs are commonly unstructured in nature, and this irregularity makes the optimization even harder for graph algorithms.

Most distributed graph computations utilize distributed adjacency lists of vertices which are usually presented by a compressed sparse row format (CSR) to decrease the memory requirement. This representation can be viewed as a one-dimensional (1D) layout of a vertex adjacency matrix of a graph since it is based on row-wise partitioning of the matrix. This approach gives a natural view

of each vertex and its out-going edges which is sufficient for graph traversal in most graph algorithms. While this approach works well in general, there are two major flaws that can degrade an overall performance. First, the 1D layout has high communication overhead because the information of adjacent vertices of one vertex is distributed among all processors. Any communication involving an update to these vertices requires all processors to participate which is usually in the form of a high-traffic all-to-all collective communication. Secondly, the 1D layout partitions a graph based only on the number of vertices. While each processors gets approximately the same number of distributed vertices of the graph, there is no guarantee that the numbers of edges in each partition are equally distributed. This can lead to load balancing issues for graph algorithms that require edge traversal since some processors might be assigned a much larger number of edges than the others. This problem is more pronounce, especially, in power-law real-world graphs in which the graphs contain very few high-degree vertices, while the majority of vertices have very low degree.

There are many approaches that have been developed to overcome the load balancing issues for a graph with high degree vertices. A vertex cut technique is an approach that has been used in many graph frameworks and algorithms [24, 136]. For any high degree vertex, it is broken down into multiple vertices connecting to each other with special edges based on graph applications. For example, these edge weights of the cut vertices are set to zero for shortest path algorithms. While this method can improve load balancing of distributed edges, it also adds an extra complexity for an additional structure to keep track of vertices that have been cut. Another approach that can improve the inter-processor communication and edge load balancing is to partition a graph by distributing both vertices and edges equally. We consider a two-dimensional (2D) graph layout that is commonly used in matrix algebra, and in a previous study in [19] for a breadth-first search algorithm. This approach distributes a graph based on both row and column partitioning of the graph adjacency matrix (as also known as *block partitioning*). Now, edges of each vertex can be distributed to more than one processor. Thus, it improves load balancing of distributed edges. Furthermore, the communication space of the all-to-all communication also reduces since the number of unique adjacent vertices of each partition decreases. However, additional all-gather and point-to-point communication phases are required to maintain the completeness of the graph

traversal. We discuss this 2D layout for graph algorithms later on in the chapter. Since this 2D layout applies to the underlying graph data structures without fine-tuning any specific graph algorithm, it can be applied to many distributed graph algorithms efficiently and effectively, and can be implemented as a framework for distributed graph computations. Our major contributions of this chapter are shown as follows:

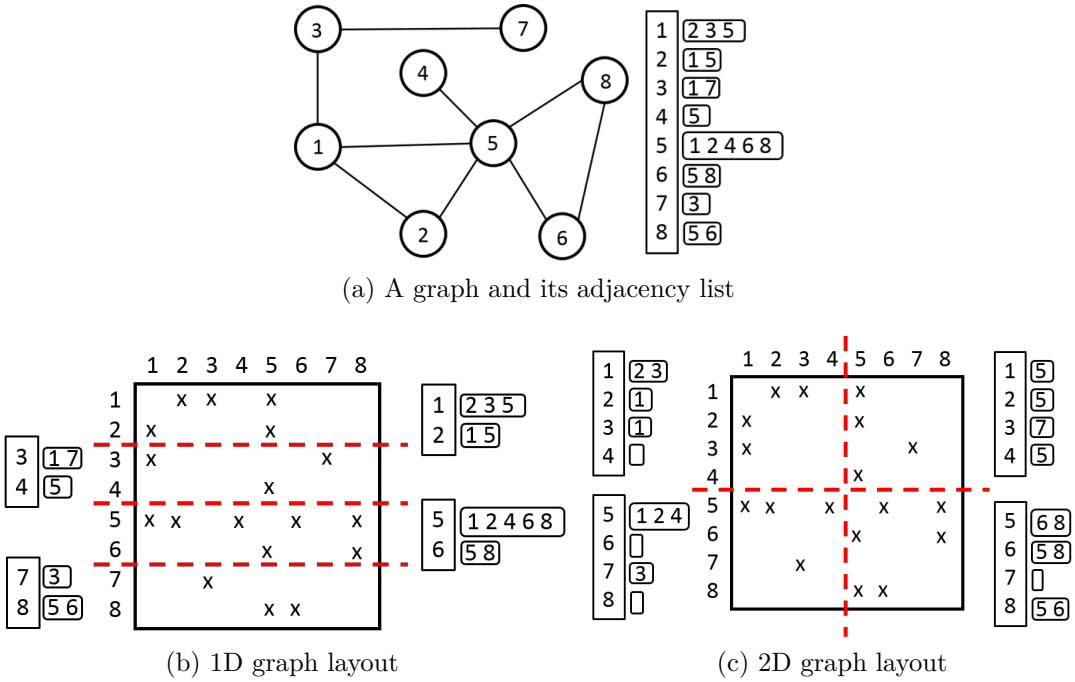
- We extend the use of the 2D layout as the underlying distributed graph data structure by combining it with various optimizations that can improve the performance of many parallel graph algorithms.
- We give detailed analysis and comparison of communication and computation between two distributed graph data structures, the traditional 1D and novel 2D graph layouts.
- We evaluate the performance of well-known graph algorithms with the 2D layout on some large-scale real-world graphs.

4.1 Overviews

In this section, we give an overview of the 2D layout and how it can be applied to graph computations. We also give a brief overview of a bulk-synchronous parallel model that is applied in our graph algorithms.

4.1.1 Two-Dimensional (2D) Graph Layout

Let $G = (V, E)$ be an undirected graph with $n = |V|$ vertices and $m = |E|$ edges. Two of the most common graph representations are an adjacency list and adjacency matrix of the graph vertices. A vertex $v \in V$ is adjacent to a vertex $u \in V$ if there is an edge from u to v or $uv \in E$. A set of vertices that are adjacent to u can be written by $adj(u)$. The adjacency list is the list of all $adj(v)$ for all $v \in V$. An example of the graph adjacency list is shown in Fig. 4.1(a). The adjacency matrix is an $n \times n$ matrix that its element $(i, j) = 1$ if $ij \in E$, otherwise, $(i, j) = 0$. Fig. 4.1(b) shows an example of the adjacency matrix. Note that we use an x instead of 1 and a blank space instead of 0 for the purpose of visualization only. The advantages of these two representations are that they are easy to manage and



(a) A graph and its adjacency list

(b) 1D graph layout

(c) 2D graph layout

Figure 4.1. An example of (a) a simple graph and its adjacency list, and (b) its 1D and (c) 2D graph layouts for four partitions, respectively.

maintain, and can be extended to a distributed graph representation effortlessly as also shown in Fig. 4.1(b) with the red dashed lines. A distributed adjacency list is simply a partition of the original adjacency list while a distributed adjacency matrix is a row partitioning of the original adjacency matrix. As the partition involves only one dimension of the matrix, we refer this partitioning approach as a one-dimensional (1D) layout. However, the obvious drawback of this layout is that only graph vertices are equally partitioned. There is no guarantee that graph edges are equally distributed and can result in an unbalanced edge distribution. As shown in our example, after applying the 1D layout, the partition that owns vertices 3 and 4 contains 3 edges while the partition that owns vertices 5 and 6 contains 7 edges. This problem is more pronounced when there are very few vertices that have more adjacencies than others. This graph characteristic is often found in scale-free graphs that arise from many real-world networks. One approach to resolve this issue is to partition the adjacency matrix along both row and column as shown in Fig. 4.1(c) with the red dashed lines. While it still does not guarantee to distribute the edges equally among partitions, it provides a better edge distribution since edges of each vertex are now distributed among multiple partitions that are in the same partition

rows, instead of residing in only one partition as in the 1D layout. Since this method partitions the adjacency matrix in a two-dimensional manner (along both row and column), we refer this partitioning method as a two-dimensional (2D) layout.

4.1.2 Bulk Synchronous Parallel for Graph Computations

A Bulk Synchronous Parallel (BPS) model is a model for parallel algorithms with a message passing abstraction. Algorithms based on the BSP model are usually simple to implement and maintain while providing low latency cost of inter-processor communication, even though a fine-grained parallelism of algorithms is more difficult to be exploited.

BSP executes in a sequence of supersteps. Each superstep consists of three main components, concurrent computation, communication and synchronization. In the concurrent computation step, each processor performs local computation simultaneously using available data that it owns. In the communication step, all processors exchange information via explicit messages so that non-local information required by each processor will be made available for the requesting processor. The communication routines are usually in the forms of collective communication such as all-to-all and all-gather communication. Finally, in the synchronization step, each processor updates its local information and wait for all processors (at each synchronization barrier) to finish their work before progressing on the next superstep. A termination can be done when there are no information to be exchanged, and all processors finish their work.

Most graph algorithms are iterative. In each iteration, the algorithms involve traversing through a set of edges and update data of vertices which the edges point to. For distributed graphs, completing all edge traversals requires communication among processors because adjacencies of local vertices can be resided on other processors. To apply BSP to graph algorithms, each iteration of the algorithms is considered a superstep. The algorithm operations are organized and combined into bulks of local computation, global communication and barrier synchronization as shown in Fig. 4.2. However, it is more practical to model iterative graph algorithms into four phases (still based on BSP) as they can be directly translated to an implementation. These four phases are; local discovery where each processor issues

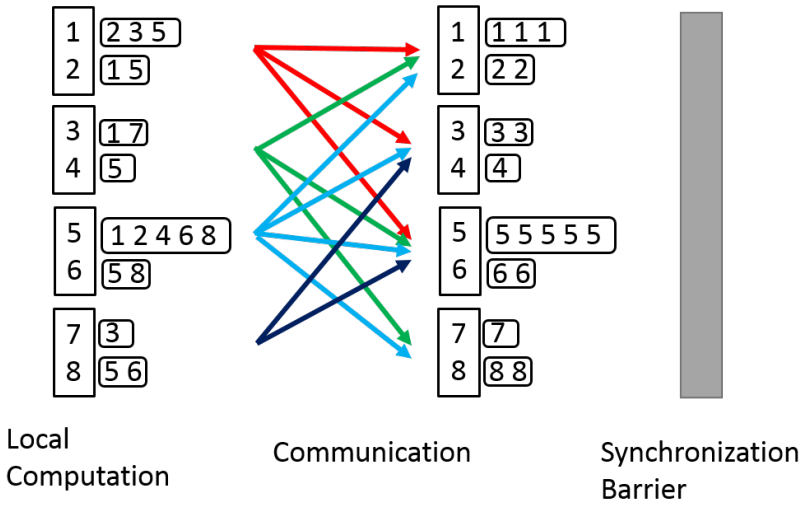


Figure 4.2. Bulk-synchronous parallel.

communication requests; edge traversal where data is exchanged among processors; local update where the transfered data is used to update the local data; and termination check where all processors check for global termination.

4.2 Graph Algorithms

In this section, we briefly give an overview of five graph algorithms, breadth-first search, approximate diameter, PageRank, connected components and Bellman-Ford single-source shortest path. These are selected graph algorithm candidates that are used to evaluate our graph framework with the 2D graph layout.

4.2.1 Breadth-First Search

Breadth-first search (BFS) is an algorithm for solving a graph traversal problem by visiting all vertices that are reachable from a particular vertex called a source vertex $s \in V$. Usually, the algorithm also returns a label (or level) of each vertex indicating how far each vertex is from s. We define a label of v as $d(v)$. Initially, $d(v)$ is set to infinity (as for unreachable from s) for all $v \in V$.

BFS starts at the source vertex s by setting $d(s) = 0$, marking s as visited and adding s to a queue called frontier. Next, for each vertex u in the frontier, all v

where $uv \in E$ are examined such that if v is not yet visited, then $d(v)$ is set to $d(u) + 1$, and v is marked as visited and is added to the frontier. Otherwise, there is no update on v. Note that v is not added to the frontier if it is already visited. Once all adjacent vertices of u are processed, u is removed from the frontier. The algorithm terminates when there is no more vertex in the frontier, and returns $d(v)$ for all vertices v.

4.2.2 Approximate Diameter

A diameter, D, of a graph is the longest distance of shortest paths between any two vertices in the graph. It is trivial that D is the largest $d(v)$ obtained from running BFS on every single vertex. Although this method sounds simple, it suffers from very high complexity and is not practical for large graphs.

Many approximation methods have been proposed to give an estimate diameter, $\hat{D}$, of a graph. One well-known, simple approximation that gives an estimate diameter $\hat{D}$ where $D/2 \leq \hat{D} \leq D$ (also referred as a 2-approximation) can be calculated by running BFS on an arbitrary vertex s to find the current largest $d(v_s)$. To tighten the approximation, ones can run BFS repeatedly from a new source vertex $s \leftarrow v_s$ until the largest $d(v_s)$ does not improve, and returns the current largest $d(v_s)$ as $\hat{D}$. In this chapter, we implement our parallel approximate diameter based on this approximation approach.

4.2.3 Connected Components

A connected component of a graph is a subgraph in which any two vertices are connected to each other by paths. The connected components problem is to identify all connected components of the graph.

The connected components can be computed in linear time in terms of the numbers of vertices and edges of the graph by using BFS, a depth-first search (DFS) or a minimum label propagation (MLB). For the method with BFS or DFS, the idea is to find all graph component using either BFS or DFS on any source vertex that has not yet been visited. Thus, the number of BFS or DFS runs is proportional to the number of components of a graph. For the method with MLB, the idea is to give each vertex a unique label or value, normally, using its vertex ID, and propagate a lowest label to cover an entire component of a graph. Thus,

the number of graph components is the number of labels left.

In this chapter, we consider the MLB algorithm to find all connected components of a graph. First, an initial label of each vertex is set to its vertex ID, and all vertices are marked active. Then, all active vertices send their labels to their neighbors, and update their labels to the lowest label which they received. If the current label is already the lowest, the vertex marks itself inactive; otherwise, it marks itself active. The algorithm terminates when all vertices are inactive, and returns the number of unique labels left as the number of all connected components of the graph.

4.2.4 PageRank

PageRank [18] is a ranking method proposed by Page et al. to rank Web pages based on numbers and ranks of other Web pages that link to them. Instead of merely counting the number of Web pages of incoming links (or incoming edges), PageRank, first, weights each edge based on the rank of which the edge is from, and computes a new rank from these weighted edges. Thus, by this ranking system, a rank computed from a few incoming edges of high-ranked pages may be higher than a rank computed from many incoming edges of low-ranked pages.

Let $u \in V$ be a Web page of a Web graph G, B_u be the set of pages that point to u or $B_u = \{v|(v,u) \in E \text{ for all } v \in V\}$, and F_u be the set of pages that u points to or $F_u = \{v|(u,v) \in E \text{ for all } v \in V\}$. Let $d_u = |F_u|$ be the degree of outgoing edges of u. Then, a simplified version of a rank of u can be defined as

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{d_v} \tag{4.1}$$

From Eq. (4.1), each page divides its rank evenly by the number of its outgoing edges, and distributes the results to all adjacencies of the outgoing edges. The new rank of each page is the sum of these numbers. The process iterates until the rank is converged. With a present of damping factor (α), the equation turns to

$$PR(u) = (1-\alpha)\frac{1}{n} + \alpha \sum_{v \in B_u} \frac{PR(v)}{d_v} \tag{4.2}$$

where α is a value between 0 to 1, but generally, it is set to 0.85. Our implementation of PageRank uses Eq. (4.2).

4.2.5 Modified Bellman-Ford Single-Source Shortest Path

Bellman-Ford is an algorithm for solving a single-source shortest path (SSSP) problem by finding the shortest distant from a particular vertex (or source vertex) s to each vertex v in the graph. In this chapter, we consider a graph with positive integer edge weights w. That is, for all $uv \in E$, $w(uv) \in I^+$. The algorithm returns a shortest distance of each vertex v to s, $d(v)$, for all $v \in V$. Similar to BFS, $d(v)$ is initially set to infinity.

There are various versions of Bellman-Ford algorithms. In this chapter, we consider a modified Bellman-Ford algorithm described as follows. First, the algorithm starts at a source vertex s by setting $d(s) = 0$, and adding s to a queue. Next, for each u in the queue, all v's where $uv \in E$ are examined such that if $d(v) > d(u)$, $d(v)$ is set to $d(u)$, and v is added to the queue. Note that the algorithm also maintains the queue in the way that it only contains unique vertices to avoid redundant computations. Once all v's are processed, u is remove from the queue. The algorithm terminates when there is no vertex in the queue, and returns $d(v)$ for all $v \in V$.

4.3 Distributed Graph Algorithms with 2D Layout

In this section, we describe how the 2D layout can be used as an underlying distributed graph data structure, and how it affects communication and computation of distributed graph computations. We also show our parallel implementations of the five selected graph algorithms, BFS, approximate diameter, PageRank, connected components and Bellman-Ford, which are previously described in Section 4.2. All of our implementations are based on the BSP model.

4.3.1 2D Layout and Its Communication

The 2D graph layout partitions vertices by distributing n/p consecutive vertices to each partition where n is the number of vertices in the graph, and $p = p_r \times p_c$ is the number of partitions where p_r and p_c are partition rows and columns, respectively. Consider the equivalent $n \times n$ adjacency matrix of the graph, the 2D layout partitions the matrix into blocks of $n/p_r \times n/p_c$ submatrices. Thus, a number of adjacencies (or edges) that each partition has is actually the number of nonzero of its assigned

submatrix. Unlike the 1D layout that all adjacencies of each vertex always belong to the same partition, the 2D layout allows adjacencies of one vertex to be distributed among partition columns that are in the same partition rows (see Fig. 4.1 in Section 4.1.1). For example, if some adjacencies of a vertex v are in a partition $p_{2,0}$, the rest of the adjacencies are distributed among partitions $p_{2,k}$ where $k = 1, ..., c-1$. In most cases, the 2D layout usually provides better edge load balancing than the 1D layout, specifically, for scale-free or power-law graphs. Note that the 1D layout is basically a special case of the 2D layout when the number of partition columns is one.

In terms of communication, since our implementations are based on the BSP model, communication is synchronous. The purpose of communication in each BSP superstep is to exchange the information that is generated from the concurrent computation among participating processors so that after the exchange, non-local information is locally available for each processor to use in the next update and synchronization steps. Note that we assume that each partition is assigned to one processor. The 1D layout requires an all-to-all communication to exchange the information, and it needs all processors to take part in. In the 2D layout, the all-to-all communication only occurs along the processor columns. Although an additional transpose (i.e., a specific point-to-point communication) is required before the exchanged information is locally available, the 2D layout still has less communication overhead than the overhead from the 1D layout as discussed in [19].

4.3.2 Parallel Implementations

The template of our 2D graph algorithms is shown in Alg. 5. It consists of five phases, three for local computations and two for communication. These local computations are *init*() where necessary data structures and parameters for each algorithms are initiated, *discovery*() where edge traversals to all adjacencies of active vertices are performed, and *update*() where local information such as vertex data, iteration/phase parameters and a list of active vertices are updated. The communication phases include *allgatherv*() where each processor gathers the data from all active vertices along its processor rows, and *alltoall_trans* where data exchanging operations consisted of all-to-all communication along processor columns and send/receive transpose between two processors are occurred. In most cases, the

communication phases are the same for any graph algorithm. Next, we describe how each algorithm is implemented based on the template.

Algorithm 5 Push-model graph algorithm templete

```
1: init(VERTEX_DATA, QUEUE)
2: while QUEUE ≠ ∅ do
3:    allgatherv(QUEUE)
4:    BUFFER ← discovery(VERTEX_DATA, QUEUE)
5:    alltoallv_trans(BUFFER)
6:    (VERTEX_DATA, QUEUE) ← update(BUFFER)
7: end while
```

Breadth-First Search. The *init*() involves initializing an integer array d to infinity (or unreachable), and processing the source vertex by setting its BFS level to zero and putting it to a queue of active vertices or frontier. Note that d is used for storing the BFS levels of all local vertices. The *discovery*() processes each active vertex u in the queue. The processing includes generating pairs (v, dv) where $v \in adj(u)$ for all u and $dv = d(u) + 1$, and putting these pairs to a send buffer. Finally, the *update*() updates the level array d locally using a receive buffer such that if $d(v) > dv$, $d(v) = dv$. It also determines new active vertices (which are the vertices that have never been updated before), and adds them to the queue. The algorithm terminates when no processor has any active vertex in the queues.

Approximate Diameter. Our algorithm repeatedly runs BFS until the longest length to any $v \in V$ from any new source vertex does not improve, and returns this length as the approximate diameter. Thus, we can implement the algorithm by just reusing our BFS implementation. The algorithm only needs to keep track of any new vertex that gives the longest BFS path, and uses it as a new source vertex.

Connected Components. We implement an MLB algorithm to find the connected components of the graph. The *init*() initializes each vertex u to be active, and sets its label $d(u)$ using its vertex ID. The *discovery*() generates pairs (v, dv) where $v \in adj(u)$ for all active vertices u in the queue and $dv = d(u)$, and puts these pairs to a send buffer. The *update*() updates the label of each v from the buffer of pairs (v, dv). If $d(v) > dv$, $d(v) = dv$. Otherwise, v is marked inactive. The algorithm progresses until there is no active vertex. The number of components

of the graph is the number of vertices that their labels never been updated.

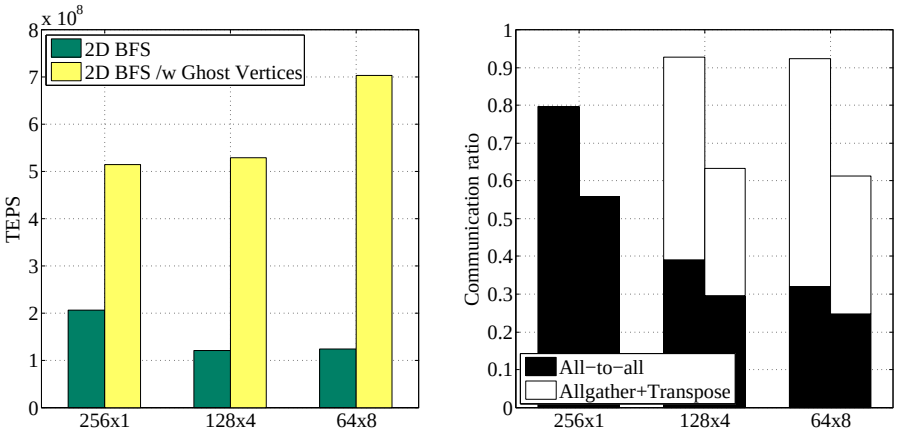
PageRank. The *init*() initializes an array d to $1.5/n$ which is now a PageRank array of local vertices. For the PageRank algorithm, all vertices are participated in computation in every iteration. Thus, every vertex is active, and there is no need to maintain a queue of active vertices. The all-gather communication is now gathering PageRank values along processor rows. The *discovery*() generates pairs (v, dv) where $v \in adj(u)$ for all local vertices u and $dv = d(u)/degree(u)$, and puts these pairs to a send buffer. The *update*() sums up all dv for each v, stores it to $d(v)$, and computes $d(v) = 1.5/n + 8.5 \times d(v)$. The loop termination is also replaced by checking whether overall RageRank is converged (i.e., there is no significant improvement in the PageRank values).

Modified Bellman-Ford SSSP. Similar to BFS, the *init*() initializes an array d to infinity, and updates the source vertex by setting its tentative distance to zero and putting it to a queue of active vertices. Note that d is now storing tentative distance of all local vertices, The *discovery*(), again, generates pairs (v, dv) where $v \in adj(u)$ for all active vertices u in the queue and $dv = d(u) + w(u, v)$, and puts these pairs to a send buffer. The *update*() updates a tentative distance of each v from the buffer of pairs (v, dv). If $d(v) > dv$, $d(v) = dv$. However, adding a new active vertex to the queue needs some extra work as it can cause redundancy. Only v that its $d(v)$ is recently updated is added to the queue, since this recently updated v will carry out its update to all of its adjacencies.

4.3.3 Optimizations

To further improve the performance of the graph algorithms with the 2D layout. We introduce several optimizations that helps avoid unnecessary work that can occur during communication and computation phases.

Ghost vertices. The technique of applying ghost vertices has been used in many distributed graph algorithms. The idea of the method is to locally cache any non-local vertex information that are adjacent to all local vertices. Thus, some vertices may have more than one copies across the processors. Although this technique

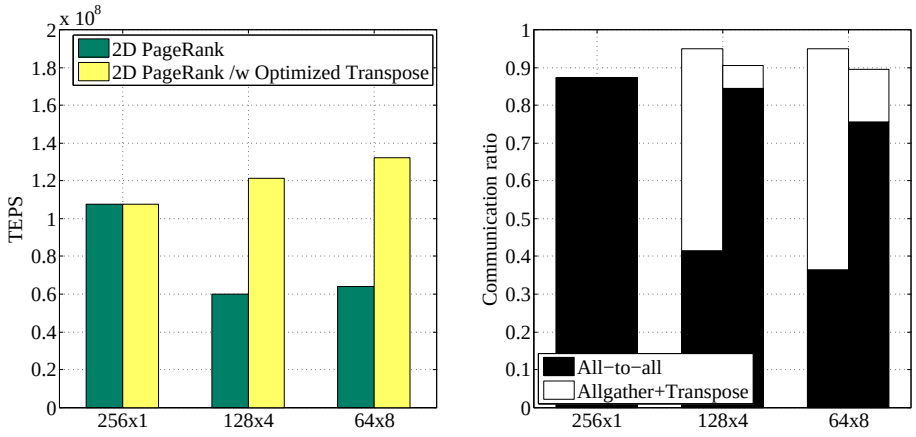


(a) TEPS (b) Communication

Figure 4.3. The performance of BFS with the 2D graph layout and ghost vertices on synthetic graphs generated using Graph500 with scale 27.

introduces more data redundancy to the graph data structure, some communication can be avoided as, in many cases, non-local vertex information can be retrieved from these ghost vertices. For example, BFS and SSSP algorithms can check their ghost vertices before sending the new distances of those vertices to the processors who own them. If the ghost vertices already have lower distances, the current distance can be discarded. Fig. 4.3 shows an example of the algorithm performance when the 2D graph layout and ghosting technique are used. The figures show a significant improvement of (a) TEPS and (b) communication ratio (e.g., lower all-to-all communication) of BFS on synthetic graphs generated using Graph500 with the edge scale of 27.

Optimized transpose. After involving the all-to-all communication along the processor columns in the 2D layout, the information are grouped so that each group belong to one processor. Even though this information is not yet to be local, it can be processed before transposing, to further reduce the size of messages. The idea is that this information can be aggregated to the size n/p which can be much smaller than the unprocessed information so that a high communication overhead can be avoid during the transposition (e.g., for PageRank, the size of unprocessed messages is proportional to the number of distributed edges which is much larger than n/p). We call this an optimized transpose, and it takes place before the



(a) TEPS (b) Communication

Figure 4.4. The performance of RageRank with the 2D graph layout and optimized transpose on synthetic graphs generated using Graph500 with scale 27.

transpose communication. This optimization is applied in every iteration of the PageRank algorithm since after the all-to-all communication, the message size is always much larger than n/p. However, in other algorithms, the optimization only triggers when the information after the all-to-all exchange is at least twice the size of n/p. Fig. 4.4 shows an example of the algorithm performance when applying the 2D graph layout and optimized transpose. The results are from the PageRank algorithm running on synthetic graphs generated using Graph500 with scale 27. There is 2x performance improvement in terms of TEPS (see Fig. 4.4(a)), and much lower all-gather and transpose communication ratio (see Fig. 4.4(b)).

4.4 Results and Discussion

4.4.1 Experimental Setup

Our experiments are run on StarCluster [122], a virtual cluster on top of Amazon Web Service (AWS) Elastic Compute Cloud (EC2) [7]. The cluster occupies with the MPICH2 complier version 1.4.1. We emulate using 32 instances of AWS EC2 m3.2xlarge which each consists of 8 cores of Intel Xeon E2-2670 v2 (Ivy Bridge) processors and 30 GB of memory.

The five algorithms used in our experiments, as mentioned in previous sections,

are breadth-first search (BFS), approximate diameter (APD), connected components (CNC), PageRank (PRK) and SSSP modified Bellman-Ford (MBF). In the experiments, both synthetic and real-world graphs are used. For synthetic graphs, they are generated from the Graph500 reference implementation [54]. The graph generator is based on the R-MAT random graph model with the parameters similar to those use in the default Graph500 benchmark (i.e. parameter a, b, c, and d are set to 0.59, 0.19, 0.19 and 0.05, respectively). We use graphs with scale 27 (i.e., 2^{27} vertices) and edge factor 16 (i.e., an average of 16 edges per vertex). For real-world graphs, we obtain two graphs from Standford Large Network Dataset Collection (SNAP) [119], it-2004 and sk-2005, which have approximately 41 million vertices and 1.1 billion edges, and approximately 50 million vertices and 1.9 billion edges, respectively. For the weighted graphs, we randomly generate the edge weights uniformly in [1 512] using Random123 library [100], and assign to the already existing graphs.

4.4.2 Performance of 2D Graph Layout

We show the performance of the five algorithms with the 2D graph layout with one, four and eight columns (labeled as $p \times 1$, $(p/4) \times 4$, and $(p/8) \times 8$, respectively) on 64, 128, and 256 MPI tasks. The results are shown in Fig. 4.5 in TEPS (i.e., the number of graph edges traversed per second). In all cases, increasing the number of columns from 1 to 8 improves the algorithm performance. The improvement can easily be seen on all algorithms when running on g500-s27. This graph also gives the best algorithm scalability comparing to other real-world graphs. For BFS, the performance of the algorithms with the 2D layout (with 4 columns) can be from 1.2× to close to 2× when comparing with the algorithms with the 1D layout. Both BFS and APD give very similar results since APD is basically the multiple runs of BFS. However, APD has lower scalability as it requires more synchronization between BFS runs. CNC, on the other hand, gives lightly lower performance than both BFS and APD as the algorithm involves more vertices during communication phases. However, the algorithm still gives good strong scalability on all graph types while APD shows some performance degradation on it-2004 and sk-2005. PRK show low performance on all graph types as it has high complexity in both computation (from computing PageRank values) and communication (from full traversing of

every vertices in each iteration). However, it gives very good strong scalability as the algorithm has high computation overhead (which is suited for parallel computing) compared to other algorithms. MBF gives the worst performance of all algorithms since the algorithm does not scale well on distributed machines as it has high redundant computation and communication.

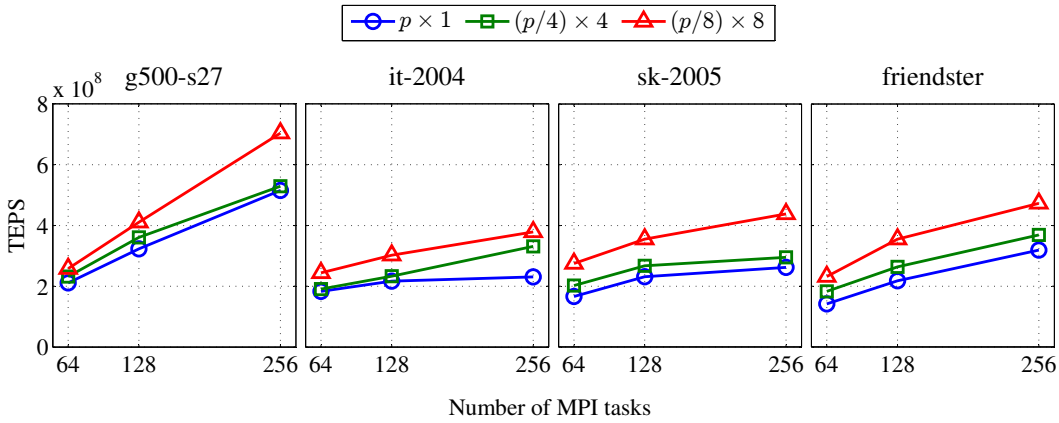
4.4.3 Communication Analysis

One of the advantages of the 2D graph layout is that it reduces communication space and the number of messages of the all-to-all communication which has high communication overhead. This improvement can be seen in all cases in Fig. 4.6. All algorithms except PageRank require less than 50% of the all-to-all communication when the 2D layout is used comparing to the communication of the 1D layout. BFS, APD and MBF show very similar results in terms of communication ratio. The overall communication increases when increasing the number of processors. However, it decreases when the number of processor columns changing from 4 to 8. For CNC, the communication ratio is mostly higher than those of BFS, APD and MBF since CNC is an algorithm with a fan-in approach that involves more vertices and leads to more communication overhead. The high communication from PRK can be expected as the algorithm has the highest communication comparing to others (i.e., full vertex traversals in every iteration). However, the 2D layout can be used to lower the all-to-all communication.

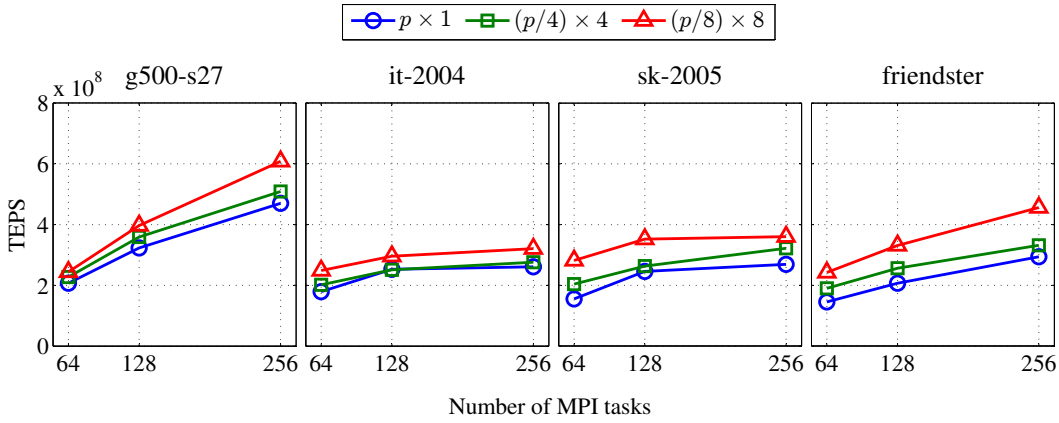
4.5 Conclusions and Future Work

We propose a distributed graph framework based on BSP with the 2D layout. Since our framework is synchronous, it is easy to apply to various iterative graph algorithms. With the use of the 2D layout, communication space and the number of messages are reduced. Furthermore, the edge distribution is also improved over the traditional 1D layout. Although the 2D layout adds some extra communication from all-gather and transpose communication, overall communication overhead of the algorithms is reduced resulting in a better algorithm scalability.

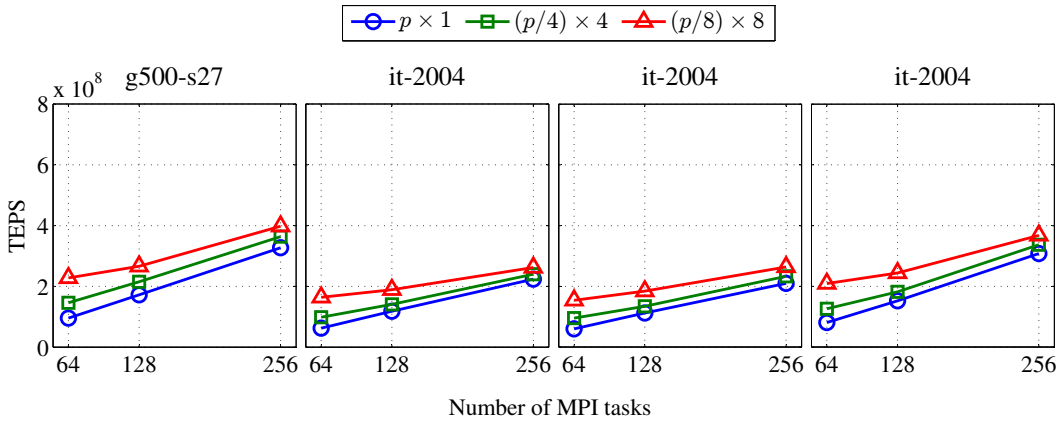
We show the performance of the 2D graph layout framework on five graph algorithms, breadth-first search, approximate diameter, connected components,



(a) Breadth-First Search (BFS)



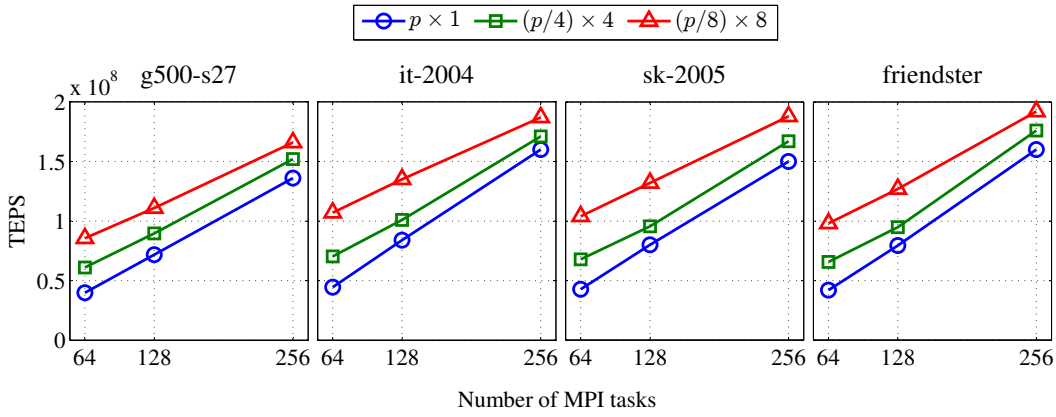
(b) Approximate Diameter (APD)



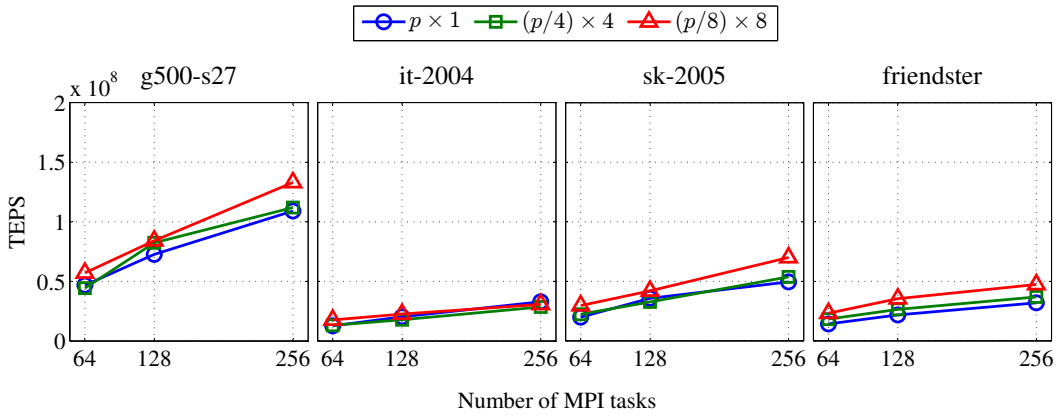
(c) Connected Components (CNC)

Figure 4.5. The performance of distributed graph algorithms with the 2D graph layout.

PageRank and modified Bellman-Ford SSSP, on a number of synthetic and real-



(d) PageRank (PRK)

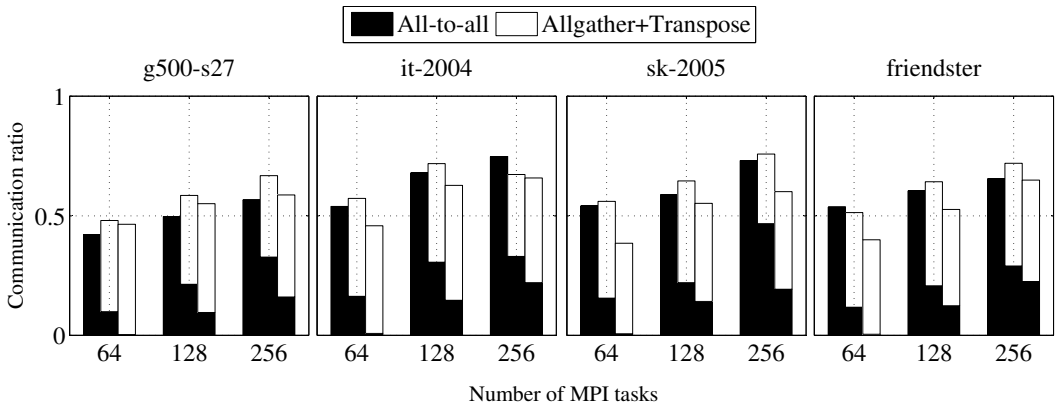


(e) Modified Bellman-Ford SSSP (MBF)

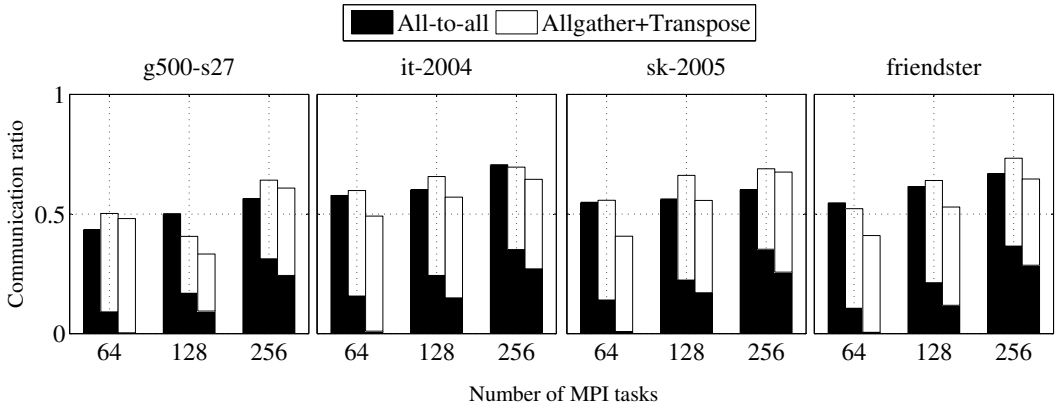
Figure 4.5. (Cont.) The performance of distributed graph algorithms with the 2D graph layout.

world graphs. Our experiment shows that the 2D graph algorithms give better performance than the 1D graph algorithms in all cases. The use of the 2D layout also significantly decreases the all-to-all communication which is the main communication overhead in parallel graph algorithms.

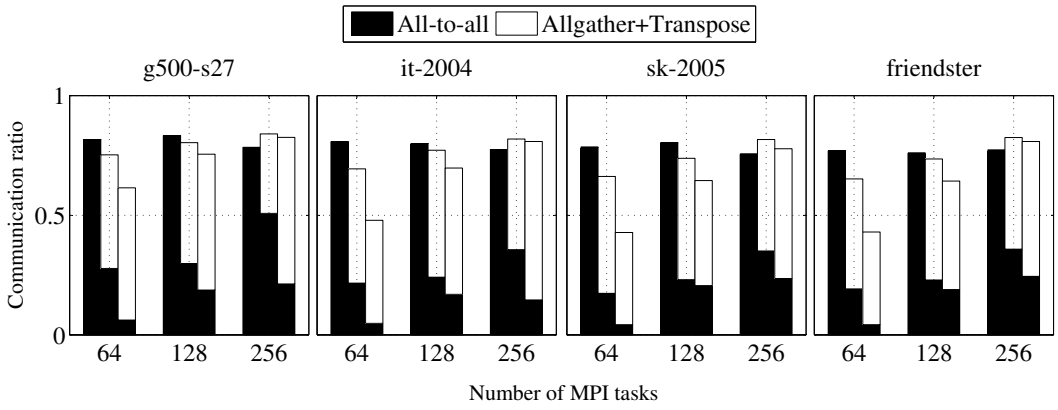
Currently, we only apply a simple distribution such that one partition is assigned to one processor. However, there are many approaches to map the 2D graph partitions to processors that can give various advantages. One method that we consider to apply in the future is to assign multiple partitions to one processors to increase the throughput of graph algorithms using the framework. With this over-partitioning method, it is possible to achieve better load balancing by overlapping some parts of computation and communication of the algorithm. The other mapping



(a) Breadth-First Search (BFS)



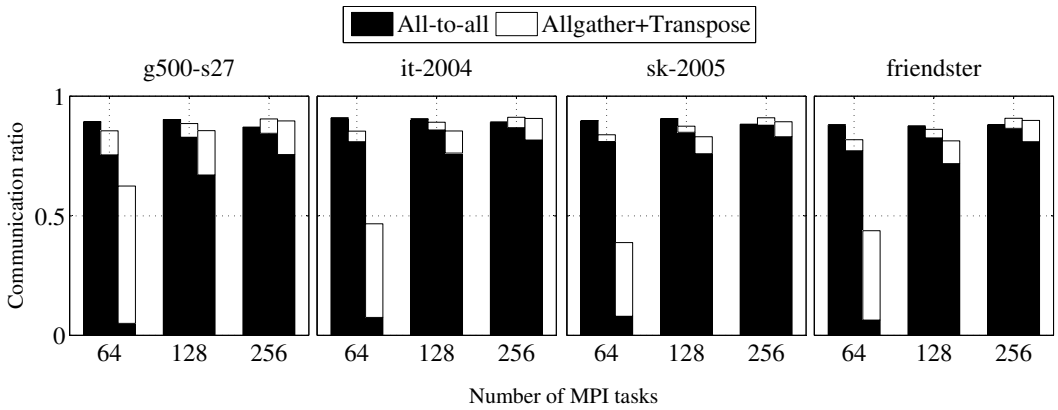
(b) Approximate Diameter (APD)



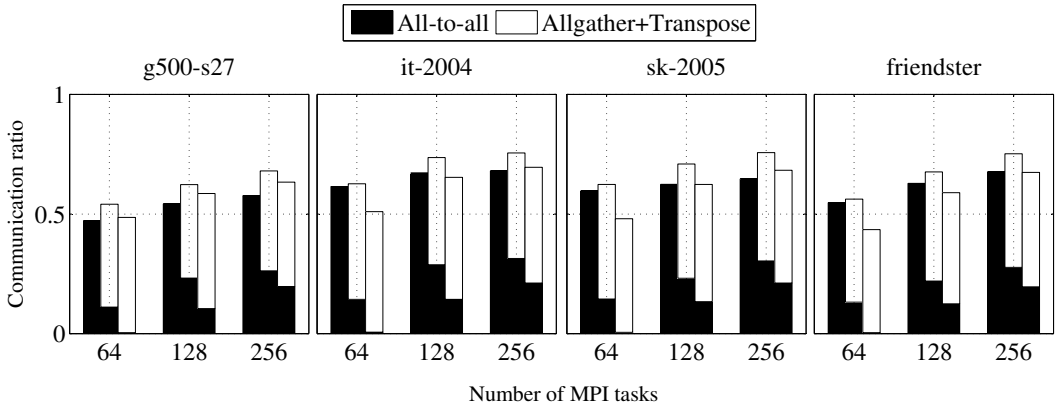
(c) Connected Components (CNC)

Figure 4.6. The breakdown communication of distributed graph algorithms with the 2D graph layout.

method is to assign a partition to each processor in some other ways instead of in



(d) PageRank (PRK)



(e) Modified Bellman-Ford SSSP (MBF)

Figure 4.6. (Cont.) The breakdown communication of distributed graph algorithms with the 2D graph layout.

order as it can possibly give better overall communication results. We also consider implement our framework to support shared-memory machines to further improve overall communication on those architectures.

Chapter 5
Parallel Single Source Shortest Path Algorithms

With the advance of online social networks, World Wide Web, e-commerce and electronic communication in the last several years, data relating to these applications has become exponentially larger. These data are usually analyzed using graphs modeling relations among data entities. However, processing these graphs is challenging not only from a tremendous size of the graphs that is usually in terms of billions of vertices and trillions of edges, but also from graph characteristics such as sparsity, irregularity and scale-free degree distributions that are difficult to manage.

Large-scale graphs are commonly stored and processed across multiple machines or in distributed environments due to a limited capability of a single machine (i.e., limited availability of memory to process an entire graph). However, current graph analyzing tools which have been optimized and used on sequential systems cannot directly be used on these distributed systems without scalability issues. Thus, novel graph processing and analysis are required, and efficient parallel graph computations are mandatory to handle these large-scale graphs effectively.

Single-Source Shortest Path (SSSP) is a well-known graph computation that has been studied for more than half a century. It is one of the most common graph analytical analysis for many graph applications such as networks, communication, transportation, electronics and so on. There are many algorithms that have been proposed such as the well-known Dijkstra's algorithm [39] and Bellman-Ford [11,44]. However, these algorithms are designed for serial machines, and do not efficiently work for parallel environments. As a result, many researchers have been studied

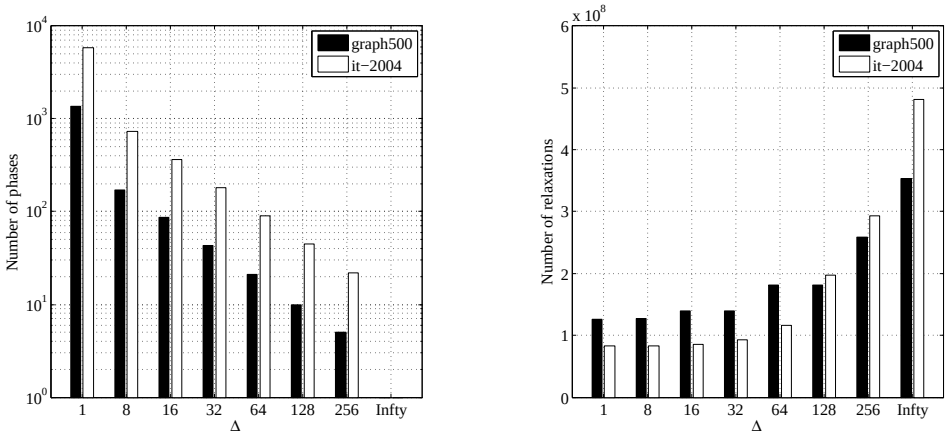
and proposed parallel SSSP algorithms or implemented SSSP as parts of their parallel graph frameworks. Some well-known graph libraries and frameworks include the Parallel Boost Graph Libray [55], GraphLab [79, 80] and PowerGraph [51], Galois [45] and ScaleGraph [36]. More recent frameworks have been proposed based on Hadoop sytems [133] such as Cyclops [27], GraphX [136] and Mizan [70]. For standalone implementations of SSSP, most recent implementations usually are for GPU parallel systems such as [35, 131, 138]. However, high performance GPU architectures are still not widely available and they also require good CPUs to speed up the overall performance. Other environments that SSSP have been developed up on are shared-memory architectures. These SSSP implementations include [84, 99, 115].

In this chapter, we focus on designing and implementing efficient SSSP algorithms for distributed-memory systems since there are only few SSSP implementations for this type of architectures. We aware of a recent SSSP study of Chakaravarthy et al. [24] that is proposed for massively parallel systems, IBM Blue Gene/Q (Mira). Their SSSP implementations have applied various optimizations and techniques to achieve very good performance such as direction optimization (or a push-pull approach), pruning, vertex cut and hybridization. However, many techniques used are specifically for SSSP algorithms or a limited variety of graph algorithms. In our case of SSSP implementations, most of our techniques are more flexible and can be extended to other graph algorithms while still achieving good performance. We also experiment on various graph types and distributed-memory environments to see how well our algorithms perform in various conditions. Our main contributions include:

- Novel SSSP algorithms that combine advantages of various well-known SSSP algorithms.
- A utilization of a two-dimensional graph layout to reduce communication overhead and improve load balancing of SSSP algorithms.
- A distributed cache-like optimization that filters out unnecessary SSSP updates and communication to further increase the overall performance of the algorithms.
- A detailed evaluation of the SSSP algorithms on various large-scale graphs

and environments.

5.1 Case Study: SSSP Algorithm Performance



(a) Number of phases (b) Number of relaxations

Figure 5.1. The numbers of phases and relaxations of the Δ-stepping algorithm with different Δ.

Let $G = (V, E)$ be a graph with $n = |V|$ vertices and $m = |E|$ directed edges with non-negative weights. (For the case of undirected edges, it can be viewed as directed edges pointing in both directions.) In general, the input to SSSP is a source vertex, $s \in V$. SSSP finds the shortest paths between s and each $v \in V$. Most SSSP algorithms usually are based on two classical approaches, label-setting and label-correcting. For the label-setting, each vertex is relaxed only once, and is marked as settled after its relaxation. The relaxation order is a key of this approach to maintain the algorithm validation. The Dijkstra's algorithm [39] belongs to this category with the use of a priority queue that keeps all active vertices in the queue in ascending order based on their distances from s. The algorithm is work-efficient, but provides very limited concurrency to exploit. The label-correcting approach is, on the other hand, more amenable to parallelization while it is not work-optimal. Each vertex may be relaxed multiple times and in any order since there is no priority given to the relaxation order of active vertices. The Bellman-Ford algorithm [11] is an example of this approach. The algorithm relaxes n vertices in n iterations, and each vertex may be relaxed more than once. The Δ-stepping algorithm [88] is a special SSSP algorithm that utilizes both label-setting and label-correcting

approaches to compromise between work-efficiency and concurrency in the algorithm with the use of a Δ parameter and Δ-size buckets. The relaxation of vertices in each bucket is based on label-correcting while the bucket itself is based on label-setting. Thus, adjusting the size of Δ determines the tradeoff between work-efficiency and concurrency.

Fig. 5.1 shows (a) numbers of phases and (b) numbers of edge relaxations with different Δ values on a synthetic graph generated using Graph500 with scale 27 (g500-s27) and a real-world graph (it-2004). The performance of the Dijkstra's algorithm and Bellman-Ford are shown with Δ equal to 1 and Infty, respectively, as they are equivalent algorithms. Increasing Δ results in decreasing the number of phases as buckets with larger Δ can hold and relax more vertices than smaller Δ buckets. Thus, the larger Δ, the more concurrency. However, increasing Δ also yields higher numbers of relaxations as it increases a chance of each vertex to be relax multiple times. Thus, the larger Δ, the less work efficiency. Note that it-2004 requires higher numbers of phases to process because it has a larger graph diameter than g500-s27.

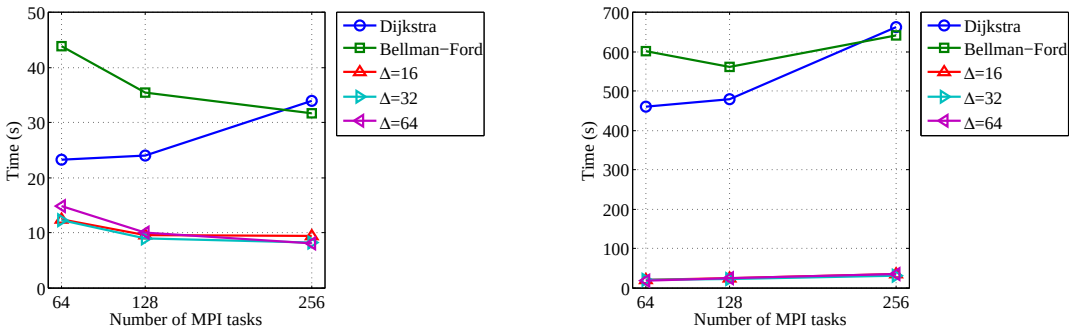
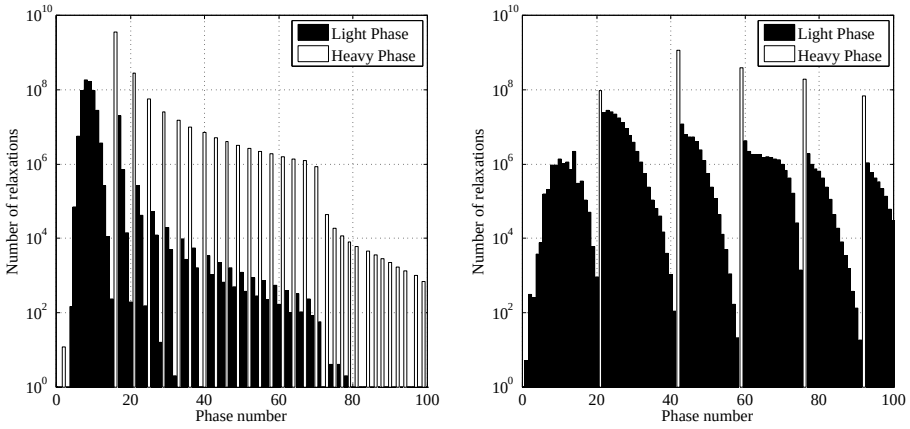


Figure 5.2. The execution time of the Dijkstra's, Bellman-Ford and Δ-stepping algorithms on a synthetic graph generated using Graph500 with scale 27 (g500-s27) and a real-world graph (it-2004).

The performance of the three distributed algorithms, Dijkstra's, Bellman-Ford and Δ-stepping algorithms, with different Δ on the g500-s27 and it-2004 graphs is shown in Fig. 5.2. For g500-s27, Bellman-Ford provides high algorithm concurrency, and it gives better performance scalability when the numbers of processors increase. In contrast, the Dijkstra's algorithm shows very poor performance scalability since it has very low algorithm concurrency. Fortunately, Δ-stepping can combine the

advantages of both Dijkstra's and Bellman-Ford algorithms resulting in much better performance and scalability of the algorithm. For it-2004 that has a high graph diameter, both Dijkstra's and Bellman-Ford algorithms give very poor performance. On the other hand, Δ-stepping still shows much better performance than the other two algorithms even though Δ-stepping does not show good scaling for this graph.



(a) g500-s27 (b) it-2004

Figure 5.3. The numbers of relaxations on the first 100 phases of the Δ-stepping algorithm with $\Delta = 32$.

Fig. 5.3 shows the numbers of relaxations on the first 100 phases of Δ-stepping algorithm with $\Delta = 32$ on g500-s27 and it-2004. Note that g500-s27 and it-2004 require 123 and 954 phases to complete the full SSSP execution, respectively. The results from both graphs show that the first half of the algorithm execution is dominated by light phase relaxations while the later half is dominated by heavy phase relaxations. During early phases, most vertices of the graphs are still unsettled, and this results in more work in light phases. At some points when a large portion of vertices are settled, more work is shifted to heavy phases as the number of insertions of unsettled vertices to current buckets decreases.

5.2 Novel Parallel SSSP Implementations

5.2.1 General Parallel SSSP for Distributed-Memory Systems

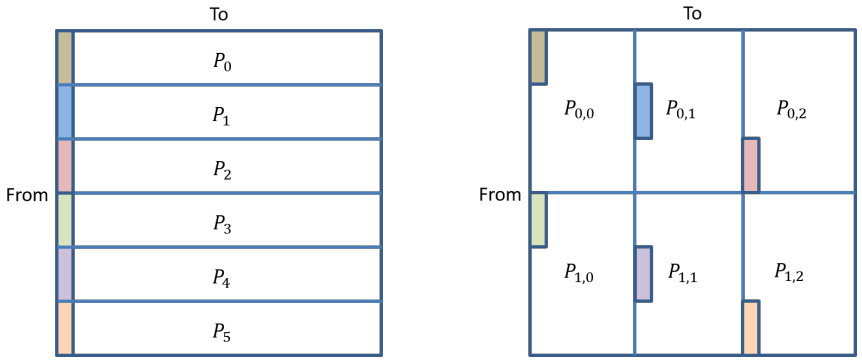
Our SSSP implementation is based on a bulk-synchronous Δ-stepping algorithm for distributed-memory implemented in [94]. The algorithm composes of three main steps, a local discovery, an all-to-all exchange and a local update for both light and heavy phases. In the local discovery step, each processor looks up to all adjacencies v of its local vertices u in the current bucket, and generates corresponding tentative distances $dtv = d(u) + w(u, v)$ of those adjacencies. Note that, in the light phase, only adjacencies from light edges are considered, while, in the heavy phase, only adjacencies from heavy edges are processed. For each edge uv, a pair (v, dtv) is generated, and is added to a queue called QRequest. The all-to-all exchange step distributes these pairs in QRequest to make them local to processors so that each processor can use these information to update a local tentative distance list in the local update step. An edge relaxation is part of the local update step that involves updating vertex tentative distances and adding/removing vertices to/from buckets based on their current distances.

There are some additional optimizations to the data structure and algorithm. The full list of the optimizations is shown in [94].

5.2.2 Parallel SSSP with 2D graph layout

We consider a two-dimensional (2D) graph layout previously studied in [19] for breadth-first search. This approach optimizes the underlying graph data structures, and gives an order of magnitude improvement over general distributed graph frameworks without fine-tuning any specific graph algorithm. Thus, it can be applied to any distributed graph algorithm efficiently and effectively, and can be implemented as an efficient framework for distributed graph computations. The idea of this approach is to partition a sparse adjacency matrix of a graph into grid blocks of p_r rows and p_c columns, and to force each inter-processor communication to occur only on one dimension (either row or column) at a time. Thus, it reduces the inter-processor communication space. If we partition a sparse adjacency matrix of a graph with n vertices into $p = p_r \times p_c$ partitions, with the traditional 1D layout, each set of n/p consecutive rows of the matrix is assigned to one partition

(see Fig. 5.4(a)). Alternatively, we can partition the adjacency matrix into grid blocks, and assign each block to one partition or processor (see Fig. 5.4(b)). With $p = p_r \times p_c$ processors, the communication space can be reduced from $p_r \times p_c$ to p_r for the all-to-all communication. Furthermore, this approach provides better load balancing of edges of a graph than the 1D layout approach as any dense row of a high degree vertex can now be distributed across multiple processors instead of only one processor as in the 1D layout.

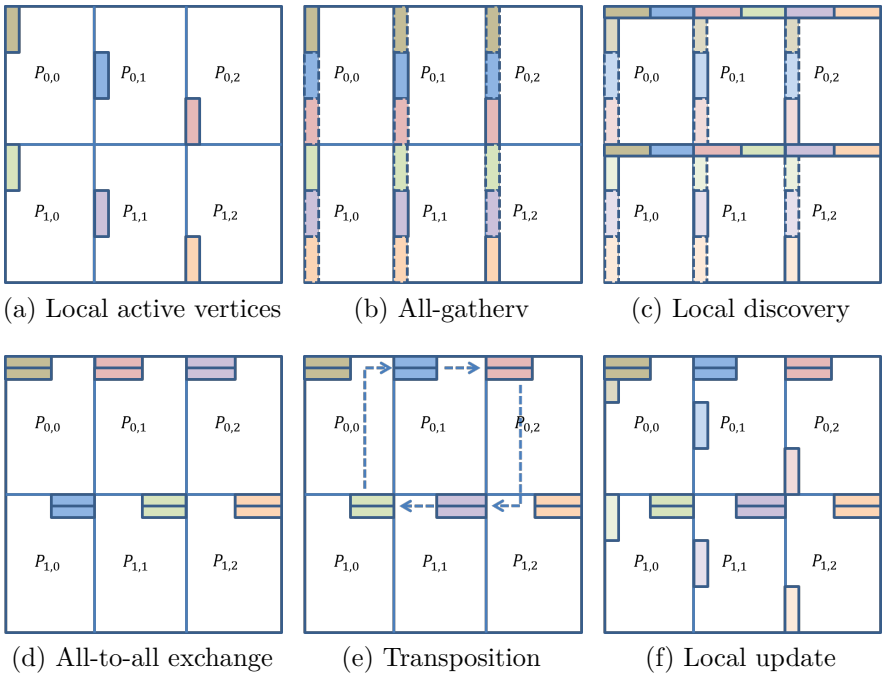


(a) With 1D layout (b) With 2D layout

Figure 5.4. The graph distributions by partitioning the equivalent adjacency matrix of the graph.

To apply the 2D graph layout to the Δ-stepping algorithm, each of the three steps needs to be modified according to the changes in the vertex and edge distributions. While the vertices are distributed in similar manner as in the 1D graph layout, edges are now distributed differently. Previously in the 1D layout, all outgoing edges of local vertices are assigned to one processor. However, with the 2D layout, these edges are now distributed among row processors that have the same row rank. Fig. 5.5(a) illustrates the partitioning of vertices and edges for the 2D layout.

In the local discovery step, there is no need to modify the original routine. The only work that needs to be done is by adding a pre-processing phase that merges all current buckets along the processor rows by using a row-wise all-gather operation. The goal is to gather all active vertices that belong to the same processor rows since edge information of one vertex is distributed to all processors in the same processor rows. Thus, each processor requires to know all active vertices that are in the current bucket of their neighbor processor rows before the discovery step can take place. After the current bucket is merged (see Fig. 5.5(b)), each processor can now simultaneously work on generating pairs (v, dtv) of its local active vertices



(a) Local active vertices (b) All-gatherv (c) Local discovery

(d) All-to-all exchange (e) Transposition (f) Local update

Figure 5.5. The main SSSP steps when applying the 2D layout.

(see Fig. 5.5(c)).

In the all-to-all exchange step, the purpose of this step is to distribute the generated pairs (v, dtv) to the processors that are responsible to maintain those data relating to vertices v. In our implementation, we use two sub-communications, a column-wise all-to-all exchange and a send-receive transposition. The column-wise all-to-all puts all information pairs that belong to the same owner onto one processor. Fig. 5.5(d) shows a result of this all-to-all exchange operation. After that, each processor sends and receives these pair lists to the actual owner processors. The latter communication can be viewed as a matrix transposition as shown in Fig. 5.5(e).

In the local update step, the original routine can be kept the same, but the data structure of the buckets needs to be changed. In the 2D layout, local tentative distances of each processor only correspond to a block of adjacencies and does not include all adjacencies of one vertex. Instead of only storing active vertices in buckets, both vertices and their current tentative distances need to be stored in the buckets so that each processor knows the distance information without initiating any other communication when the buckets are merged in the next local

discovery step. Fig. 5.5(f) illustrates this local update step. Since all pairs (v, dtv) are local, each processor can update the tentative distances of their local vertices simultaneously.

Algorithm 6 Distributed SSSP with the 2D Graph Layout

```
 1: for local u do
 2:    d[u] = ∞
 3: end for
 4: current = 0
 5: if onwer(s) = rank then
 6:    d[s] = 0
 7: end if
 8: if onwerRow(s) = rankRow then
 9:    add pair (s, 0) to Bucket[current]
10: end if
11: while Bucket ≠ ∅ do {Globally check}
12:    while Bucket[current] ≠ ∅ do {Globally check}
13:       for pair (u, du) in Bucket[current] do
14:          for light edge (u, v) do
15:             dtv = du + w(u, v)
16:             add pair (v, dtv) to QRequest
17:          end for
18:          add pair (u, du) to QHeavy
19:       end for
20:       Alltoallv (row-wise) and Transpose of QRequest
21:       for pair (v, dtv) in QRequest do
22:          Relax(v, dtv)
23:       end for
24:       Allgatherv (column-wise) of Bucket[current]
25:    end while
26:    for pair (u, du) in QHeavy do
27:       for heavy edge (u, v) do
28:          dtv = du + w(u, v)
29:          add pair (v, dtv) to QRequest
30:       end for
31:    end for
32:    Alltoallv (row-wise) and Transpose of QRequest
33:    for pair (v, dtv) in QRequest do
34:       Relax(v, dtv)
35:    end for
36:    current = current + 1 {Move to next bucket}
37:    Allgatherv (column-wise) of Bucket[current]
38: end while
```

Our complete SSSP algorithm with the 2D graph layout is shown in Alg. 6. The processing of light and heavy phases is shown in lines 9-18 and lines 19-27, respectively. Alg. 7 shows the relaxation procedure used in Alg. 6 (lines 22 and 34).

Algorithm 7 Vertex Relaxation for Distributed SSSP

```
Relax(v, dtv)
if d[v] > dtv then
   old = d[v]/Δ, new = dtv/Δ
   Remove pair (v, d[v]) from Bucket[old]
   Add pair (v, dtv) to Bucket[new]
   d[v] = dtv
end if
```

5.2.3 Other Optimizations

To further improve the algorithm performance, we apply other three optimizations, a cache-like optimization, a heuristic Δ increment and a direction optimization. The detailed explanation is as follows.

Cache-like Optimization: We maintain a tentative distance list of all unique adjacencies of the local vertices as a local cache. This list holds the recent values of tentative distances of all unique adjacent vertices. When a new tentative distance is generated (during the discovery step), this newly generated distance is compared to the local copy in the list. If the new distance is shorter, it will be processed in the regular manner by adding the generated pair to the QRequest, and the local copy in the list is updated to this value. However, if the new distance is longer, it will be discarded since the remote processors will eventually discard the value during the relaxation anyway. Thus, with a small tradeoff of additional data structures and computations, this approach can significantly avoid unnecessary work that involves both communication and computation in later steps.

Heuristic Δ Increment: The idea of this optimization is from an observation of the Δ-stepping algorithm that the algorithm provides a good performance in early iterations when a small Δ is used since it can avoid a large portion of redundant work during processing light phases. Meanwhile, with a larger Δ, the algorithm provides a good performance in later iterations since most vertices of the graph are settled so that a portion of redundant work is low. Thus, more algorithm concurrency provides more benefit. In other words, a small Δ results in low overhead during light phases, but leads to high overhead during heavy phases. On the other hand, a large Δ decreases work in heavy phases, but increases work in

light phases. The algorithm with Δ that can be adjusted when needed can provide better algorithm performance. Thus, from this observation, instead of using a fixed value of Δ, our heuristic optimization processes by starting with a small value of Δ. Once some thresholds are met, the value of Δ is then increased (usually to ∞) to speed up the later iterations.

Direction Optimization: This optimization is a heuristic approach first introduced in [10] for breadth-first search (BFS). Conventional BFS algorithms usually proceed in an top-down approach such that, in every iteration, each vertex in a frontier checks all of its neighbors whether they are not yet visited, adds them to the frontier, and then marks them as visited. The algorithm terminates whenever there is no vertex in the frontier. The algorithm performance is based on the total number of adjacent vertices of all vertices in this frontier. The more adjacencies, the more work that needs to be done. Furthermore, if there are some high degree vertices that need to be processed in later iterations, most work will be discarded since most adjacencies of those vertices are already settled. From this observation, the bottom-up approach can come to play for efficiently processing of vertices in the frontier. Instead of using the top-down approach, it can be done in the reverse direction. To avoid processing large numbers of adjacent vertices, the algorithm checks unvisited vertices and marks them visited if their neighbors are in the frontier. With a heuristic determination, the algorithm can alternately switch between top-down and bottom-up approaches to achieve an optimal performance. Since the discovery step in SSSP is done in similar manner as BFS, Chakaravarthy et. al. [24] adapts a similar technique called a push-pull heuristic to their Δ-stepping-based SSSP algorithms. The algorithms proceed with a push (similar to the top-down approach) by default during the early stages of the execution. If the forward communication volume of the current bucket is greater than the request communication volume of all later buckets combined, the algorithms switch to the pull approach. This push-pull heuristic considerably improves an overall performance of the algorithm. The main reason of the improvement is because of the lower communication volume, thus, the consequent computation also decreases.

5.2.4 Summary of implementations

In summary, we implement five SSSP algorithms:

1. **SP1a:** The SSSP algorithm based on Δ-stepping with the cache-like optimization
2. **SP1b:** SP1a with the direction optimization (push-pull heuristic)
3. **SP2a:** SP1a with the 2D graph layout
4. **SP2b:** SP2a with the Δ increment heuristic

The main difference among the algorithms is the level of optimizations that varies from 1D to 2D graph layouts and some heuristic approaches introduced in each algorithm.

5.3 Performance Results and Analysis

5.3.1 Experimental Setup

Our experiments are run on a virtual cluster using StarCluster [122] with the MPICH2 complier version 1.4.1 on top of Amazon Web Service (AWS) Elastic Compute Cloud (EC2) [7]. We use 32 instances of AWS EC2 m3.2xlarge. Each instance consists of 8 cores of high frequency Intel Xeon E5-2670 v2 (Ivy Bridge) processors with 30 GB of memory. We also experiment on a smaller computing cluster, CyberStar [34], which is available for our use at The Pennsylvania State University. The CyberStar cluster consists of 192 Dell PowerEdge R610 severs. Each provides two quad-core Intel Nehalem processors running at 2.66 GHz with 24 GB of RAM. On this system, the codes are complied using OpenMPI version 4.8.2. We use up to 128 cores of the CyberStar cluster. Note that the reason of most of our experiments are done on Amazon EC2 is because of the availability of numbers of processors that we can use (up to 256). Although CyberStar provides a better performance for high performance computing, the numbers of processors that we can use is very limited (up to 128).

The graphs that we use in our experiments are listed in Tab. 5.1. The g500-s27 is a synthetic graph generated from the Graph500 reference implementation [54]. The

Table 5.1. The list of graphs that are use in our experiments.

Graph	Vertices	Edges	Reference
g500-s27	134 M	2.1 B	[54]
it-2004	41 M	1.1 B	[15]
sk-2005	50 M	1.9 B	[15]
friendster	65 M	1.8 B	[137]
orkut	3 M	117 M	[137]
livejournal	4 M	68 M	[137]

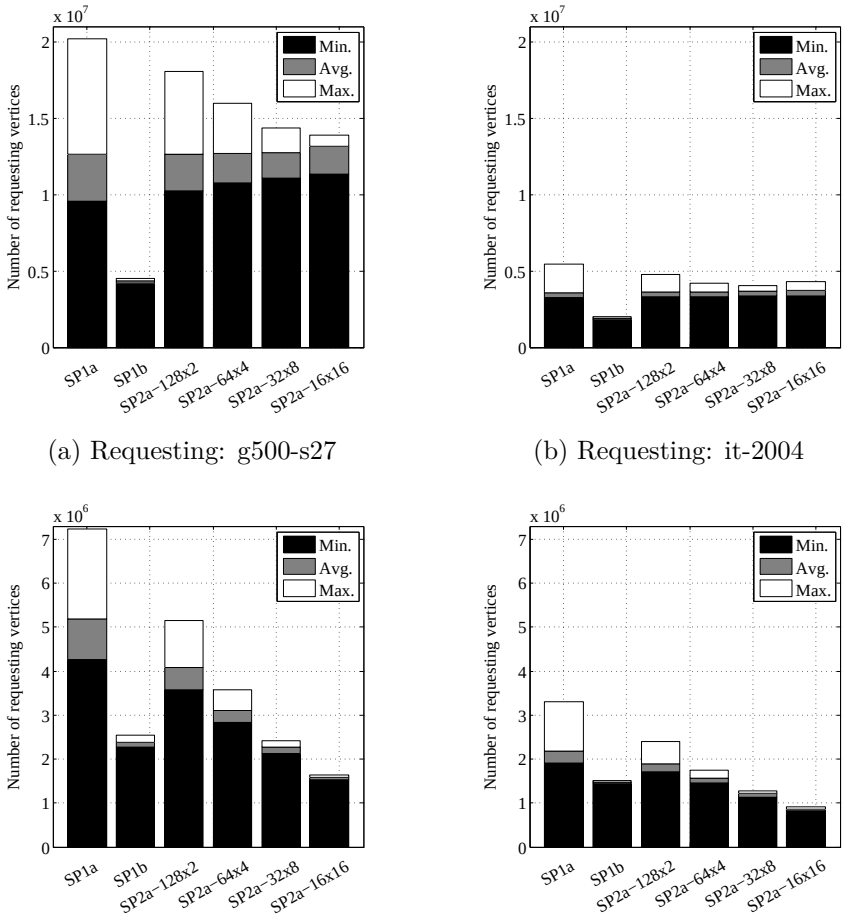
graph generator is based on the R-MAT random graph model with the parameters similar to those use in the default Graph500 benchmark (i.e., parameter a, b, c and d are set to 0.59, 0.19, 0.19 and 0.05, respectively). In this experiment, we use the graph scale of 27 (i.e., 2^{27} vertices) with edge factor of 16 (i.e., an average of 16 degrees per vertex). The other six graphs are real-world graphs that are obtained from Standford Large Network Dataset Collection (SNAP) [119] and The University of Florida Sparse Matrix Collection [120]. The edge weights of all graphs are randomly, uniformly generated in [1 512] using Random123 library [100].

We fix the value of Δ to 32 for all algorithms. Please note that this Δ might not be the optimal value in all test cases, but, in our initial experiments on the systems, it gives good performance most of the time. To get the optimal performance in all cases is a challenging problem since since Δ needs to be changed accordingly to the systems (such as CPU, network bandwidth and latency) and the graph (such as size, type of distribution and number of partitions). For more discussion about the Δ value that can affect the performance of the Δ-stepping algorithm, please see [94].

5.3.2 Algorithm and Communication cost Analysis

The 2D layout improves the performance of the SSSP algorithms by decreasing the (all-to-all) communication space and improving load balancing. In this subsection, we further look into the performance of the algorithms with the 2D layout on different numbers of columns.

In the 2D layout, when the number of columns increases, it further decreases the all-to-all communication overhead, and improves the edge distribution among partitions. Consider processing a graph with n vertices and m edges using $p = r \times c$ processors. The all-to-all and all-gather communication spaces are usually



(a) Requesting: g500-s27

(b) Requesting: it-2004

(c) Sending: g500-s27

(d) Sending: it-2004

Figure 5.6. The number of (a,b) requesting and (c,d) sending vertices to be relaxed during the highest relaxation phase of the Δ-stepping algorithm with the 2D layout on g500-s27 and it-2004 using different combinations of processor rows and columns on 256 MPI tasks.

proportional to r and c, respectively. The maximum number of messages for each all-to-all communication is m/c while the maximum number of messages for each all-gather is n/r. In each phase, processor $p_{i,j}$ requires to interact with processors $p_{k,j}$ for the all-to-all communication where $1 \leq k \leq r$, and with processors $p_{i,l}$ for the all-gather communication where $1 \leq l \leq c$. By setting $r = 1$ and $c = p$, the algorithms do not need any all-to-all communication, but the all-gather communication now requires all processors to participate.

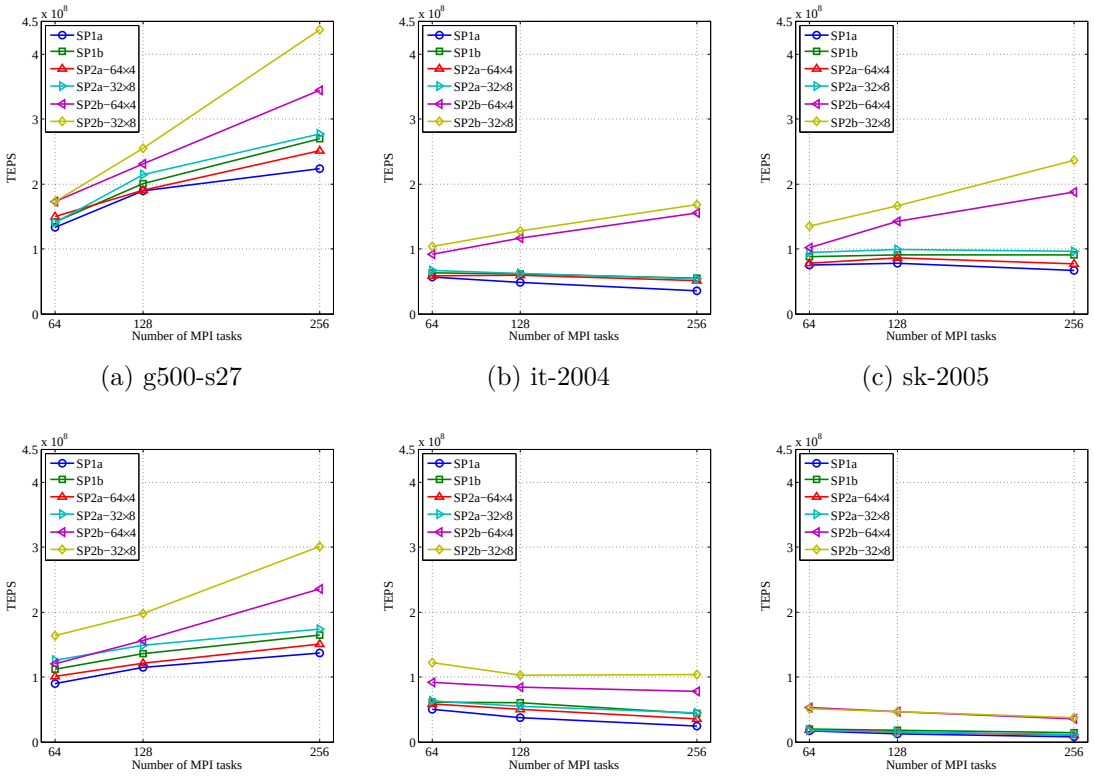
When running the SSSP algorithms on scale-free graphs, a majority of the entire computation and communication are usually spent on few phases of the

algorithms that require to process on very high degree vertices. The Fig. 5.6 shows the average, minimum and maximum vertices to be (a,b) requested and (c,d) sent for relaxation during one of these phases that consumes the most time of the algorithm execution of SP1a, SP1b and SP2a on g500-s27 and it-2004 with 256 partitions. Note that we use the abbreviation SP2a-R×C for the SP2a algorithm with R and C processor rows and columns, respectively. For example, SP2a-64×4 is the SP2a algorithm with 64 row and 4 column processors (e.g., 256 processors in total). The improvement of load balancing of the requested vertices for relaxation can easily be seen in Fig. 5.6(a,b) as the minimum and maximum numbers of the vertices decrease on both graphs from SP1a to SP1b and SP1a to SP2a. The improvement from SP1a to SP1b is significant as the optimization is specifically implemented for this issue (i.e., the overhead during the high-requested phases) by switching from the push to pull method as the numbers of vertices involved in these phases are much lower. On the other hand, SP2a still processes the same number of vertices, but in lower communication space and better load balancing. Not only load balancing of the communication improves, but the numbers of (average) messages among inter-processors also reduce as we can see in Figs. 5.6(c,d). However, there are some limitations of both SP1b and SP2a. For SP1b, the push-pull heuristic may not trigger in some phases that have high computation and communication if the costs of both push and pull approaches are not much different. For SP2a, although increasing numbers of columns can improve load balancing of distributed edges and decrease (all-to-all) communication in every phase, the all-gather communication also increases linearly as the number of columns increases. There is no optimal number of columns that gives the best algorithm performance since it depends on various factor such as the number of processors, the size of the graph and other system specifications.

5.3.3 Benefits of 2D SSSP Algorithms

We experiment on six different graphs to see how each algorithm performs under various circumstances. Fig. 5.7 shows the algorithm performance on AWS EC2 up to 32 nodes of 8-core m3.2xlarge (or up to 256 MPI tasks). Although SP1b can significantly reduce computation and communication during the high-requested phases, its overall performance is similar to SP2a. The SP2b algorithm gives

the best performance in all cases, and it also gives the best algorithm scalability when numbers of processors increase. The peak performance of SP2b-32×8 of approximately 0.45 GTEPS can be observed on g500-s27 with 256 MPI tasks which is approximately 2× faster than the performance of SP1a on the same setup. The SP2b algorithm also shows good scaling on large graphs (e.g., g500-s27, it-2004, sk-2005 and friendster).



(a) g500-s27 (b) it-2004 (c) sk-2005

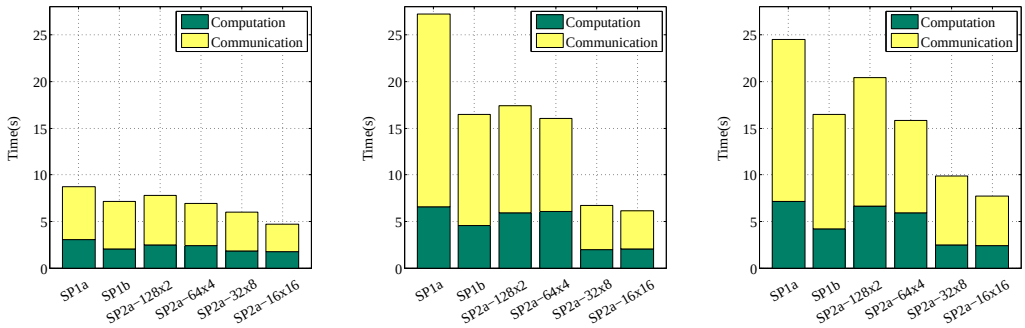
(d) friendster (e) orkut (f) livejournal

Figure 5.7. The performance of SSSP algorithms on six graphs with up to 256 MPI tasks.

5.3.4 Communication Cost Analysis

The 2D graph layout helps reducing the communication volume and space, and improving load balancing. Fig. 5.8 shows the execution time of each algorithm in terms of computation and communication on six graphs. More than half of the time for all algorithms is spent on communication as the networks of AWS EC2 is not optimized for data transferring for high performance parallel computing. The

improvement of SP1b over SP1a is from the reduction of computation overhead as numbers of processing vertices in some phases are reduced. On the other hand, SP2a has lower communication overhead than the overhead of SP1a as the communication space is decreased from the use of the 2D layout. The SP2b algorithm further improves the overall performance by introducing more concurrency in the later phases resulting in lower both computation and communication overhead during the algorithm executions.

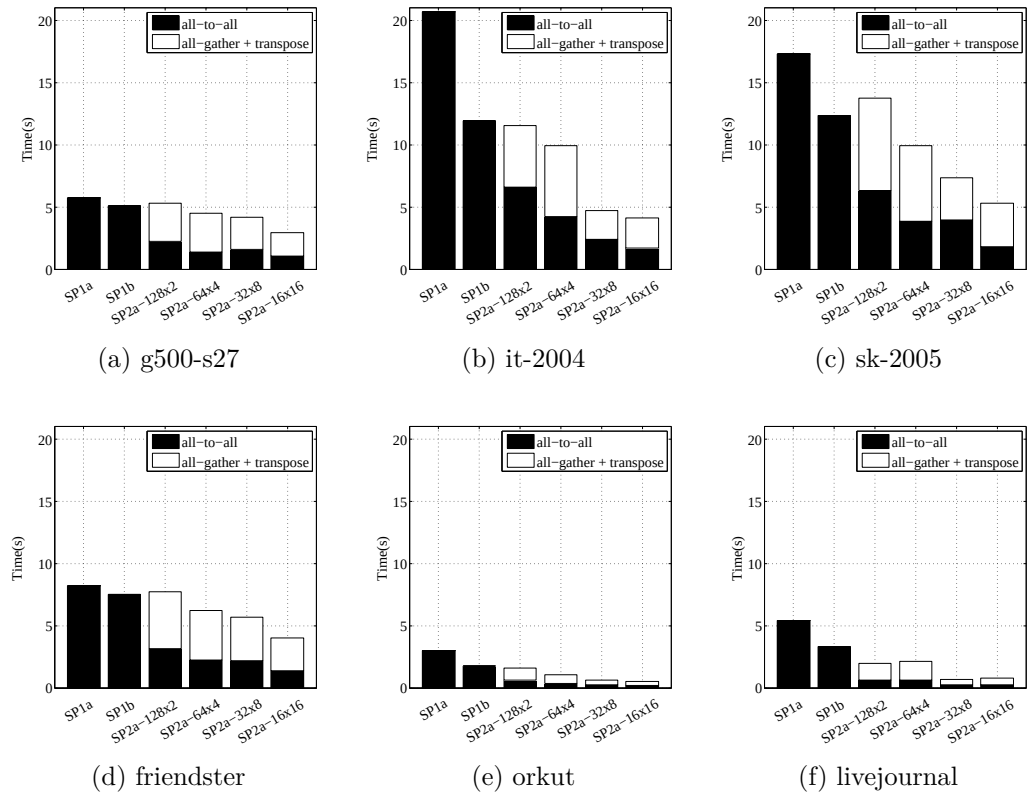


(a) g500-s27 (b) it-2004 (c) sk-2005

(d) friendster (e) orkut (f) livejournal

Figure 5.8. Communication and computation time of SSSP algorithms on 256 MPI tasks.

Fig. 5.9 shows the communication time break down of all algorithms. We can see that when the number of processor rows increases, it affects the communication by decreasing the all-to-all communication, and slightly increasing the all-gather and transpose communication. In all cases, the SP2b algorithm shows the least communication overhead with up to 10× faster for the all-to-all communication and up to 5× faster for the total communication.



(a) g500-s27 (b) it-2004 (c) sk-2005

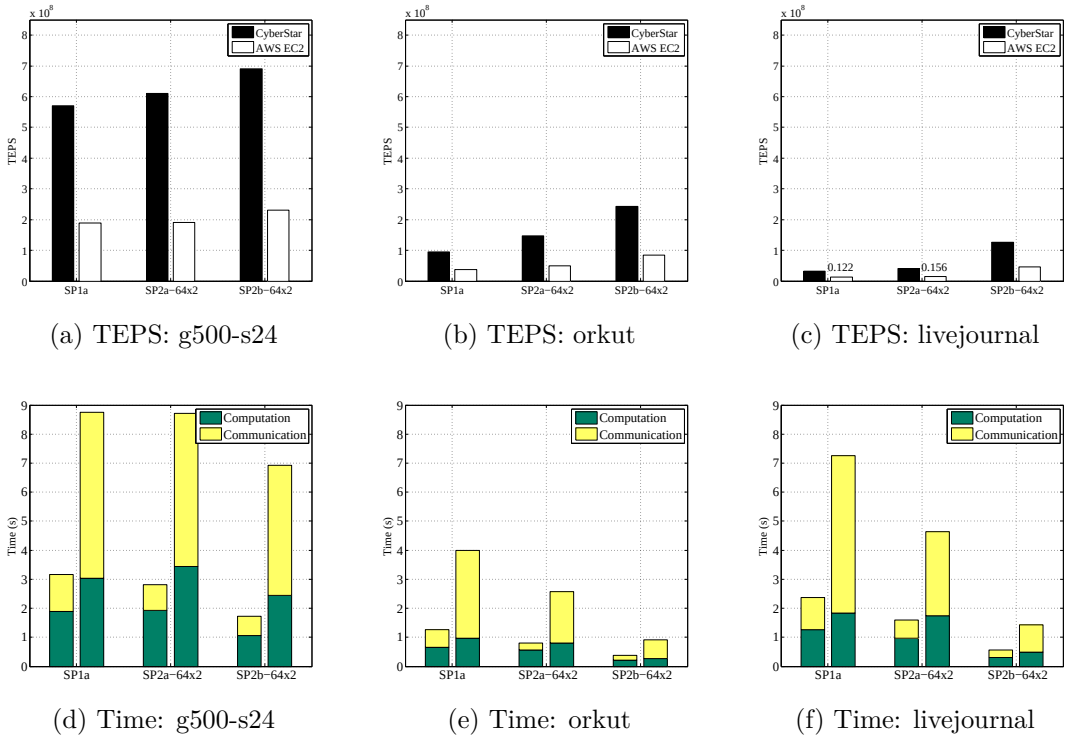
(d) friendster (e) orkut (f) livejournal

Figure 5.9. Communication breakdown of SSSP algorithms on 32 computing nodes.

5.3.5 Cross-Architecture Performance

The results from the experiments that we have done so far show very limited performance since they are obtained from the experiments on the virtual cluster, StarCluster, on top of Amazon EC2 instances. Many devices of instances are not up to the standard of most high performance computing systems. For example, the commodity network on EC2 m3.2xlarge that we use has the maximum bandwidth of only up to 125 MB/s while most standard computing clusters usually equip with the InfiniBand interconnect that provides up to 10 GB/s. For comparison purposes, we show the results that can be obtained from an actual high performance cluster, in this case, the CyberStar cluster. Note that we could not perform the full experiments on CyberStar due to the limited resources.

We perform a comparison of three SSSP algorithms, SP1a, SP2a-64×2 and SP2b-64×2 on CyberStar and AWS EC2. The experiments are done with 128 MPI tasks on both systems (i.e., 16 nodes on CyberStar and 8 nodes on AWS EC2). The



(a) TEPS: g500-s24 (b) TEPS: orkut (c) TEPS: livejournal

(d) Time: g500-s24 (e) Time: orkut (f) Time: livejournal

Figure 5.10. Performance comparison of three SSSP algorithms on CyberStar (left bar) and AWS EC2 (right bar). The experiments are performed on g500-s24 (2^{24} vertices), orkut and livejournal graphs with 128 MPI tasks.

algorithms run on g500-s24 (2^{24} vertices), orkut and livejournal graphs, and the performance is shown in Fig. 5.10. The performance of the three SSSP algorithms on CyberStar shows much better results than on AWS EC2 as we can see that the algorithms provide up to 3× the performance on AWS EC2 (Figs. 5.10(a-c)). Figs. 5.10(d-f) show the break down computation and communication time of the algorithms, respectively. With the InfiniBand interconnection, communication time of the SSSP algorithms on CyberStar shows a significant improvement, and takes only around one-fifth of the communication time on AWS EC2.

5.3.6 Comparison to Other Works

Many parallel graph frameworks have been recently proposed, and have reported good performance such as GraphX [136], Cyclops [27], Mizan [70] and Scale-Graph [36]. In this section, we perform a performance comparison of our 2D SSSP implementations with SSSP implementations in GraphX and Cyclops. Please

note that GraphX and Cyclops are not MPI-based as we use in our graph implementations. Despite the differences of frameworks, compilers and inter-processor communication approaches, the comparison is constructed under the same computational environments i.e., AWS EC2 m4.10xlarge 4 instances in which each instance comprises of 40 vCPUs and 160GB of RAM, and is tested on sk-2005, friendster and it-2004 graphs.

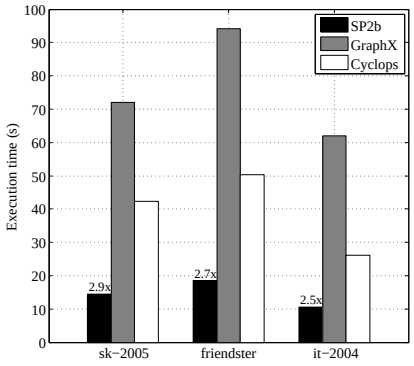


Figure 5.11. The performance comparison of SSSP implementations on various graph processing frameworks.

The SSSP implementations in both GraphX and Cyclops are based on the Dijsktra's algorithm. As a result, our SSSP implementation shows much better performance in all cases as shown in Fig. 5.11. Our SSSP implementations perform up to 5× faster than the SSSP implementation on GraphX, and up to 2× faster than the SSSP implementation on Cyclops.

5.4 Conclusion and Future Work

We propose efficient SSSP algorithms that combine the advantages of Dijkstra's and Bellman-Ford algorithms. Our algorithms reduce both communication and computation overhead from the utilization of the 2D graph layout, the cache-like optimization and the Δ increment heuristic. The 2D layout improves the algorithm performance by decreasing the communication space of the all-to-all communication from p to r where $p = r \times c$ is the number of total processors (assuming that each processor owns one partition); r and c are the number of processor rows and columns, respectively. Furthermore, the layout also improves the load balancing of

distributed graphs, especially, with scale-free graphs that contain few high degree vertices, since the adjacencies of these vertices are now distributed among multiple processors. The cached-like optimization avoid unnecessary workloads for both communication and communication by filtering out all update requests that are known to be discarded. Finally, by increasing the Δ values while the algorithms progress, we can improve the concurrency of the algorithms in the later iterations without high tradeoff in the work redundancy.

Currently, our algorithms are based on the bulk-synchronous processing for distributed-memory systems. We plan to extend our algorithms to also support the shared-memory parallel systems (i.e., the hybrid OpenMP and MPI implementation) that can further reduce the inter-processor communication of the algorithm. Another possible approach to improve the algorithm performance is to overlap the communication and computation by over-partitioning the graph and assigning multiple partitions to each processor. Thus, the computation and communication can be overlapped among partitions on the same processor.

Chapter 6
Parallel Approximate Graph Coloring Algorithms

One of fundamental concepts in parallel computing is to determine data dependencies in a program. The more dependence among data, the less parallelism of the programs. These dependencies are usually uncovered by identifying independent sets of data. Data that belongs to the same set is independent to each other, and can be processed simultaneously without any race conditions or invalid execution results.

A commonly used technique to discover such the sets is to use a graph coloring method. This method is widely used to determine concurrency in many parallel scientific computing such as some iterative methods for sparse linear systems [65], preconditioners [61], adaptive mesh refinements [66] and mesh optimizations [13].

Graph coloring is an assignment of colors or labels to elements of a graph based on some given constraints. These graph elements can be vertices or edges of the graph, and their corresponding coloring problems are usually referred as vertex and edge coloring, respectively. In this context, we consider only vertex coloring, and also refer it as graph coloring throughout the chapter.

One of most common types of vertex coloring is to color graph vertices such that no two adjacent vertices are given the same color. Let $G = (V, E)$ be an undirected graph where V and E are sets of vertices and edges of the graph, respectively. Graph coloring is to assign colors to all $v \in V$ such that, for any $u, v \in V$ and $uv \in E$, u and v cannot have the same color. Once all vertices are colored, it can be viewed as a partition of V into q independent sets where q is the number of colors used in the coloring process. The minimum number of colors required for a

given graph is called the chromatic number. However, determining the chromatic number of a graph is NP-Complete. Thus, graph coloring problems are usually solved using heuristic approaches to find an approximate solution because of their NP-Completeness. A general approximate coloring usually processes in the way that, first, all vertices are ordered by some ordering criterion. Then, each vertex is picked in order based on its rank, and is given a minimum color that are not used by their adjacencies. The method progresses until all vertices are colored. This algorithm is shown in Alg. 8. There are several ordering heuristics that can be used as discussed in [3, 56]. However, the two most well-known, efficient ordering are the largest-degree-first [132] and smallest-degree-last [86] orderings.

Algorithm 8 Sequential approximate graph coloring algorithm

```
1: set v.color to no color for all v
2: ordering v based on a given condition
3: for each v from the ordering do
4:     color ← min. color not used by neighbors
5:     v.color ← color
6: end for
```

In this Chapter, we propose scalable approximate (vertex) graph coloring algorithms for distributed-memory systems. The algorithms use a heuristic technique that iteratively colors graph vertices based on their priorities. Our parallel implementation is based on a bulk synchronous parallel model with the 2D graph layout that is presented in Chapter 4. Our main contributions are as follows:

- Approximate graph coloring algorithms that are scalable and can be used on large-scale distributed systems.
- A utilization of the 2D graph layout to reduce communication overhead and improve load-balancing of approximate graph coloring algorithms.
- A detailed evaluation of the approximate graph coloring algorithms on various large-scale graphs and distributed-memory machines.

6.1 Parallel Approximate Graph Coloring: Prior Work

Parallel approximate graph coloring has been studied on various types of platforms such as shared-memory [48], distributed-memory [17, 22, 47], and multicore and

multithreaded systems [23, 81]. These implementations are based on efficient, sequential heuristic approaches. However, the major flaw of these algorithms is that they do not provide much parallelism to be exploited because of data dependencies in the algorithms; thus, their scalability is limited. As a result, most parallel coloring algorithms are not actually true formulation of efficient, sequential algorithms (i.e., they do not give the same solution) as they need to be modified to provide more parallelism, especially, for algorithms for distributed-memory systems that vertex information is scattered among independent memory spaces. Hence, designing and implementing scalable, efficient approximate graph coloring algorithms for distributed-memory systems is very challenging.

We consider two recent algorithms for a parallel approximate graph coloring problem for distributed graphs. The algorithms are shown in Algs. 9 [47] and 10 [22]. Even though both algorithms are based on the bulk synchronous parallel model, Alg. 9 uses a distributed coloring method that requires less communication as it is designed for Hadoop distributed systems [133] while Alg. 10 is a parallel coloring based on the sequential algorithm that attempts to minimize coloring conflicts.

Algorithm 9 BSP Distributed graph coloring algorithm

```
1: set v.color to no color for all v
2: color ← 0
3: while some v with no color do
4:    for all v with no color do
5:       generate pairs (v, rank)
6:       buffer ← (v, rank)
7:    end for
8:    comm. exchange buffer
9:    for all v with no color do
10:      get all (v, rank) from buffer
11:      if v.rank > all rank then
12:         v.color ← color
13:      end if
14:   end for
15:   color ← color + 1
16: end while
```

Alg. 9 progresses in an iterative fashion. In each iteration, all active (or uncolored) vertices send their information to all neighbors. This information usually relates to the ordering heuristics used in the algorithms such as vertex

degree and vertex ID. Once all vertices receive the information from their neighbors, they color themselves with a distinct color (e.g., one color for each iteration) and mark themselves inactive if they own the maximal of the information. This information usually includes, but are not limited to, maximum degree or maximum ID among neighbors. The algorithm terminates once all vertices are inactive (e.g., all vertices are colored).

Algorithm 10 SCP Framework

```
1: set all vertices to no color
2: S ← set of subset partitions of size s
3: while S is not empty do
4:    for each U ∈ S do
5:       color U with sequential algorithm
6:       exchange colors of boundary vertices
7:       adjust colors of boundary vertices
8:    end for
9: end while
```

Alg. 10, on the other hand, uses a sequential coloring algorithm on each subgraph and performs recoloring any boundary vertex that has conflict colors. Given a partition of a graph, the algorithm progresses by partitioning each subgraph with a fix size s. Then, the algorithm colors vertices of each subpartition with a serial coloring algorithm. At the end of each coloring phase, processors exchange the colors of boundary vertices of subgraphs. If any boundary vertex has a conflict color, the algorithm determines whether this vertex should be recolored based on its randomized priority. If the vertex has less priority, it is added to a list for recoloring. The algorithm repeats these three steps of coloring, exchanging boundary vertex colors and detecting conflicts on all subpartitions. The algorithm terminates once all vertices have been colored. Note that the present of subpartitions with size s on the subgraphs is to minimize the number of color conflicts during the coloring phase. The smaller the size s, the lower the conflicts. However, the algorithm will require more iterations to completely recolor all vertices that will lead to more overhead from synchronization and communication.

6.2 Parallel Graph Coloring: 2D layout

The 2D graph layout for distributed parallel algorithms has been studied in [19] for breadth-first search. It provides significant improvement over a traditional (1D) graph layout in terms of communication space and edge load balancing, especially in large-scale real-world graphs with scale free distributions such as in social networks.

Our algorithms combine advantages of various graph coloring algorithms and apply the 2D graph layout to further improve the performance of parallel coloring algorithms while still providing coloring accuracy close to the sequential coloring algorithm. Our two approximate coloring algorithms are based on Algs. 9 and 10 which we call SCP2d and BSP2d, respectively. The main improvement of algorithms with the 2D graph layout is that edges of each vertex can now be distributed to multiple processors instead of residing on the same processor as in the 1D layout. This 2D layout can be viewed as *block* partitioning of adjacency matrix of the graph (i.e., partitioning the matrix by both row and column) which is commonly used in matrix algebra.

The main difference between the 1D and 2D graph coloring algorithms is the communication pattern. In the 2D layout, out-going edges of a vertex are distributed across multiple processors, or more precisely, this edge information of one vertex are distributed among processor rows. Thus, to access the complete information of all vertex adjacencies, an additional all-gather communication (among processors in the same communication rows) is required. However, the main communication of the algorithms which is in the form of an all-to-all communication is decreased. In the 1D layout, all processors need to participate in the same all-to-all communication while, in the 2D layout, the all-to-all communication occurs only among processors that belong to the same communication columns (or column processors). Thus, for the 2D layout with $p = \sqrt{p} \times \sqrt{p}$ processors, the all-to-all communication only occurs among groups of $\sqrt{p}$ processors instead of all p processors as in the 1D layout.

Table 6.1. The list of graphs and meshes used in our experiments.

Graph	Vertices	Edges	Reference
bmw3_2	227 K	5.5 M	[119]
hood	221 K	4.8 M	[119]
msdoor	416 K	9.3 M	[119]
pwtk	218 K	5.6 M	[119]
menger_sponge	6 M	154 M	[96]
luer_connector	10 M	253 M	[96]
nlpkkt200	16 M	440 M	[120]
nlpkkt240	28 M	761 M	[120]
g500-s27	134 M	2.1 B	[54]
it-2004	41 M	1.1 B	[15]
sk-2005	50 M	1.9 B	[15]
friendster	65 M	1.8 B	[137]

6.3 Performance Results and Analysis

6.3.1 Experimental Setup

Our experiments were run on two settings. First, it is an Intel i7 laptop (8-core processor) with 16GB of memory. This machine runs experiments that compare the number of colors from our algorithms with [22]. The other system is StarCluster [122], a virtual cluster on top of Amazon Web Service (AWS) Elastic Compute Cloud (EC2) [7]. The cluster occupies with the MPICH2 complier version 1.4.1. We emulate using 32 instances of AWS EC2 m3.2xlarge which each consists of 8 cores of Intel Xeon E2-2670 v2 (Ivy Bridge) processors and 30 GB of memory. This system is for the experiments on scalability of the algorithms on large dataset.

In the experiments, the algorithms have been evaluated on real-word graphs and meshes which are obtained from The University of Florida Sparse Matrix Collection [120], Stanford Network Analysis Platform [119] and The Laboratory for Web Algorithmics [15]. The list of complete inputs is shown in Tab. 6.1. Note that g500-s27 is a synthetic graph generated using Graph500 [54] with vertex and edge scales of 27 and 16, respectively.

6.3.2 Performance of Approximate Coloring Algorithms

We compare the accuracy of the two algorithms, i.e., a number of colors which are returned from the algorithms with the 2D graph layout using one and four processor columns on small meshes, bmw3_2, hood, msdoor and pwtk. (A lower number of colors indicates better accuracy.) The result is shown in Tab. 6.2. Note that * indicates an unavailable result as the algorithms with the 2D layout requires at least four processors with four processor columns. The numbers of colors from SCP2d are about half of the results from BSP2d. The reason is that BSP2d is based on a distributed graph coloring heuristic that uses an entirely different approach from the sequential coloring algorithm. However, increasing the number of processors in BSP2d does not change coloring accuracy of the algorithm. In contrast with SCP2d, when the number of processors increases, the accuracy of the algorithm also decreases since more partitions introduces more color conflicts along the boundary vertices.

		SCP2d		BSP2d	
		$p \times 1$	$(p/4) \times 4$	$p \times 1$	$(p/4) \times 4$
bmw3_2	$p = 1$	48	*	84	*
	$p = 16$	57	58	92	92
hood	$p = 1$	42	*	76	*
	$p = 16$	47	48	82	82
msdoor	$p = 1$	42	*	76	*
	$p = 16$	47	48	90	90
pwtk	$p = 1$	48	*	82	*
	$p = 16$	54	54	90	90

Table 6.2. The number of colors that are returned from the BSP2d and SCP2d algorithms on bmw3_2, hood, msdoor and pwtk graphs with one and four processor columns.

The performance of the SCP2d and BSP2d algorithms in terms of TEPS (i.e., edges traversed per second) on StarCluster on large meshes (e.g., menger_sponge, luer_connector, nlpkkt200 and nlpkkt240) is shown in Tab. 6.3. Please note that the TEPS in this case is a normalized time per edges. Increasing the number of columns from 1 to 8 in both 2D algorithms does not improve much performance (e.g., only up to four and seven percent improvement on SCP2d and BSP2d, respectively) as the numbers of edges are likely equally distributed even when using the traditional 1D layout. The advantage of the 2D layout that reduces the communication space

is negligible as it also introduces extra communication in the form of the all-gather communication. In most cases, BSP2d provides better performance and scalability over SCP2d, despite BSP2d requires more colors on the same graphs and meshes.

		BSP2d		SCP2d	
		$p \times 1$	$(p/8) \times 8$	$p \times 1$	$(p/8) \times 8$
menger_sponge	$p = 64$	4.23	**4.38**	3.32	**3.38**
	$p = 128$	**6.53**	6.44	4.58	**4.68**
	$p = 256$	**8.60**	8.37	5.58	**5.64**
luer_connector	$p = 64$	3.78	**3.86**	3.17	**3.22**
	$p = 128$	**6.10**	5.98	4.56	**4.62**
	$p = 256$	8.01	**8.14**	5.55	**5.72**
nlpkkt200	$p = 64$	3.05	**3.07**	2.66	**2.85**
	$p = 128$	**4.87**	4.82	3.98	**4.03**
	$p = 256$	**6.82**	6.65	**5.21**	5.16
nlpkkt240	$p = 64$	2.50	**2.59**	2.20	**2.22**
	$p = 128$	3.58	**3.69**	**3.09**	3.03
	$p = 256$	**4.63**	4.57	**3.90**	3.78

Table 6.3. The performance (TEPS $\times 10^7$) of BSP2d and SCP2d on large meshes.

The performance of the SCP2d and BSP2d algorithms in terms of TEPS on StarCluster on big social graphs (e.g., g500-s27, it-2004, sk-2005 and friendster) is shown in Tab. 6.4. Increasing the numbers of columns from 1 to 8, in this case, improves the performance on both algorithms as the edge distribution of scale-free graphs on the 2D layout is more balance than the distribution on the 1D layout. The performance improvement can be up to 10 and 19 percents for SCP2d and BSP2d, respectively. Furthermore, BSP2d gives better scalability in all cases when the numbers of processors increases although SCP2d provides better performance in general in terms of the number of colors.

6.3.3 Communication Cost Analysis

The break down computation and communication costs of the BSP2d and SCP2d algorithms on selected meshes and graphs are shown in Figs. 6.1 and 6.2, respectively. The four bars of each plot group indicates the running time of SCP2d with 1 and 8 columns, and BSP2d with 1 and 8 columns, respectively. As the degree distribution of vertices in meshes is uniform, the 2D layout does not provide any significant improvement over the 1D layout, and sometimes, the overall execution time also

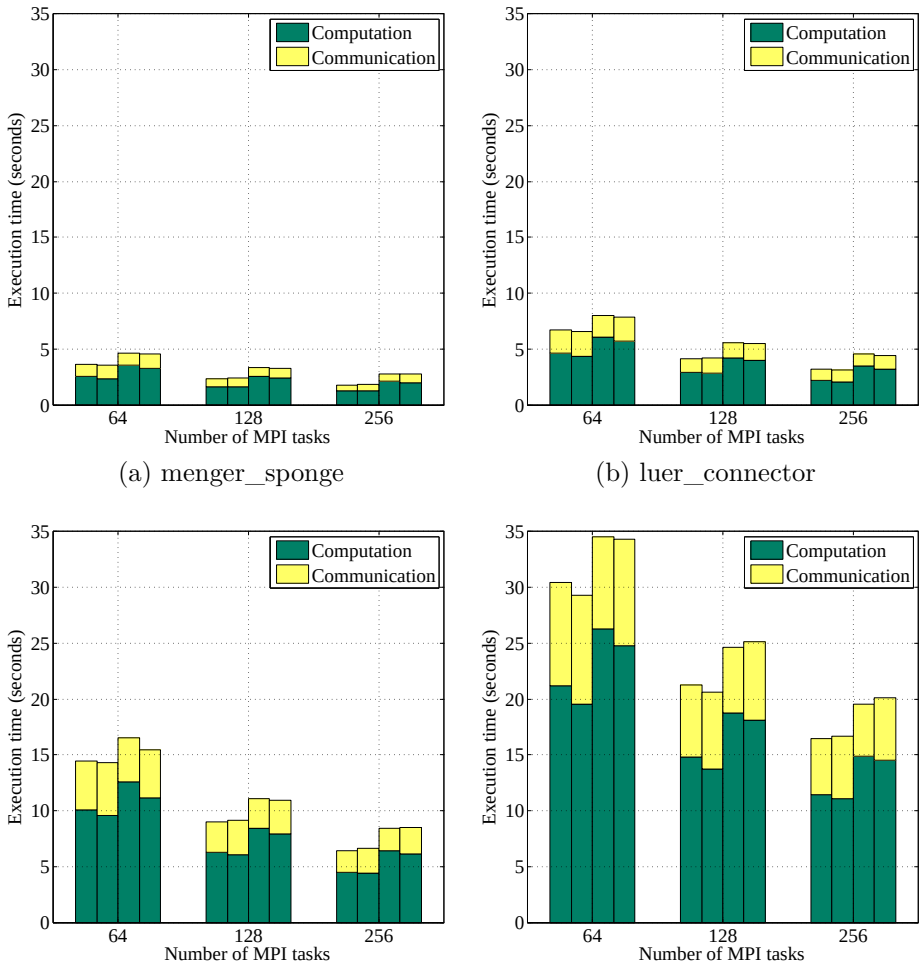
		BSP2d		SCP2d	
		$p \times 1$	$(p/8) \times 8$	$p \times 1$	$(p/8) \times 8$
g500-s27	$p = 64$	0.96	**1.02**	0.79	**0.90**
	$p = 128$	1.32	**1.44**	1.18	**1.40**
	$p = 256$	1.87	**2.03**	1.82	**1.99**
it-2004	$p = 64$	0.35	**0.37**	0.26	**0.29**
	$p = 128$	0.53	**0.57**	0.37	**0.44**
	$p = 256$	0.75	**0.81**	0.57	**0.68**
sk-2005	$p = 64$	0.57	**0.60**	0.39	**0.44**
	$p = 128$	0.87	**0.88**	0.65	**0.75**
	$p = 256$	1.13	**1.22**	1.09	**1.17**
friendster	$p = 64$	0.89	**0.93**	0.63	**0.75**
	$p = 128$	1.19	**1.29**	0.92	**1.01**
	$p = 256$	1.47	**1.62**	1.33	**1.45**

Table 6.4. The performance (TEPS $\times 10^7$) of BSP2d and SCP2d on large graphs.

increases when the number of columns increases from 1 to 8 (see Fig. 6.1(d)). On the other hand, the results of both algorithms on social graphs reveal a notable improvement in both computation and communication costs when the number of columns increases from 1 to 8. This is from the characteristic of social graphs that has a power law degree distribution (i.e., very few vertices in the graphs contain higher vertex degree than others). Thus, the 2D layout can provide more balance of the edge distribution among processors than the distribution of the 1D layout.

6.4 Conclusion and Future Work

We have propose parallel approximate graph coloring algorithms based on [47] and [22] combining with the 2D graph layout and other optimization techniques. Our algorithms show improved performance and scalability over the approximate algorithms with the traditional 1D layout. The main reasons are from better load balancing and lower communication from the use of the 2D layout. From our experiment, the BSP2d algorithm provides better performance over the SCP2d algorithm in terms of the running time in all cases while it has less accuracy (i.e., it requires more colors) than the SCP2d algorithm. The main reason is that BSP2d requires less synchronization because there is no need for the algorithm to manage any color conflict among processors. On the other hand, SCP2d requires higher



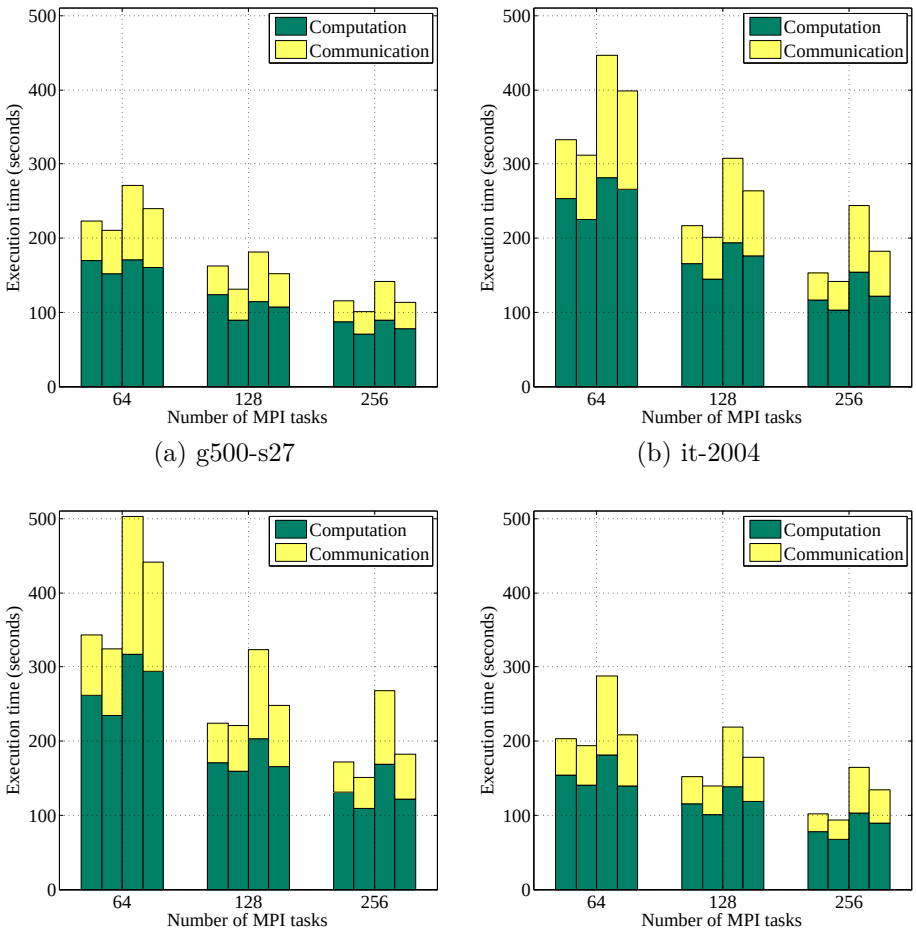
(a) menger_sponge (b) luer_connector

(c) nlpkkt200 (d) nlpkkt240

Figure 6.1. The performance of the three algorithms on large meshes, (a) menger_sponge, (b) luer_connector, (c) nlpkkt200 and (d) nlpkkt240 meshes. The four bars of each plot group indicates the running time of SCP2d with 1 and 8 columns, and BSP2d with 1 and 8 columns, respectively.

algorithm synchronization as it needs to resolve color conflicts which gives better results in terms of coloring accuracy.

Our algorithms are currently focused on distributed-memory systems which can suffer from the scalability issues when the number of processors increases. We plan to implement parallel coloring algorithms that can utilize both multicore and multiprocessor systems to further improve the scalability of the algorithms.



(a) g500-s27 (b) it-2004

(c) sk-2005 (d) friendster

Figure 6.2. The performance of the three algorithms on large social graphs, (a) g500-s27, (b) it-2004, (c) sk-2005 and (d) friendster graphs. The four bars of each plot group indicates the running time of SCP2d with 1 and 8 columns, and BSP2d with 1 and 8 columns, respectively.

Chapter 7
MDEC: MeTiS-Based Domain Decomposition for Parallel 2D Mesh Generation

The general domain decomposition problem is to decompose the domain of interest into several smaller, non-overlapping domains (called subdomains) based upon some criterion (typically for parallel computation) such as: load balancing, computation requirements, or data dependency. Relevant domains of interest for domain decomposition are: sets, vectors, matrices, or geometries. In this chapter, we are concerned with the decomposition of geometric domains.

Domain decomposition techniques have been employed in parallel numerical algorithms in order to decompose a large, complex problem into many smaller, simpler subproblems which can be solved in parallel. Within the context of parallel numerical algorithms, domain decomposition is typically employed before the main computation begins. For example, domain decomposition methods are often used in the numerical solution of partial differential equations by the finite element method, or other such techniques, in order to decompose the domain into several subdomains on which the PDE is solved. In this example, geometric domains or meshes can be partitioned across the processors so that the numerical PDE can be solved in a distributed manner. Some techniques which have been successfully

The work of this chapter has been published in:
[95] T. Panitanarak and S.M. Shontz, "MDEC: MeTiS-based domain decomposition for parallel 2D mesh generation," Proceedings of the 2011 International Conference on Computational Science, June 2011, pp. 302-311.

used for mesh partitioning include: geometric mesh partitioning [50] (which strives to divide the mesh into equal-sized regions or submeshes with a small number of interconnecting edges), coordinate bisection [57] (which partitions the vertices of the mesh after projection onto one of the coordinate axes), spectral bisection [98] (which partitions the mesh according to the eigenvectors of the Laplacian of its connectivity graph), and multilevel Kernighan-Lin [58] (which partitions the mesh into a sequence of successively smaller graphs, uses the spectral method to partition the smallest graph, and propagates the partition back through the hierarchy; the Kernighan-Lin method is used to refine the partition). Mesh partitioning remains an active area of research, as various decompositions of the domain can lead to different levels of parallelism in the resulting numerical algorithms.

The main contributions of this chapter are as follows:

- A fast domain decomposition for parallel 2D mesh generation that generates good quality subdomains.
- An analysis of the algorithm that guaranteed good boundary angles.
- A comparison with other domain decompositions for parallel 2D mesh generation.

7.1 The Domain Decomposition Problem for Parallel Mesh Generation

Despite the fact that domain decomposition is only applied before the main computation step (as described in the previous section), decomposition of the geometry for the purposes of parallel mesh generation is also desired if the size of the geometric domain is very large or if more accuracy is needed in the numerical solution of the PDE. The remainder of this chapter focuses on parallel computational techniques for 2D mesh generation.

Parallel mesh generation starts with decomposing the geometric domain into many smaller non-overlapping subdomains. The resulting subdomains are then meshed in parallel. During the mesh generation process, communication between processors may be required in order to preserve the conformality of the overall

mesh. However, communication might not be required at all if all the conformal points are predetermined [30].

A review of various parallel mesh generation algorithms is provided in [30]. In that chapter, the mesh generation techniques are divided into two categories. The first category of techniques includes mesh generation algorithms for which each subdomain is meshed sequentially. The second category includes techniques for which the degree of coupling between the processors is what defines the degree of communication between the processors in order to preserve conformity of the overall mesh.

A recent domain decomposition algorithm specifically designed and used for parallel mesh generation is the Medial Axis Domain Decomposition (MADD) algorithm [74]. This algorithm decomposes the geometric domain in a divide-and-conquer fashion. The MADD algorithm decomposes the geometric domain by first discretizing the domain boundary. Second, it finds the approximate medial axis of the geometric domain using centroids of the coarse mesh. (This is a boundary conforming Delaunay triangulation of the points created in the previous step). These are actually the nodes of a Voronoi triangulation. Third, it partitions the graph of the Voronoi nodes into two subsets. Fourth, it uses a subset of the Voronoi nodes and connects them to the triangle boundary points to make separators (i.e., segments of the boundary) to separate the two subdomains. Finally, it recursively calls the first four steps using the generated subdomains as inputs until the desired number of subdomains is achieved. For more details on the algorithm, the reader is referred to [74].

In [74], it was noted that using the background mesh directly for the decomposition can lead to small boundary angles. This is undesirable in that the resulting subdomains may lead to less-balanced subdomains and to issues with load balancing within the context of parallel mesh generation. In the next section, we will discuss this further and will describe a way to resolve the small boundary angles. Furthermore, we will use our technique as the basis for a domain decomposition approach for triangular meshes.

7.2 MeTiS-Based Domain Decomposition (MDEC) with Guaranteed Good Boundary Angles (i.e., Boundary Angles Greater Than $60°$)

Our MeTiS-based Domain Decomposition (MDEC) procedure begins with the generation of an initial triangular background mesh on the geometric domain of interest. Next, the mesh is partitioned into the desired number of subdomains. For a given edge of a triangular element, if the edge belongs to elements that belong to different partitions, the edge is used as a separator (i.e., as a segment of the boundary of the final subdomain). Since the final boundary is constructed from edges of existing elements, the boundary angles can be as small as the angles of an element. In order to generate a partition with subdomains containing boundary angles that are at least 60°, a background mesh which satisifies the 60° angle constraint must be generated. Unfortunately, it is not practical to mesh the geometric domain with triangular elements containing boundary angles that are all greater than 60°, as most mesh generation algorithms cannot generate such meshes.

Fortunately, background meshes with element angles greater than 30° can be employed, and it is practical for some algorithms to generate a decomposition of the background mesh into subdomains such that the each subdomain contains boundary angles that are greater than 60°. To achieve this, we can perform an adjustment to the decomposition as follows. First, note that in a decomposition of the boundary of the background mesh, the boundary angles can either be less than, equal to, or greater than 60°, because they can be taken directly from one element angle (since two edges of a single element can be used as separators or as boundary segments) or from the summation of two or more element angles, respectively. The idea of the adjustments is to consider the elements which provide separators with two edges and make some modifications so as to eliminate the small angles. This is the idea of our MeTiS-based Domain Decomposition algorithm.

We define a *bad triangle* to be a triangle that has two edges that are used as separators and an angle between the two edges that is less than 60°. As was described in the previous paragraph, the subdomain boundary is less than 60° if some separators stem from bad triangles. Bad triangles can be classified into three groups as follows. Case 1 occurs when one edge belonging to a bad triangle is

an external segment. Case 2 occurs when two of the triangle's edges are internal separators of two subdomains. Finally, Case 3 happens when the triangle's edges are internal separators of three or more subdomains.

Next, we will describe some techniques for making adjustments (on a case-by-case basis) to the bad triangles to achieve subdomains with a minimum boundary angle of at least 60°. In the next section, we provide a proof that our domain decomposition technique results in such a decomposition into subdomains.

We now describe the techniques for adjusting the bad triangles in order to yield a decomposition into subdomains with the desired boundary angle property. The techniques for improving the small angles in the bad triangles will be performed according to the type of bad triangle. First, suppose that edges v_1v_2, and v_1v_3 are two edges of the bad triangle. For the bad triangle, which can be identified as belonging to Case 1, assume that v_1v_2 is an external separator. Then, we can simply replace v_1v_2 with v_1v_3, or we can use v_2v_3 as an alternative separator instead of v_1v_3 if the angle $v_1v_2v_3$ is greater than 60° but less than 120° (to avoid creating another bad triangle). Otherwise, we can add midpoint v_m of v_1v_2 and replace v_1v_2 with v_2v_m. The reason for using the midpoint instead of the point v_p, where v_3v_p is perpendicular to v_1v_2, is that v_p can be off of the line v_1v_2, and we want to balance v_1v_m and v_mv_2 in order to obtain a good mesh. For a bad triangle in Case 2, we can easily replace v_1v_2 and v_1v_3 with v_2v_3. Finally, for a bad triangle in Case 3, we can add the incenter v_i of the triangle and can replace v_1v_2 and v_1v_3 with v_1v_i, v_2v_i, and v_3v_i. The reader is referred to Figure 7.1 for an illustration of the adjustments in each case.

7.3 The MDEC Algorithm and Implementation

We now describe the MDEC algorithm and its implementation.

7.3.1 MDEC Algorithm

The steps of the MDEC algorithm are as follows.

1. Mesh the geometric domain with an angle constraint of 30°.

2. Convert the mesh to a graph such that one node is placed in the graph to

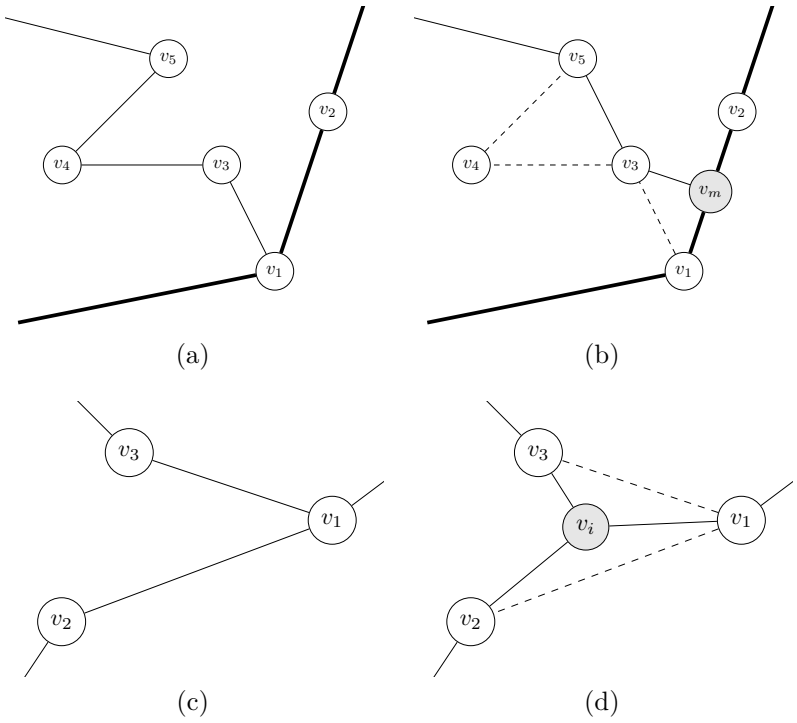


Figure 7.1. Adjustment methods for removing the small angles in the bad triangles as implemented by MDEC: The white nodes represent vertices in the initial coarse mesh, and the thick lines are the external boundary. Figure 7.1a and Figure 7.1b illustrate the adjustments made for bad triangles in Case 1 and Case 2. From Figure 7.1c to Figure 7.1d, the adjustment for Case 3 is shown.

represent each triangle and an edge in the graph represents an adjacency relation between two triangles (i.e., the two triangles are neighbors).

3. Partition the graph in Step 2 into the desired number of subdomains.

4. Insert separators into the partitions in Step 3 to create the initial subdomains and detect the bad angles in each subdomain (as indicated in Cases 1-3 below).

5. Fix the bad angles (i.e., a boundary angle that is formed using two edges, i.e., v_1v_2 and v_1v_3, in the same triangle) labeled Cases 1-3.

   Case 1 v_1v_2 form an external boundary of the geometric domain, and v_1v_3 form an internal boundary between two subdomains.

   - If angle $v_1v_2v_3$ is between 60° and 120°, replace v_1v_3 with v_2v_3.

- Otherwise, add the midpoint v_m of v_1v_2 and replace v_1v_3 with v_3v_m.

Case 2 v_1v_2 and v_1v_3 form an internal boundary between two subdomains.

- Replace both v_1v_2 and v_1v_3 with v_2v_3.

Case 3 v_1v_2 and v_1v_3 form an internal boundary amongst three or more subdomains.

- Add the incenter v_i and replace both v_1v_2 and v_1v_3 with v_iv_2 and v_iv_3.

The following theorem demonstrates that the MDEC algorithm guarantees a mesh with good boundary angles.

Theorem. *The MDEC algorithm generates minimum boundary angles of at least* $60°$.

Proof. We consider the angle improvements given for Cases 1-3 above and show that they guarantee boundary angles greater than $60°$. For Case 2, it is easy to see that any angle less than $60°$ is eliminated with the replacement of the $180°$ angle. The angle improvements for Case 1 and Case 3 can be demonstrated as follows.

Consider Case 1. In the case that $60° \leq v_1v_2v_3 < 120°$, when v_1v_3 is replaced with v_2v_3, a new angle greater than $60°$ is obtained. In addition, the formation of a bad triangle that shares v_2v_3 is avoided. In the case that $v_1v_2v_3 < 60°$ or $v_1v_2v_3 \geq 120°$, v_1v_3 is replaced by v_3v_m, where v_m is the midpoint of v_1v_2. Bad triangle $\Delta v_2v_1v_3$ is split into two triangles, namely $\Delta v_1v_mv_3$ and $\Delta v_2v_mv_3$. Since $30° \leq v_mv_1v_3 < 60°$, if $30° \leq v_mv_2v_3 < 60°$, we obtain $v_1v_mv_3 \geq 60°$. Instead, if $120° \leq v_mv_2v_3 < 150°$, we obtain $v_1v_mv_3 \geq 60°$. Similarly, it can be shown that $v_2v_mv_3 \geq 60°$.

We now consider Case 3. In this case, v_1v_2 and v_1v_3 are replaced by v_1v_i, v_2v_i and v_3v_i, where v_i is the incenter of triangle $\Delta v_2v_1v_3$. The bad triangle $\Delta v_2v_1v_3$ is then split into three triangles, namely $\Delta v_1v_iv_2$, $\Delta v_2v_iv_3$, and $\Delta v_1v_iv_3$. Assume $v_1v_iv_2 < 60°$. Then, $v_iv_1v_2 +$

$v_i v_2 v_1 \geq 120°$. Multiplying both sides of the inequality by two yields: $2(v_i v_1 v_2 + v_i v_2 v_1) = 2v_i v_1 v_2 + 2v_i v_2 v_1 = v_3 v_1 v_2 + v_3 v_2 v_1 \geq 240°$. This contradicts the fact that the interior angles of a triangle must sum to 180°. Thus, $v_1 v_i v_2 \geq 60°$. This is also true for $v_2 v_i v_3$ and for $v_1 v_i v_3$.

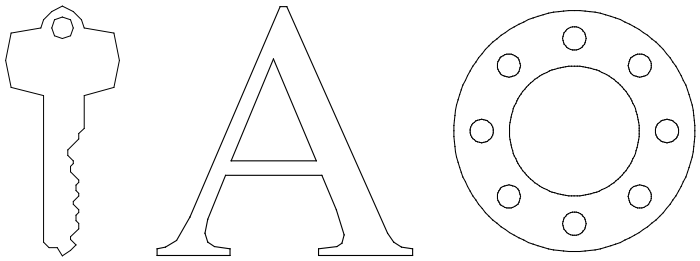
□

7.3.2 MDEC Implementation

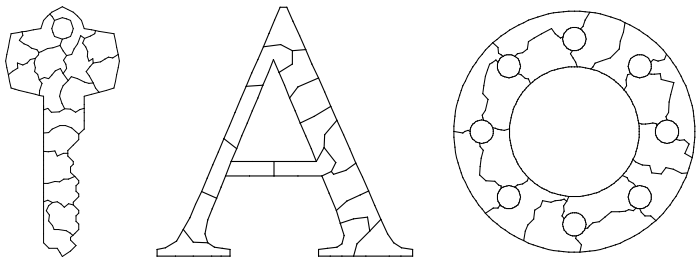
In our implementation of MDEC, the Triangle [112] and MeTiS [67] software packages are used. Triangle is used to generate an initial background mesh satisfying a boundary angle constraint of 30° and an area constraint (based on the area of the geometric domain). MeTiS is used as a graph partitioner for Step 3 of the MDEC algorithm. To achieve our goal of balancing the area of the resulting subdomains, the node weight is set to the area of the triangle. To evaluate our decomposition, PCDM, a parallel 2D constrained Delaunay mesh generation technique introduced by Chernikov and Chrisochoides in [74] is used to generate parallel meshes on the subdomains. The objective of PCDM is to reduce communication between processors that shares an interface by providing asynchronous communication with aggregation of small messages. In addition, we compare the results of our algorithm with MADD, as the primary decomposition routine currently used with PCDM is MADD.

7.4 Numerical Experiments

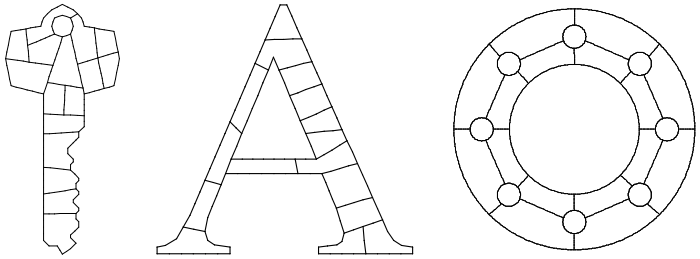
We perform two numerical experiments in order to test our MDEC domain decomposition algorithm. First, we use MDEC to decompose our geometric domains of interest, i.e., the key, A, and pipe models, into 128, 256, 512, 1024 and 2048 subdomains, respectively, and measure the domain decomposition time, the area of each subdomain, and the minimum boundary angle (i.e., as computed over all subdomain boundary angles). (The figures of the three models are shown in Figure 7.2a.) Second, we generate decompositions of the same models into 4, 8, 16, 32, 64, and 128 subdomains, use them as inputs to PCDM, and measure the mesh generation time and the element quality of the resulting mesh. The second experiment was performed on the Cyberstar compute cluster [34] available to the



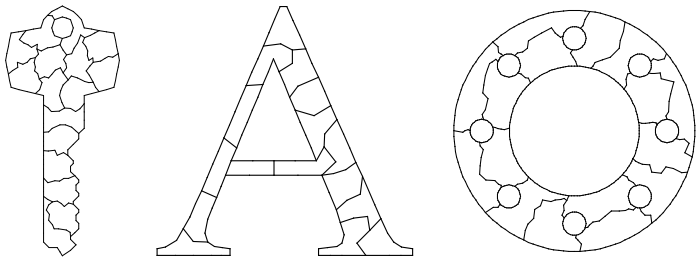
(a) The three models used in the experiments.



(b) The decomposition from the MDEC algorithm.



(c) The decomposition from the MADD algorithm.



(d) The decomposition from the pMeTiS algorithm

Figure 7.2. (a) The three geometric models used in the experiments: Key, A, and Pipe, and their decompositions into 16 subdomains as generated by the (b) MDEC, (c) MADD, and (d) pMeTiS domain decomposition algorithms.

researchers with 1, 2, 4, 8, and 16 Intel Nahalem processors, respectively. For both experiments, we compare our results to those obtained for the MADD algorithm [30] and the MeTiS algorithm (pMeTiS) [67]. It should be noted that pMeTiS performs a domain decomposition that is identical to the domain decomposition generated by MDEC if no boundary angles are fixed in the bad triangles.

7.4.1 The Domain Decompositions

Figures 7.2b, 7.2c and 7.2d illustrate the decompositions of the MDEC, MADD, and pMeTiS algorithms into 16 subdomains, respectively. Tables 7.1, 7.2 and 7.3 show the decomposition time, the minimum boundary angle, and the subdomain area for decompositions of the geometric models into 128, 256, 512, 1024, and 2048 subdomains using the MDEC, MADD and pMeTiS algorithms on the Key, A, and Pipe models, respectively.

Since MDEC and pMeTiS can generate the desired number of subdomains simultaneously instead of employing a divide-and-conquer approach as is the case for the MADD algorithm, the decomposition times for MDEC and pMeTiS are lower than the decomposition time for MADD. Our results show that MDEC requires approximately 87%, 94% and 93% less time than MADD for decomposing the Key, A, and Pipe models, respectively. The percentages reported above (and throughout this paragraph) are the averages as computed over all of the subdomains in the experiment of interest. For example, the above percentages are averages as computed over 128-2048 subdomains.

In comparison with pMeTiS, MDEC requires approximately 25%, 26% and 17% more time for the domain decomposition than does pMeTiS. However, the angles of the subdomain boundaries which are generated by MDEC are higher than those generated by either of the other two algorithms in most cases. MDEC generates subdomains with boundary angles that are approximately 9% better (in terms of the number of degrees of the angle) than those generated by MADD and are approximately 48% better than the angles generated by pMeTiS. In addition, MDEC decomposes the geometric domain into subdomains with boundary angles that are guaranteed to be greater than 60°, whereas there is no such guarantee for the subdomains generated by MADD or pMeTiS. Note that for a small number of subdomains, MADD can decompose the geometric domain into subdomains with

# of subdom.	Dec. time (secs)	Min. angle (°)	Subdomain area		
			Min.	Max.	Avg.
128	0.078	60.4844	24.2267	27.5068	26.1094
256	0.109	60.1282	12.1624	13.8609	13.0547
512	0.224	60.0812	5.9584	7.0230	6.5273
1024	0.389	60.0344	3.0483	3.4524	3.2637
2048	0.827	60.0066	1.4978	1.7318	1.6318

(a) MDEC algorithm

# of subdom.	Dec. time (secs)	Min. angle (°)	Subdomain area		
			Min.	Max.	Avg.
128	0.499	55.1059	12.4219	39.0910	26.1094
256	0.857	56.3551	6.2222	18.3606	13.0547
512	1.825	54.6888	2.9356	9.7813	6.5273
1024	3.463	55.1502	1.4366	4.7357	3.2637
2048	6.865	54.1880	0.6954	2.3434	1.6318

(b) MADD algorithm

# of subdom.	Dec. time (secs)	Min. angle (°)	Subdomain area		
			Min.	Max.	Avg.
128	0.046	35.5640	25.3577	26.8671	26.1094
256	0.078	32.7425	12.5834	13.4458	13.0547
512	0.140	33.3454	6.1160	6.2171	6.5273
1024	0.328	31.2630	3.1384	3.3619	3.2637
2048	0.796	31.1219	1.5424	1.6824	1.6318

(c) pMeTiS algorithm

Table 7.1. Decompositions of the Key domain generated by the MDEC, MADD, and pMeTiS algorithms (showing number of subdomains, decomposition time, minimum angle, and subdomain area)

boundary angles greater than 60°. The boundary angles are greater than those generated by the MeTiS algorithm (see Table 7.3c for 128 subdomains).

In terms of the subdomain area, pMeTiS does the best job of balancing the subdomain areas for all geometric domains considered. That is, the difference between the minimum and maximum subdomain areas is lower than the corresponding differences for the MDEC and MADD algorithms. The minimum and maximum subdomain areas are shown in Tables 7.1, 7.2, and 7.3). However, unlike MADD, MDEC does a very good job of balancing the subdomain area and does nearly as well as pMeTiS in this regard. Because pMeTiS does not achieve our goal of generating subdomains with reasonable minimum boundary angles, we no longer consider it in this chapter.

# of subdom.	Dec. time (secs)	Min. angle (°)	Subdomain area		
			Min.	Max.	Avg.
128	0.021	60.0933	87.3546	175.5632	131.4490
256	0.046	60.0902	44.4354	82.1791	65.7245
512	0.077	60.0379	14.7750	40.9076	32.8622
1024	0.156	60.0290	8.9520	22.5653	16.4311
2048	0.343	60.0105	4.5646	11.1691	8.2156

(a) MDEC algorithm

# of subdom.	Dec. time (secs)	Min. angle (°)	Subdomain area		
			Min.	Max.	Avg.
128	0.343	54.7301	67.7831	193.7770	131.4490
256	0.702	54.4969	37.1646	93.0400	65.7245
512	1.451	54.6933	12.4522	49.6854	32.8622
1024	3.073	54.8099	6.3131	24.4724	16.4311
2048	6.067	54.0625	3.2245	12.4306	8.2156

(b) MADD algorithm

# of subdom.	Dec. time (secs)	Min. angle (°)	Subdomain area		
			Min.	Max.	Avg.
128	0.015	30.7138	96.6917	148.2100	131.4490
256	0.031	30.9247	46.9538	75.1200	65.7245
512	0.062	31.2089	21.2383	36.3605	32.8622
1024	0.125	30.5489	11.1123	18.5072	16.4311
2048	0.249	30.2315	5.4319	9.4990	8.2156

(c) pMeTiS algorithm

Table 7.2. Decompositions of the A domain generated by the MDEC, MADD, and pMeTiS algorithms (showing number of subdomains, decomposition time, minimum angle, and subdomain area)

7.4.2 Parallel Mesh Generation

To evaluate and compare the quality of the decompositions from the MDEC and MADD algorithms, the subdomains generated by the algorithms were given to PCDM as input. The quality of the decompositions was evaluated based on the output meshes generated by the PCDM algorithm. In this subsection, we focus on the results from the Key model, as the results for the other models studied are very similar.

Figure 7.3 shows the running time of PCDM when using the outputs of the MDEC and MADD domain decomposition algorithms as inputs to PCDM, respectively. As seen in the figure, both algorithms yield subdomains upon which meshes

can be generated in parallel in approximately the same amount of time using PCDM. It is very hard to distinguish which algorithm is better when evaluated in this sense. The figures also show that increasing the number of processors does not always decrease the running time. In particular, when the number of processors exceeds the number of subdomains for either the MDEC or the MADD domain decomposition algorithms, the amount of time required increases.

By comparing the results for the MDEC and MADD algorithms, as shown in Figure 7.4, we can see that it is only for the case of eight subdomains that PCDM can generate a mesh in parallel faster for the subdomains of the MADD algorithm than for those of the MDEC algorithm. For other numbers of subdomains, the running times of PCDM for the inputs of both the MADD and MDEC algorithms

# of subdom.	Dec. time (secs)	Min. angle (°)	Subdomain area		
			Min.	Max.	Avg.
128	0.047	60.0302	80.0668	88.0002	84.4502
256	0.078	60.0030	39.5866	44.1174	42.2251
512	0.156	60.0015	19.7528	22.3155	21.1125
1024	0.312	60.0051	9.8531	11.1607	10.5563
2048	0.749	60.0012	4.9143	5.5810	5.2781

(a) MDEC algorithm

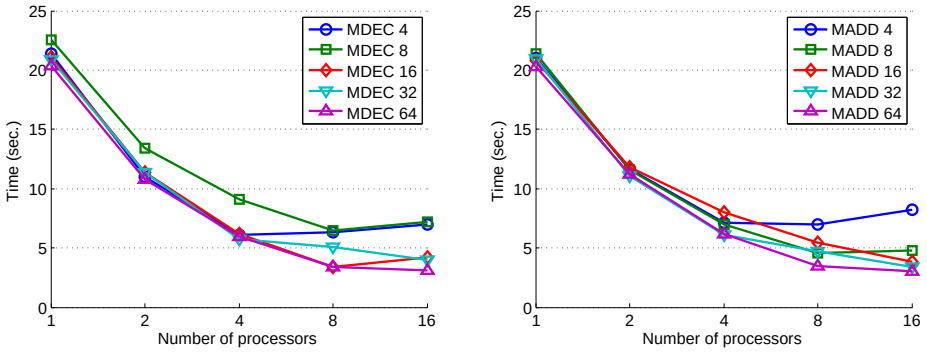
# of subdom.	Dec. time (secs)	Min. angle (°)	Subdomain area		
			Min.	Max.	Avg.
128	1.124	61.3000	33.7301	127.5020	84.4502
256	*	*	*	*	*
512	2.294	54.5434	9.6054	30.3949	21.1125
1024	4.320	54.0761	5.1874	15.5628	10.5563
2048	7.536	54.0761	1.9638	7.7769	5.2781

(b) MADD algorithm

# of subdom.	Dec. time (secs)	Min. angle (°)	Subdomain area		
			Min.	Max.	Avg.
128	0.031	31.0892	82.0408	86.9324	84.4502
256	0.062	30.6625	39.6250	43.4881	42.2251
512	0.140	30.0303	20.2833	21.7462	21.1125
1024	0.296	30.2154	10.2489	10.8724	10.5563
2048	0.701	30.0723	5.0019	5.4364	5.2781

(c) pMeTiS algorithm

Table 7.3. Decompositions of the Pipe domain generated by the MDEC, MADD, and pMeTiS algorithms (showing number of subdomains, decomposition time, minimum angle, and subdomain area). A * denotes algorithmic failure.



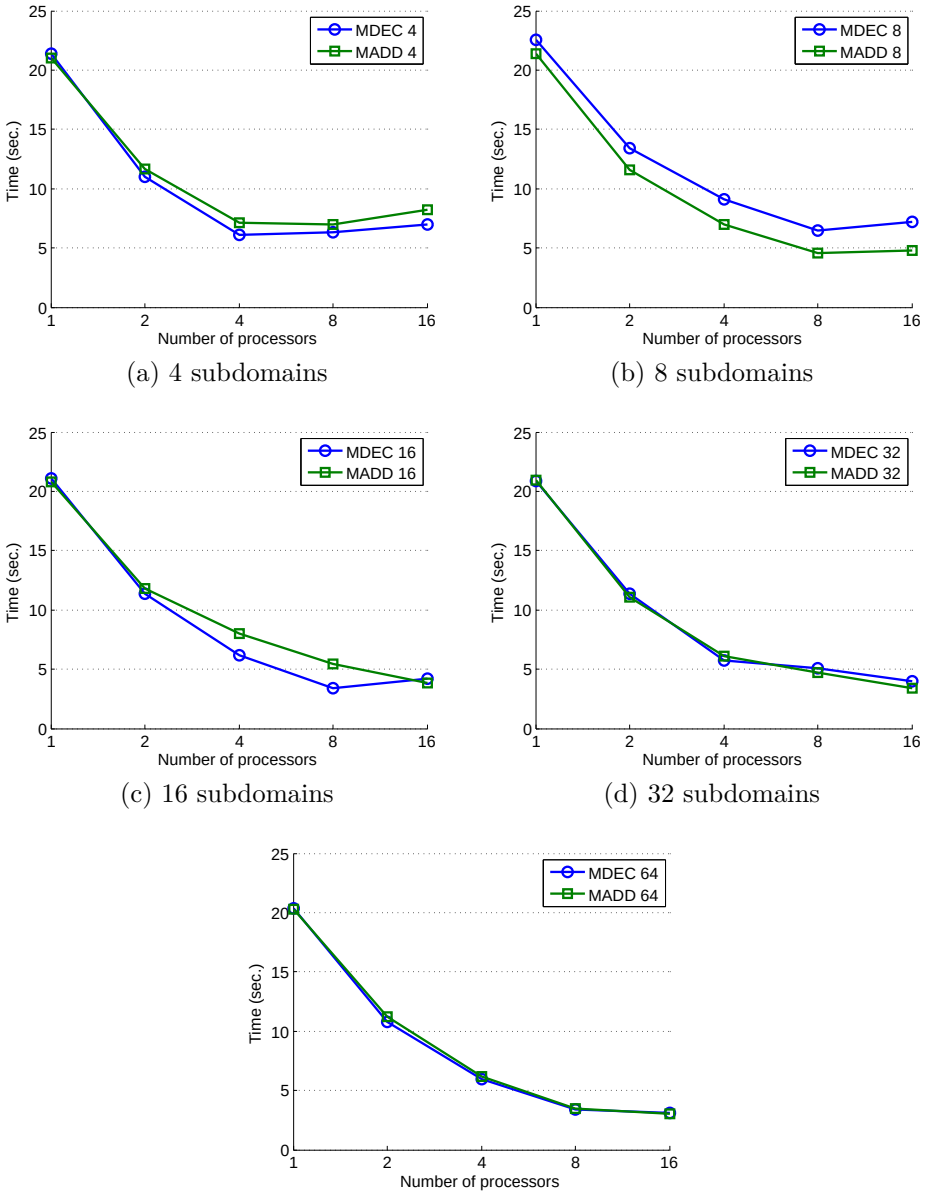
(a) MDEC algorithm (b) MADD algorithm

Figure 7.3. Triangulation time for 4, 8, 16, 32 and 64 subdomains of the Key model using the MDEC and MADD algorithms.

are very similar.

We use the element angles of the triangular elements to assess the qualities of the meshes generated by PCDM using both the MADD and MDEC subdomains as input. In each case, we normalize the number of triangles and report the percentages of triangles in each range of angles, as the meshes generated from different subdomains contain different numbers of triangles. However, all of the meshes generated by PCDM contain approximately 77 million triangles for the geometric models considered here. Both the MDEC and MADD decompositions yield similar final meshes as shown in Figure 7.5. The histograms give the percentages of triangles in each range of angles for two different subdomain/processor configurations. In particular, the results shown in Figure 7.5a are for the final meshes generated by PCDM using the MDEC and MADD subdomains with 1 processor and 4 subdomains. Similarly, Figure 7.5b shows the results for the final meshes generated with 16 processors and 64 subdomains using the PCDM algorithm. No real difference is observed between the element qualities of the PCDM meshes generated on the MDEC and MADD subdomains.

When the numbers of subdomains or processors are increased, the angle quality of the resulting meshes is very similar. In Figure 7.6a, when the number of subdomains is increased, the distribution of the angles does not change much. Only in the 64-subdomain case are there some changes which can be easily seen. In particular, there is an increase in the number of triangles with angles in the 40-50 degree range. However, this increase in the percentage of triangles is less than 2%. When the number of processors is increased, all cases give very similar results as



(a) 4 subdomains

(b) 8 subdomains

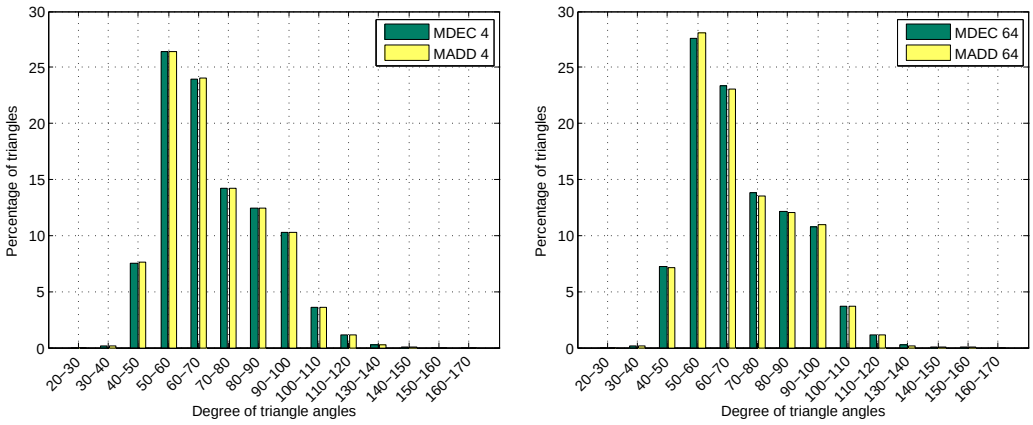
(c) 16 subdomains

(d) 32 subdomains

(e) 64 subdomains

Figure 7.4. Comparison of triangulation times for the PCDM algorithm using the subdomains from the MDEC and MADD algorithms as input for various numbers of subdomains on the Key model, respectively.

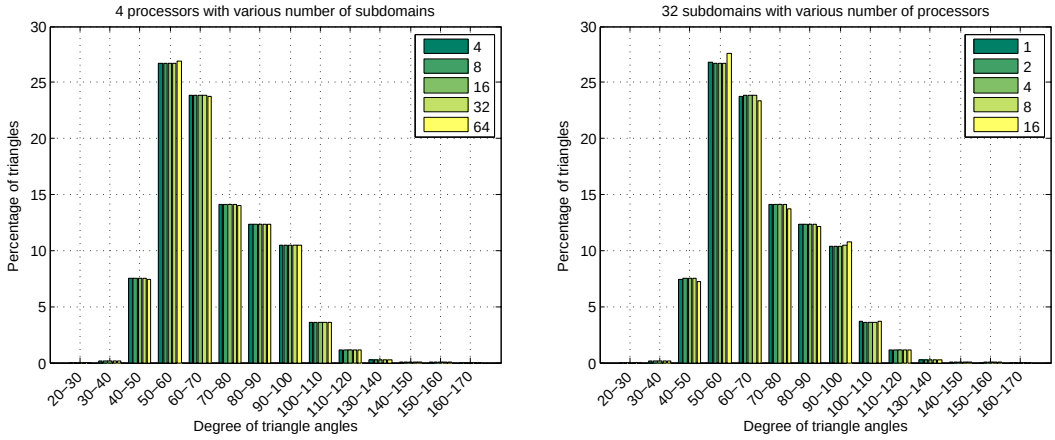
those shown in Figure 7.6b. From our experiments, we can conclude that the meshes generated by the PCDM algorithm using the MADD and MDEC subdomains as input, are similar in terms of mesh generation time and angle quality. Moreover, the angle quality of elements in the final mesh is not sensitive to changes in the



(a) 4 subdomains (b) 64 subdomains

Figure 7.5. Percentage of triangles in various angle ranges for various decompositions of the Key model using the MDEC and MADD algorithms.

numbers of subdomains and processors.



(a) 4 processors (b) 32 subdomains

Figure 7.6. Percentages of triangles in various angle ranges for the decompositions of the Key model

7.5 Conclusions and Future Work

We described a novel domain decomposition technique called the Metis-based Domain Decomposition (MDEC) algorithm for use with 2D parallel mesh generation.

The MDEC algorithm provides a good domain decomposition in terms of the decomposition time, the boundary angles of the resulting subdomains, and balancing of the subdomain areas. The MDEC algorithm yields better boundary angles, subdomain areas, and decomposition times than the MADD algorithm. However, because the decomposition of the MDEC algorithm is based on a background mesh, MDEC cannot be used to generate a decomposition with guaranteed good boundary angles ($\geq 60^\circ$) if the background mesh does not provide triangle elements with all angles $\geq 30^\circ$.

There are several possibilities for future work. First, we plan to develop an extension of the MDEC algorithm and corresponding theory to handle quadrilateral background meshes. Second, we note that if the weight of the node used in the graph partitioned routine is changed, a different domain composition may result. For example, in this chapter, we set the weight of the node to be the area of the corresponding triangle, which led to a balancing of the areas of the resulting subdomains. If, however, the node weights are each set to 1, the triangle density would be balanced. We plan to study whether or not this particular type of decomposition will prove useful for balancing the workload of the parallel mesh generator if the mesh generation is focused on the triangle density in certain areas of the domain. Finally, we plan to extend the MDEC algorithm to handle 3D meshes containing tetrahedral and/or hexahedral elements. In this case, there are two main issues which need to be considered: the choice of angle to focus on (i.e., the dihedral angle versus the solid angle) and how to guarantee good boundary angles.

Chapter 8
A Parallel Log Barrier-Based Mesh Warping Algorithm for Distributed-Memory Machines

8.1 Introduction

As computational simulations become more complex and are often multiphysics and multiscale in nature, it is important that meshes be generated and manipulated in parallel on either parallel clusters or multicore machines. The SciDAC Interoperable Technologies for Advanced Petascale Simulations (ITAPS) Center [63] is one example of a large project that addresses the needs of petascale mesh simulations. Furthermore, several parallel mesh generation techniques have been developed (see [31] for a survey); recent techniques have been developed for parallel Delaunay mesh generation (e.g., [6, 28, 29, 46, 89]), parallel advancing front mesh generation (e.g., [37, 76–78]), and parallel edge subdivision mesh generation (e.g., [21, 37, 92, 102, 103, 135]).

There are several areas of active research involving parallel post processing of meshes. For example, parallel mesh quality improvement and untangling algorithms have been developed which employ numerical optimization methods to untangle the mesh and improve its quality by repositioning the nodes [12, 107]. Parallel remeshing and mesh adaptation methods have also been proposed which alter the

The work of this chapter has been accepted for publication in:
[96] T. Panitanarak and S.M. Shontz, "A parallel log barrier-based mesh warping algorithm for distributed memory machines," Accepted to Engineering with Computers, April 2017.

mesh topology in order to improve its quality; often times this is done in response to a change in the PDE solution [52, 72, 82].

However, in regards to parallel mesh warping, only a few algorithms have been developed for use in computational simulations with deforming domains. For such applications, the mesh must be updated in parallel at each time step in response to the deforming boundary of the geometric domain. Parallel algorithms have been developed which combine mesh warping with topological operations [108, 109, 111]. Other parallel mesh warping algorithms have been designed for use in computational fluid dynamics (CFD) applications [49, 129]. More recent research on parallel mesh warping algorithms includes meshless techniques developed in [42, 83] but still focuses on CFD applications.

In this chapter, we propose parallel LBWARP, a parallel log barrier-based mesh warping algorithm for distributed memory systems. Parallel LBWARP is a parallel formulation of the general-purpose, geometric mesh warping algorithm named LBWARP which was proposed by Shontz and Vavasis [113]. Even though LBWARP is computationally intensive when a single deformation is applied, it is rather efficient when multiple deformations are performed. In this case, the computational complexity and also the overall run time of the algorithm decreases significantly. We discuss this advantage of parallel LBWARP in more detail in the chapter. The remainder of the chapter is organized as follows. In Sections 8.2 and 8.3, we provide overviews of the sequential LBWARP algorithm and sparse linear solvers used by the LBWARP algorithm, respectively. Then, we describe our parallelization of the sequential LBWARP method and introduce parallel LBWARP in Section 8.4. In Section 8.5, an analysis of the run time of the parallel LBWARP algorithm is discussed. We describe several numerical experiments which were designed to test the performance of our parallel LBWARP method on 3D domains and the resulting run times, speedup, and strong and weak scalability results in Section 8.6. In Sections 8.7, we demonstrate an application of parallel LBWARP on heart motion problems. In Section 8.8, we summarize our work and discuss some future research possibilities related to extensions of our parallel algorithm.

8.2 An Overview of LBWARP

LBWARP is a log barrier-based mesh warping algorithm which was proposed by Shontz and Vavasis [113]. The algorithm consists of three main steps. The first step is to generate a set of local weights (or inverse distances) describing the relationship of each interior node to its neighbors using a log barrier technique. These sets of weights are computed using an interior point method from nonlinear programming. Next, the boundary nodes are deformed by applying a transformation given by the user. Lastly, a system of linear equations is constructed from the sets of weights in the first step and the new positions of the transformed boundary nodes from the second step. This linear system is then solved for the final positions of the interior nodes.

Assume that a 3D mesh has m and n interior and total nodes, respectively. To compute the weights for interior nodes, a local optimization problem is solved for the coordinates of each interior node using a log barrier technique. The optimization problem is as follows

$$\begin{aligned} & \max_{w_{ij}, j \in N_i} \sum_{j \in N_i} \log(w_{ij}) \\ \text{subject to} \quad & w_{ij} > 0, \\ & \sum_{j \in N_i} w_{ij} = 1, \\ & x_i = \sum_{j \in N_i} w_{ij} x_j, \\ & y_i = \sum_{j \in N_i} w_{ij} y_j, \\ \text{and} \quad & z_i = \sum_{j \in N_i} w_{ij} z_j, \end{aligned} \tag{8.1}$$

where w_{ij} is the weight of node j acting on node i, N_i is the set of neighbors of node i, (i.e., $j \in N_i$ if and only if node j is connected to node i), and x_i, y_i, and z_i are the xyz-coordinates of node i. Since the objective function along with the constraints forms a strictly convex optimization problem, there exists a unique solution which can be found using an interior point method provided an initial feasible point exists [113]. In this chapter, we solve equation (8.1) using the projected Newton method [14] with an initial feasible point in the interior of the domain as described

in [113].

Consider the convex combinations of x_i, y_i, and z_i in (8.1), by defining a weight matrix A where $A(i,j) = -w_{ij}$ and $A(i,i) = 1$. Assuming the interior nodes are numbered first, and the boundary nodes are numbered last, the matrix A can be written as $[A_I \; A_B]$. A_I is an $m \times m$ matrix specifying how each node is connected to each of its interior neighbors, and A_B is an $m \times (n-m)$ matrix specifying how each interior node is connected to each of its boundary neighbors. Similarly, let x, y, and z be vectors of the x-, y-, and z-coordinates of all of the mesh nodes, respectively. Thus, we have $x = [x_I \; x_B]^T$, $y = [y_I \; y_B]^T$, and $z = [z_I \; z_B]^T$. Each of the subvectors x_I, y_I, and z_I contain coordinates of the m interior nodes, and each of the subvectors x_B, y_B, and z_B contain coordinates of the $n-m$ boundary nodes. Hence, we can write the resulting linear system in (8.1) as follows

$$A_I[x_I \, y_I \, z_I] = \; -A_B[x_B \, y_B \, z_B]. \tag{8.2}$$

Once the representation of the initial mesh has been determined, a user-specified boundary deformation can be applied as follows:

$$[x_B \, y_B \, z_B] \rightarrow \; [\hat{x}_B \, \hat{y}_B \, \hat{z}_B]. \tag{8.3}$$

Final positions of the interior nodes in the deformed mesh, i.e., $\hat{x_I}, \hat{y_I}$, and $\hat{z_I}$ are then determined by solving the following linear system based on equation (8.2) and the updates in equation (8.3);

$$A_I[\hat{x}_I \, \hat{y}_I \, \hat{z}_I] = \; -A_B[\hat{x}_B \, \hat{y}_B \, \hat{z}_B]. \tag{8.4}$$

See [113] for more details.

Although the LBWARP algorithm is initially computationally intensive due to the construction of the weight matrix, once the weight matrix has been computed, the LBWARP algorithm can reuse this matrix in additional mesh deformations provided an LU factorization method is used to solve the linear system. In other words, only the boundary deformation and the linear solution steps are performed for additional deformations. The complexity is then typically $O(m^2)$ for additional deformations but depends on the number of nonzero elements in A_I, i.e., $nnz(A_I)$ per deformation [16].

8.3 Sparse Linear Solvers with Multiple Right-Hand Sides

The choice of linear solver which is employed to solve (8.4) affects the runtime of the warping algorithm. Certainly, the sparsity and structural symmetry of A_I should be taken into account when selecting a linear solver. Note that the complexity of iterative linear solvers is usually in the form of $O(m^2)$. However, since our matrices are both very sparse and well-conditioned, the solver runtime is closer to O(m) for these problems.

Moreover, each deformation requires three linear solves, i.e., one for each of the x-, y- and z-coordinates (for 3D tetrahedral meshes). Each linear solve employs the same left-hand side matrix but a different right-hand side (RHS) vector. Thus, a single linear solver which takes into account the above properties and is able to address the multiple right-hand side problem should be employed.

Both direct and iterative methods can be used to solve (8.4), and both categories of methods have their advantages and disadvantages. Direct solvers directly support multiple RHS vectors, but their use can increase the number of nonzero elements in the matrix during row elimination. Thus, matrix reordering is required to minimize any nonzero fill-in. On the other hand, iterative solvers do not require reordering but instead need to be modified to support multiple RHS.

Thus, in this chapter, we consider the use of three different parallel sparse linear solvers for the solution of (8.4) in parallel LBWARP. Next, we give an overview of the serial versions of these methods.

Block BiCG (BiConjugate Gradient) [91] is a modified version of the biconjugate gradient (BiCG) method [43] to support multiple RHS. Block BiCG is essentially identical to the standard BiCG method with the only difference being that operations are performed with multivectors instead of single vectors. Thus, both methods are based on the conjugate gradient (CG) method [59] with an extension to provide a capability for solving nonsymmetric linear systems $Ax = b$. As in CG, the method uses search directions,

$$d^{(i+1)} = r^{(i+1)} + \beta_i d^{(i)}$$

to update the residuals, $r^{(i+1)}$, and solution approximations, $x^{(i+1)}$, such that

$$r^{(i+1)} = r^{(i)} - \alpha_i A d^{(i)} \quad \text{and} \quad x^{(i+1)} = x^{(i)} + \alpha_i d^{(i)},$$
$$\text{where} \quad \alpha_i = r^{(i)^T} r^{(i)} / d^{(i)^T} A d^{(i)} \quad \text{and} \quad \beta_i = r^{(i+1)^T} r^{(i+1)} / r^{(i)^T} r^{(i)}.$$

To handle nonsymmetric systems, instead of using only one orthogonal sequence of residuals and conjugate directions as in CG, BiCG uses two mutually orthogonal sequences (or two sequences of residuals and conjugate directions). In addition to computing $d^{(i)}$ and $r^{(i)}$, $\tilde{d}^{(i)}$ and $\tilde{r}^{(i)}$ are also computed similarly by replacing A with A^T. Moreover, the computations of α_i and β_i are replaced by

$$\alpha_i = \tilde{r}^{(i)^T} r^{(i)} / \tilde{d}^{(i)^T} A d^{(i)} \quad \text{and} \quad \beta_i = \tilde{r}^{(i+1)^T} r^{(i+1)} / \tilde{r}^{(i)^T} r^{(i)}.$$

Thus, the cost per iteration of BiCG is approximately twice that of CG. The cost per iteration of BiCG is approximately the cost of computing two inner products, five scalar-vector multiplications and additions, and two matrix-vector products or $O(12n + 2n^2)$, where n is the length of the vector x.

To handle multiple RHS, BiCG can be modified to solve all RHS at once or to solve the system $AX = B$ where X and B are $n \times c$ matrices and c is the number of right-hand side vectors. To update the approximations to the solution, the following formulas are now used

$$D^{(i+1)} = R^{(i+1)} + D^{(i)} \mathrm{B}^{(i)}, \qquad R^{(i+1)} = R^{(i)} - A D^{(i)} \mathrm{A}^{(i)},$$
$$\tilde{D}^{(i+1)} = \tilde{R}^{(i+1)} + \tilde{D}^{(i)} \mathrm{B}^{(i)}, \qquad \tilde{R}^{(i+1)} = \tilde{R}^{(i)} - A \tilde{D}^{(i)} \mathrm{A}^{(i)},$$

and

$$X^{(i+1)} = X^{(i)} + D^{(i)} \mathrm{A}^{(i)}$$

where

$$\mathrm{A}^{(i)} = (\tilde{D}^{(i)^T} A D^{(i)})^{-1} \tilde{R}^{(i)^T} R^{(i)} \quad \text{and} \quad \mathrm{B}^{(i)} = (\tilde{R}^{(i)^T} R^{(i)})^{-1} \tilde{R}^{(i+1)^T} R^{(i+1)}.$$

Note that D and R are now $n \times c$ matrices, while A and B are $c \times c$ matrices.

As we can see, the complexity of the algorithm has increased, as it requires more matrix operations. Furthermore, the convergence and stability of the algorithm are also affected. Thus, additional efforts to maintain these properties are needed such as deflation, i.e., removing some right-hand sides from the process (see [91, 117]). Since operations on vectors are now performed using matrices, for c right-hand sides, the complexity of block BiCG is $O(12cn + 2cn^2)$.

Block GMRES (Generalized minimal residual) [118] is a modified version of GMRES [105] to support multiple RHS using a similar approach as the block BiCG method. GMRES itself is an extension of the minimal residual (MINRES) method which can only be used to solve symmetric systems. Similar to MINRES, it approximates the solution by generating a sequence of orthogonal vectors with minimal residual. However, without symmetry, all previously generated vectors are needed and must be retained to construct the approximations as follows:

$$x^{(i)} = x^{(0)} + y_1 v^{(1)} + \cdots + y_i v^{(i)},$$
$$\text{where } \; y_k \; \text{ minimizes } \; \|b - Ax^{(i)}\| \; \text{ and } \; v^{(i+1)} = w^{(i)}/\|w^{(i)}\|.$$

For each i, $w^{(i)}$ is initialized with $Av^{(i)}$ and explicitly updated by $w^{(i)} = w^{(i)} - (w^{(i)}, v^{(k)})v^{(k)}$ for $k = 1, \ldots, i$. By defining $r^{(0)} = b - Ax^{(0)}$ and the $(k+1) \times k$ upper Hessenburg matrix H_k from the Arnoldi's relation $AV_k = V_{k+1}H_k$ where V_k is the orthonormal basis of the Krylov subspace $K_k(A, r^{(0)})$ build by the Arnoldi's procedure, we can compute $y_k = \|r^{(0)}\|H_k^{-1}e_1$. The cost of the the k^{th} iteration of GMRES (without restarting) is approximately the total time of $k+1$ inner products, $k+1$ scalar-vector multiplications and additions, and one matrix-vector product or $O(3(k+1)n + n^2)$. Consequently, the extension to block GMRES results in $O(3(k+1)cn + cn^2)$ complexity.

LU **Decomposition** is a factorization of a matrix A into LU[1], where L and U are $n \times n$ unit lower triangular and upper triangular matrices, respectively. This factorization can be used to indirectly solve the linear system $Ax = b$ with the equivalent system $LUx = b$ which can be solved by performing two triangular solves

[1]The factor LU exists only if A is nonsingular. In the event an element on the diagonal of A is zero or nearly zero, partial pivoting is required.

as follows:

$$Ax = b \Leftrightarrow LUx = L(Ux) = Ly = b.$$

More specifically, forward substitution is performed to solve for y from $Ly = b$, and then backward substitution is performed to solve for x from $Ux = y$.

The LU decomposition can be directly applied to any matrix including sparse matrices. However, a row operation during the decomposition can result in the generation of additional nonzero elements which were previously zero (called nonzero fill-in). This increases memory usage and the number of computations in the algorithm. In this chapter, we use nested dissection, a fill-reducing ordering, in order to minimize nonzero fill-in which occurs during the LU decomposition. For a dense $n \times n$ matrix, the complexity of a standard LU decomposition is $O(\frac{2}{3}n^3)$. However, the algorithm can be specialized for matrices based on their sparsity pattern. This lowers the complexity in practice.

8.4 Parallel LBWARP

In this section, we describe our parallel formation of the LBWARP algorithm; the resulting method is referred to as parallel LBWARP. The parallel LBWARP method contains three steps, i.e., weight generation, boundary deformation, and linear solution.

8.4.1 Parallelization of Weight Generation Step

As described in the previous section, the first step of the LBWARP algorithm is to use optimization to generate a set of local weights which specifies how a given interior node is represented as a convex combination of its neighbors. To parallelize this step, it is important that the interior nodes are equally distributed among the processors to balance the workload. Assume that p processors are used for solving m optimization problems corresponding to the m interior nodes. A subset of $\lceil m/p \rceil$ consecutive interior nodes are assigned to each processor for simplicity to identify the ownership of the distributed interior nodes. As the computation during the weight generation step is node-based, we represent a mesh as a graph

such that each graph node and edge represent the corresponding mesh node and edge, respectively, and the connectivity among nodes is preserved.

Before the actual computation of the weight matrix begins, the neighbors of each subset of the interior nodes are pre-computed and sent to the corresponding processors. There will be redundant copies of nodes among processors, i.e., one node can be assigned to more than one processor, and the amount of memory required by each processor varies since the number of neighbors for each subset of the interior nodes is different. This redundancy in sending boundary nodes will not scale arbitrarily. Even though mesh partitioning can be used to balance the distribution, it also introduces additional complexity, and can decrease the overall performance.

Fig. 8.1 demonstrates our partitioning approach. First, interior nodes are determined as shown (Fig. 8.1(a)). Then, they are equally distributed to the processors (Fig. 8.1(b)). After that, the neighbors of each subset of interior nodes are computed and sent to the corresponding processors (Fig. 8.1(c)). Once all processors receive their subsets of interior nodes and the subset's neighboring nodes, they compute their local weights independently in an embarrassingly parallel manner using the projected Newton method without communication during the weight computations.

During the weight generation step, the $m \times n$ matrix $A = [A_I \; A_B]$ is constructed by formulating and solving m local optimization problems. More specifically, solving a single optimization problem for the i^{th} interior node given by (8.1) yields the i^{th} row of A. With the parallel approach described above, since each processor acquires a subset of approximately $\lceil m/p \rceil$ consecutively numbered interior nodes and all of their neighbors, approximately $\lceil m/p \rceil$ consecutive rows of the matrix A can be generated simultaneously on p processors without communication. After the processors finish generating the local weights for their assigned interior nodes, each processor owns non-overlapping, consecutive rows of the weight matrix A, (i.e., processor zero generates rows one through $\lceil m/p \rceil$, processor two generates rows $\lceil m/p \rceil + 1$ through $2\lceil m/p \rceil$, and so on).

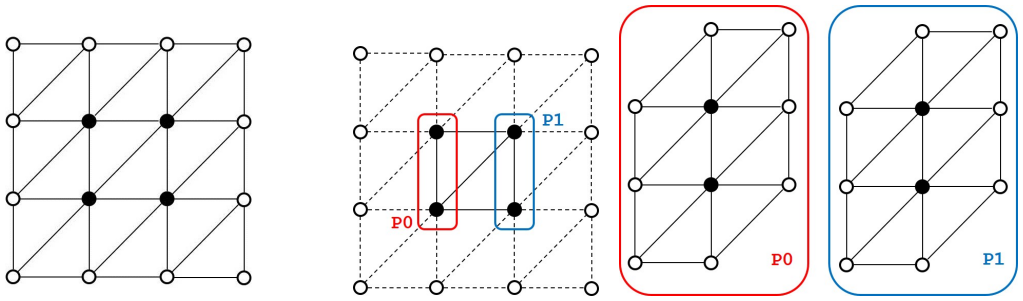


Figure 8.1. (a) The original mesh before partitioning; the black and white nodes represent the interior and boundary nodes, respectively. (b) For load-balancing purposes, first, only interior nodes are partitioned and sent to processors. (c) Then, neighbors of each subset of interior nodes are computed and sent to the corresponding processors.

8.4.2 Parallelization of Boundary Deformation Step

The next step is to apply the deformation of the boundary nodes in parallel. This step involves computing the right-hand side of equation (8.4). Since part of the matrix A_B is generated and owned locally by each processor during the weight generation step, the boundary deformation step can also be performed in an embarrassingly parallel manner by sending each processor the coordinates of the boundary nodes, i.e., $\hat{x}_B$, $\hat{y}_B$, and $\hat{z}_B$. Although there is redundancy in all of the processors owning the entire set of boundary nodes, this allows the processors to simultaneously compute their respective parts of the right-hand side vector without communication. This is equivalent to computing a portion of the right-hand side vector locally on each processor.

8.4.3 Parallelization of Linear Solution Step

The last step is to solve the three linear systems in equation (8.4) for the final positions of the interior nodes in the deformed mesh, i.e., $\hat{x}_I, \hat{y}_I$, and $\hat{z}_I$. The parallel linear solver which is used in the linear solution step with the distributed matrix A_I should take advantage of its sparsity and also support multiple right-hand side vectors.

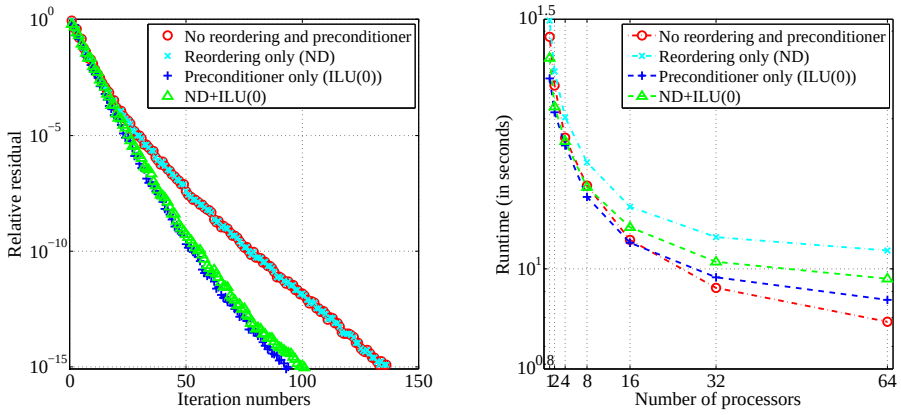
To this end, we consider parallel versions of block BiCG, block GMRES, and LU decomposition for distributed memory machines for use in the linear solution step of parallel LBWARP. We refer to them as DistBlBiCG, DistBlGMRES, and

DistLU, respectively. We chose these three algorithms based on our preliminary experiments (e.g., the deformation of the Menger sponge mesh shown in Fig. 8.5 from (a) to (b)) with the parallel block GMRES and SuperLU_DIST algorithms [73] implemented in the Amesos2 and Belos packages [9] for the Trilinos project [128]. Our preliminary results with the Trilinos package demonstrated good speedups when solving linear systems based on multiple RHS solvers on matrices generated from the weight generation step. We implement our own linear solvers as we wish for the mesh and linear algebra calculations to be implemented in the same manner; this is important for our timing tests. The data suggests our linear solvers are a bit faster and more scalable, which is an added benefit of implementing the solvers ourselves.

Distributed block BiCG and distributed block GMRES: Our implementations of DistBlBiCG and DistBlGMRES are based on [8, 91, 117, 118]. The $n \times n$ sparse matrix is distributed based on a row-wise distribution among the p processors. Thus, each processor owns a non-overlapping $\lceil n/p \rceil$ consecutive rows of the original matrix. A RHS vector corresponding to the matrix is also distributed in a similar manner. Thus, matrices that normally cannot be fit in memory on a single processor can be processed with this approach. However, there is a trade-off in terms of processing time since some matrix-matrix and/or matrix-vector operations typically require off-processor information in the form of message-passing communication. The two main parallel routines in the main loop of both solvers are matrix-vector multiplication and vector dot product. For parallel matrix-vector multiplication, this can be done by distributing the vectors (d or v for DistBlBiCG or DistBlGMRES, respectively) to all processors according to the rows of matrix that each processor owns. Then, each processor multiplies the received vector with its own rows and the result vector is stored. For parallel dot product, each processor computes the partial result of the inner product from the rows which it owns. After that, these partial results are summed globally using MPI_Reduce to obtain the inner product.

We performed a set of preliminary experiments in order to determine whether or not it was necessary to perform either reordering on A_I or preconditioning when solving the linear system in equation (8.4). We experimented with the use of the nested dissection (ND) reordering and application of an ILU(0) preconditioner

on block GMRES using the Belos package. Typical results from our preliminary experiments can be seen in Fig. 8.2. Although applying the preconditioner increases the convergence rate (Fig. 8.2(a)), it only reduces the linear solution time when up to 16 processors are used. For a larger number of processors, (i.e., 32 processors or more), the overhead of computing the preconditioner becomes more visible, as the size of our deforming mesh problem is not large enough (i.e., our preliminary experiments are for a deforming mesh with approximately 6M nodes) for this strategy to pay off (Fig. 8.2(b)). (For this problem, the linear solution step takes less than 10% of the overall warping time, which is around 20 seconds.) Given the relatively low condition number of the weight matrix (i.e., 37 for this problem), the linear systems can be solved without use of a preconditioner and can still obtain a good convergence rate. In such cases, it should be clear that is it the most beneficial to employ block GMRES without either reordering or preconditioning. Thus, we do not use either reordering or preconditioning when solving the linear systems in equation (8.4) with the DistBlBiCG and DistBlGMRES solvers.



(a) Convergence rate (b) Runtime

Figure 8.2. The effect of the nested dissection reordering and ILU(0) preconditioner on (a) convergence rate and (b) runtime of the block GMRES solver. Note that a log scale is used for the vertical axis in (b).

Distributed LU: Like other sparse LU factorizations, our DistLU algorithm consists of reordering to reduce fill-in, symbolic factorization, numerical factorization, and triangular solves. While partial pivoting is essential for general matrices, the factorization of A_I does not require partial pivoting since A_I is weakly diagonally

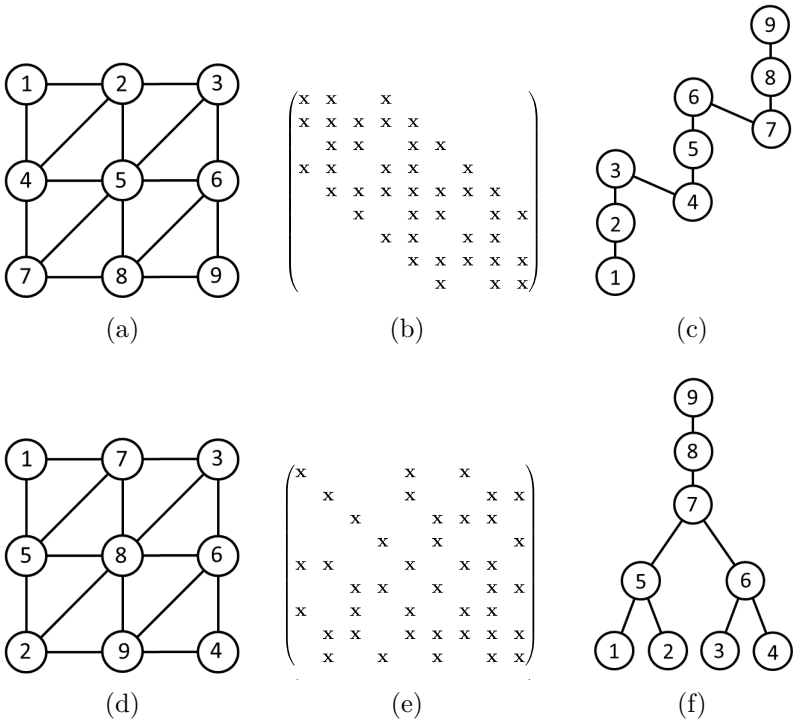


Figure 8.3. A mesh with natural ordering (a), its adjacency matrix (b), and its elimination tree (c). The same mesh after applying nested dissection reordering (d) and the corresponding adjacency matrix (e) and elimination tree (f). Note that '**x**' indicates a nonzero element in the matrix.

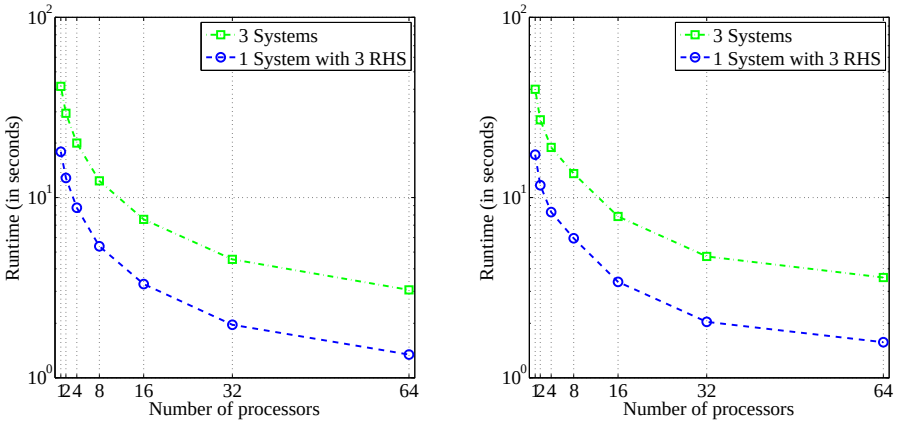
dominant. Without partial pivoting, some communication and computation can be avoided. Also, based on the symmetric structure of A_I, the symbolic factorization (which is used to determine the fill-in in the L and U factors) is easier to compute than for unsymmetric matrices. More specifically, our parallel sparse LU factorization is similar to SuperLU_DIST [73] introduced by Li *et al.* but is simpler in terms of the amount of computation performed and the complexity. We apply the ND reordering [2] which reduces fill-in for the sparse matrix A_I using MeTiS [69]. The algorithm finds an elimination ordering of the matrix using a divide and conquer approach. The new elimination ordering can be used to exploit the matrix columns that can be updated simultaneously, (i.e., they are independent). The result of the nested dissection algorithm also yields an elimination graph or task graph of the matrix to use as an elimination order of all of the columns. Fig. 8.3 illustrates meshes with their natural ordering and after applying a nested dissection reordering along with their adjacency matrices and elimination trees.

A mesh with natural ordering (Fig. 8.3(a)), its adjacency matrix and elimination tree are shown in Figs. 8.3(b) and 8.3(c), respectively. The elimination ordering is sequential and dependent on the previous nodes. Thus, to be able to perform elimination of the k^{th} column of the matrix, all columns from one up to $(k-1)$ need to be eliminated first. Figure 8.3(d) shows the same mesh as above after nested dissection reordering is applied. Similarly, its adjacency matrix and elimination tree are shown in Figs. 8.3(e) and 8.3(f), respectively. Now, since nodes 1-4 are pairwise independent, columns 1-4 of the matrix can be eliminated simultaneously. Similarly, columns 5-6 can also be eliminated at the same time.

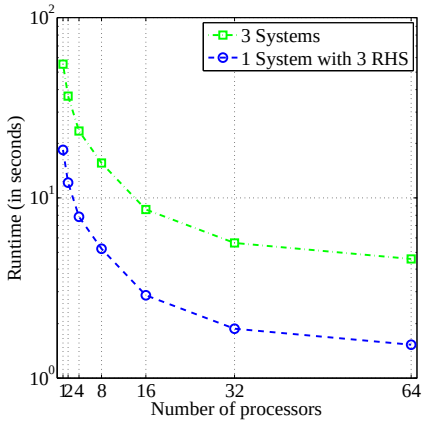
Benefits of parallel linear solvers with multiple RHS support: A comparison of the runtimes for the three parallel sparse linear solvers, i.e., the DistBlBiCG, DistGMRES, and DistLU algorithms, with and without multiple RHS vector support, is shown in Fig. 8.4. Recall that solving equation (8.4) for the final positions of the interior nodes in the deformed mesh requires three linear systems to be solved, i.e., one linear system per nodal coordinate. For systems without multiple RHS vector support, the linear systems are solved independently. Whereas, the nodal coordinates are solved for simultaneously when linear solvers with multiple RHS vector support are employed. Simultaneous linear solves result in reduced runtime because the overhead during the initialization and some additional operations can be combined and reused in order to avoid redundant computation.

8.4.4 Parallel LBWARP Algorithm

The complete parallel LBWARP algorithm is shown in Algorithm 11. Assume that interior nodes are distributed in a row-wise distribution among all processors. Thus, each processor owns non-overlapping $\lceil n/p \rceil$ interior nodes including their x, y, and z coordinates, and the completed target boundary coordinates, $\hat{x}_B$, $\hat{y}_B$, and $\hat{z}_B$. The first step of the algorithm (lines 1-6) is to generate neighbor lists which will be used in the next step. It involves **MPI_Alltoallv** communication to exchange the coordinates of non-local neighbors. After that, each processor can compute its local weights independently without any communication as shown in lines 7-14. Since each interior node corresponds to one row in the weight matrix, a set of local interior nodes at processor p results in the partial weight matrix rows, A.



(a) DistBlBiCG (b) DistBlGMRES



(c) DistLU

Figure 8.4. A comparison of the runtimes for the three parallel sparse linear solvers when solving with and without multiple RHS vectors: (a) DistBlBiCG, (b) DistBlGMRES, and (c) DistLU.

The rest of the algorithm focuses on construction and solution of the system (8.2) for $\hat{x}_I$, $\hat{y}_I$, and $\hat{z}_I$ using one of the parallel sparse linear solvers, i.e., DistBlBiCG, DistBlGMRES, or DistLU.

Lines 15-19 show the boundary deformation step. Each processor has its own copy of $\hat{x}_B$, $\hat{y}_B$, and $\hat{z}_B$. Thus, the processors simultaneously compute their respective parts of the right-hand side vectors using equation (8.4). Finally, the linear systems shown in equation (8.4) can be solved for $\hat{x}_I$, $\hat{y}_I$, and $\hat{z}_I$ using one of the three parallel sparse linear solvers, DistBlBiCG, DistBlGMRES, or DistLU.

Algorithm 11 Parallel LBWARP algorithm

```
// Generate neighbor lists
for all p processors in parallel do
    generate neighbor lists of all local interior nodes
    request coordinates of non-local neighbors using MPI_Alltoallv
end for
synchronization using MPI_Barrier
// Step 1: Generate a weight matrix, A = [A_I  A_B]
for all p processors in parallel do
    for each local node v do
        solve the optimization at v for the weight matrix row v
    end for
end for
synchronization using MPI_Barrier
// Step 2: Compute the right-hand side vectors in equations (8.4)
for all p processors in parallel do
    compute b_x = −A_B x̂_B, b_y = −A_B ŷ_B, b_z = −A_B ẑ_B
end for
synchronization using MPI_Barrier
// Step 3: Solve the linear systems (8.4) for x̂_I, ŷ_I, and ẑ_I
solve A_I x̂_I = b_x, A_I ŷ_I = b_y, A_I ẑ_I = b_z using
    either DistBlBiCG, DistBlGMRES or DistLU
```

8.5 Parallel Analysis

As described in Section 8.4, parallel LBWARP consists of three main steps, i.e., the weight generation, the boundary deformation, and linear solution the same as does LBWARP. We now discuss the performance gain in each these steps of parallel LBWARP.

Assume that the maximum time to solve a single optimization problem in (8.1) to determine the weights for interior node i is t_{op}. Thus, LBWARP takes at most mt_{op} total time to compute the sets of weights for the m interior nodes.

In the case of parallel LBWARP, for analysis purposes, assume optimal load balancing across all available p processors and that each processor is assigned to work on approximately $\lceil m/p \rceil$ distinct interior nodes. Hence, each processor takes at most $\lceil m/p \rceil t_{op}$ time to compute its own set of local weights. Since all processors can work simultaneously without any communication among them during this step, the total time to compute all weights is still $\lceil m/p \rceil t_{op}$. There is some additional

computation and communication to generate neighbor lists for the interior nodes which can be viewed as a conversion from the original mesh representation to a graph representation. It is easy to see that if the nodes belong to the same element, they are neighbors of one another. Assume that there are n_e elements in the mesh. For LBWARP, to find all possible neighbor relationships of the interior nodes, all mesh elements have to be visited. Thus, the total time for the sequential conversion is $n_e t_{con}$ where t_{con} is the maximum time required to inspect a single mesh element. For parallel LBWARP, if $\lceil n_e/p \rceil$ elements are distributed among all p processors, each processor can visit its elements in $\lceil n_e/p \rceil t_{con}$ time. However, a local neighbor list generated by each processor is not yet complete since each interior node belongs to multiple mesh elements and those elements may be distributed to different processors. To obtain the complete neighbor list for a given processor, each processor needs to gather neighbor lists from other processors which requires at most $(\lceil m/p \rceil pd)t_{com}$ or mdt_{com} time where d is the maximum degree of a node and t_{com} is the time to send a single message. Thus, the total time required in the weight generation step is bounded above by $mt_{op} + n_e t_{con}$ for LBWARP and $\lceil m/p \rceil t_{op} + \lceil n_e/p \rceil t_{con} + mdt_{com}$ for parallel LBWARP. In general, the time used for distribution and gathering is very small compared to the time for solving the optimization problems used to generate the weights. Fortunately, since there is no further communication after all local neighbor lists have been generated in the weight generation step, this step of parallel LBWARP is very scalable in terms of both strong and weak scaling.

The boundary deformation step involves parallel matrix-vector multiplication using each right-hand side vector as shown in equation (8.4).

Since rows of A_B have already been distributed among the processors, the only work that needs to be done is to distribute the vectors $\hat{x}_B$, $\hat{y}_B$ and $\hat{z}_B$ to the processors so that the processors can simultaneously compute portions of the right-hand side vectors based on the part of A_B that each one owns. Assuming the time to compute the right-hand side vectors sequentially is t_c, the approximate run time in parallel is $\lceil m/p \rceil t_c$.

In the linear solution step, the time required to solve the system depends on the algorithm used for this step. For iterative solvers, similar to the sequential versions, the complexity is based on the number of iterations required for convergence (and the time per iteration) which is approximately $O(n^2/p)$ per iteration with

some communication overhead. For the distributed LU, the complexity is based on $nnz(A_I)$ and the structure of A_I, which is approximately $M = \sum_{k-1}^{n} c_k r_k$, where c_k and r_k are the numbers of off-diagonal elements in each column of block column k and each row of block row k, respectively. Thus, the parallel runtime is approximately $O((nnz(A_I)+M)/p+H(p))$, where $H(p)$ is communication overhead of p processors from broadcasting messages.

8.6 Numerical Experiments

In order to test the performance of our parallel LBWARP algorithm, we perform several numerical experiments on 3D tetrahedral meshes. The implementation of parallel LBWARP is in C/C++ using the message-passing interface (OpenMPI version 1.7.3). We use the dense and sparse vector and matrix routines in the Eigen library [64] where vector or matrix operations are required. All of our experiments were run on the CyberStar cluster available for our use at The Pennsylvania State University [34]. More specifically, 192 Dell PowerEdge R610 servers were used. Each server provides 2 quad-core Intel Nehalem processors running at 2.66 GHz and 24 GB of RAM. In all of our experiments, we use only one core per server node in order to reduce the amount of duplicate data storage within the node. This allows for more efficient memory usage, which is important when performing parallel mesh warping on large-scale meshes (with tens of millions of elements). The step in our parallel LBWARP algorithm that requires the most memory is the neighbor computation step since the algorithm needs to hold both the mesh structure (read from a file) and the neighbor list (or graph that is generated from the mesh) and also the memory to hold vertex coordinates during this step. Although these steps are performed in parallel, the tasks will also need to be distributed to the appropriate processors. For both DistBlBiCG and DistBlGMRES, the relative convergence tolerance and the maximum numbers of iterations are set to 10^{-5} and 10^3, respectively. The initial guess vector for both algortihms is set to zero. The GMRES restart iteration is 30.

The 3D domains that we use in our experiments, i.e., Menger sponge and Luer connector, are shown in Figs. 8.5(a) and (d), respectively. Meshes on these domains were generated using TetGen [116] and have approximately 6M and 9M nodes, respectively. (Note that M here is defined as one million.) The deformed boundaries

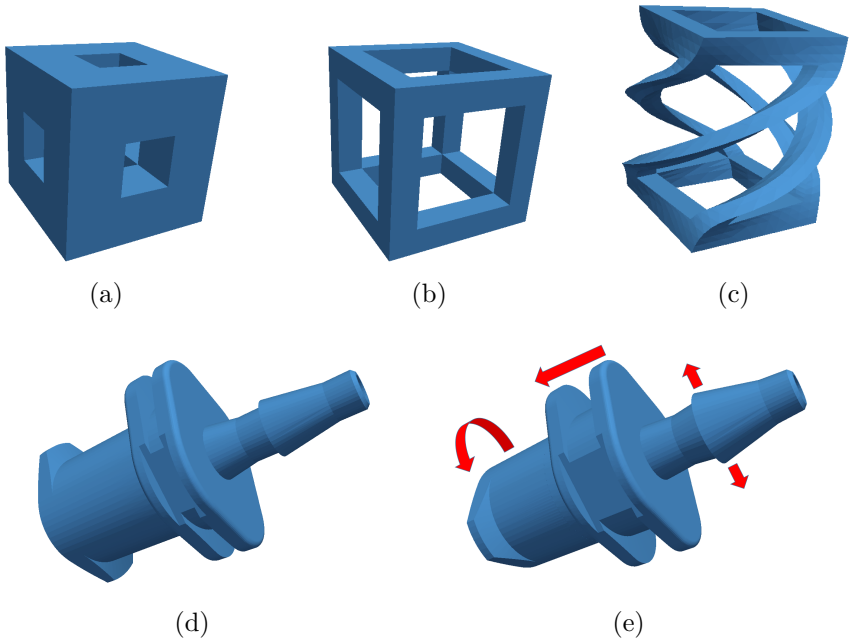


Figure 8.5. (a) The Menger sponge domain and its two (b and c) deforming boundaries, and (d) the Luer connector domain and its (e) deforming boundary.

of the Menger sponge are shown in Figs. 8.5(b) and (c), and the deformed boundary of the Luer connector domains is shown in Fig. 8.5(e). These boundaries are used in the boundary deformation step of the mesh warping process. For the Menger sponge mesh, the first deformation (Fig. 8.5(b)) was generated by increasing the size of all square holes by 50%. Mesh elements were compressed in one dimension and stretched in the other two dimensions. The latter deformation is much more pronounced near the hole. The second deformation (Fig. 8.5(c)) was generated by applying the first deformation, and then counter-clockwise twisting the model by 90 degrees while increasing the height of the model by 30%. In this case, mesh elements were extremely compressed and twisted, as we can see in the figure. For the Luer connector mesh, the deformation was generated by increasing the size of the small tube on the top, extending the gap between the two middle plates, and rotating the lowest plate by 90 degrees. With these deformations, the mesh elements around the top are affected by two-dimensional expansion. The mesh elements around the middle of the model are stretched in one dimension. Finally, the mesh elements around the bottom are both compressed and distorted. Statistics for both meshes, such as the numbers of nodes and tetrahedral elements in the

meshes and the mesh quality (as measured by the mean ratio (MR) mesh quality metric[2]) before and after the mesh deformation process are shown in Tables 8.1 and 8.2, respectively. Fig. 8.6 shows the spy plot of A_I for a coarse mesh of the initial Menger sponge model with (a) natural ordering and (b) nested dissection ordering, respectively. Note that the coarse mesh is used only for the purpose of visualizing the spy plot of A_I. As we can see from the figure, the mesh yields a very sparse matrix, A_I. A spy plot of A_I for the initial Luer connector model shows a similar result.

Table 8.1. The sizes of the Menger sponge and Luer connector meshes.

Mesh	# nodes	# elements
Menger Sponge	6,025,426	37,723,148
Luer Connector	10,523,992	59,291,516

Table 8.2. The mean ratio (MR) mesh quality of the Menger sponge and Luer connector meshes.

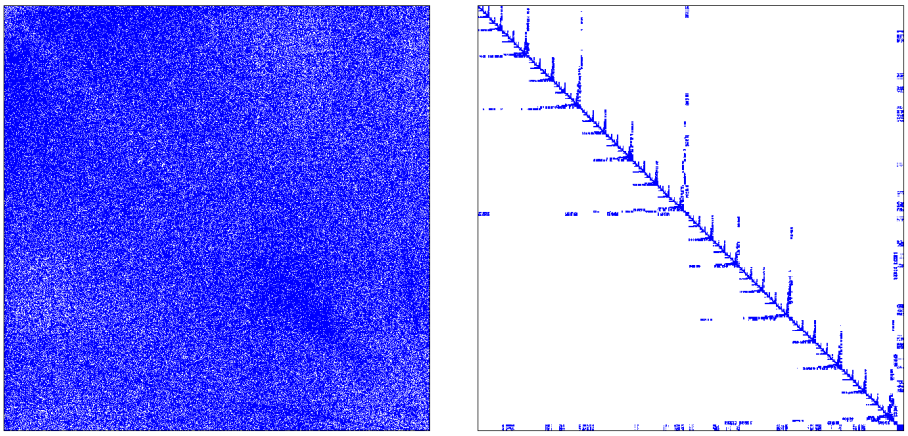
Mesh	Initial Mesh Quality (MR)			Final Mesh Quality (MR)		
	Max.	Avg.	Min.	Max	Avg.	Min.
Menger Sponge	1.0000	0.7842	0.2196	1.0000	0.3894	0.0177
(twisted)	-	-	-	1.0000	0.2059	0.0102
Luer Connector	1.0000	0.7180	0.1926	1.0000	0.2877	0.0159

Figs. 8.7(a) and (b) show the total runtime and speedup of the parallel LBWARP algorithm using DistLU for the linear solution step on the Menger sponge and Luer connector meshes running on different numbers of processors, respectively. (Note a log scale is used for the vertical axis.) For the Menger sponge mesh, although there are two different deformations, the total runtime for both deformations is the same when using DistLU in the linear solution step, as they generate and solve the same weight matrix. The experiments on the two meshes gave very similar results in terms of runtime and speedup. They both achieve speedup very close to the ideal

[2]The MR mesh quality metric is given by

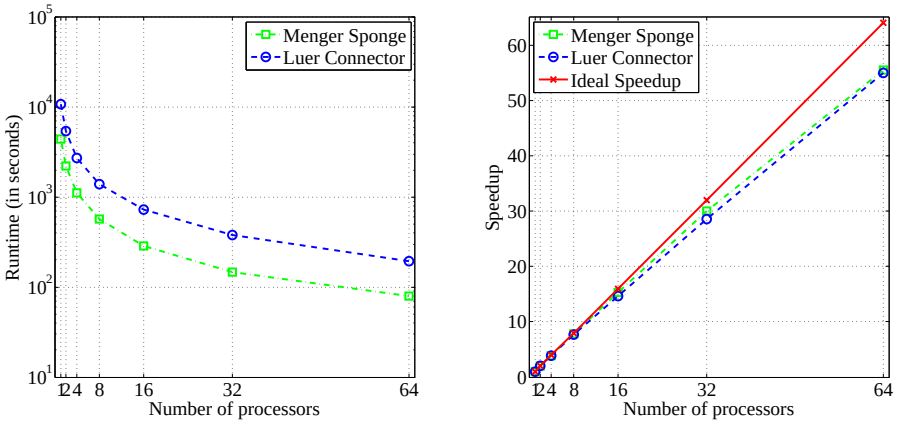
$$\eta = \frac{12(3v)^{2/3}}{\sum_{0 \le i < j \le 3} l_{ij}^2} \text{ [75]}, \tag{8.5}$$

where v and l_{ij} denote the volume and various edge lengths of the tetrahedron, respectively. Note the ideal mesh quality occurs when $\eta = 1$, and 0 denotes a degenerate tetrahedron. The range of the metric is 0 to 1. Higher values denote better quality.



(a) Menger sponge: Natural ordering (b) Menger sponge: Nested Dissection ordering

Figure 8.6. The spy plots of A_I for a coarse mesh on the initial Menger sponge model with (a) natural ordering and (b) nested dissection reordering, respectively.



(a) Total runtime (b) Speedup

Figure 8.7. (a) The total runtime and (b) speedup of parallel LBWARP using DistLU on the Menger sponge and Luer connector meshes.

speedup for a small number of processors. This slightly decreases as the number of processors increases. The runtime for the weight generation step dominates the overall time as shown in Tables 8.3 and 8.4 resulting in good strong scaling of the algorithm since this step is the most effective one to parallelize. We observe some slight performance deterioration due to the overhead in the pre-processing step, i.e., computation of the neighbor lists. However, the overall scalability of the algorithm

Table 8.3. Breakdown of the runtime (in seconds) for parallel LBWARP: neighbor computation, weight generation, boundary deformation, and linear solution steps using DistLU on the Menger sponge mesh.

# procs.	Neighbor Computation	Weight Generation	Boundary Deformation	Linear Solution
1	48.74	4311.76	7.43	26.08
2	32.95	2169.22	5.27	16.72
4	21.23	1097.54	3.55	10.98
8	16.12	539.98	2.74	7.68
16	11.59	274.56	1.77	4.95
32	7.56	142.92	1.24	3.69
64	3.78	76.59	0.86	3.12

Table 8.4. Breakdown of the runtime (in seconds) for parallel LBWARP: neighbor computation, weight generation, boundary deformation, and linear solution steps using DistLU on the Luer connector mesh.

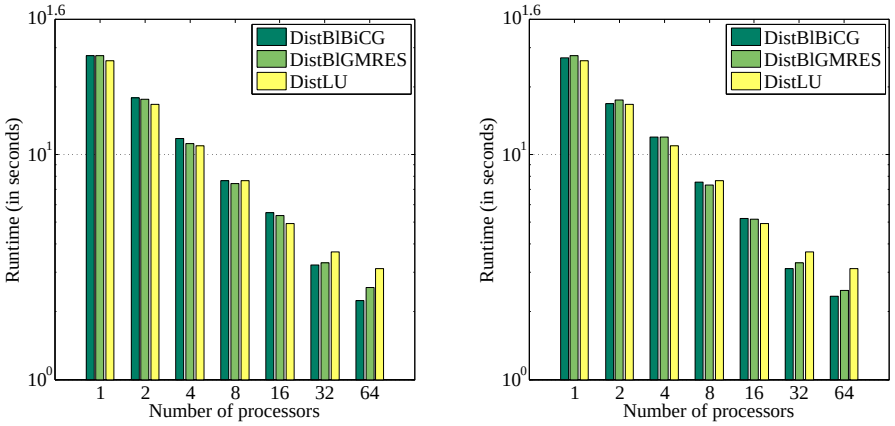
# procs.	Neighbor Computation	Weight Generation	Boundary Deformation	Linear Solution
1	102.41	10,593.24	13.98	35.93
2	65.20	5,365.32	11.02	23.29
4	44.52	2,679.88	7.51	16.06
8	32.74	1,342.59	5.95	11.32
16	20.21	704.73	4.32	8.63
32	14.82	342.67	3.64	6.64
64	6.18	186.99	2.37	5.75

Table 8.5. Weak scaling results for parallel LBWARP using DistBlBiCG on the Luer connector mesh.

# procs.	Avg. # nodes per proc.	Avg. # elements per proc.	Time (s.)	Scaling factor
1	203,182.00	1,032,988.00	132.64	1.00
2	196,294.00	1,062,265.50	151.12	0.88
4	218,869.00	957,434.50	168.15	0.79
8	184,865.38	855,358.00	171.63	0.77
16	152,833.00	907,685.44	175.23	0.76
32	163,799.56	966,950.56	180.74	0.73
64	164,437.38	926,430.94	182.31	0.73

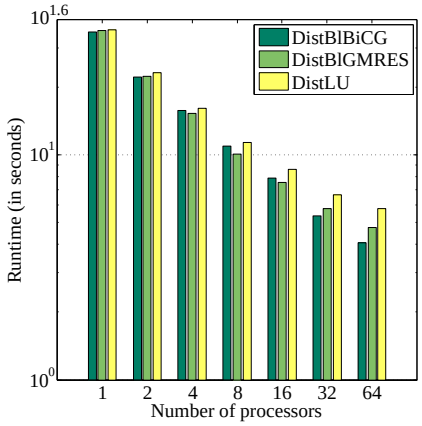
is still good up to 64 processors. We have not extended our experiments to more than 64 processors due to limited processor accessibility on the Cyberstar cluster.

We compare the performance of the three linear solvers, i.e., the DistBlBiCG,



(a) Menger sponge

(b) Twisted Menger sponge



(c) Luer connector

Figure 8.8. The runtime (in seconds) for the parallel sparse linear solvers for the (a) Menger sponge, (b) Luer Connector, and (c) twisted Menger sponge meshes.

DistBlGMRES, and DistLU solvers, as shown in Fig. 8.8. The figure shows the runtime of the linear solution step of the parallel LBWARP algorithm for the (a) Menger sponge, (b) twisted Menger sponge, and (c) Luer connector meshes. The DistLU solver gives good performance on a smaller number of processors and on smaller meshes, (i.e., with 6M nodes for the Menger sponge mesh), with up to 8 processors, as we can see from Fig. 8.8(a) and Fig. 8.8(b). However, for more than 8 processors, the iterative solvers are more scalable, and yield a lower runtime. With larger meshes, (i.e., with 9M nodes for the Luer connector mesh), both iterative solves perform better than the direct solver in all cases (see Fig. 8.8(c)). Moreover, the lower complexity of the DistBlBiCG solver results in the best scalability on the

largest number of processors in the experiments.

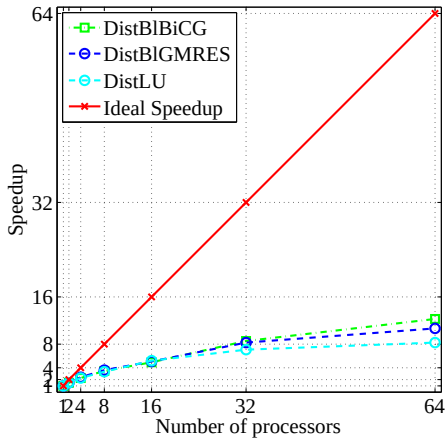


Figure 8.9. The speedup for the three parallel sparse linear solvers for the Menger sponge mesh.

The speedups of the three linear solvers when applied to the Menger sponge mesh are shown in Fig. 8.9. (Since the speedup results look similar for the other meshes, we only include the result for one mesh.) As can be seen in the figure, the speedup of these linear solvers is far from ideal. In particular, the linear solution step does not scale as well as the weight generation step. However, since the algorithm spends the majority of the runtime in the weight generation step, this does not affect the performance of the algorithm very much.

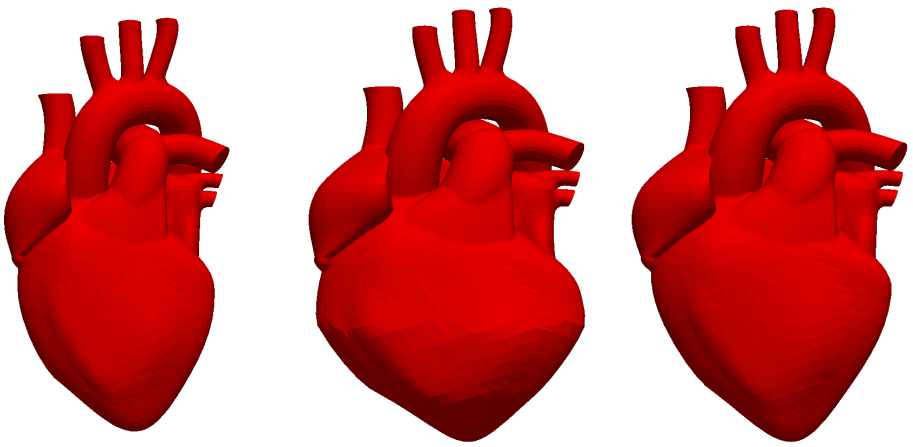
The weak scaling results of parallel LBWARP is shown in Tab. 8.5. The results are obtained from the parallel LBWARP algorithm with the DistBlBiCG solver on various sizes of Luer connector meshes. In the table, the average numbers of nodes and elements per processor, timing and weak scaling factor, for the parallel LBWARP algorithm on up to 64 processors, are given. The ideal successive scaling factor should all be one. However, in our case, there are some variables that affect the results. For instance, the average numbers of nodes and elements are not the same when the number of processors increases. Since the mesh is unstructured, it is hard to generate the mesh that contains the exact numbers of nodes and elements as we want. Such weak scaling results are typical for parallel unstructured mesh computations.

8.7 Multiple Mesh Deformations in a Heartbeat Simulation

One of the main advantages of parallel LBWARP is the re-usability of the weight matrix for additional deformations. Although the overall runtime for parallel LBWARP is dominated by the weight generation step, the weight matrix is generated only once and can be used for a series of deformations. This greatly reduces the computational complexity of the algorithm. Moreover, we can also apply linear solvers that support multiple RHS problems to further reduce the overall runtime. This can be done by computing the RHS vectors (from the boundary deformation step) for all deformations and solving the relevant linear systems simultaneously. Note this can only be done, however, in cases where all of the boundary deformations are known at once. In this section, we demonstrate the performance of parallel LBWARP on multiple mesh deformations with weight matrix re-utilization in a heartbeat simulation.

Heartbeat simulations have been developed in [4, 60, 62]. Such simulations (and their corresponding visualizations) may aid clinicians in medical diagonsis/treatment and may also be used for education. In addition, simulations may aid in obtaining a deeper understanding of a particular biological phenomenon of the heart and its ventricular systems. For example, beating heart meshes can be used to simulate the bioelectricity, biomechanics, and calcium dynamics of the human heart. For this application, we focus on applying multiple deformations to the initial heart mesh which are representative of actual heart motion. Note our heart motion simulation is symbolic and does not correspond to motion obtained from experimental data.

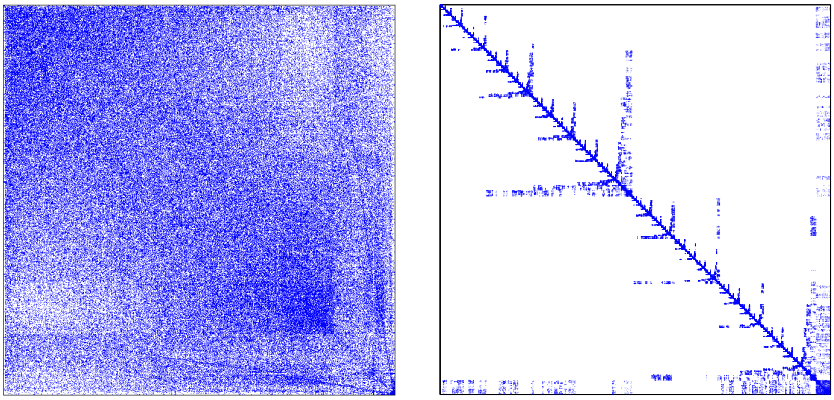
The initial heart domain was obtained from a model in GrabCAD [53], a community database of CAD models. The initial volume heart mesh has approximately 5M nodes and 30M tetrahedral elements and was generated using TetGen. The deformed boundaries are deformations of the surface meshes based on the initial volume heart mesh. After warping the initial mesh to the first deformed boundary, we consequently perform a series of deformations of the original mesh to other target boundaries. We demonstrate the use of multiple deformations with parallel LBWARP. We experiment by computing the deformations from the initial mesh (Fig. 8.10(a)) to five different deformations. Figs. 8.10(b) and (c) shows the sample



]

(b) (c)

Figure 8.10. Simulation of the heartbeat cycle. The initial motion of the heart is shown in (a), whereas (b) and (c) show sample deformations of the heart at two different timesteps within the cycle.



(a) Natural ordering (b) ND reordering

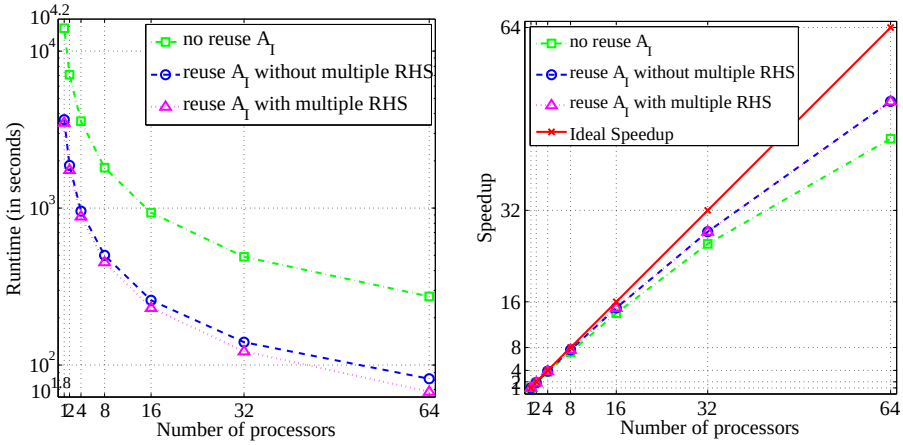
Figure 8.11. Spy plots of A_I for the initial heart mesh with (a) natural ordering and (b) ND reordering.

motions of these five deformations. Spy plots of A_I for the initial heart mesh with (a) natural ordering and (b) nested dissection reordering are shown in Fig. 8.11. The mesh quality is shown in Table 8.6 as mneasured by the MR mesh quality metric. The average MR remains fairly constant throughout the mesh deformation process; only the maximum MR increases throughout the deformation process. This was also observed for the cardiology application in [113]. Note the noticeable decrease in mesh quality is from large deformations of the initial mesh to the target

geometric domains. Smaller deformation steps could be taken (similar to those by the methods in [97,114]) if less change in the mesh quality is needed per deformation step. In our case, we are interested in how well the algorithm performs with large deformations.

Table 8.6. The mean ratio (MR) mesh quality of the heart meshes.

# Deformation	Mesh Quality (MR)		
	Max.	Avg.	Min.
0	1.0000	0.6484	0.1745
1	1.0000	0.2746	0.0248
2	1.0000	0.2643	0.0220
3	1.0000	0.3051	0.0207
4	1.0000	0.2786	0.0202

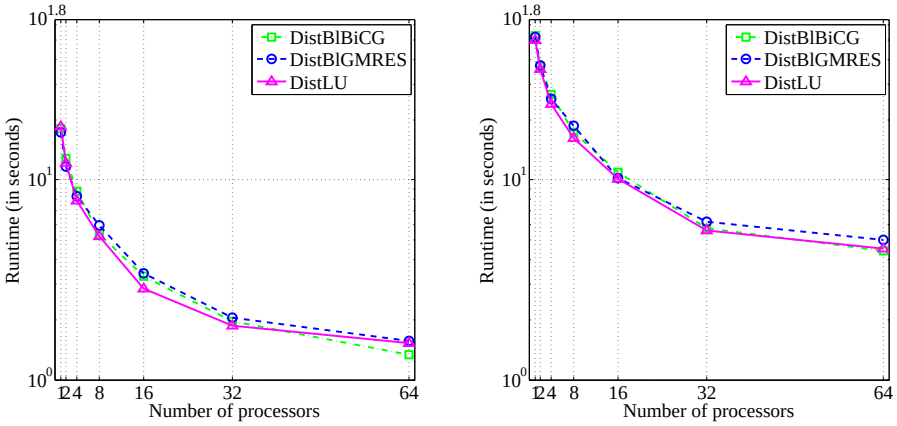


(a) Runtime (b) Speedup

Figure 8.12. (a) The total runtime and (b) speedup for four deformations of the heart mesh using DistLU as a solver.

Fig. 8.12 shows the total runtime and speedup of the parallel LBWARP algorithm for four deformations of the heart mesh using the DistLU algorithm as the linear solver. As shown in Fig. 8.12(a), reusing A_I can significantly reduce the total runtime of the algorithm. For k deformations, when A_I is reused, the algorithm is close is close to k-times faster than without reuse for sufficiently large k. The algorithm with and without multiple RHS support for the linear solution step does not show much difference in runtime. (It is around 5% faster for four deformations on 64 processors. This is because the linear solution step takes less than three

percent of the total time.) As shown in Fig. 8.12(b), the speedup of the algorithm improves when the algorithm reuses A_I, whereas the use or non-use of multiple RHS support does not hardly affect the speedup of the algorithm. Although the advantage of employing multiple RHS support is much less than the advantage of reusing A_I, the combination of both approaches is rather advantageous when DistLU is used on large problems.



(a) One deformation (b) Four deformantions

Figure 8.13. The runtime for the linear solver step for (a) one deformation and (b) four deformations.

We also compare the performance of the three parallel sparse linear solvers, i.e., DistBlBiCG, DistBlGMRES and DistLU, on the linear solution step of parallel LBWARP on the heart meshes as shown in Fig. 8.13. Fig 8.13(a) gives a runtime comparison between the three algorithms for one deformation (i.e., by solving the linear systems with three RHS). The DistLU algorithm gives the best performance over both the DistBlBiCG and DistBlGMRES algorithms for lower numbers of processors, (i.e., up to 32 processors). However, the complexity of the algorithm does not scale well, and, thus, the DistBlBiCG and DistBlGMRES algorithms give better performance for larger numbers of processors, (i.e., more than 32 processors). Since DistBlBiCG has the lowest complexity and also scales well when increasing the numbers of processors, it shows the best performance on 64 processors. With 64 processors, the performance of DistBlBiBG is approximately 17% and 13% faster than that of DistBlGMRES and DistLU, respectively.

Despite the higher complexity, the direct solver has the advantage of reusing the L and U factors and reduces further the overall runtime compared with the parallel

block iterative solvers when solving multiple RHS problems on small numbers of processors. Fig 8.13(b) shows the runtimes of the three parallel sparse linear solvers for four deformations (or by solving the linear system with 12 RHS vectors). The advantage of reusing the L and U factors for multiple linear solves on up to 32 processors can be seen in this figure. Once the L and U factors of the matrix have been computed, they can be used to (triangular) solve with additional RHS vectors in much less time. However, due to the high complexity of the sparse LU algortihm, it has the worst scalability. For large numbers of processors, we can see that the DistLU algorithm shows worse performance than the DistBlBiCG algorithm which has lower complexity and better strong scalability. With 64 processors, the performance of DistBlBiBG is approximately 13% and 2% faster than those of DistBlGMRES and DistLU, respectively.

8.8 Conclusions and Future Research

We have proposed a parallel formulation of the LBWARP algorithm in [113] for warping tetrahedral meshes on distributed memory machines. The algorithm generates the p distributed neighbor lists from the input mesh in which all interior nodes are numbered first and the boundary nodes are numbered last and sends each distributed neighbor list to a processor (assuming there are at least p processors available). Once the processors receive their neighbor lists, they perform the local weight generation for nodes in their neighbor list in parallel and without any communication. After that, the mesh boundary is deformed in parallel. Parallel LBWARP distributes the entries of the deformed boundary to the corresponding processors based on the rows of the weight matrix that the processors have generated. Finally, the linear system, which is based on the weight matrix and the boundary deformation, is solved for the final coordinates of the interior nodes in the deformed mesh using one of three parallel sparse linear solvers, i.e., the distributed block BiCG, distributed block GMRES, and distributed LU algorithms. These solvers support multiple right-hand side vectors which reduces the overall runtime of the parallel LBWARP algorithm since it otherwise requires solution of a sparse linear system with three right-hand side vectors for each deformation of a tetrahedral mesh.

Our experimental results show good strong scalability and speedup on several

3D tetrahedral meshes as a result of efficiently parallelizing the most time consuming step in the algorithm, i.e., the weight generation step. This step takes approximately 80-90% of the algorithm's overall runtime. However, we implement it in an embarrassingly parallel manner in order to avoid all communication during this step. Weak scaling results typical of those for unstructured meshes are also demonstrated. In regards to the performance of the linear solvers, the DistBlBiCG and DistBlGMRES algorithms generally perform well on a large number of processors, (i.e., $p > 32$). On the other hand, the DistLU algorithm performs better on a small number of processors, (i.e., $p \leq 32$) and large numbers of deformations due to the reuse of the L and U factors. Our experiments only use up to 64 processors due to the limited computing system. We expect the two iterative algorithms to outperform the DistLU algorithm when using more than 64 processors or when warping larger meshes (e.g., meshes with more than ten million nodes), as they have lower algorithm complexity and memory requirements.

We applied parallel LBWARP to a heartbeat simulation and demonstrated its performance on a sequence of mesh deformations. For multiple mesh deformations, once the weight matrix has been computed, the parallel LBWARP algorithm can reuse this matrix to determine the interior nodes for the other mesh deformations. That is, only the boundary deformation and linear solution steps are needed which further reduces the algorithm complexity and runtime. With the use of parallel sparse linear solvers that support multiple right-hand side vectors, the overall runtime can be reduced even further.

Possibilities for future research include extension of parallel LBWARP to other mesh element types such as hexahedral elements on 3D domains. Another possible avenue for research is the implementation of a parallel hybrid OpenMP/MPI LBWARP algorithm which can utilize both intra- and inter-node parallelism, as shared memory architectures are becoming increasingly more common. In regards to the parallel sparse linear solvers, our current implementations of the DistBlBiCG and DistBlGMRES algorithms apply row-wise partitioning and distribution of the weight matrix. It is possible to further improve their performance by applying a block matrix partition and distribution. Determination of ways to reuse a portion of the computations performed by these parallel iterative methods when multiple deformations are applied is also of interest.

Chapter 9
Conclusions and Future Work

This dissertation introduces scalable graph and mesh computations for distributed-memory systems. We study some state-of-the-art graph algorithms and data structures, and some domain decomposition and mesh warping techniques. New algorithms and implementations are developed based on these analyses and evaluations.

In the first part of the dissertation, we propose scalable graph computations that efficiently utilize distributed-memory machines. We evaluate, analyze, and compare the performance of various single-source shortest path algorithms that are implemented for distributed-memory machines. We then improve their performance by applying a 2D graph layout as an underlying distributed graph data structure. The algorithms with the 2D graph layout reduce overall communication overhead and improve load balancing, especially for real-world graphs that have a power law degree distribution. Some implementation optimizations are also introduced, such as unique adjacency that act as a local cache to avoid unnecessary communication and data reduction. This handles most of the duplicate data to lower the memory requirement and reduces unnecessary communication and computation. We further extend the study of the 2D graph layout and optimizations on other well-known graph algorithms, such as breadth-first search, approximate diameter, connected components, PageRank, and modified Bellman-Ford, and evaluate their performance on numerous types of graphs including both large-scale real-world and synthetic Graph500 graphs. Finally, we analyze various approximate graph coloring algorithms implemented on the 2D graph layout. Most of the algorithms are hard to parallelize because of data dependencies in the algorithms. Some algorithms focus on the accuracy over the performance, and vice versa. We show that the 2D

graph layout can improve the performance of the algorithms while maintaining the accuracy.

In the second part of the dissertation, we focus on some parallel mesh computations and introduce a domain decomposition algorithm for 2D parallel mesh generation and a parallel mesh warping algorithm. Our domain decomposition algorithm is based on the MeTiS partitioner and is combined with heuristics to improve bad subdomain angles. The algorithm generates subdomains with good quality comparable to other more complex domain decomposition methods, such as medial axis domain decomposition. Our algorithm has a much lower subdomain generation time due to its low complexity. Next, we propose a parallel mesh warping algorithm, i.e., parallel LBWARP, that is based on a serial log barrier-based mesh warping technique. The formulation of the parallel algorithm utilizes a modified distributed adjacency list with ghost vertices as an underlying data structure which avoids most of the communication in the algorithm. We also provide three parallel sparse linear solvers, i.e., DistBlockBiCG, DistBlockGMRES, and DistLU, that support multiple right-hand side vectors. These linear solvers are motivated by Trilinos solvers. The use of these solvers further reduces the overall execution time of the parallel LBWARP algorithm since the algorithm requires a solution of the sparse linear systems with three right-hand sides per deformation (for 3D meshes). The algorithm provides good performance and strong scalability due to the low communication overhead.

Although this dissertation highlights many efficient algorithms, data structures, and techniques for graph and mesh computations, there are still many possibilities for improvements to be explored. First, all of our parallel implementations are currently based on the bulk synchronous parallel model which explicitly separates communication and computation from one another. While it is simple and provides good performance, it has high synchronization overhead. One approach to further improve the algorithms' performance is to overlap communication and computation which will result in less idle time. Second, the current scope of this dissertation is only limited to distributed-memory systems. These computing environments require inter-communication via explicit messages resulting in high communication latency. In reality, as for current architectures, single processors are usually equipped with multiple cores that work on the same memory space. Intra-communications can be done using read/write from/to the shared memory which yield much lower

latency during communication. We plan to extend our implementations to utilize this type of architecture and create more scalable efficient hybrid implementations (i.e., utilizing both MPI and OpenMP). Finally, we plan to make an open source framework combining all of our graph implementations and an a mesh toolkit for parallel mesh algorithms so that it can help research communities in parallel graph and mesh computations.

Bibliography

[1] A. Abou-rjeili and G. Karypis, *Multilevel algorithms for partitioning power-law graphs*, in Proc. 20^{th} International Parallel and Distributed Processing Symposium, 2006, pp. 16–575.

[2] G. J. Alan, *Nested dissection of a regular finite element mesh*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 345–363.

[3] J. Allwright, R. Bordawekar, P. Coddington, K. Dincer, and C. Martin, *A comparison of parallel graph coloring algorithms*, Tech. Report SCCS-666, Northeast Parallel Architectures Center at Syracuse University, 1995.

[4] A. Amano, K. Kanda, T. Shibayama, Y. Kamei, and T. Matsuda, *Model generation interface for simulation of left ventricular motion*, Electronics and Communications in Japan (Part II: Electronics), 90 (2007), pp. 87–98.

[5] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM Journal on Matrix Analysis and Applications, 17 (1996), pp. 886–905.

[6] C. D. Antonopoulos, X. Ding, A. N. Chernikov, F. Blagojevic, D. S. Nikolopoulos, and N. P. Chrisochoides, *Multigrain parallel Delaunay mesh generation*, in Proc. 19^{th} Annual International Conference on Supercomputing, ACM Press, 2005, pp. 367–376.

[7] *Amazon Web Services: Amazon Elastic Compute Cloud*. `http://aws.amazon.com/ec2/`, last accessed July 2015.

[8] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods*, vol. 43, SIAM, 1994.

[9] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist, *Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems*, Scientific Programming, 20 (2012), pp. 241–255.

[10] S. Beamer, K. Asanović, and D. Patterson, *Direction-optimizing breadth-first search*, Scientific Programming, 21 (2013), pp. 137–148.

[11] R. Bellman, *On a routing problem*, Quarterly of Applied Mathematics, 16 (1958), pp. 87–90.

[12] D. Benítez, E. Rodríguez, J. M. Escobar, and R. Montenegro, *Performance evaluation of a parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes*, in Proc. 22^{nd} International Meshing Roundtable, Springer, 2014, pp. 579–598.

[13] D. BenÃdtez, E. RodrÃdguez, J. Escobar, and R. Montenegro, *Performance evaluation of a parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes*, in Proc. 22^{nd} International Meshing Roundtable, 2014, pp. 579–598.

[14] D. P. Bertsekas, *Projected Newton methods for optimization problems with simple constraints*, SIAM Journal on Control and Optimization, 20 (1982), pp. 221–246.

[15] P. Boldi and S. Vigna, *The WebGraph framework I: Compression techniques*, in Proc. 13^{th} International World Wide Web Conference, Manhattan, USA, 2004, ACM Press, pp. 595–601.

[16] M. Botsch, D. Bommes, and L. Kobbelt, *Efficient linear system solvers for mesh processing*, in Proc. 11^{th} IMA International Conference on the Mathematics of Surfaces, 2005, pp. 62–83.

[17] D. Bozdağ, A. H. Gebremedhin, F. Manne, E. G. Boman, and U. V. Catalyurek, *A framework for scalable greedy coloring on distributed-memory parallel computers*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 515–535.

[18] S. Brin and L. Page, *The anatomy of a large-scale hypertextual web search engine*, Computer networks and ISDN systems, 30 (1998), pp. 107–117.

[19] A. Buluç and K. Madduri, *Parallel breadth-first search on distributed memory systems*, in Proc. 2011 International Conference on High Performance Computing, Networking, Storage and Analysis, 2011.

[20] C. Burstedde, L. C. Wilcox, and O. Ghattas, *p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM Journal on Scientific Computing, 33 (2011), pp. 1103–1133.

[21] J. G. Castanos and J. E. Savage, *Pared: A framework for the adaptive solution of PDEs*, in Proc. 8^{th} IEEE Symposium on High Performance Distributed Computing, 1999, pp. 133–140.

[22] U. V. Catalyurek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen, *Distributed-memory parallel algorithms for matching and coloring*, in 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IEEE, 2011, pp. 1971–1980.

[23] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, *Graph coloring algorithms for multi-core and massively multithreaded architectures*, Parallel Computing, 38 (2012), pp. 576–594.

[24] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal, *Scalable single source shortest path algorithms for massively parallel systems*, in Proc. 28^{th} International Parallel and Distributed Processing Symposium, 2014, pp. 889–901.

[25] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A recursive model for graph mining*, in Proc. 2004 SIAM International Conference on Data Mining, 2004.

[26] F. Checconi and F. Petrini, *Traversing trillions of edges in real-time: Graph exploration on large-scale parallel machines*, in Proc. 28^{th} International Parallel and Distributed Processing Symposium, 2014, pp. 425–434.

[27] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan, *Computation and communication efficient graph processing with distributed immutable view*, in Proc. 23^{rd} International Symposium on High-performance Parallel and Distributed Computing, ACM, 2014, pp. 215–226.

[28] A. N. Chernikov and N. P. Chrisochoides, *Parallel guaranteed quality Delaunay uniform mesh refinement*, SIAM Journal on Scientific Computing, 28 (2006), pp. 1907–1926.

[29] N. Chrisochoides, A. Chernikov, A. Fedorov, A. Kot, L. Linardakis, and P. Foteinos, *Towards exascale parallel Delaunay mesh generation*, in Proc. 18^{th} International Meshing Roundtable, 2009, pp. 319–336.

[30] N. P. Chrisochoides, *A survey of parallel mesh generation methods*, Tech. Report BrownSC-2005-09, Brown University, 2005. Also appears as a chapter, in: Are Magnus Bruaset, Aslak Tveito (Eds.), Numerical Solution of Partial Differential Equations on Parallel Computers, Springer, 2006.

[31] N. P. Chrisochoides, *A survey of parallel mesh generation methods*, Tech. Report SC-2005-09, Brown University, 2005.

[32] J. Crobak, J. Berry, K. Madduri, and D. Bader, *Advanced shortest paths algorithms on a massively-multithreaded architecture*, in Proc. 1^{st} Workshop on Multithreaded Architectures and Applications, 2007.

[33] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, et al., *Scalable earthquake simulation on petascale supercomputers*, in Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, 2010, pp. 1–20.

[34] *CyberSTAR: A scalable terascale advanced resource for discovery through computing*. `http://www.ics.psu.edu/infrast/index.html`, last accessed September 2016.

[35] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens, *Work-efficient parallel GPU methods for single-source shortest paths*, in Proc. 28^{th} International Parallel and Distributed Processing Symposium, vol. 28, 2014.

[36] M. Dayarathna, C. Houngkaew, and T. Suzumura, *Introducing ScaleGraph: An X10 library for billion scale graph analytics*, in Proc. 2012 ACM SIGPLAN X10 Workshop, ACM, 2012, p. 6.

[37] H. L. De Cougny and M. S. Shephard, *Parallel refinement and corasening of tetrahedral meshes*, International Journal for Numerical Methods in Engineering, 46 (1999), pp. 1101–1125.

[38] R. Dial, *Algorithm 360: Shortest path forest with topological ordering*, Communications of the ACM, 12 (1969), pp. 632–633.

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

[40] N. Edmonds, T. Hoefler, and A. Lumsdaine, *A space-efficient parallel algorithm for computing betweenness centrality in distributed memory*, in Proc. 2010 International Conference on High Performance Computing, 2010.

[41] N. Edmonds, J. Willcock, and A. Lumsdaine, *Expressing graph algorithms using generalized active messages*, in Proc. 27^{th} International Conference on Supercomputing, 2013.

[42] O. Estruch, O. Lehmkuhl, R. Borrell, C. P. Segarra, and A. Oliva, *A parallel radial basis function interpolation method for unstructured dynamic meshes*, Computers and Fluids, 80 (2013), pp. 44–54.

[43] R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Numerical Analysis, Springer, 1976, pp. 73–89.

[44] L. A. Ford, *Network flow theory*, Tech. Report Report P-923, The Rand Corporation, 1956.

[45] *Galois*. `http://iss.ices.utexas.edu/?p=projects/galois`, last accessed September 2015.

[46] J. Galtier and P. L. George, *Prepartioning as a way to mesh subdomains in parallel*, in Proc. Joint ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Genration, 1997, pp. 107–122.

[47] N. Gandhi and R. Misra, *Performance comparison of parallel graph coloring algorithms on BSP model using hadoop*, in Proc. 2015 International Conference on Computing, Networking and Communications, Feb 2015, pp. 110–116.

[48] A. H. Gebremedhin, F. Manne, and T. Woods, *Speeding up parallel graph coloring*, in Applied Parallel Computing. State of the Art in Scientific Computing, Springer, 2004, pp. 1079–1088.

[49] T. Gerhold and J. Neumann, *The parallel mesh deformation of the DLR TAU-code*, in New results in numerical and experimental fluid mechanics VI, notes on numerical fluid mechanics and multidisciplinary design, vol. 96, Springer, 2008, pp. 162–169.

[50] J. R. Gilbert, G. L. Miller, and S.-H. Teng, *Geometric mesh partitioning: Implementation and experiments*, SIAM Journal on Scientific Computing, 19 (1998), pp. 2091–2110.

[51] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, *PowerGraph: Distributed graph-parallel computation on natural graphs*, in Proc. 10^{th} USENIX Symposium on Operating Systems Design and Implementation, vol. 12, 2012, p. 2.

[52] G. J. Gorman, G. Rokos, J. Southern, and P. H. Kelly, *Thread-parallel anisotropic mesh adaptation*, in New Challenges in Grid Generation and Adaptivity for Scientific Computing, Springer, 2015, pp. 113–137.

[53] *GrabCAD*. `https://grabcad.com`, last accessed January 2016.

[54] *Graph500 benchmark*. `http://www.graph500.org/`, last accessed August 2015.

[55] D. Gregor and A. Lumsdaine, *The Parallel BGL: A generic library for distributed graph computations*, Parallel Object-Oriented Scientific Computing, 2 (2005), pp. 1–18.

[56] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, *Ordering heuristics for parallel graph coloring*, in Proc. 26^{th} ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, 2014, pp. 166–177.

[57] M. T. Heath and P. Raghavan, *A cartesian parallel nested dissection algorithm*, SIAM Journal on Matrix Analysis and Applications, 16 (1995), pp. 235–253.

[58] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, in Proc. 1995 ACM/IEEE Conference on Supercomputing, June 1995.

[59] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards Vol, 49 (1952).

[60] P. J. Hunter, A. J. Pullan, and B. H. Smaill, *Modeling total heart function*, Annual Review of Biomedical Engineering, 5 (2003), pp. 147–177.

[61] D. Hysom and A. Pothen, *A scalable parallel algorithm for incomplete factor preconditioning*, SIAM Journal on Scientific Computing, 22 (2001), pp. 2194–2215.

[62] T. Ijiri, T. Ashihara, N. Umetani, T. Igarashi, R. Haraguchi, H. Yokota, and K. Nakazawa, *A kinematic approach for efficient and robust simulation of the cardiac beating motion*, PloS ONE, 7 (2012), p. e36706.

[63] *Interoperable Technologies for Advanced Petascale Simulations (ITAPS) Center*. `http://www.itaps.org/`, last accessed September 2016.

[64] B. Jacob and G. Guennebaud, *Eigen*. `http://eigen.tuxfamily.org`.

[65] M. T. Jones and P. E. Plassmann, *Scalable iterative solution of sparse linear systems*, Parallel Computing, 20 (1994), pp. 753–773.

[66] M. T. Jones and P. E. Plassmann, *Parallel algorithms for adaptive mesh refinement*, SIAM Journal on Scientific Computing, 18 (1997), pp. 686–708.

[67] G. Karapis and V. Kumar, *MeTiS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 4.0. University of Minnesota*. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[68] D. R. Karger and C. Stein, *A new approach to the minimum cut problem*, Journal of the ACM, 43 (1996), pp. 601–640.

[69] G. Karypis and V. Kumar, *A fast and highly quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing, 20 (1999), pp. 359–392.

[70] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, *Mizan: A system for dynamic load balancing in large-scale graph processing*, in Proc. 8^{th} ACM European Conference on Computer Systems, ACM, 2013, pp. 169–182.

[71] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, *Optimistic parallelism requires abstractions*, in Proc. 28^{th} ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007.

[72] C. Lachat, C. Dobrzynski, and F. Pellegrini, *Parallel mesh adaptation using parallel graph partitioning*, in Proc. 11^{th} World Congress on Computational Mechanics, 2014, pp. 1–12.

[73] X. S. Li and J. W. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Transactions on Mathematical Software, 29 (2003), pp. 110–140.

[74] L. Linardakis and N. Chrisochoides, *Algorithm 870: A static geometric medial axis domain decomposition in 2D Euclidean space*, ACM Transactions on Mathematical Software, 34 (2008), pp. 1–28.

[75] A. Liu and B. Joe, *Relationship between tetrahedron shape measures*, BIT Numerical Mathematics, 34 (1994), pp. 268–287.

[76] R. Löhner, *A* 2^{nd} *generation parallel advancing front grid generator*, in Proc. 21^{st} International Meshing Roundtable, 2013, pp. 457–474.

[77] R. Löhner, J. Camberos, and M. Marsha, *Unstructured scientific compuation on scalable multiprocessors*, in Parallel Unstructured Grid Generation, P. Hehrotra and J. Saltz, eds., MIT Press, 1990, pp. 31–64.

[78] R. Löhner and J. R. Cebral, *Parallel advancing front grid generation*, in Proc. 8^{th} International Meshing Roundtable, 1999, pp. 67–74.

[79] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, *Distributed GraphLab: A framework for machine learning and data mining in the cloud*, Proc. VLDB Endowment, 5 (2012), pp. 716–727.

[80] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, *GraphLab: A new framework for parallel machine learning*, Computing Research Repository, (2010).

[81] H. Lu, M. Halappanavar, D. Chavarria-Miranda, A. Gebremedhin, and A. Kalyanaraman, *Balanced coloring for parallel computing applications*, in Proc. 2015 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2015, pp. 7–16.

[82] Q. Lu, M. S. Shephard, S. Tendulkar, and M. W. Beall, *Parallel mesh adaptation for high-order finite element methods with curved element geometry*, Engineering with Computers, 30 (2014), pp. 271–286.

[83] E. Luke, E. Collins, and E. Blades, *A fast mesh deformation method using explicit interpolation*, Journal of Computational Physics, 231 (2012), pp. 586–601.

[84] 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*, in Proc. 9^{th} Workshop on Algorithm Engineering and Experiments, 2007, pp. 23–35.

[85] 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 Proc. 2010 ACM SIGMOD International Conference on Management of Data, 2010, pp. 135–146.

[86] D. W. Matula and L. L. Beck, *Smallest-last ordering and clustering and graph coloring algorithms*, Journal of the ACM, 30 (1983), pp. 417–427.

[87] D. J. Mavriplis and S. Pirzadeh, *Large-scale parallel unstructured mesh computations for three-dimensional high-lift analysis*, Journal of Aircraft, 36 (1999), pp. 987–998.

[88] U. Meyer and P. Sanders, Δ*-stepping: A parallelizable shortest path algorithm*, Journal of Algorithms, 49 (2003), pp. 114–152.

[89] D. Nave, N. Chrisochoides, and L. P. Chew, *Guaranteed-quality parallel Delaunay refinement for restricted polyhedral domains*, in Computational Geometry: Theory and Applications, vol. 28, 2004, pp. 191–215.

[90] D. Nguyen, A. Lenharth, and K. Pingali, *A lightweight infrastructure for graph analytics*, in Proc. 24^{th} ACM Symposium on Operating Systems Principles, New York, NY, USA, 2013, ACM, pp. 456–471.

[91] D. P. O'Leary, *The block conjugate gradient algorithm and related methods*, Linear Algebra and its Applications, 29 (1980), pp. 293–322.

[92] L. Oliker, R. Biswas, and H. Gabow, *Parallel tetrahedral mesh adaptation with dynamic load balancing*, Parallel Computing Journal, (2000), pp. 1583–1608.

[93] T. Panitanarak and K. Madduri, *Performance analysis of single-source shortest path algorithms on distributed-memory systems*, Tech. Report CSE #14-003, The Pennsylvania State University, 2014.

[94] ——, *Performance analysis of single-source shortest path algorithms on distributed-memory systems*, in Proc. 6^{th} SIAM Workshop on Combinatorial Scientific Computing, 2014, p. 60.

[95] T. Panitanarak and S. M. Shontz, *MDEC: MeTiS-based domain decomposition for parallel 2D mesh generation*, in Proc. 2011 International Conference on Computational Science, 2011, pp. 302–311.

[96] ——, *A parallel log barrier-based mesh warping algorithm for distributed-memory machines*, Engineering with Computers, (2016). Submitted March 18.

[97] J. Park, S. M. Shontz, and C. S. Drapaca, *A combined level set/mesh warping algorithm for tracking brain and cerebrospinal fluid evolution in hydrocephalic patients*, Image-Based Geometric Modeling and Mesh Generation, Lecture Notes in Computational Vision and Biomechanics, 3 (2013), pp. 107–141.

[98] A. Pothen, H. D. Simon, and K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 430–452.

[99] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, *Managing large graphs on multi-cores with graph awareness*, in Proc. 2012 USENIX Conference on Annual Technical Conference, 2012.

[100] *Random123: A library of counter-based random number generators.* `http://www.thesalmons.org/john/random123/`, last accessed January 2017.

[101] M. Rasquin, C. Smith, K. Chitale, E. S. Seol, B. Matthews, J. Martin, O. Sahni, R. Loy, M. S. Shephard, and K. E. Jansen, *Scalable fully implicit finite element flow solver with application to high-fidelity flow control simulations on a realistic wing design*, Computing in Science and Engineering, 16 (2014), pp. 13–21.

[102] M. C. Rivara, C. Carlderon, D. Pizaro, A. Fedorov, and N. Chrisochoides, *Parallel decoupled terminal-edge bisection algorithm for 3D meshes*, Engineering with Computers, (2005).

[103] M. C. Rivara, D. Pizarro, and N. Chrisochoides, *Parallel refinement of tetrahedral edges using terminal-edge bisection algorithm*, in Proc. 13^{th} International Meshing Roundtable, 2004.

[104] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, 2003.

[105] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 856–869.

[106] S. Salihoglu and J. Widom, *GPS: A graph processing system*, in Proc. 25^{th} International Conference on Scientific and Statistical Database Management, SSDBM, 2013, pp. 22:1–22:12.

[107] S. P. Sastry and S. M. Shontz, *A parallel log-barrier method for mesh quality improvement and untangling*, Engineering with Computers, 30 (2014), pp. 503–515.

[108] P. Selwood, M. Berzins, and P. Dew, *3D parallel mesh adaptivity: Data-structures and algorithms*, in Proc. 8^{th} SIAM Conference on Parallel Processing for Scientific Computing, SIAM, 1997.

[109] P. Selwood, N. Verhoeven, J. Nash, M. Berzins, N. Weatherill, P. Dew, and K. Morgan, *Parallel mesh generation and adaptivity: Partitioning and analysis*, in Proc. 1996 Parallel CFD Conference, 1996.

[110] B. Shao, H. Wang, and Y. Li, *Trinity: A distributed graph engine on a memory cloud*, in Proc. 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, 2013, pp. 505–516.

[111] M. Shephard, J. E. Flaherty, C. L. Bottasso, H. L. de Cougny, C. Özturan, and M. L. Simone, *Parallel automated adaptive analysis*, Parallel Computing, 23 (1997), pp. 1327–1347.

[112] J. Shewchuk, *Triangle: Engineering a 2D quality mesh generator and delaunay triangulator*, in Applied Computational Geometry Towards Geometric Engineering, vol. 1148 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1996, pp. 203–222.

[113] S. M. Shontz and S. A. Vavasis, *A mesh warping algorithm based on weighted Laplacian smoothing*, in Proc. 12^{th} International Meshing Roundtable, 2003, pp. 147–158.

[114] ——, *Analysis of and workarounds for element reversal for a finite element-based algorithm for warping triangular and tetrahedral meshes*, BIT, Numerical Mathematics, 50 (2010), pp. 863–884.

[115] J. Shun and G. E. Blelloch, *Ligra: A lightweight graph processing framework for shared memory*, in Proc. 18^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, 2013, pp. 135–146.

[116] H. Si, *TetGen: A quality tetrahedral mesh generator and three-dimensional Delaunay triangulator*. http://tetgen.berlios.de/.

[117] V. Simoncini, *A stabilized QMR version of block BICG*, SIAM Journal on Matrix Analysis and Applications, 18 (1997), pp. 419–434.

[118] V. Simoncini and E. Gallopoulos, *Convergence properties of block GMRES and matrix polynomials*, Linear Algebra and Its Applications, 247 (1996), pp. 97–119.

[119] *SNAP: Stanford Network Analysis Project*. https://snap.stanford.edu/data/, last accessed August 2015.

[120] *The University of Florida Sparse Matrix Collection*. https://www.cise.ufl.edu/research/sparse/matrices/, last accessed August 2015.

[121] *STAMPEDE - Texas Advanced Computing Center*. https://www.tacc.utexas.edu/stampede/, last accessed January 2017.

[122] *StarCluster*. http://star.mit.edu/cluster/, last accessed July 2015.

[123] *Amazon statistics and facts*. http://expandedramblings.com/index.php/amazon-statistics/, last accessed September 2016.

[124] *Facebook newsroom*. http://newsroom.fb.com/, last accessed September 2016.

[125] *Netflix statistics and facts*. http://expandedramblings.com/index.php/netflix_statistics-facts/, last accessed September 2016.

[126] *Large scale simulation of a helicopter engine jet and an axial fan*. http://http://www.teraflop-workbench.org, last accessed March 2016.

[127] M. Stoer and F. Wagner, *A simple min-cut algorithm*, Journal of the ACM, 44 (1997), pp. 585–591.

[128] *Trilinos Project*. http://trilinos.org/, last accessed September 2016.

[129] H. M. Tsai, A. S. Wong, J. Cai, Y. Zhu, and F. Liu, *Unsteady flow calculations with a parallel multiblock moving mesh algorithm*, American Institute of Aeronautics and Astronautics Journal, 39 (2001), pp. 1021–1029.

[130] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, *Asynchronous large-scale graph processing made easy*, in Proc. 2013 Conference on Innovative Data Systems Research, 2013.

[131] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, *Gunrock: A high-performance graph processing library on the GPU*, in Proc. 20^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, 2015, pp. 265–266.

[132] D. J. Welsh and M. B. Powell, *An upper bound for the chromatic number of a graph and its application to timetabling problems*, The Computer Journal, 10 (1967), pp. 85–86.

[133] T. White, *Hadoop: The definitive guide*, O'Reilly Media, Inc., 2012.

[134] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, *Active Pebbles: Parallel programming for data-driven applications*, in Proc. 2011 International Conference on Supercomputing, 2011, pp. 235–244.

[135] R. Williams, *Adaptive parallel meshes with complex geometry*, in Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, 1991.

[136] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, *GraphX: A resilient distributed graph system on spark*, in Proc. 1^{st} International Workshop on Graph Data Management Experiences and Systems, GRADES '13, 2013, pp. 2:1–2:6.

[137] J. Yang and J. Leskovec, *Defining and evaluating network communities based on ground-truth*, Computing Research Repository, abs/1205.6233 (2012).

[138] J. Zhong and B. He, *Medusa: Simplified graph processing on GPUs*, IEEE Transactions on Parallel and Distributed Systems, 25 (2014), pp. 1543–1552.

Vita

Thap Panitanarak

Thap Panitanarak received his B.S. degree in Mathematics from Chiangmai University, Thailand, in 2005. He received his M.S. degree in Computer Science from Western Michigan University in 2009. He continued his studies by entering the Ph.D. program in the Computer Science and Engineering Department at The Pennsylvania State University in September 2009. Both of his M.S. and Ph.D. studies have been supported by The Royal Thai Government Scholarship. His research interests lie in high-performance computing and parallel graph and mesh algorithms.