EFFICIENT DISTRIBUTED GRAPH ALGORITHMS FOR HIGH PERFORMANCE COMPUTING CONTEXTS

Ian Bogle

Submitted in Partial Fullfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

Approved by:
George M. Slota, Chair
Karen Devine
Boleslaw Szymanski
Mohammed Zaki

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[August 2022]
Submitted June 2022

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

There exists a rich and long-standing body of literature on the topic of efficient, parallel graph algorithms. For many problems, an optimal shared memory approach has been described and refined by numerous studies. However, as our scientific computing applications increase in scale, so too do the graphs intrinsic to their representation of the world. The large scale of graph data poses a significant problem for analytics with well-established efficient shared memory approaches: They have not been tested in distributed memory environments. To make matters worse, it is not clear that every shared memory approach would be efficient in distributed memory, let alone easy to implement.

We explore the problem space of distributed memory graph algorithms targeting scientific computing applications running on High Performance Computing (HPC) systems. Through this work, we have seen immense value gained from tailoring graph algorithms specifically to the problem at hand. We were able to make use of mesh data and other application data to see our approach execute at a fraction of a percent of a single simulation step of our target application. We also anecdotally note that incorporating efficient pre-processing into the simulation pipeline can save valuable time that would otherwise be spent exporting meshes, and running the pre-processing outside of the HPC platform.

Additionally, we explore how to efficiently leverage all computing power at our disposal. We implemented a hierarchically parallel graph coloring framework that is able to execute efficiently on HPC systems with GPU resources or with only CPU resources. We use an architecture-aware approach to selecting algorithms we experimentally determined to be more efficient given certain hardware resources. Assembling such a framework is not a trivial feat, as build processes for these large scientific computing libraries can be complex. Our runtime experiments show that overall, our new approach is faster than a similar MPI-only framework, and importantly uses few more colors in general. This capability to adapt to different architectures is vital to performant distributed algorithms, as it allows the users of the algorithm to leverage all hardware at their disposal.

Finally, we explore directly porting shared memory graph algorithms to distributed memory, and find that this approach is not guaranteed to yield efficient implementations. We study two implementations of graph biconnectivity algorithms, a well-known parallel algorithm that is known to be efficient, and a newer graph algorithm that is competitive

in shared memory, and was selected due to its use of simple subroutines. We find that in general it can be difficult to obtain an efficient distributed memory implementation from an algorithm only formulated for shared memory. Additionally, there are specific implementation details that can make certain shared memory algorithms very difficult to implement. A common approach to biconnectivity is the construction of a secondary graph, which is much simpler to achieve in shared memory than it is in distributed memory. We show runtimes for our implementations of several shared memory algorithms including Breadth First Search (BFS), descendant counting, preorder labeling, and constructing spanning forests. Our implementations which follow simpler, but theoretically less optimal approaches tend to outperform the optimal shared memory algorithms we implemented in distributed memory. It is important to note that this does not mean that shared memory approaches cannot inform our distributed memory approaches, just that it can be difficult to intuit what approaches will be the most performant in distributed memory.

CHAPTER 1
INTRODUCTION

1.1 Introduction

This thesis explores graph algorithms by investigating novel graph algorithms, their implementations on mainstream High Performance Computing (HPC) hardware, and their benefits to numerous applications from a variety of fields. A graph in this context is an abstraction that uses a set of vertices and edges to represent different objects and the connections between them. Many fields use graphs to represent complex networks, including sociology, biology, and computational science [1]–[4], and as these networks get larger the need for efficient graph algorithms grows. Efficient graph algorithms that operate on large-scale inputs are difficult to develop. DARPA lists both computation at scale and capturing the dynamics of large-scale distributed networks in its 23 challenge questions [5]. Additionally, graph algorithms tend to be difficult to implement in distributed memory [6].

This thesis focuses on developing novel algorithms for graph connectivity and graph coloring in modern distributed-memory HPC systems. These problems were chosen because they directly target scientific computing applications which require them. One of the main themes of this thesis is exploring how to best parallelize distributed graph algorithms to fully leverage modern HPC hardware.

Modern HPC hardware is perpetually being improved and iterated upon. At the start of the work presented in this thesis, the HPC system housed at RPI was AMOS, a Blue Gene/Q with 5K nodes with 80K cores and 80TB of RAM. AMOS was a homogeneous system, using a large number of CPU-based nodes with no accelerators. Currently, the HPC system housed at RPI is AiMOS, which has 268 nodes, each of which has 2 IBM Power9 processors at 3.15 GHz, 4x NVIDIA Tesla V100 GPUs with 16 GB of memory connected via NVLink, 512 GB of RAM, and 1.6 TB Samsung NVMe Flash memory. This shift to a system with fewer nodes equipped with powerful accelerators is indicative of the current general direction of HPC systems, as currently 8 of the top 10 systems listed on the Top500 are heterogeneous systems [7].

1.2 Relevant Problems

Graph connectivity is a set of problems that generally determine which parts of a graph are connected to each other in a certain manner. The most relevant graph connectivity problem to this thesis is graph biconnectivity, which identifies vertices which will disconnect a graph when removed. Graph biconnectivity is useful for assessing network resiliency [8], fault tolerance in ad hoc networks [9], and detecting mechanisms in meshes for structural dynamics [10]. Efficient serial and shared-memory parallel algorithms exist for this problem [8], [11]–[13], but until this work there were no distributed memory biconnectivity algorithms capable of efficiently processing large-scale networks on HPC systems. Additionally, many of these efficient serial and shared memory parallel algorithms use subroutines that do not translate well into distributed memory.

Graph coloring is a problem in which colors are assigned to vertices based on some criteria. The simplest formulation of the graph coloring problem is distance-1 graph coloring, where each vertex must have a different color from its neighboring vertices. Each graph has what is called a chromatic number, which is the smallest number of colors required for a valid distance-1 coloring on a graph. In general, finding a coloring that uses the fewest possible colors is an NP-Hard problem. However, heuristic approaches generally get close enough to the chromatic number for many practical applications [14]. Practical applications for coloring largely center on finding parallelism in scientific computing applications [4], [15]–[17], memory management in compilers [18], [19], and finding short-circuits in circuit designs [20]. Work has been done on efficient serial [17], [21], shared memory parallel [15], [22]–[24], and distributed memory approaches [25], [26] for many coloring formulations. This work spans a variety of hardware as well, from multithreaded CPUs [15], [23] to individual GPUs [15], [27], as well as distributed memory approaches run with a single thread per node [25], and run with multiple threads per node [26]. Up to this point, there has not been a distributed approach that leverages multiple GPUs, which is a common modern HPC architecture.

1.3 Thesis Organization

The remaining sections of this thesis are organized into 4 chapters. Chapter 2 investigates a distributed memory multi-GPU implementation of distance-1, distance-2, and

partial distance-2 graph coloring. The content of this chapter is composed from the following papers:

- **Distributed Memory Graph Coloring Algorithms for Multiple GPUs** published in *IA3 2020*
- **Parallel Graph Coloring Algorithms for Distributed GPU Environments** published in *Parallel Computing 2022*

Chapter 3 explores a novel algorithm for detecting degenerate features in ice sheet meshes used in land-ice simulations, which also has potential applications to distributed graph biconnectivity. It also explores a novel approach to graph biconnectivity based on the ideas behind the ice sheet algorithm. The content of this chapter is composed from the following papers and presentations:

- **Parallel Graph Algorithms to Remove Degenerate Features from Ice-Sheet Meshes, and Determine Biconnectivity**, a talk given at *SIAM CSE 2019 at the Minisymposium on Theoretical and computation advancements in Ice-Sheet Modeling*
- **A Parallel Graph Algorithm for Detecting Mesh Singularities in Distributed Memory Ice Sheet Simulations**, published in *ICPP 2019*

Chapter 4 explores an efficient generalization of our ice sheet algorithm to solve graph biconnectivity, and presents a novel distributed implementation of generalized LCA traversals. The content of this chapter is composed from the following paper:

- **Distributed Algorithms for the Graph Biconnectivity and Least Common Ancestor Problems**, published in *ParSocial*, a workshop of *IPDPS22*

Chapter 5 explores efficient distributed implementations for graph biconnectivity and several subroutines used in shared memory biconnectivity algorithms. We plan to submit this work for publication in the near future.

CHAPTER 2
DISTRIBUTED GRAPH COLORING ON MULTIPLE GPUS

2.1 Chapter Introduction

In this chapter, we introduce an implementation of a distributed coloring framework capable of leveraging shared-memory parallelism on CPU and GPU-based platforms. This allows our graph coloring implementation to leverage whatever local parallelism is available to the application calling our coloring. We also show that our implementation of this framework is competitive with another distributed implementation of the same framework that uses no on-node parallelism. This implementation also represents the first distributed multi-GPU coloring algorithm, to our knowledge.

2.2 Graph Coloring Introduction

We present new multi-GPU, distributed memory implementations of distance-1, distance-2, and partial distance-2 graph coloring. *Distance-1 graph coloring* assigns *colors* (i.e., labels) to all vertices in a graph such that no two neighboring vertices have the same color. Similarly, *distance-2 coloring* assigns colors such that no vertices within *two hops*, also called a "two-hop neighborhood," have the same color. *Partial distance-2 coloring* is a special case of distance-2 coloring, in which only one set of a bipartite graph's vertices are colored. Usually, these problems are formulated as NP-hard optimization problems, where the number of colors used to fully color a graph is minimized. Serial heuristic algorithms have traditionally been used to solve these problems, one of the most notable being the DSatur algorithm of Brélaz [21]. More recently, parallel algorithms [15], [25] have been proposed; such algorithms usually require multiple *rounds* to correct for improper *speculative* colorings produced in multi-threaded or distributed environments.

There are many useful applications of graph coloring. Most commonly, it is employed to find concurrency in parallel scientific computations [15], [16]; all data sharing a color can

This chapter previously appeared as: I. Bogle, G. M. Slota, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring algorithms for distributed GPU environments," *Parallel Comput.*, vol. 110, May 2022. Art. no. 102896

Portions of this chapter previously appeared as: I. Bogle, E. G. Boman, K. Devine, S. Rajamanickam, and G. M. Slota, "Distributed memory graph coloring algorithms for multiple GPUs," In *2020 IEEE/ACM 10th Workshop on Irregular Appl.: Architectures and Algorithms (IA3)*, Nov. 2020, pp. 54-62.

be updated in parallel without incurring race conditions. Other applications use coloring as a preprocessing step to speed up the computation of Jacobian and Hessian matrices [17] and to identify short circuits in printed circuit designs [20]. Partial distance-2 coloring in particular is used to color sparse Jacobian matrices [28]. Despite the intractability of minimizing the number of colors for non-trivial graphs, such applications benefit from good heuristic algorithms that produce small numbers of colors. For instance, Deveci et al. [15] show that a smaller number of colors used by a coloring-based preconditioner reduces the runtime of a conjugate gradient solver by 33%.

In particular, this work is motivated by the use of graph coloring as a preprocessing step for distributed scientific computations such as automatic differentiation [4]. A common use-case of automatic differentiation is for nonlinear solvers, which can require the calculation of derivatives multiple times during a single solve. For such applications, where graph coloring would also need to be done multiple times, assembling the associated graphs on a single node to run a sequential or shared-memory coloring algorithm may not be feasible [25]. As such, we focus on running our algorithms on the parallel architectures used by the underlying target applications. These architectures are typically highly distributed, with multiple CPUs and/or GPUs per node. Therefore, we specifically consider in this work coloring algorithms that can use the "MPI+X" paradigm, where the Message Passing Interface (MPI) library is used in distributed memory and "X" is multicore CPU or GPU acceleration. Though, we also acknowledge that for other scientific computations where graph coloring is used as a true preprocessing step, running a coloring algorithm offline might be the desired approach, as serial algorithms can often have better output quality.

2.2.1 Contributions

We present and examine two MPI+X implementations of distance-1 coloring as well as one MPI+X implementation of distance-2 coloring and an MPI+X implementation of partial distance-2 coloring. In order to run on a wide variety of architectures, we use the Kokkos performance portability framework [29], [30] for on-node parallelism and Trilinos [31] for distributed MPI-based parallelism. The combination of Kokkos and MPI allows our algorithms to run on multiple multicore CPUs or multiple GPUs in a system. For this paper, we focus on the performance of our algorithms in MPI+GPU environments. For distance-1 coloring of real-world networks, our algorithms achieve up to 2.38x speedup on

128 GPUs compared to a single GPU, and only a 2.23% increase in the number of colors on average. For distance-2 coloring, our algorithm achieves up to 33x speedup and, on average, a 7.5% increase in the number of colors. We also demonstrate good weak scaling behavior up to 128 GPUs for graphs with up to 12.8 billion vertices and 76.7 billion edges.

2.2.2 Extension of Prior Work

This work is an extension of our previous distributed-memory coloring work published at the 2020 IA^3 Workshop [32]. There are several changes and additions included in this extension. We improved the recoloring performance of the local distance-2 algorithm, resulting in a speedup over previous results. We added a new distributed conflict resolution heuristic that improves both color quality and average runtime on our distance-1 implementation while incurring very little computational cost. We additionally implemented an approach to solve to a new problem variant, partial distance-2 coloring. We improved our experimental timing methodology to be more representative of our real-world use cases, and we now show updated performance results for all implementations.

2.3 Background

2.3.1 Coloring Problem

While there exist many definitions of the "graph coloring problem," we specifically consider variants of distance-1 and distance-2 coloring. Consider graph $G = (V, E)$ with vertex set V and edge set E. *Distance-1 coloring* assigns to each vertex $v \in V$ a color $C(v)$ such that $\forall (u, v) \in E, C(u) \neq C(v)$. In *distance-2 coloring*, colors are assigned such that $\forall (u, v), (v, w) \in E, C(u) \neq C(v) \neq C(w)$; i.e., all vertices within two hops of each other have different colors. *Partial Distance-2 coloring* is a special case of distance-2 coloring in which $\forall (u, v), (v, w) \in E, C(u) \neq C(w)$; it is typically applied to bipartite graphs in which only one set of the vertices is given colors (thus, the designation "partial"). When a coloring satisfies one of the above constraints, it is called *proper*. The goal is to find proper colorings of G such that the total number of different colors used is minimized.

2.3.2 Coloring Background

While minimizing the number of colors is NP-hard, serial coloring algorithms using greedy heuristics have been effective for many applications [14]. The serial greedy algorithm in Algorithm 1 colors vertices one at a time. Colors are represented by integers, and the smallest usable color is assigned as a vertex's color. Most serial and parallel coloring algorithms use some variation of greedy coloring, with algorithmic differences usually involving the processing order of vertices or, in parallel, the handling of conflicts and communication.

Algorithm 1 Serial greedy coloring algorithm

```
procedure SerialGreedy(Graph G = (V, E))
    C(∀v ∈ V) ← 0                          ▷ Initialize all colors as null
    for all v ∈ V in some order do
        c ← the smallest color not used by a neighbor of v
        C(v) ← c
```

Conflicts in a coloring are edges that violate the color-assignment criterion; for example, in distance-1 coloring, a conflict is an edge with both endpoints sharing the same color. Colorings that contain conflicts are not proper colorings, and are referred to as *pseudo-colorings*. Pseudo-colorings arise only in parallel coloring algorithms, as conflicts arise only when two vertices are colored concurrently. A coloring's "quality" refers to the number of colors used; higher quality colorings of a graph G use fewer colors, while lower quality colorings of G use more colors.

It has been observed that the order vertices are visited affects the number of colors needed. Popular vertex ordering heuristics for greeding coloring include largest-degree-first, smallest-degree-last, and saturation degree [33]. These orderings are highly sequential and do not allow much parallelism. However, relaxations of those orderings can allow some parallelism [34]. In this work, we consider the above degree-based heuristic for conflict resolution instead, where concurrency issues are not a concern in our implementation.

2.3.3 Parallel Coloring Algorithms

There are two popular approaches to parallel graph coloring. The first concurrently finds independent sets of vertices and concurrently colors all of the vertices in each set. This approach was used by Jones and Plassmann [22]. Osama et al. [35] found independent sets on a single GPU and explored the impact of varying the baseline independent set algorithm.

The second approach, referred to as "speculate and iterate" [14], [23], colors as many vertices as possible in parallel and then iteratively fixes conflicts in the resulting pseudo-coloring until no conflicts remain. Gebremedhin et al. [14], Çatalyürek et al. [23] and Rokos et al. [24] present shared-memory implementations based on the speculate and iterate approach. Deveci et al. [15] present implementations based on the speculate and iterate approach that are scalable on a single GPU. Distributed-memory algorithms such as those in [25], [26] use the speculate and iterate approach. Grosset et al. [27] present a hybrid speculate and iterate approach that splits computations between the CPU and a single GPU, but does not operate on multiple GPUs in a distributed memory context. Sallinen et al. [36] demonstrated how to color very large, dynamic graphs efficiently. Besta et al. [33] developed shared memory coloring algorithms and analyzed their performance. They compared to both Jones-Plassman and speculative methods, but only on multicore CPU.

Bozdağ et al. [25] showed that, in distributed memory, the speculative approach is more scalable than methods based on the independent set approach of Jones and Plassmann. Therefore, we choose a speculative and iterative approach with our algorithms.

2.3.4 Distributed Coloring

In a typical distributed memory setting, an input graph is split into subgraphs that are assigned to separate processes. A process's *local graph* $G_l = \{V_l + V_g, E_l + E_g\}$ is the subgraph assigned to the process. Its vertex set V_l contains *local vertices*, and a process is said to *own* its local vertices. The intersection of all processes' V_l is null, and the union equals V. The local graph also has non-local vertex set V_g, with such non-local vertices commonly referred to as *ghost vertices*; these vertices are copies of vertices owned by other processes. To ensure a proper coloring, each process needs to store color state information for both local vertices and ghost vertices; typically, ghost vertices are treated as read-only. The local graph contains edge set E_l, which are edges between local vertices, and E_g, which are edges containing at least one ghost vertex as an endpoint. Bozdağ et al. [25] also defines two subsets of local vertices: *boundary vertices* and *interior vertices*. Boundary vertices are locally owned vertices that share an edge with at least one ghost; interior vertices are locally owned vertices that do not neighbor ghosts. For processes to communicate colors associated with their local vertices, each vertex has a unique global identifier (GID).

Algorithm 2 shows the general approach presented by Bozdağ et al. They note that in-

Algorithm 2 Zoltan's Distributed Coloring Framework

```
procedure ZOLTAN-DIST-COLOR(
 Local Graph G = (V_l + V_g, E_l + E_g), s, colors)
   U ← V_l                                          ▷ U is the list of vertices to color
   while Any process has vertices to color do
      P ← Partition of U into subsets of size s
      for all set u in P do
         for all vertex v in u do
            colors[v] ← a valid color
         send colors of boundary verts to remote copies
         receive colors from remote processes
      Wait to receive all messages
      R ← 0                                         ▷ R is the set of vertices to recolor
      for all boundary vertex v ∈ U do
         if ∃ (v,w) ∈ E_g, where colors[v] = colors[w] and rand(v) < rand(w) then
            R ← R ∪ {v}
      U ← R
```

terior vertices on each process can be colored independently without causing remote conflicts. They also attempt to reduce the number of distributed conflicts by coloring the boundary in "batches". Distributed coloring conflicts can only occur when neighboring vertices are colored concurrently. Their approach colors groups of boundary vertices and then communicates the updated colors, which allows remote processes to avoid conflicts when coloring subsequent batches of vertices. Additionally, the framework uses asynchronous communication, rather than synchronous collectives.

2.4 Methods

We present four hybrid MPI+GPU algorithms, called Distance-1 (D1), Distance-1 Two Ghost Layer (D1-2GL), Distance-2 (D2), and Partial Distance-2 (PD2). D1 and D1-2GL solve the distance-1 coloring problem, D2 solves distance-2 coloring, and PD2 solves a variation of distance-2 coloring. We leverage Trilinos [31] for distributed MPI-based parallelism and Kokkos [29] for on-node parallelism. KokkosKernels [30] provides baseline implementations of distance-1, distance-2, and partial distance-2 coloring algorithms that we use and modify for our local coloring and recoloring subroutines.

Our four proposed algorithms follow the same basic framework, which builds upon the aformentioned work of Bozdağ et al. [25]. In our approach, we color all *local* vertices first.

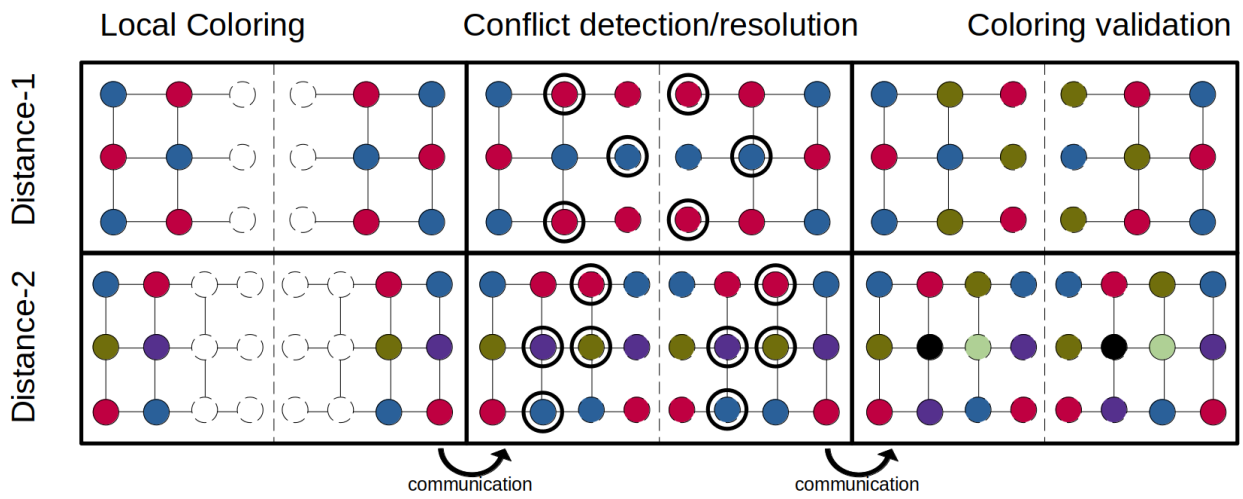

Figure 2.1: Example of our implementation of the distributed-memory coloring framework on a small graph with two processes. First, each process colors their local graph in shared-memory and communicates local colors to ghost copies. Second, we detect conflicts consistently and recolor the vertices local to each process, and communicate local colors to ghost copies. Third, we verify that the coloring is proper. If the coloring contains further conflicts, we go back to conflict detection/resolution.

Then, after communicating boundary vertices' colors, we fix all conflicts. Several rounds of conflict resolution and communication may be needed to resolve all conflicts. We found that this approach was generally faster than the batched boundary coloring of Bozdağ et al., and it allowed us to use existing parallel coloring routines in KokkosKernels without substantial modification.

Algorithm 3 demonstrates the general approach for our three speculative distributed algorithms, while Figure 2.1 shows our approach in action for a small example graph with two processes. First, each process colors all local vertices with a shared-memory algorithm. Then, each process communicates its boundary vertices' colors to processes with corresponding ghosts. Processes detect conflicts in a globally consistent way and remove the colors of conflicted vertices. Finally, processes locally recolor all uncolored vertices, communicate updates, detect conflicts, and repeat until no conflicts are found. The implementation differences between the problem variants involve modifications to the local coloring routines and how we detect and resolve conflicts.

Algorithm 3 Distributed-Memory Speculative Coloring

```
procedure Parallel-Color(
  Local Graph G_l = {V_l + V_g, E_l + E_g},GID)
    colors ← Color(G_l, colors)                    ▷ Initially color local graph
    Communicate colors of boundary vertices
    conflicts ← Detect-Conflicts(G_l, colors, GID)
    Allreduce(conflicts, SUM)                      ▷ Global sum conflicts
    while conflicts > 0 do
        gc ← current colors of all ghosts
        colors = Color(G_l, colors)                ▷ Recolor conflicted
                                                   ▷ vertices
        Replace ghost colors with gc
        Communicate updated boundary colors
        conflicts ← Detect-Conflicts(G_l, colors, GID)
        Allreduce(conflicts, SUM)                  ▷ Global sum conflicts
    return colors
```

2.4.1 Distributed Boundaries

Figure 2.2 shows the sets of boundary vertices for distance-1 and distance-2 formulations of graph coloring. A process' distance-1 boundary vertices are its owned vertices that have neighbors owned by other processes. Its distance-2 boundary vertices are its owned vertices whose neighbors have neighbors owned by other processes. These sets allow us to optimize our distributed conflict detection, as only vertices in the boundary may conflict with a vertex on another process.

2.4.2 Distance-1 Coloring (D1)

Our Distance-1 method begins by independently coloring all owned vertices on each process using the GPU-enabled algorithms by Deveci et al. [15] VB_BIT and EB_BIT in KokkosKernels [30]. VB_BIT uses vertex-based parallelism; each vertex is colored by a single thread. VB_BIT also uses compact bit-based representations of colors to make it performant on GPUs. EB_BIT uses edge-based parallelism; a thread colors the endpoints of a single edge. EB_BIT also uses the compact color representation to reduce memory usage on GPUs. For graphs with skewed degree distribution (e.g., social networks), edge-based parallelism typically yields better workload balance between GPU threads. We observed that for graphs with a sufficiently large maximum degree, edge-based EB_BIT outperformed vertex-based VB_BIT on Tesla V100 GPUs. Therefore, we use a simple heuristic based on maximum

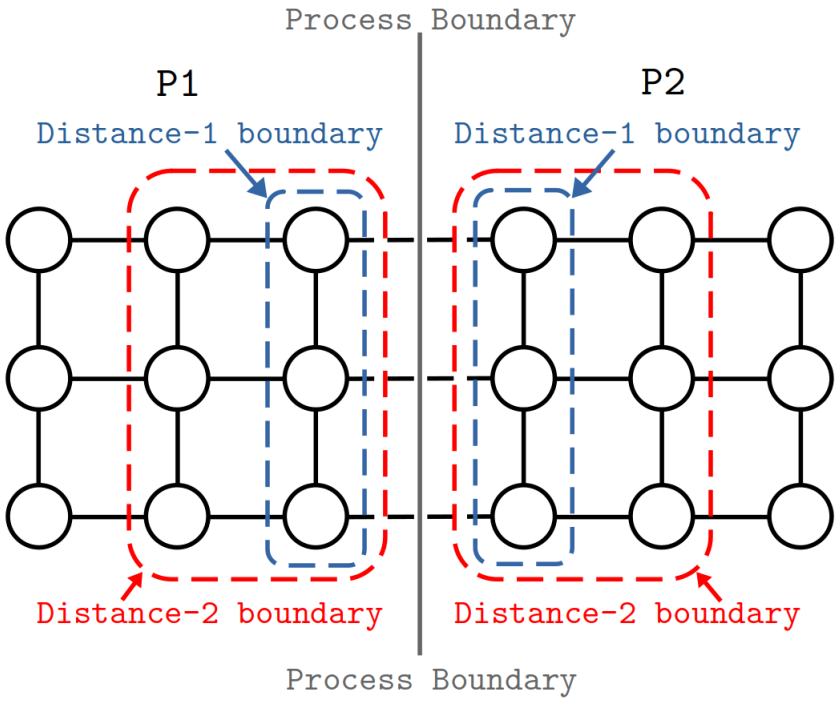

Figure 2.2: Definition of boundary vertex sets for different coloring instances.

degree: we use EB_BIT for graphs with maximum degree greater than 6000; otherwise, we use VB_BIT.

Algorithm 4 Distance-1 conflict detection

```
procedure Detect-Conflicts-D1(
  Local Graph G_l = {V_l + V_g, E_l + E_g}, colors, GID)
  conflicts ← 0
  for all v ∈ V_g do in parallel
    for all ⟨v, u⟩ ∈ (E_g) do
      conflicts ← conflicts + Check-Conflicts(v, u, ...)
      if colors[v] = 0 then
        break
  return conflicts
```

Algorithm 4 shows the conflict detection component of Algorithm 3. This algorithm runs on each process using its local graph G_l. It detects conflicts across processor boundaries and uncolors vertices to resolve the conflicts before recoloring.

After the initial coloring, only boundary vertices can be in conflict with one another[1].

[1]As suggested by Bozdağ et al., we considered reordering local vertices to group all boundary vertices together for ease of processing. This optimization did not show benefit in our implementation, as reordering tended to be slower than coloring of the entire local graph.

Algorithm 5 Algorithm to identify and resolve conflicts

```
procedure CHECK-CONFLICTS(v, u, colors, GID, recolorDegrees)
    if colors[v] = colors[u] then
        if recolorDegrees and degree(v) < degree(u) then
            colors[v] ← 0
        else if recolorDegrees and degree(u) < degree(v) then
            colors[u] ← 0
        else if rand(GID[v]) > rand(GID[u]) then
            colors[v] ← 0
        else if rand(GID[u]) > rand(GID[v]) then
            colors[u] ← 0
        else if GID[v] > GID[u] then
            colors[v] ← 0
        else
            colors[u] ← 0
        return 1
    return 0
```

We perform a full exchange of boundary vertices' colors using collective communication functions implemented in the Zoltan2 package of Trilinos [31]. After the initial all-to-all boundary exchange, we only communicate the colors of boundary vertices that have been recolored. After each process receives its ghosts' colors, it detects conflicts by checking each owned boundary vertex's color against the colors of its neighbor. The conflict detection is done in parallel over owned boundary vertices using Kokkos. The overall time of conflict detection is small enough that any imbalance resulting from our use of vertex-based parallelism is insignificant relative to end-to-end times for the D1 algorithm.

Once we have identified all conflicts, we again use VB_BIT or EB_BIT to recolor the determined set of conflicting vertices. We modified KokkosKernels' coloring implementations to accept a "partial" coloring and the full local graph, including ghosts. (Our initial coloring phase did not need ghost information.) We also modified VB_BIT to accept a list of vertices to be recolored. Such a modification was not feasible for EB_BIT.

Before we detect conflicts and recolor vertices, we save a copy of the ghosts' colors (gc in Algorithm 4). Then we give color zero to all vertices that will be recolored; our coloring functions interpret color zero as uncolored.

To prevent the coloring functions from resolving conflicts without respecting our conflict resolution rules (thus preventing convergence of our parallel coloring), we allow a process

to temporarily recolor some ghosts, even though the process does not have enough color information to correctly recolor them. The ghosts' colors are then restored to their original values in order to keep ghosts' colors consistent with their owning process. Then, we communicate only recolored owned vertices, ensuring that recoloring changes only owned vertices.

2.4.3 Distributed Recoloring Using Vertex Degrees

When a conflict is found, only one vertex involved in the conflict needs to be recolored. Since conflicts happen on edges between two processes' vertices, both processes must agree on which vertex will be recolored.

In previous distributed-memory coloring frameworks [25], [32] this recoloring choice was made by random numbers which were seeded on the global identifier of each vertex. Since the random numbers were generated by global identifiers, they are consistent across remote processes without incurring any communication. No other heuristics for distributed-memory conflict resolution have been explored in the literature, to our knowledge.

We propose a new algorithm for selecting vertices to be recolored in the conflict phase, based on prioritizing by vertex degrees. This idea was inspired by the effectiveness of largest-first and smallest-last ordering in the serial greedy algorithm. To the best of our knowledge, prioritizing the distributed recoloring of lower degree vertices is a novel approach to distributed coloring conflict resolution. In this approach, shown in Algorithm 5, when recolorDegrees is true, our conflict detection prioritizes recoloring the lower degree vertex involved in a distributed conflict. For vertices with equal degree, we adopt the random conflict resolution scheme of Bozdağ et al. in which the conflicted vertex with the higher random number generated from its global identifier (GID) is chosen for recoloring.

The idea behind our recolorDegrees heuristic is that recoloring vertices with large degrees will likely result in giving those vertices a higher color, while recoloring vertices with a smaller degree may be able to use a smaller color for that vertex. Additionally, recoloring vertices with fewer neighbors means that it is less likely that we recolor neighboring vertices concurrently which can reduce the number of conflicts that arise during distributed recoloring. We show that this approach generally decreases runtime for distance-1 coloring, and reduces the number of colors used. In our experiments, recolorDegrees reduces our color usage by 8.9% and runtime by roughly 7% for D1 on average. It achieves a maximum speedup of 45%, and a maximum color reduction of 39% over using D1 without recolorDegrees.

Note that on regular mesh-like graphs where most vertices have the same degree, this new approach behaves similarly to the the random number approach. We observe an average speedup of 1.021x on the more regular graphs in our test set, and a 1.107x average speedup on the more skewed graphs. Additionally, we see an average 4.2% color reduction on the regular graphs, while skewed graphs have an average color reduction of 9.2%. These differences are partially due to our heuristic reducing to the random number heuristic in many cases on more regular graphs, but they are also due to the fact that the baseline random number scheme already performs quite well on regular inputs, leaving less room for improvement than the skewed graphs. Consequently, the best improvement in both runtime and color usage for our heuristic is seen on a skewed graph.

We finally note that we compute the vertex degrees only once. Possible variations include using a "dynamic" degree based on how many neighbors have been colored or the "saturation degree" (how many colors the colored neighbors have been assigned). We reserve an investigation of these variations for future work.

2.4.4 Two Ghost Layers Coloring (D1-2GL)

Algorithm 6 Second Ghost Layer Construction

```
procedure CONSTRUCT-2GL(
  Local Graph G_l = {V_l + V_g, E_l + E_g}, GID, LIDMap)
  ghostGIDs ← GID[v] ∀ v ∈ V_g
  communicate ghostGIDs to remote procs
  recvdGIDs ← received buffer from remote procs
  sendDegrees ← 0
  for all g in range(0, recvdGIDs.size()) do
    sendDegrees[g] ← degree(LIDMap[recvdGIDs[g]])
  communicate sendDegrees to remote procs
  ghostDegrees ← received buffer from remote procs
  recvdAdjs ← adjacencies for entries in recvdGIDs
  communicate recvdAdjs to remote procs
  ghostAdjs ← received buffer from remote procs
```

Our second algorithm for distance-1 coloring, D1-2GL, follows the D1 method, but adds another ghost vertex "layer" to the subgraphs on each process. In D1, a process' subgraph does not include neighbors of ghost vertices unless those neighbors are already owned by the process. In D1-2GL, we include all neighbors of ghost vertices (the two-hop neighborhood of

local vertices) in each process's subgraph, giving us "two ghost layers." To the best of our knowledge, this approach has not been explored before with respect to graph coloring.

This method can reduce the total amount of communication relative to D1 for certain graphs by reducing the total number of recoloring rounds needed. This reduction in rounds is possible by effectively increasing the amount of local recoloring work. In particular, for mesh or otherwise regular graphs, the second ghost layer is primarily made up of interior vertices on other processes. Interior vertices are never recolored, so the colors of the vertices in the second ghost layer are fixed. Each process can then directly resolve more conflicts in a consistent way, thus requiring fewer rounds of recoloring. Fewer recoloring rounds results in fewer collective communications.

However, in D1-2GL, each communication is more expensive than in D1, because a larger boundary from each process is communicated. Also, in irregular graphs, the second ghost layer often does not have mostly interior vertices. The relative proportion of interior vertices in the second layer also gets smaller as the number of processes increases. Additionally, for each process to consistently resolve their local conflicts in a distance-1 coloring, they need to recolor vertices in the first ghost layer, resulting in increased recoloring workloads relative to D1. For the extra ghost layer to pay off, it must reduce the number of rounds of communication enough to make up for the increased cost of each communication and the added recoloring work.

To construct the second ghost layer on each process, processes exchange the adjacency lists of their boundary vertices; this step is needed only once. The general approach for this addition is outlined in Algorithm 6. It would be possible to implement this adjacency exchange without first sending ghost GIDs from each process, but that step allows us to establish the order in which the adjacency information will be received, thus making the construction of the compressed sparse row (CSR) representation much easier. Additionally, there is hidden complexity in this procedure due to MPI's message size limits. For skewed graphs and networks with many edges, the ghost adjacencies can easily outgrow MPI's size limits[2], so it is necessary to split each adjacency send into rounds. That detail is omitted from the algorithm listing for clarity. After the ghosts' connectivity information is added, we use the same coloring approach as in D1.

[2]Per the most recent MPI 4.0 Standard (June 2021), C bindings still use 32-bit integers for counts and offsets of arrays.

Table 2.1: Summary of D1 and D2 input graphs. δ_{avg} refers to average degree and δ_{max} refers to maximum degree. Values listed are after preprocessing to remove multi-edges and self-loops. k = thousand, M = million, B = billion.

Graph	Class	#Vertices	#Edges	δ_{avg}	δ_{max}	Memory (GB)
ldoor	PDE Problem	0.9 M	21 M	45	77	0.32
Audikw_1	PDE Problem	0.9 M	39 M	81	345	0.59
Bump_2911	PDE Problem	2.9 M	63 M	43	194	0.96
Queen_4147	PDE Problem	4.1 M	163 M	78	89	2.5
soc-LiveJournal1	Social Network	4.8 M	43 M	18	20 k	0.67
hollywood-2009	Social Network	1.1 M	57 M	99	12 k	0.86
twitter7	Social Network	42 M	1.4 B	35	2.9 M	21
com-Friendster	Social Network	66 M	1.8 B	55	5.2 k	27
europe_osm	Road Network	51 M	54 M	2.1	13	1.2
indochina-2004	Web Graph	7.4 M	194 M	26	256 k	2.9
MOLIERE_2016	Document Mining	30 M	3.3 B	80	2.1 M	49
rgg_n_2_24_s0	Synthetic Graph	17 M	133 M	15	40	2.1
kron_g500-logn21	Synthetic Graph	2.0 M	182 M	87	214 k	2.7
mycielskian19	Synthetic Graph	393 k	452 M	2.3 k	196 k	6.7
mycielskian20	Synthetic Graph	786 k	1.4 B	3.4 k	393 k	21
hexahedral	Weak Scaling	12.5 M – 12.8 B	75 M – 76.7 B	6	6	1.2 GB – 1.1 TB

We optimize our conflict detection for both distance-1 implementations by looking through only the ghost vertices' adjacencies (E_g), as they neighbor all local boundary vertices. Our local coloring algorithms require our local graphs to have undirected edges to ghost vertices, so this optimization is trivial for both D1 and D1-2GL.

2.4.5 Distance-2 Coloring (D2)

Our distance-2 coloring algorithm, D2, builds upon both D1 and D1-2GL. As with distance-1 coloring, we use algorithms from Deveci et al. in KokkosKernels for local distance-2 coloring. Specifically, we use NB_BIT, which is a "net-based" distance-2 coloring algorithm that uses the approach described by Taş et al. [37]. Instead of checking for distance-2 conflicts only between a single vertex and its two-hop neighborhood, the net-based approach detects distance-2 conflicts among the immediate neighbors of a vertex. Our D2 approach also utilizes a second ghost layer to give each process the full two-hop neighborhood of its boundary vertices. This enables each process to directly check for distance-2 conflicts with local adjacency information. To find a distance-2 conflict for a given vertex, its entire two-hop neighborhood must be checked for potential conflicting colors.

Algorithm 7 shows conflict detection in D2 for each process. We again use vertex-based parallelism while detecting conflicts; each thread examines the entire two-hop neighborhood of a vertex v. The input argument V_b is the set of distance-2 boundary vertices (as in Figure 2.2), which we precompute. As with distance-1 conflict detection, we identify all

Algorithm 7 Distance-2 conflict detection

```
procedure DETECT-D2-CONFLICTS(
  Local Graph G_l = {V_l + V_g, E_l + E_g}, V_b, colors, GID, doPartialColoring)
  conflicts ← 0
  for all v ∈ V_b do in parallel
    for all ⟨v, u⟩ ∈ (E_l + E_g) do
      if not doPartialColoring then
        conflicts ← conflicts + Check-Conflicts(v, u, ...)
        if colors[v] = 0 then
          break
      for all ⟨u, x⟩ ∈ (E_l + E_g) do
                                        ▷ u is one hop and x is two hops from v
        conflicts ← conflicts + Check-Conflicts(v, x, ...)
        if colors[v] = 0 then
          break
      if colors[v] = 0 then
        break
  return conflicts
```

local conflicts and use either the degree of each vertex or a random number generator to ensure that vertices to be recolored are chosen consistently across processes. The iterative recoloring method of D1 then also works for D2 — we recolor all conflicts, replace the old ghost colors, and then communicate local changes.

2.4.6 Partial Distance-2 Coloring (PD2)

We have also implemented an algorithm, PD2, that solves the partial distance-2 coloring problem. Partial distance-2 coloring is similar to distance-2 coloring, but it detects and resolves only two-hop conflicts. Typically, partial distance-2 coloring is used on non-symmetric graphs. A bipartite graph $B(V_s, V_t, E_B)$ is constructed from directed graph $G(V, E)$ with an undirected edge $\langle v_s \in V_s, v_t \in V_t \rangle \in E_B$ for each directed edge $\langle v_s, v_t \rangle \in E$; colors are needed only for vertices in V_s. Partial distance-2 coloring colors only one set of the vertices in the bipartite graph, which is why it is termed a *partial* coloring. In algorithm 7, when doPartialColoring is false, the algorithm detects all distance-2 conflicts. When doPartialColoring is true, it only detects two-hop conflicts for the partial coloring. Currently, our PD2 implementation must color all vertices in the bipartite representation of the graph; applications can ignore colors for vertices in V_t. Removing this limitation is a subject for future work.

2.4.7 Partitioning

We assume that target applications partition and distribute their input graphs in some way before calling these coloring algorithms. In our experiments, we used XtraPuLP v0.3 [38] to partition our graphs. Determining optimal partitions for coloring is not our goal in this work. Rather, we have chosen a partitioning strategy representative of that used in many applications. We partition graphs by balancing the number of edges per-process and minimizing a global edge-cut metric. This approach effectively balances per-process workload and helps minimize global communication requirements.

2.5 Experimental Setup

We performed scaling experiments on the AiMOS supercomputer housed at Rensselaer Polytechnic Institute. The system has 268 nodes, each equipped with two IBM Power 9 processors clocked at 3.15 GHz, 4x NVIDIA Tesla V100 GPUs with 16 GB of memory connected via NVLink, 512 GB of RAM, and 1.6 TB Samsung NVMe Flash memory. Inter-node communication uses a Mellanox Infiniband interconnect configured in a fat-tree topology. We compile with xlC 16.1.1 and use Spectrum MPI with GPU-Direct communication disabled.

We chose to disable GPU-Direct communication in order to keep our build process feasible, as we rely on several dependencies between KokkosKernels and Trilinos. The work of Potluri et al. [39] and Venkatesh et al. [40] also demonstrate there is little expected benefit with GPU-direct communication for large blocking collective communications. Our implementation depends on optimized local coloring subroutines, which unfortunately cannot be used to leverage overlapped communication and computation. Our boundary exchanges for most test inputs are also quite large. Thus, we would expect little to no benefit with GPU-Direct enabled. We, however, also acknowledge that for specific test inputs, GPU-Direct or other optimizations such as CUDA streams, asynchronous memory operations, and non-blocking point-to-point MPI communications may provide speedups from our current approach. These additional optimizations were omitted from this current work for the sake of implementation simplicity, but they might make for promising future work.

The graphs we used to test D1 and D2 are listed in Table 2.1. Most of the graphs are from the SuiteSparse Matrix Collection [41]. The maximum degree δ_{max} can be considered an upper bound for the number of colors used, as any incomplete, connected, and undirected

graph can be colored using δ_{max} colors [42]. We selected many of the same graphs used by Deveci et al. to allow for direct performance comparisons. We include many graphs from Partial Differential Equation (PDE) problems because they are representative of graphs used with Automatic Differentiation [4], which is a target application for graph coloring algorithms. We also include social network graphs and a web crawl to demonstrate scaling of our methods on irregular real-world datasets. We preprocessed all graphs to remove multi-edges and self-loops, and we used subroutines from HPCGraph [43] for efficient I/O.

We compare our implementation against distributed distance-1 and distance-2 coloring in the Zoltan [44] package of Trilinos. Zoltan's implementations are based directly on Bozdağ et al. [25]. Zoltan's distributed algorithm for distance-2 coloring requires only a single ghost layer, and to reduce conflicts, the boundary vertices are colored in small batches.

Our methods D1, D1-2GL, D2, and PD2 were run with four GPUs and four MPI ranks (one per GPU) per node. Zoltan uses only MPI parallelism; it does not use GPU or multicore parallelism. For consistency, we use four MPI ranks per node with Zoltan, and use the same number of nodes for experiments with Zoltan and our methods. We used Zoltan's default coloring parameters; we did not experiment with options for vertex visit ordering, boundary coloring batch size, etc. We used the same partitioning method across all of our comparisons.

This comparison attempts to simulate the expected use-case of our codebase in preprocessing graphs for an application. In this scenario, the number of ranks used by the application is fixed. Adding ranks for the coloring step would require expensive re-partitioning and would not be feasible in the scientific applications that we target. Our experiments determine the direct benefit of an application calling our routine instead of the currently viable alternative of Zoltan.

We omit direct comparison to single-node GPU coloring codes such as CuSPARSE [45], as we use subroutines for on-node coloring from Deveci et al. [15]. Deveci et al. have already performed a comprehensive comparison between their coloring methods and those in CuSPARSE, reporting an average speedup of 50% across a similar set of test instances. As such, we are confident that our on-node GPU coloring is representative of the current state-of-the-art.

2.6 Results

For our experiments, we compare overall performance for D1 and D2 on up to 128 ranks versus Zoltan. Our performance metrics include execution time, parallel scaling, and number of colors used. We do not include the partitioning time for XtraPuLP or I/O; we assume target applications will load, partition, and distribute their graphs. Each of the results reported represents an average of five runs.

2.6.1 Distance-1 Performance

We summarize the performance of our algorithms relative to Zoltan using the performance profiles in Figure 2.3. Performance profiles plot the proportion of problems an algorithm can solve for a given relative cost. The relative cost is obtained by dividing each approach's execution time (or colors used) by the best approach's execution time (or colors used) for a given problem. In these plots, the line that is higher represents the best performing algorithm. The further to the right that an algorithm's profile is, the worse it is relative to the best algorithm. D1-baseline does not consider vertex degree when doing distributed recoloring (e.g., recolorDegrees is false in Algorithm 5). D1-recolor-degree represents our novel approach that recolors distributed conflicts based on vertex degree (e.g., recolorDegrees is true in Algorithm 5).

We ran D1-baseline, D1-recolor-degree and Zoltan with 128 MPI ranks to color the 15 SuiteSparse graphs in Table 2.1. D1-baseline and D1-recolor-degree used MPI plus 128 Tesla V100 GPUs, while Zoltan used MPI on 128 Power9 CPU cores across 32 nodes (four MPI ranks per node). Some skewed graphs (e.g., hollywood-2009) did not run on 128 ranks on Zoltan or D1-baseline; in those cases we use the largest run that completed for both approaches. Figure 2.3a shows that D1-recolor-degree outperforms both Zoltan and D1-baselines in terms of execution time in these experiments. D1-baseline and D1-recolor-degree are very similar in terms of runtime performance, but D1-recolor-degree is the fastest approach for 60% of the graphs, D1-baseline is fastest on 26%, and Zoltan is fastest on 13%. Zoltan is faster than our approaches on two of the smallest graphs, Audikw_1, and ldoor. D1-baseline is faster than D1-recolor-degrees on four graphs which are more varied in application and structure: Bump_2911, com-Friendster, rgg_n_2_24_s0, and twitter7. There are four graphs for which D1-baseline and D1-recolor-degrees runtime performance differ

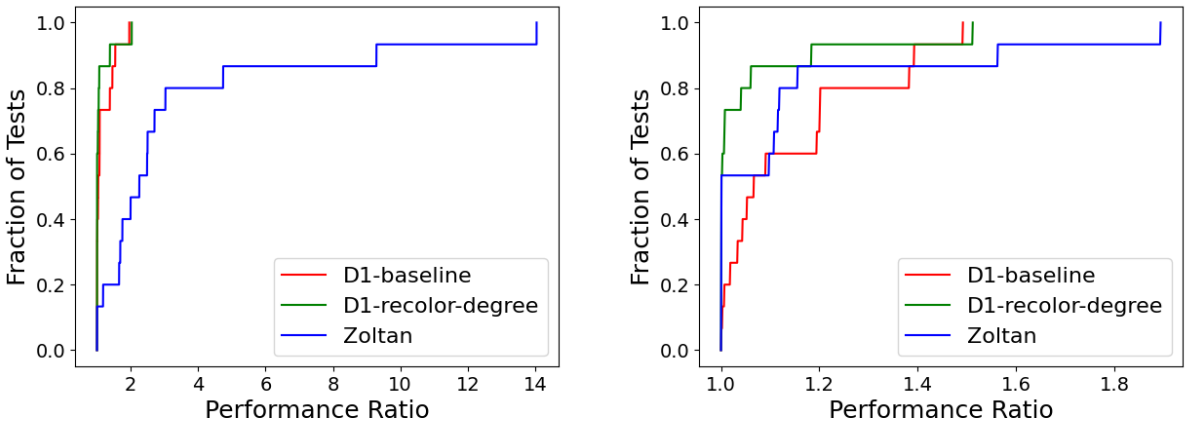

Figure 2.3: Performance profiles comparing D1-baseline and D1-recolor-degree on 128 Tesla V100 GPUs with Zoltan's distance-1 coloring on 128 Power9 cores in terms of (a) execution time and (b) number of colors computed for the graphs listed in Table 2.1.

substantially: Audikw_1 (D1-baseline is 32% faster), ldoor (D1-recolor-degrees is 42% faster), mycielskian19 (D1-recolor-degrees is 45% faster), and mycielskian20 (D1-recolor-degrees is 38% faster). D1-baseline has at most a 10x speedup over Zoltan (with the mycielskian20 graph) and at worst an 1.95x slowdown relative to Zoltan (with ldoor), while D1-recolor-degrees achieves at most a 14x speedup over Zoltan (on mycielskian20), and at worst a 2x slowdown (on Audikw_1).

Figure 2.3b shows that Zoltan outperforms D1-baseline in terms of color usage, but D1-recolor-degree is much more competitive. Both Zoltan and D1-recolor-degree use the fewest colors in 53% of experiments; Zoltan and D1-recolor-degree tie on a single graph. D1-baseline uses the fewest number of colors on a single graph, for which it ties D1-recolor-degree. D1-recolor-degree uses more colors than D1-baseline for two graphs (indochina-2004 and twitter7); in both graphs, D1-baseline uses roughly 1% fewer colors. On average, D1-recolor-degree uses 8.9% fewer colors than D1-baseline, and in the best case, it reduces color usage 39% relative to D1-baseline (mycielskian19). On average, D1-recolor-degree uses 4% fewer colors than Zoltan. In the worst case, D1-recolor-degree uses 51% more colors than Zoltan (twitter7); in the best case, D1-recolor-degree uses 53% fewer colors than Zoltan (mycielskian20).

Because the performance of D1-recolor-degree is generally better than that of D1-

baseline, all further distance-1 coloring results use D1-recolor-degree, and we refer to D1-recolor-degree as D1 going forward.

2.6.2 Distance-1 Strong Scaling

Figure 2.4 shows strong scaling times for Queen_4147 and com-Friendster. These graphs are selected for presentation because they are the largest graphs of their respective problem domains. Data points that are absent were the result of out-of-memory issues or execution times (including I/O and partitioning) that were longer than our single job allocation limits. D1 scales better on the com-Friendster graph than on Queen_4147, as the GPUs can be more fully utilized with the much larger com-Friendster graph. For Queen_4147, D1 on 128 GPUs shows a speedup of around 2.38x over a single GPU. D1 uses 12% fewer colors than Zoltan in the 128 rank run on Queen_4147, as well as running 1.75x faster than Zoltan on that graph. For com-Friendster, D1 is roughly 4.6x faster than Zoltan in the 128 rank run, and only uses 0.6% more colors than Zoltan.

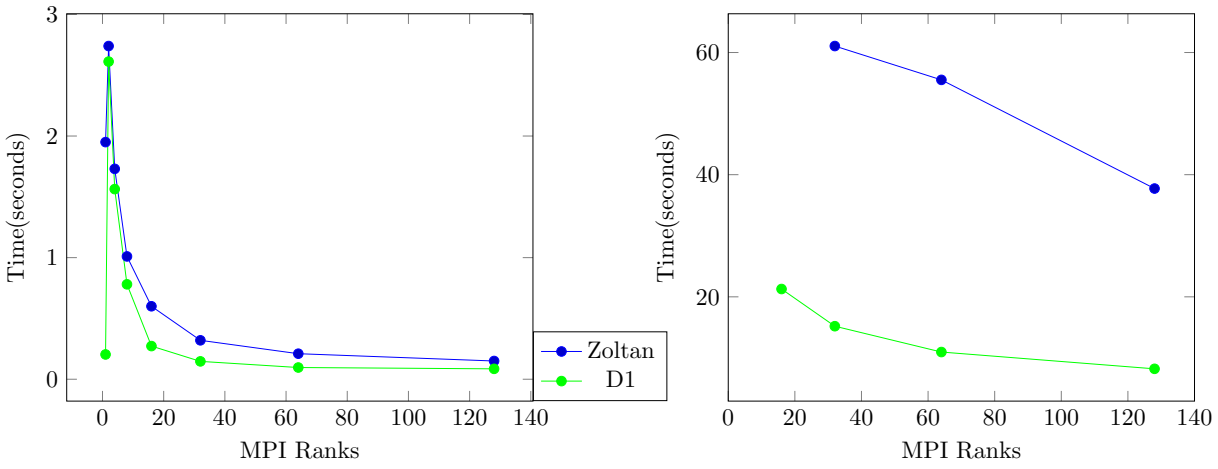

Figure 2.4: Zoltan and D1 strong scaling on (a) Queen_4147 and (b) com-Friendster.

For graph processing in general, it is often difficult to demonstrate good strong scaling relative to single node runs. From the Graph500.org benchmark (June 2021 BFS results) [46], the relative per-node performance difference in the metric of "edges processed per second" between the fastest multi-node results and fastest single node results are over 100x. For coloring on GPUs, graphs that can fit into a single GPU do not provide sufficient work

parallelism for large numbers of GPUs, and multi-GPU execution incurs communication overheads and additional required rounds for speculative coloring. However, on roughly half of the graphs that fit on a single GPU, D1 with 128 GPUs achieves an average speedup of 1.9x over a single GPU. D1 achieves a maximum speedup of 2.43x on the mycielskian20 graph. For the other half of the graphs, D1 does not show a speedup over a single GPU. On small or highly skewed graphs that fit on a single GPU, speedup is limited, due to the communication overheads and work imbalances that result from distribution even with relatively good partitioning. Distributed coloring is valuable even for these small problems, however, as parallel applications using coloring typically have distributed data that would be expensive to gather into one GPU for single-GPU coloring.

On average over all the graphs, D1 uses 38% more colors than the single GPU run, while Zoltan uses 53.6% more colors than the single GPU run. Such large color usage increases are mostly due to the Mycielskian19 and Mycielskian20 graphs. These graphs were generated to have known minimum number of colors (chromatic numbers) of 19 and 20, respectively, and our single GPU runs use 19 and 21 colors to color those graphs. Both D1 and Zoltan have trouble coloring these graphs with so few colors in distributed memory, but our D1 implementation colors these graphs in fewer colors than Zoltan. Without these two outliers, the average color increase from the single GPU run is only 2.23% for D1, and Zoltan decreases color usage by 0.1% on average. Zoltan's higher coloring quality is due to its inherently lower concurrency.

Figure 2.5 shows the total communication and computation time associated with each run. For both graphs, the dominant scaling factor is computation. Specifically, the computational overhead associated with recoloring vertices in distributed memory is the dominant scaling factor. However, strong scaling is good on both graphs, despite the fact that adding more ranks to a problem also increases the number of vertices that need to be recolored. Figure 2.5b shows that D1 scales to more ranks on com-Friendster, primarily because of the graph's larger size.

2.6.3 Distance-1 Weak Scaling

The greatest benefit of our approach is its ability to efficiently process massive-scale graphs. We demonstrate this benefit with a weak-scaling study conducted using uniform 3D hexahedral meshes. The meshes were partitioned with block partitioning along a single

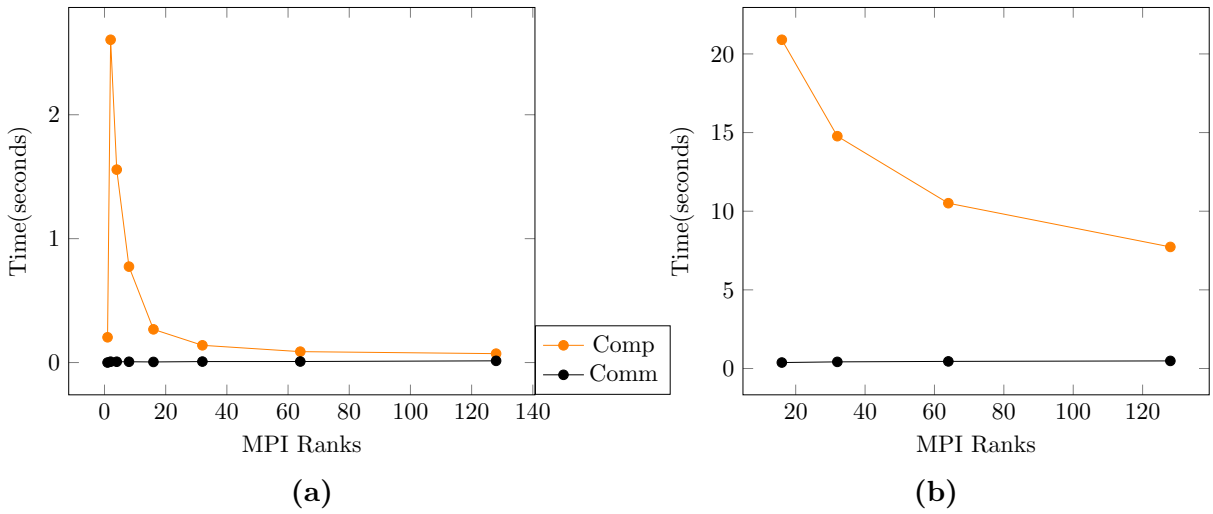

Figure 2.5: D1 communication time (Comm) and computation time (Comp) from 1 to 128 GPUs on (a) Queen_4147 and (b) com-Friendster.

axis, resulting in the mesh being distributed in "slabs." Larger meshes were generated by doubling the number of elements in a single dimension to keep the per-process communication and computational workload constant. Each distinct per-process workload increases the boundary by a factor of two, which correspondingly increases communication and recoloring overhead for distributed runs. We run with up to 100 million vertices per GPU, yielding a graph of 12.8 billion vertices and 76.7 billion edges in our largest tests; **this graph was colored in less than two seconds**.

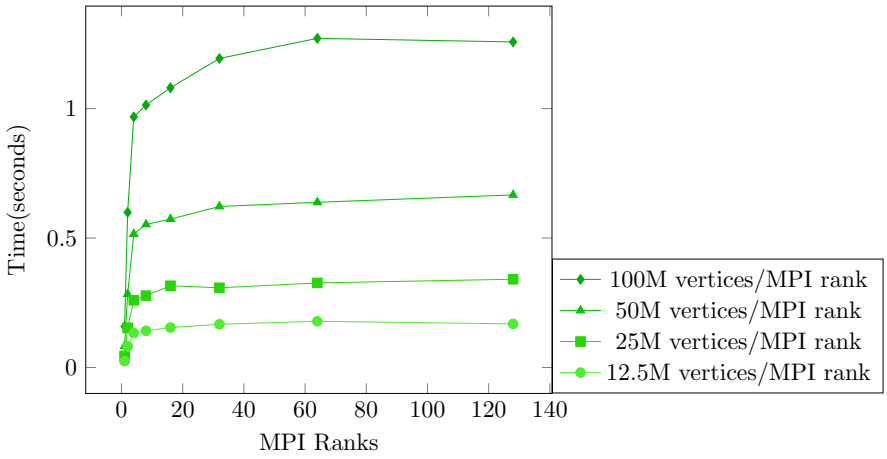

Figure 2.6: Weak scaling of D1 on 3D mesh graphs. Tests use 12.5, 25, 50, and 100 million vertices per GPU.

Figure 2.6 shows that the single rank runs for each workload are similar, indicating

that communication and recoloring overhead are the dominant scaling factors for this study. In increasing the boundary size by a factor of two, we do not necessarily increase the number of distributed conflicts by two, especially in such a regular graph. The smaller workloads all have similar and relatively small recoloring workloads, which is why they show more consistent weak scaling than the 100 Million vertex per rank experiment. That particular experiment does substantially more recoloring than the others, resulting in its increase in runtime as the number of ranks increases. We have found that for extremely regular meshes like these, the number of vertices on process boundaries is the primary driver for recoloring workload magnitudes with D1.

2.6.4 D1-2GL Performance

In general, D1-2GL reduces the number of collective communications used in the distributed distance-1 coloring. Figure 2.7a compares the number of communication rounds for D1-baseline and D1-2GL on the Queen_4147 input for 2 to 128 MPI ranks, averaged over five runs. With 128 ranks on this graph, D1-2GL method reduces the number of rounds by 25% on average, resulting in an overall speedup of 1.18x on 128 ranks. D1-2GL also provides speedups over D1-baseline with other small and regular graphs: 1.17x with Audikw_1 and 1.2x with ldoor. Examing the performance of these algorithms on Queen_4147 in more detail, we first observe that communication costs are around 8% of the total runtime for both D1-baseline and D1-2GL. However, Figure 2.7c shows that in total D1-2GL communicates up to twice as much data as D1 for Queen_4147 due to the additional ghost layer. Despite this, Figure 2.7b shows that D1-2GL recolors fewer vertices for most MPI rank counts. But as the rank count increases and more vertices comprise the boundary layers, the computation costs for D1-2GL also increases.

Despite the observed additional overhead, on Queen_4147 D1-2GL is able to see a speedup over D1-baseline due to the reduction in rounds, which reduces serialization of recoloring work. Unfortunately, for highly structured graphs such as those we used for our weak scaling experiments, D1-baseline does not use many rounds to begin with, so D1-2GL sees no speedup. For dense and skewed inputs, D1-2GL has both increased communication costs and increased recoloring workloads at scale, so D1-baseline outperforms D1-2GL on these graphs on our test system. Because of the above, we generally see the greatest benefit of D1-2GL on small mesh-like graphs. However, in distributed systems with much higher

latency costs, we would expect D1-2GL would be considerably more competitive on average.

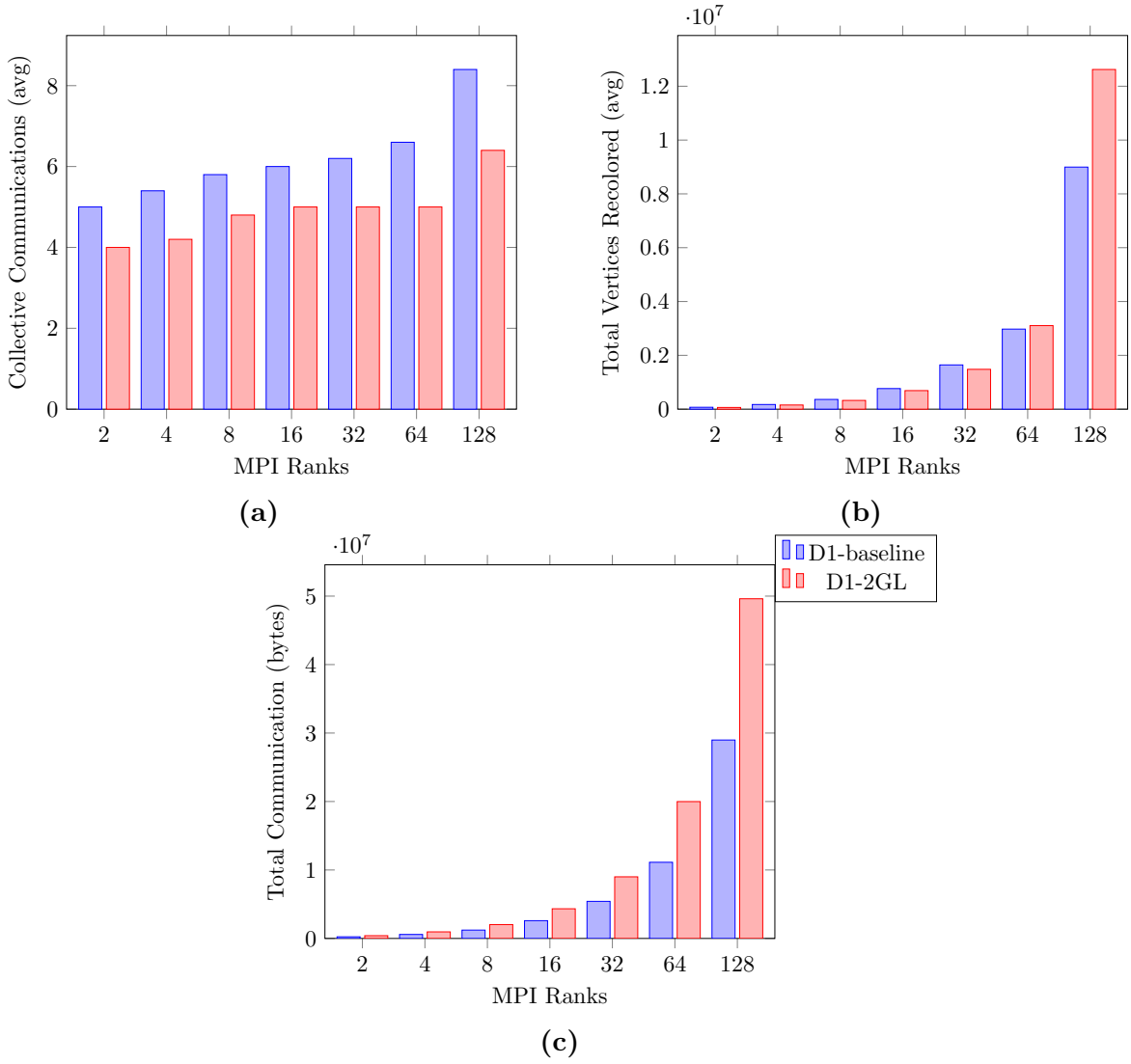

Figure 2.7: D1 vs D1-2GL comparisons of (a) Communication rounds, (b) Recoloring workloads, and (c) Total communication in bytes.

2.6.5 Distance-2 Performance

We compare our D2 method to Zoltan's distance-2 coloring using eight graphs from Table 2.1: Bump_2911, Queen_4147, hollywood-2009, europe_osm, rgg_n_2_24_s0, ldoor, Audikw_1, and soc-LiveJournal1. We use the same experimental setup as with the distance-1 performance comparison. Figure 2.8a shows that D2 compares well against Zoltan in terms of execution time, with D2 outperforming Zoltan on all but two graphs. In the best case, we see an 8.5x speedup over Zoltan on the Queen_4147 graph.

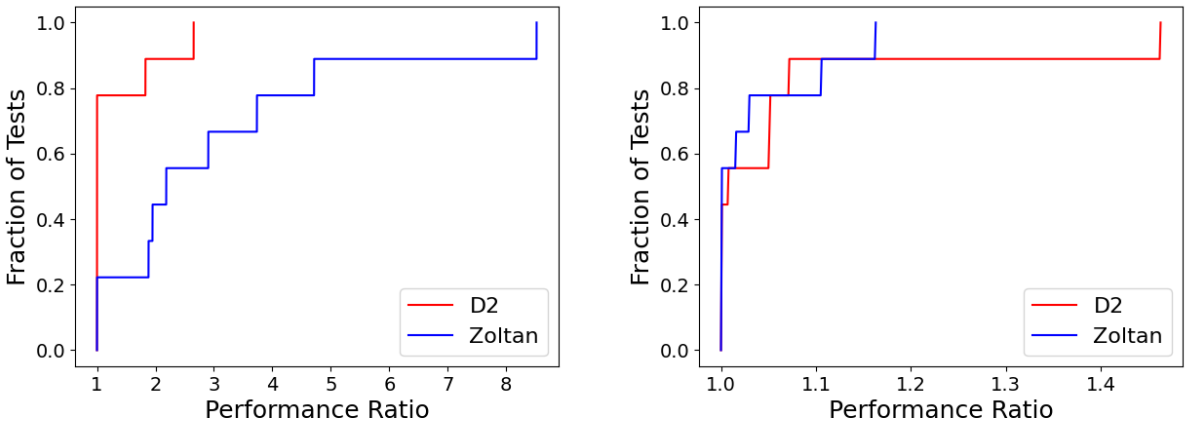

Figure 2.8: Performance profiles comparing D2 on 128 Tesla V100 GPUs with Zoltan's distance-2 coloring on 128 Power9 cores in terms of (a) execution time and (b) number of colors computed for a subset of graphs listed in Table 2.1.

Figure 2.8b shows that D2 has similar color usage as Zoltan. D2 and Zoltan each produce fewer colors in half of the experiments. In all but one case in which Zoltan uses fewer colors, D2 uses no more than 10% more colors. Interestingly, the number of colors used by D2 on the soc-LiveJournal1 graph is unchanged with one and 128 GPUs. Zoltan outperforms D2 with respect to runtime on skewed graphs because Zoltan has distance-2 optimizations which reduce communication overhead and minimize the chance for distributed conflicts. Implementing such optimizations within our code is a target for future work.

2.6.6 Distance-2 Strong Scaling

Figures 2.9a and 2.9b show the strong scaling behavior of D2 and Zoltan on Bump_2911 and Queen_4147. Bump_2911 shows that D2 scales better initially than Zoltan, and with 128 ranks, D2 is 2.9x faster than Zoltan, using 0.7% more colors. Queen_4147 shows better scaling for D2 as well; with 128 ranks, D2 is 8.5x faster than Zoltan and uses 10% fewer colors.

On average over the eight graphs, D2 exhibits 4.29x speedup on 128 GPUs over a single GPU, and uses 7.5% more colors than single GPU runs. Speedup is greater with D2 than D1 because distance-2 coloring is more computationally intensive, and thus has a larger work-to-overhead ratio.

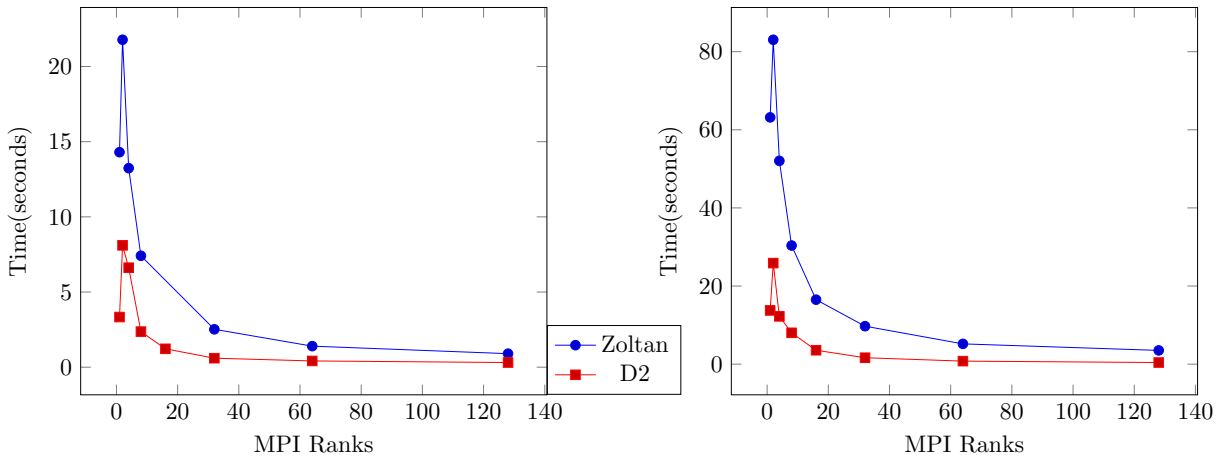

Figure 2.9: D2 and Zoltan strong scaling for distance-2 coloring on (a) Bump_2911 and (b) Queen_4147.

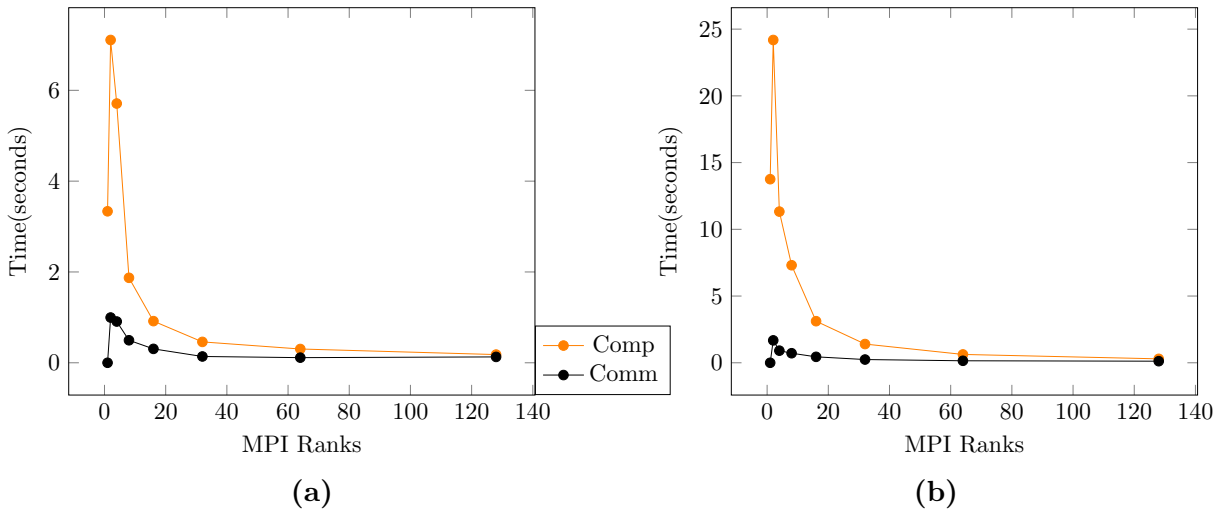

Figure 2.10: D2 communication time (comm) and computation time (comp) from 1 to 128 GPUs for (a) Bump_2911 and (b) Queen_4147.

Figures 2.10a and 2.10b show the communication and computation breakdown of D2 on Bump_2911 and Queen_4147. Bump_2911 shows computation and communication scaling for up to 128 ranks, while color usage increases by only 0.6%. In general, the relative increase in color usage from a single rank for distance-2 coloring is less than for distance-1 coloring. The number of colors used for distance-2 coloring is greater than for distance-1; therefore, a similar absolute increase in color count results in a lower proportional increase.

Table 2.2: Summary of the graphs used for PD2 tests. Statistics are for the bipartite representation of the graph (Section 2.4.6). δ_{avg} is average degree and δ_{max} is maximum degree. Numeric values listed are after preprocessing to remove multi-edges and self-loops. k = thousand, M = million.

Graph	Class	#Vtx	#Edges	δ_{avg}	δ_{max}
Hamrle3	Circuit Sim.	2.9 M	5.5 M	3.5	18
patents	Patent Citations	7.5 M	14.9 M	1.9	1k

2.6.7 Distance-2 Weak Scaling

Figure 2.11 demonstrates the weak scaling behavior for D2. The same hexahedral mesh graphs were used as in the D1 weak scaling experiments. In general, D2 has fairly consistent weak scaling. Note that relative to the distance-1 problem, distance-2 has a much higher complexity. This explains the larger observed differences in execution time between the test instances on a single node. Weak scaling to large process counts is observed to be quite good for all workloads. The largest 76.7 billion edge test still completes in under a minute on 128 GPUs.

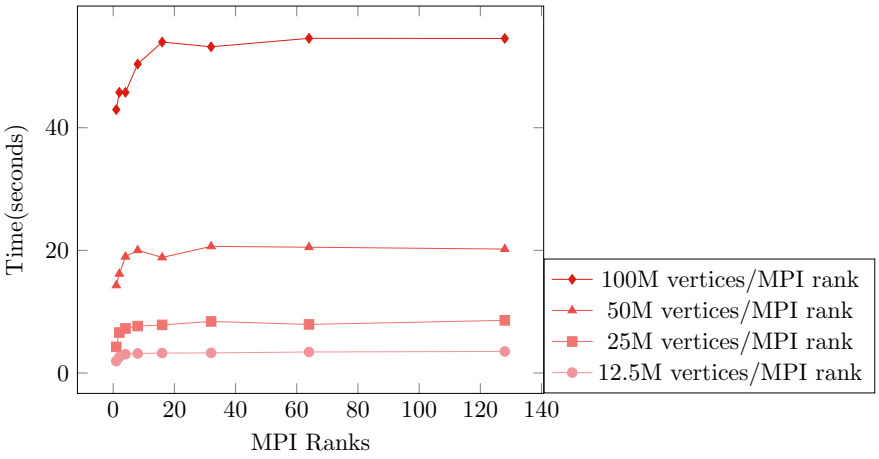

Figure 2.11: Distance-2 weak scaling of D2 on 3D mesh graphs.

2.6.8 Partial Distance-2 Strong Scaling

Table 2.2 shows the graphs that we used to compare our PD2 implementation against Zoltan. Partial distance-2 coloring is typically used on non-symmetric and bipartite graphs; the graphs in Table 2.2 are representative of application use cases. We give metrics reported

for the bipartite representation of the graph (as described in Section 2.4.6). Partial distance-2 colorings typically are needed for only a subset of the vertices in a graph, but our PD2 implementation colors all vertices in the graph. In contrast, Zoltan colors only vertices that would be colored in typical partial distance-2 coloring. Modifying our local coloring routines to only color the necessary vertices is an avenue for future work.

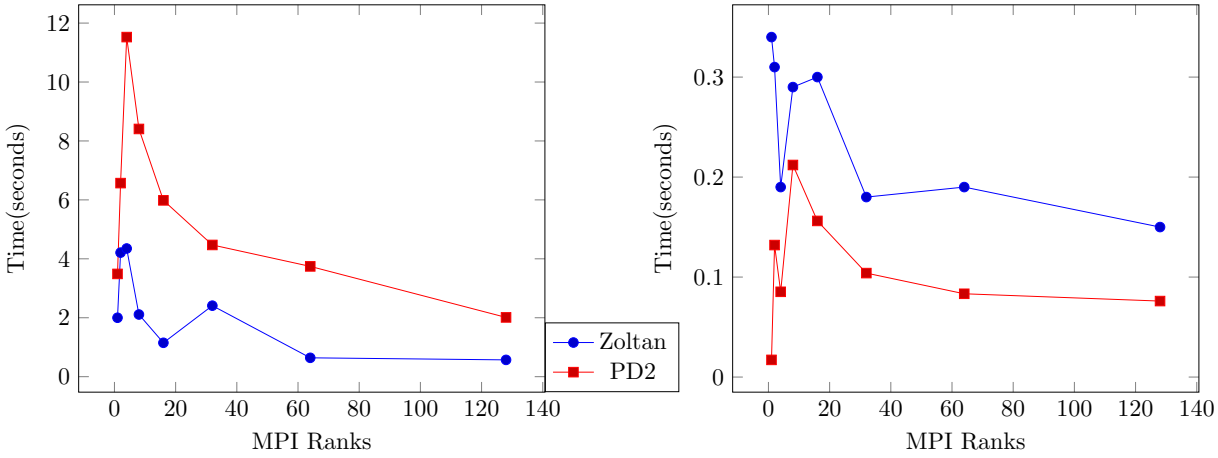

Figure 2.12: PD2 strong scaling for partial distance-2 coloring on (a) patents and (b) Hamrle3.

Figure 2.12 shows the strong scaling behavior of PD2. The experiment with Hamrle3 on four ranks benefits from a particularly good partitioning that results in less recoloring for both PD2 and Zoltan relative to other process configurations. A detailed investigation into the impact of partition quality on coloring performance would make for interesting future work. For the patents graph, D2 has a particularly heavy recoloring workload for four ranks, resulting in a large increase in runtime from two to four ranks. Even though PD2 is coloring more vertices than Zoltan in these tests, PD2 achieves roughly 2x speedup on 128 ranks with Hamrle3. With patents, Zoltan is faster than PD2; this result can be attributed partially to Zoltan's optimized recoloring scheme that reduces the number of conflicts introduced while recoloring distributed conflicts. PD2 achieves a 1.73x speedup over a single GPU with the patents graph, while it did not show any speedup from a single GPU with Hamrle3. Figure 2.12a shows that, with the patents graph, Zoltan is faster on one core than a single GPU. This speedup is attributed to Zoltan's coloring fewer vertices than PD2; when Zoltan colors the same number of vertices as PD2, their single rank runtimes are closer. Investigating the cause of this result is another subject for future research.

For these two graphs, PD2 uses a very similar number of colors as Zoltan. PD2 uses at most 10% more colors in the distributed runs. This difference is typically only one to five colors more than Zoltan.

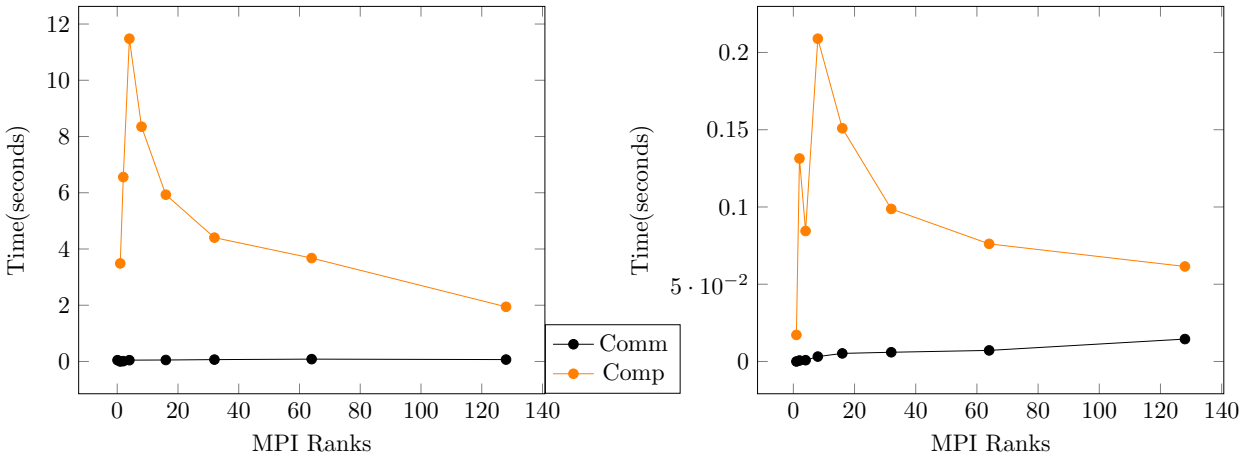

Figure 2.13: PD2 communication time (comm) and computation time (comp) from 1 to 128 GPUs on (a) patents and (b) Hamrle3.

Figure 2.13 shows that computation is the main factor in the scaling behavior of PD2. In distributed runs, the largest factor of the runtime is the computation overhead involved in recoloring distributed conflicts. Figure 2.13b shows an unexpected decrease in computation for the Hamrle3 graph for four ranks, which is due to a decrease in the recoloring workload. PD2's recoloring workload is approximately 25,000 vertices per rank in most experiments, but the four-rank experiment has a recoloring workload of 9,000 vertices per rank. Figure 2.13a shows that the four-rank PD2 run has a much longer computation time than expected; this is due to the total distributed recoloring workload increasing by a factor of six. Additionally, the 64-rank run with the patents graph shows slightly less computational scaling than expected, due to an increase in recoloring rounds. Increasing recoloring rounds serializes recoloring computation and incurs more rounds of communication, resulting in a runtime increase. Optimizing recoloring to reduce subsequent conflicts and reduce the number of recoloring rounds necessary in D2 and PD2 are also subjects for future research.

2.7 Conclusion

This chapter explored an implementation of a distributed graph coloring framework that is able to run on multiple GPUs, as well as leverage other local shared memory parallelism. We showed that our approach is competitive with the previous Zoltan approach in both runtime and number of colors used. Our implementation can produce distance-1, distance-2, and partial distance-2 colorings, in order to support a variety of scientific computing applications including Automatic Differentiation. The next chapter will discuss an efficient tailored algorithm for solving a specific graph connectivity problem for a land-ice simulation.

CHAPTER 3
ICE SHEET CONNECTIVITY

3.1 Chapter Introduction

The previous chapter examined a multi-GPU framework for distributed graph coloring, in support of scientific computing applications such as Automatic Differentiation In this chapter, we introduce a graph algorithm that solves a very specific connectivity problem for a land-ice simulation. The previous approach was implemented in a MATLAB script that did not execute on the same platform as the application. Our approach not only improved on the previous approach's runtime by orders-of-magnitude, but was also implemented in distributed memory, allowing it to be called during a simulation. This cuts down on the time it takes to output data from the simulation and transfer it to the MATLAB script, as well as the time taken to solve the connectivity problem. This chapter also explores how to generalize the specific ice sheet algorithm to solve graph biconnectivity in distributed memory.

3.2 Degenerate Features in Ice Sheet Meshes

Modeling sea-level rise (SLR) is important in climate modeling. A major factor contributing to SLR is mass loss from the Greenland and Antarctic ice sheets [47]. To predict SLR accurately, ice sheet dynamics are simulated using large-scale parallel computational models (e.g. [2], [48], [49]). Typically, these models assume that ice flow is reduced by friction with bedrock, and fail to simulate regions, such as icebergs, where the ice is floating and partially or completely detached from the land ice. It is therefore important to detect these regions during simulation so that they can be properly handled.

Most large scale ice sheet models employ finite-element or finite-volume discretizations using unstructured ice sheet meshes. These are either 2D or 3D meshes generated by extruding 2D basal meshes. Therefore, detecting regions that are problematic for the solvers reduces to detecting "degenerate features" in 2D basal ice sheet meshes as detailed in [50]. These degenerate features can develop over the course of an ice sheet simulation; therefore,

This chapter previously appeared as: I. Bogle, K. Devine, M. Perego, S. Rajamanickam, and G. M. Slota, "A parallel graph algorithm for detecting mesh singularities in distributed memory ice sheet simulations," In *Proc. of the 48th Int. Conf. on Parallel Process.*, Aug. 2019, pp. 1-10.

it is of paramount importance that the detection can be efficiently performed at runtime on meshes held in distributed memory.

In our study, we consider only 2D conformal meshes, such as the basal triangular and quadrilateral meshes used in the Albany Land Ice (Albany-LI) [3] velocity solver component of the MALI model [49]. Zou et al. [51] considered the detection of degenerate features in non-conformal block-structured grids; their proposed method works efficiently for data generated by adaptive mesh refinement algorithms. However, they detect only a subset of the degenerate features that we are required to detect.

The degenerate mesh features we must detect are similar to the biconnected components of a graph. Biconnected components are maximal subgraphs of a graph such that the removal of any single vertex will not disconnect the subgraph. Zou et al. only considers detection of connected components. Connected components are maximal subgraphs such that at least one path exists between all vertex pairs within the subgraph.

We view a mesh as an undirected graph, with mesh vertices corresponding to vertices in the graph, and graph edges defined by the element connectivity in the mesh. Finding the biconnected components of the graph reveals vertices that are single points of failure, such as floating chunks of ice that are about to become icebergs. While our study focuses on ice sheet simulations, biconnectivity algorithms are also useful for fault tolerance in ad hoc networks [9], and detecting mechanisms in meshes for structural dynamics [52]. There are shared-memory algorithms for graph biconnectivity (e.g., Tarjan and Vishkin [53]), but to support parallel ice sheet simulations such as Albany-LI, distributed-memory algorithms that do not rely on a global view of the graph are needed.

Our Contributions: We present an efficient, distributed-memory algorithm for detecting degenerate features in ice sheet meshes. We implemented our algorithm in the Zoltan2 [54] graph algorithms library and demonstrated it with the Albany-LI solver. We show that our algorithm is fast enough to run at every step of a simulation, taking at most 0.4% of the runtime of a single solver step in Albany-LI using a 13M element mesh. We explore the algorithm's performance with synthetic meshes in which we vary the size and number of degenerate features, and show that the algorithm performs well for meshes that approximate real ice sheet meshes.

3.3 Background and Related Work

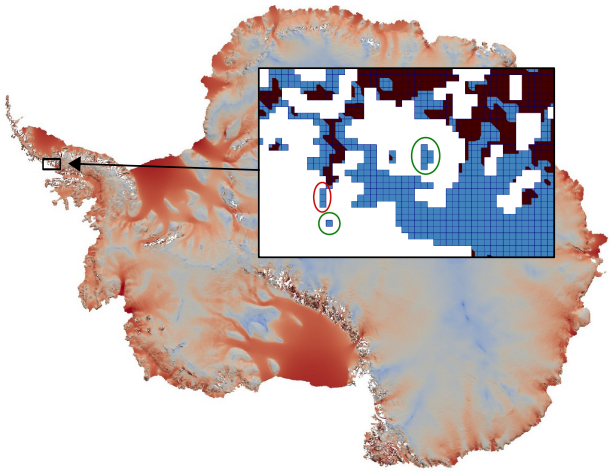

Figure 3.1: Antarctic Ice Sheet colored with ice surface velocity magnitude (red = fast, blue = slow), and mesh detail showing a realistic map of floating (light blue) and grounded (brown) ice and degenerate features marked with green (icebergs) and red (hinges) circles.

Degenerate features of an ice sheet mesh correspond to parts of the ice that are floating and either detached from grounded ice (icebergs) or loosely attached to it (hinged peninsulas). Examples of these features are shown in the mesh of Antarctica in Figure 3.1. We refer to vertices corresponding to places where the ice is in contact with the ground as "grounded" vertices and to vertices corresponding to places where the ice is floating as "floating" vertices. Icebergs (i.e., floating islands) are connected components of the mesh that are constituted entirely of floating vertices. Hinged peninsulas, or "hinges," are portions of the mesh that are constituted of floating vertices and that can be disconnected from the mesh by the removal of a single vertex. Both icebergs and hinged peninsulas can negatively impact solver convergence and need to be identified as degenerate features. In fact, in these regions, ice flow equations are typically ill-posed because the ice velocity is known only up to translations and/or rotations.

Given a mesh and information about whether each mesh vertex is grounded or floating, our goal is to determine whether or not each mesh vertex is part of a degenerate feature.

Degenerate Feature Detections Algorithms: The previous detection algorithm for Albany-LI was implemented as a standalone MATLAB code [50] and used as a pre-processor to Albany-LI simulations. To remove floating islands, Tuminaro et al. used a Breadth-First

Search (BFS) from every unvisited grounded vertex. Vertices that remained unvisited after the BFS operations had to be part of a floating island. Tuminaro et al. used graph coloring to remove hinged peninsulas. Each mesh vertex was colored so that no neighboring vertices had the same color. Then, for each color, they temporarily removed vertices of that color and re-ran the floating island algorithm; hinged peninsulas would become floating islands when the colored hinge vertices were temporarily removed. This approach generally had runtimes of several minutes when running on meshes with millions of elements.

A closely related problem was addressed in [51], considering meshes generated by adaptive mesh refinement (AMR) algorithms. This work found icebergs (but not hinged penisulas) in multi-level, multi-resolution meshes. Their method detects connected components at each level of the AMR structure and joins them together globally. To find connected components at a single level, they use the SAUF algorithm [55], a two-pass labeling algorithm that assigns temporary labels to each vertex on the first pass, uses a Union-Find data structure to determine which temporary labels are equivalent, and finalizes labels on the second pass. In the distributed implementation, Zou et. al run the first pass of SAUF on each process, considering only local vertices. Then they exchange ghost label information, and construct the final set of labels by sending all local Union-Find structures to an elected process on the same AMR level. After assigning labels on a level-by-level basis, they do a similar procedure to assign final labels for the entire AMR structure. Each unique label then corresponds to a connected component across the entire AMR structure. Similarly, Harrison et al. [56] present a connected component algorithm for 3D meshes based on union-find operations; they demonstrate its performance on 2197 processor with two-billion element meshes. However, these connected component approaches find icebergs only. Our biconnected component method detects both icebergs and floating peninsulas that are connected to the main ice sheet by a single point — features that create significant challenges for solvers.

Biconnectivity Algorithms: Graph biconnectivity decomposition algorithms seek to identify all maximal biconnected subgraphs as well as all cut vertices within some graph. Cut vertices, commonly referred to as articulation points or articulation vertices, are single vertices that disconnect the graph when removed.

Biconnectivity is a well studied problem, with a work-optimal serial algorithm presented by Hopcroft and Tarjan [57]. This work optimal algorithm uses Depth First Search (DFS) to identify biconnected components. DFS algorithms, however, are not easy to parallelize, as

discussed by Tarjan and Vishkin [53]. Tarjan and Vishkin [53] present a parallel algorithm for finding biconnected components in a concurrent-read, concurrent-write parallel RAM model, where each processor has access to shared memory. This parallel algorithm reduces the problem of biconnectivity to the problem of connectivity in an auxiliary graph. The auxiliary graph can be constructed without using DFS, and its connectivity can then be found efficiently in shared memory.

Two shared memory algorithms for biconnectivity exploit simple graph operations such as BFS and color propagation [58] to identify articulation points. The BFS-based algorithm does BFS sweeps to determine whether children of certain vertices can reach other vertices on their parents' level; if so, the parent is not an articulation point. The coloring version also does an initial BFS; it then uses color propagation rules to prevent certain color labels from being passed through articulation points. Most recently, LCA-BiCC was proposed by M. Chaitanya and K. Kothapalli [59]. They observe that that finding bridges (edges whose removal disconnects the graph) is more parallelizable than finding articulation vertices. They use an arbitrarily rooted BFS on the input graph to find a set of non-tree edges (edges not in the BFS tree). Then they find the Lowest Common Ancestor (LCA) for each of the endpoints of the non-tree edges. They show that this set of LCA vertices is guaranteed to contain the set of all articulation points in the graph, although it can contain non-articulation points as well. Edges not visited in finding LCA vertices are bridges in the original graph; each bridge has endpoints that are articulation points. The algorithm is then applied recursively on the subgraphs formed by excluding the bridges. An efficient biconnectivity algorithm has not yet been demonstrated on a distributed graph representation. Distributed 2-edge connectivity algorithms exist, but these only find bridges.

One could use a biconnectivity algorithm to perform degenerate feature detection by identifying biconnected components and then testing each component to determine whether it is connected to ground. (Indeed, we used this strategy to verify the correctness of our proposed algorithm.) But by exploiting information about the mesh to identify potential articulation points and propagating the grounding information in the BFS operations, we can provide an efficient algorithm that is feasible for distributed memory.

3.4 Ice Sheet Feature Detection

Algorithm 8 shows a high-level view of our approach. The inputs of our algorithm are a mesh M and grounding information for each vertex indicating whether the vertex is grounded. From the mesh M, we can extract a graph $G = (V, E)$ in which vertices in V correspond to mesh vertices in M and edges in E correspond to mesh vertex adjacencies along element edges in M.

Algorithm 8 Degenerate feature detection algorithm

```
1: procedure PROP-ALG(Mesh M,grounding_info)
2:     Compute set of potential articulation points using M
3:     Extract graph G from M
4:     labels ← ∅
5:     Propagate initial grounding_info labels in G
6:     while Propagation is incomplete do
7:         Continue propagating, repropagate if necessary
8:     Return labels indicating connection to ground
```

The first step in Algorithm 8 is to identify a set of potential articulation vertices in the mesh. For correctness, this set must include at least all true articulation points. The set may include vertices that are not true articulation points; indeed, the set of all mesh vertices can be used. However, the algorithm completes more quickly if the set of potential articulation points is close to the set of true articulation points, and we can exploit mesh information to closely approximate the set of true articulation points (Section 3.4.1).

Grounding information is then passed from grounded vertices to neighboring vertices via label propagation. Each vertex v has a label that is a structure of four vertex IDs: two representing grounded vertices to which there is a path from v in G, and two that record which vertices propagated the grounded vertex IDs to v. Grounded vertices' labels are initialized with their vertex IDs, and they propagate their vertex IDs to neighboring vertices in a breadth-first manner. When a vertex receives new grounded information, it stores the grounded vertex IDs it received and propagates them further (Section 3.4.2). Initial propagation halts at potential articulation points. Then propagation is restarted, with care taken to propagate grounding information correctly through potential articulation points (Section 3.4.3). Once all vertices have been labeled correctly, propagation ends and the labels indicate whether a vertex is grounded or not. Vertices with two grounded-vertex

IDs in their labels have two paths to the ground, and thus, are not part of any degenerate features. Other vertices are part of degenerate features and will be removed by the ice sheet simulation.

3.4.1 Identifying Potential Articulation Points

When considering a geometric 2D mesh, all articulation points will be located on the boundary. Given the elements of a mesh, it is straightforward to compute the boundary edges of the mesh; the boundary edges are those that are not shared by two elements. Then, given the list of boundary edges, we determine which vertices are potential articulation points by looking at the number of boundary edges incident to each boundary vertex. If a boundary vertex has two incident boundary edges, it cannot be an articulation point; its two incident edges must either come from adjacent elements sharing some other edge incident on that boundary vertex or exists on a corner of the mesh. If a vertex has more than two incident boundary edges, it is a potential articulation point. For example, in Figure 3.2 (top), vertex B is a potential articulation point because it has more than two boundary edges, while vertex F is not.

3.4.2 Label Propagation Rules

All grounded vertices initialize the grounded vertex IDs in their labels to their own vertex IDs. During label propagation, vertices share their grounding information with neighboring vertices. Algorithm 9 shows the rules used to update a neighboring vertex's label. Vertices that are not potential articulation points may give their neighbors all (zero, one or two) of the unique grounded vertex IDs that they have. The neighboring vertices track from which vertices they received each grounded vertex ID. If a neighboring vertex already has two grounded vertex IDs, it is "full" and is not updated.

We ensure that potential articulation points send only one vertex ID to each neighbor, as an articulation point can contribute only one point of contact to the ground to any neighbor. Full potential articulation points pass their own vertex ID as a grounded vertex, rather than either of the grounded vertex IDs that they store. "Half-full" potential articulation points (those having only one grounded vertex ID in their label) pass their one grounded vertex ID. Because labels contain the vertex ID of the vertices giving a stored grounded vertex ID, a potential articulation point can determine whether or not it has previously given

the neighbor a grounded vertex ID.

Algorithm 9 Function for updating neighboring vertices labels during propagation

```
1: procedure GIVE-LABELS(curr_vtx, neighbor)
2:     if curr_vtx ∈ Potential_Articulation_Points then
3:         if curr_vtx hasn't sent anything to neighbor before then
4:             if curr_vtx has two grounded vertex IDs then
5:                 curr_vtx gives neighbor its own vertex ID as
6:                 a grounded vertex
7:             else
8:                 curr_vtx gives its only grounded vertex ID
9:                 to neighbor
10:    else
11:        curr_vtx gives neighbor any grounded vertex IDs
12:        that neighbor doesn't have
```

3.4.3 Propagation on Two Frontiers

We use two queues to manage propagation along two "frontiers": propagation from potential articulation points (*art_frontier*) and propagation from other vertices (*frontier*). As shown in Algorithm 10, the two queues separate the potential articulation points from the other vertices while propagating. Initially, grounded vertices are placed in the appropriate queue (*frontier* or *art_frontier*, depending on whether they are or are not potential articulation points). Our algorithm begins propagation from the *frontier* queue; when that queue is empty, it swaps to the *art_frontier* queue and resumes propagation. This swap-and-propagate pattern continues until both queues are empty, meaning no labels changed in the previous iteration. Separating potential articulation vertices from non-articulation vertices allows potential articulation points to accrue as much grounding information as possible before passing that grounding information along. In particular, it increases the likelihood that potential articulation points will have full labels before they propagate their values, which reduces the total number of propagation iterations we need.

3.4.4 Multi-Phase Conditions

Although rare in ice sheet meshes, features such as chains of articulation points can occur and require special handling to correctly identify all degenerate features. These situations require additional phases of the propagation algorithm. Figure 3.2 shows an example.

Algorithm 10 BFS-based label propagation algorithm

```
1: procedure BFS-Prop(frontier,art_frontier,labels,G=(V,E))
2:     if frontier.empty() then
3:         swap(frontier,art_frontier)
4:     while !frontier.empty() do
5:         curr_vtx ← frontier.pop()
6:         for all neighbors n of curr_vtx do
7:             Give-Labels(curr_vtx,n)
8:             if n's label changed then
9:                 if n ∈ Potential_Articulation_Points then
10:                    art_frontier.push(n)
11:                else
12:                    frontier.push(n)
13:        if frontier.empty() then
14:            swap(frontier,art_frontier)
```

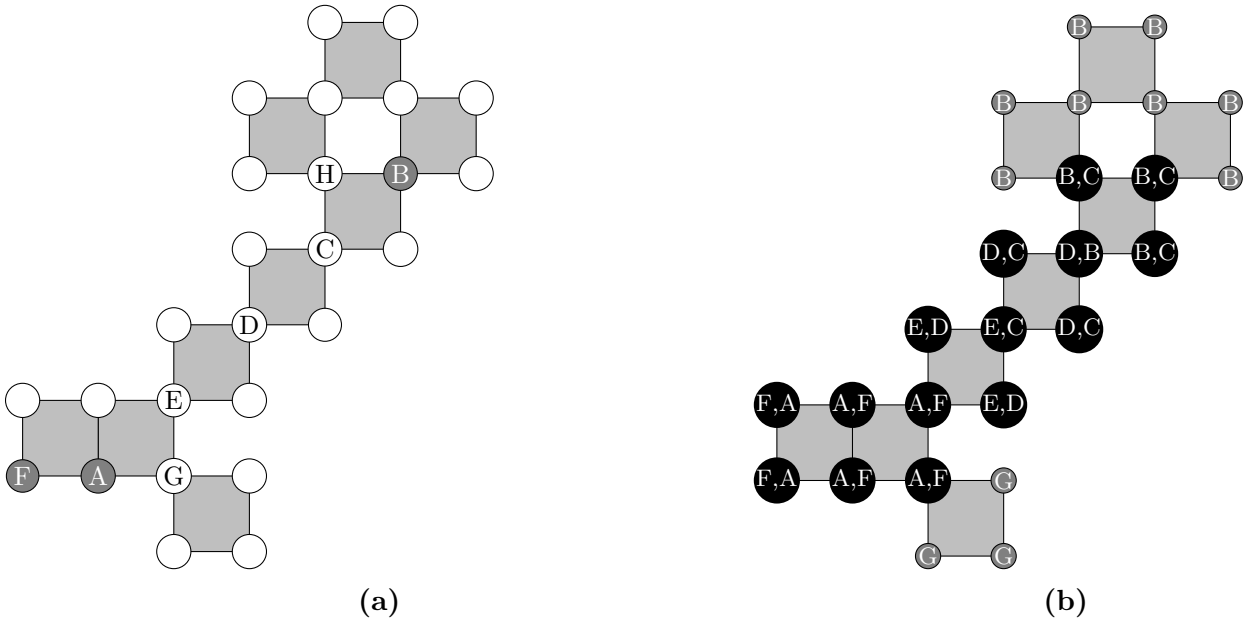

Figure 3.2: An example demonstrating the state of a mesh (a) Before propagation and (b) After propagation.

To determine whether additional propagation is needed, we identify potential articulation vertices that have two grounded vertex IDs in their label and a half-full neighbor vertex whose only grounded vertex ID is not the potential articulation point's ID. We add all such potential articulation points to a frontier, reset labels that have only one grounded vertex ID, and propagate from this new frontier. Algorithm 11 shows this addition to BFS-Prop. The multi-phase approach is required only when certain topological conditions exist in the

mesh and there is a large difference in shortest path distance between two grounded vertices and non-grounded vertices in the mesh. These conditions are rarely, if ever, encountered in practice, but we provide mitigation to ensure correctness.

Algorithm 11 Driver for BFS-Prop, Complete, Multi-Phase Solution

```
1: procedure BFS-Prop-Driver(labels, G=(V,E))
2:     art_frontier ←grounded potential articulation vertices
3:     frontier ←grounded vertices that are not potential
4:              articulation points
5:     BFS-Prop(frontier,art_frontier,labels,G)
6:     while true do
7:         for all potential articulation points v ∈ V do
8:             if v's label is full then
9:                 for all neighbors n of v do
10:                    if n's label is half-full and doesn't have
11:                  v's ID then
12:                        art_frontier.push(v)
13:        if art_frontier.empty() then
14:            break
15:        for all v ∈ V do
16:            if v's label is half full then
17:                clear v's label
18:        BFS-Prop(frontier,art_frontier,labels,G)
```

3.4.5 Propagation Example

Figure 3.2 shows an example that illustrates parts of our algorithm. Initially, only grey vertices A, B, and F are grounded; all white vertices are not grounded initially. Labels for A, B and F are initialized to their respective IDs. Vertices A and F are placed in the *frontier* queue, since they are grounded and not potential articulation points. As a potential articulation point, vertex B is placed in *art_frontier*.

Vertex A propagates its label to its neighbors and adds its neighbors to the appropriate queues depending on their potential articulation status. In particular, A gives F its label so that F is full with label (F, A). Vertex F then propagates its full label to its neighbors, and the neighbors propagate (A, F) to their neighbors. (Note that labels (F, A) and (A, F) are equivalent.) The neighbors of A and F in *frontier* then propagate label (A, F) to potential articulation points E and G, giving them both full labels.

Next, from the *art_frontier* queue, vertices B, E, and G propagate to their neighbors; because E and G have full labels (A, F), they propagate their own vertex IDs (see Algorithm 9). Not-full neighbors of E and G are placed in the *frontier* queue (since their neighbors are not potential articulation points), and the swapping between the *frontier* and *art_frontier* queues continues.

We eventually reach the state in Figure 3.2 (bottom). This state is not the final state, as the vertices labeled B do not yet have full labels. However, at this point, the queues are both empty. Algorithm 11 (lines 7-12) finds that the vertices labeled B may not be in their final state, so a new round of propagation is initiated. All half-full labels are emptied, and propagation is restarted from vertex H. We do not need to restart propagation from vertex G, so the degenerate feature connected to G ends up with no labels.

3.5 Distributed Memory Implementation

We have created a distributed memory implementation of our algorithm that is callable by parallel mesh-based applications. Because the application's mesh is likely already distributed to processors in a balanced manner for its computation, we use the same distribution of data to processors as the application. We assume the commonly used "owner computes" strategy for the application's distribution of mesh entities; that is, each processor identifies a set of unique vertices for which it is responsible. Our load balance and communication patterns are thus determined by the application's distribution of the data; however, since our runtimes are very small compared to the application's solve times, redistributing data within our method to adjust load balance is not worthwhile.

The application also provides all edges incident to its owned vertices, including edges to off-processor vertices. Then in each processor p, we store an "owned" graph vertex for each mesh vertex owned by p. We also create one layer of "ghost" vertices — copies of vertices that are owned by some processor $q \neq p$, and are neighbors of vertices owned by processor p. Using the edges provided by the application, we create a "local" graph consisting of the owned vertices, the ghost vertices and the edges.

In distributed memory, the label-propagation algorithm can maintain its "push" of label values to neighboring vertices without ill effects. We perform Algorithm 10 (BFS-Prop) independently on each processor's local graph, allowing labels to propagate to both owned

and ghost vertices. Once local propagation stops, we communicate (via MPI point-to-point messages) to push the ghost vertices' labels to their owning processor. The owned labels are then made consistent with values received from the ghost copies. A global communication (all-reduce) is used to determine whether any processor had meaningful label changes due to the exchange of ghost information. If so, the ghost vertices are updated via point-to-point communication from their owners and propagation resumes.

We use existing classes from the Tpetra [60] package of Trilinos [61] to implement this strategy. We build Tpetra Maps to describe the distribution of the owned and ghosted vertices among processors. To store labels for owned and ghosted vertices, we use the Maps to create a Tpetra FEMultiVector — a distributed vector that provides ghost-exchange communication and ghost update capabilities. The FEMultiVector was designed for finite element assembly operations, allowing processors to contribute physical values to ghost vertices and sum the contributions from multiple processors. It is templated on a scalar type, which is usually *double* in physics simulations. We, however, provide our label structure as the scalar type, and overload its summation operator to use our Give-Labels function (Algorithm 9). To use the FEMultiVector's communication capabilities, we call FEMultiVector's method beginFill() before we start the local label propagation, and endFill() to communicate the ghost vertices' labels to their owners and "sum" their values into the owned versions using the overload summation operator. To communicate the owned labels back to the ghost copies, we use the FEMultiVector method doOwnedToOwnedPlusShared().

3.6 Correctness Proofs

Here we will prove the correctness of our algorithm. First, we'll show that any vertex with two labels after propagation completes must have at least two internally-vertex-disjoint paths to grounded vertices. As mentioned in our algorithms description, the existence of at least two such paths indicates *grounded* status to the given vertex; we keep these vertices during simulation. We don't need to differentiate between vertices that were initially grounded or those that were marked grounded during propagation. Then, we'll show that our propagation rules guarantee any vertex with two internally-vertex-disjoint paths will end with two labels. Taken together, we have our primary proposition:

Proposition 3.6.1 *Under our label propagation rules, a vertex n will own two unique labels,*

l_1 and $l_2 \iff \exists P_1, P_2$, two vertex-disjoint paths from n to two unique grounded vertices, v_1 and v_2.

Proof: A is set of potential articulation points. B is a pseudo-BiCC component containing n. A pseudo-BiCC is a maximum biconnected subgraph bounded by potential articulation points. v_1 and v_2 are grounded vertices, potentially anywhere in the graph. Paths P_1, P_2 are termed (internally) vertex-disjoint if they have no vertex in common except n. By our propagation rules, n having some label l implies that $\exists P$, where P is a (minimal length) path tracing n to some grounded vertex $v \in G$, as labels only propagate along edges.

We first prove that the existence of two unique labels implies the existence of two vertex-disjoint paths to grounded vertices. Assume vertex n has two vertex identifiers in its label, v_1 and v_2.

Trivial Cases: The trivial cases are if both labels refer to vertices v_1 and v_2 which are neighbors of n, or if one label refers to n (i.e., n is grounded). The first case is obvious, and our propagation rules implies $\exists P_2$ from n to v_2 for the second case. This path would obviously be vertex-disjoint with empty path $P_1 = \{n\}$.

Nontrivial Case: Assume n has two labels referring to grounded vertices located anywhere within the graph. Per our propagation rules, this implies $\exists P_1, P_2$ from n to each of v_1, v_2. An equivalent statement to what we're trying to prove is that $\exists P = \{v_1, \ldots, n, \ldots, v_2\}$; i.e., a path from v_1 to v_2 containing n. Assuming this path doesn't exist, $\exists s$, where s is one vertex that P_1 and P_2 must traverse between both v_1 or v_2 and n. The removal of only s would therefore disconnect n from v_1 and v_2; hence, s is an articulation point and $s \in A$, as actual articulation points are a subset of A. By our propagation rules, s can only pass one label, so n having labels v_1 and v_2 is a contradiction, and therefore such an s can't exist.

We next show that the existence of two vertex-disjoint paths to grounded vertices for n implies that n terminates propagation with two labels. Assume that n has two vertex-disjoint paths, P_1 and P_2, to initially grounded vertices, v_1 and v_2. We consider three cases.

Case 1 – $n, v_1, v_2 \in B$: In this case, all three vertices considered are contained within the same pseudo-BiCC. Based on our propagation rules, the labels v_1 and v_2 will propagate from the grounded vertices. If we consider P_1 and P_2 as a minimal vertex-disjoint pair of paths, these labels will reach n without passing through some potential articulation point to

another pseudo-BiCC; B itself is biconnected, which with our propagation rules guarantees the existence of some path through any three vertices contained within it.

Case 2 – $v_1 \in B$ and $v_2 \notin B$: In this case, v_2 is contained in B', a pseudo-BiCC distinct from B. v_1's label will propagate along P_1 within B to n unimpeded. v_2's label will begin propagating along P_2, which potentially passes through some number of potential articulation points. If B' neighbors B (*Case 2.1*) with some $x \in A, B, B'$, labels from both v_1 and v_2 will reach x. When the propagation frontiers swap, x will propagate its own label along the portion of P_2 to n, giving n its second label.

If there are multiple pseudo-BiCCs between B and B', then at some $y \in A$ along P_2 labels from v_1 and v_2 will intersect. If P_1 and P_2 are both minimal paths and not just a minimal vertex-disjoint path pair (*Case 2.2*), y will propagate its label along P_2 toward n until it either reaches n or some $z \in A$, which will subsequently begin propagation of its label toward n. After some number of potential articulation points, two labels will reach $x \in A, B$, which will finally propagate its label the rest of the way to n along P_2 within B.

If P_1 and P_2 are not both minimal paths (*Case 2.3*) is when we might require multiple phases of iteration. The shorter path between v_1 or v_2 and n might propagate its label to "consume" all potential articulation points surrounding n, as demonstrated in Figure 3.2. Recall we start a new phase by clearing half-labels and re-propagating from the newly defined ground. In our new phase, we can guarantee that there exists some new ground v_3 that has a shorter path to n than v_1 or v_2 in the prior phase; at a minimum, the pseudo-BiCC where minimal paths between v_1, v_2 and n intersect will contain v_3. By guaranteeing that we decrease the distance of at least one minimal grounded path to n during each phase, we will eventually reach one of the prior cases.

Case 3 – $v_1 \notin B$ and $v_2 \notin B$: In this case, we can apply the logic used in *Case 2* on both P_1 and P_2. We omit it for brevity.

Note that in the above cases, if there are greater than just two vertex-disjoint paths from n, it simply follows that at least two unique labels will reach n. Conversely, we provide the following simple corollary to complete our correctness proof.

Corollary 3.6.2 *Under our label propagation rules, a vertex n will own less than two unique labels $\iff$ there exists less than two vertex-disjoint paths from n to unique grounded vertices.*

Proof: First assume that our algorithm terminates with n having less than two labels. If n

has two or more vertex-disjoint paths to grounded vertices, then by Proposition 3.6.1, n will have ended up with two unique labels, a contradiction.

Next, to prove the other direction of our corollary, assume that n has less than two vertex-disjoint paths from n to unique grounded vertices. If no path exists, then no label could reach n. If $\exists P_1$ from n to grounded vertex v_1, then either $v_1 \in B$ or $v_1 \notin B$. If $v_1 \in B$, then its trivial to show that the only label that will reach n is from v_1, as either there are no other grounded vertices, or paths from n from these grounded vertices must pass through $v_1 \in A$. If $v_1 \notin B$, then P_1 must pass through some $x \in A, B$. As x is a potential articulation point, it will only pass a single label, regardless of how many other grounded vertices exist outside of B with a path to n through x.

3.6.1 Complexity Discussion

Generally speaking, our propagation algorithm utilizes a "frontier" in a similar fashion to breadth-first search. A vertex will be placed on the frontier at most twice as its label set is filled. As such, the number of propagations a vertex will send along an edge during a given phase is bounded by a maximum of two. The number of phases we require is bounded above by the cardinality of our potential articulation point set. So we can give a worst-case work complexity of $O(|E||A|)$, where E is the set of edges and A is the set of potential articulation points. However, we note that we have never required more than a single phase on real data or the synthetic data in our results. In practice, we see a *linear expected work complexity* on real-world ice sheet data of $O(|E|)$. For parallel time, the number of propagation iterations is dependent on the diameter of the graph d, which for rectangular meshes grows approximately with $O(\sqrt{|V|})$, where V is the set of vertices; note that each propagation iteration itself can be parallelized in $O(1)$ time on $O(|E|)$ processors. We therefore have a worst-case time of $O(d|A|)$ and expected time of $O(d)$ on $O(|E|)$ processors.

3.7 Experimental Setup

The scaling results we present were obtained on AMOS, RPI's Blue Gene/Q housed at the Center for Computational Innovations. AMOS has 5K nodes with 80K cores and 80TB of RAM.

We used real meshes of Antarctica from the ProSPecT ice sheet project [62]. These

Table 3.1: Real (top) and synthetic (bottom) mesh data, including the numbers of vertices, elements, potential articulation points and vertices removed from the mesh.

Mesh	#Vertices	#Elements	Potential	#Removed
16km	52,465	51,087	21	0
8km	210,170	206,436	51	14
4km	841,346	831,173	174	6
2km	3,368,275	3,341,449	389	22
1km	13,479,076	13,413,766	606	65

Mesh	#Vtx (max)	#Elems (min)	#Potential (max)	#Removed (max)
ground(2km)	3,364,589	3,354,197	260	21
numdegen	~3,364,589	1,563,762	15,800	2,683,500
numcomplex	~3,368,253	2,878,139	17,313	21
longdegen	~3,365,592	1,565,262	904,858	63,000
longcomplex	~3,368,235	2,870,687	25,229	21
syn-largest	16,236,896	16,186,433	1550	96

meshes have geographic resolution from 16km to 1km. Smaller resolutions result in more refined meshes; thus, the number of elements ranges from 51K to 13.4M.

We also generated synthetic meshes by specifying the size of the central ice mass and the number and size of the degenerate and complex features. First, the central ice mesh was created by connecting a regular grid of vertices together in elements of four vertices. Then we create "complex" features, which are blocks that are similar to the central ice block, but smaller. These features are chains of elements; each one starts on the edge of the central ice sheet, has a number of intermediate ice elements, and then connects back to a different vertex on the edge of the central ice sheet. Degenerate features are similar, but they connect to the central ice sheet at only one point on its edge. Grounding information is generated randomly for the vertices in the central ice sheet and the complex features. Degenerate features are targetted to be removed, so we do not allow them to be grounded initially. As we varied the parameters we were testing, the numbers of vertices in meshes of equivalent resolution varied slightly; maximum values are reported in Table 3.1. Likewise, the orientation of the complex and degenerate features may vary slightly due to random generation.

For our scaling studies, we distributed mesh vertices equally among processors using "linear" distributions that assign $|V|/P$ vertices to each of P processors in the order of their

global vertex ID numbers. These distributions are likely not optimal with respect to locality of vertices. However, because our method uses the same parallel distribution as applications calling it, we did not investigate optimal partitioning strategies in our tests.

3.8 Results

To evaluate our method, we ran experiments on AMOS with the real and synthetic ice sheet meshes in Table 3.1. All tests correctly identified degenerated features in the meshes; thus, we focus on the performance of our method. We did weak and strong scaling studies, as well as specialized analyses to show how mesh features (number of grounded vertices, number and length of degenerated features, number and length of complex features) affect our algorithm.

3.8.1 Feature Detection Performance

3.8.1.1 Strong Scaling

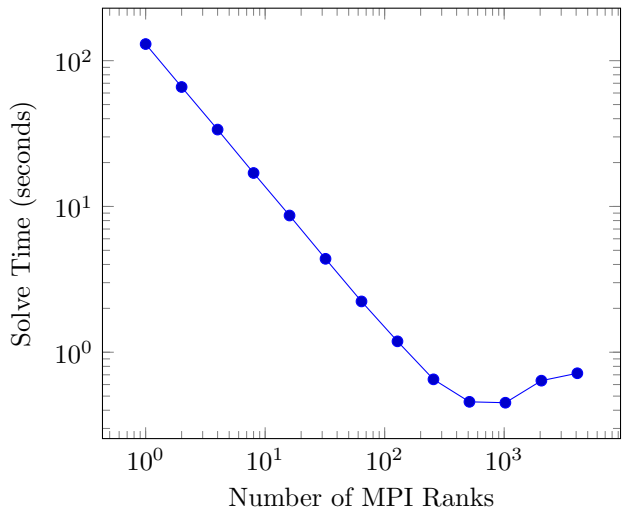

Figure 3.3: Strong scaling: runtime time using the largest real ice sheet mesh, *1km*.

The largest real mesh, *1km*, has over 13.4 million vertices. Strong scaling for this mesh is nearly perfect up to 512 MPI ranks, as shown in Figure 3.3. Beyond 512 ranks, communication increases, and computation takes longer due to a larger number of frontier switches in the propagation phase. The breakdown of computation and communication times is shown in Figure 3.4.

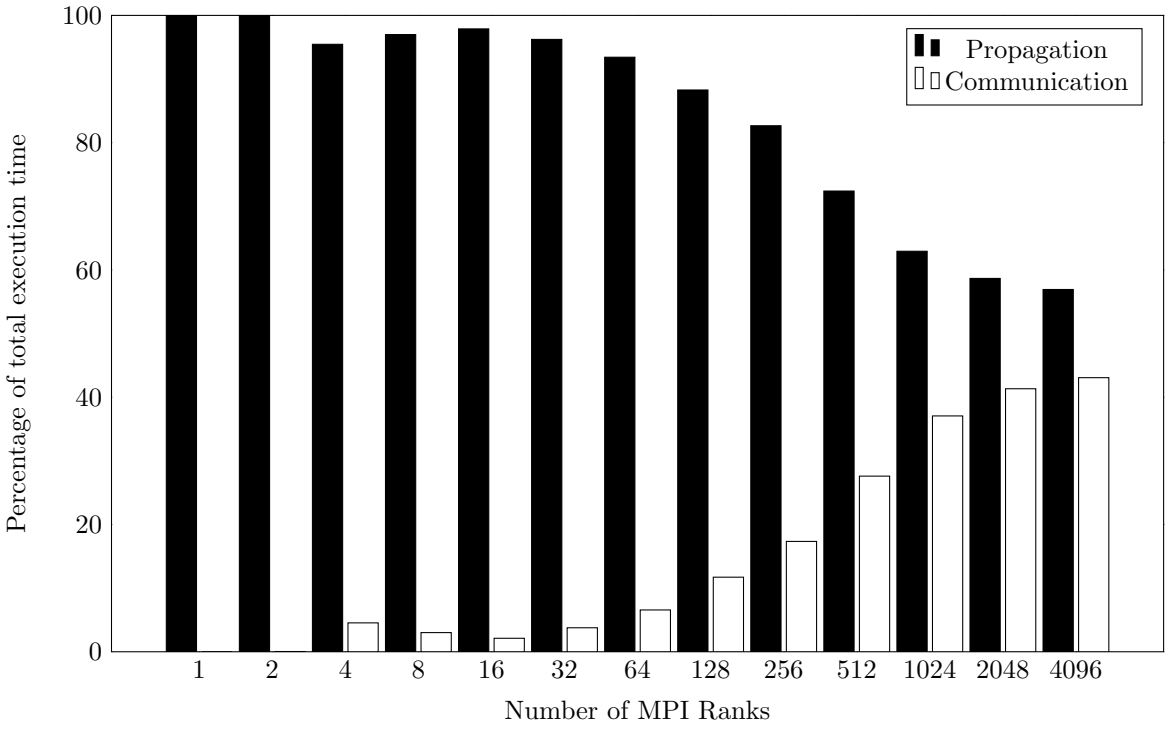

Figure 3.4: Strong scaling: percentage of runtime time for propagation and communication with real mesh *1km*.

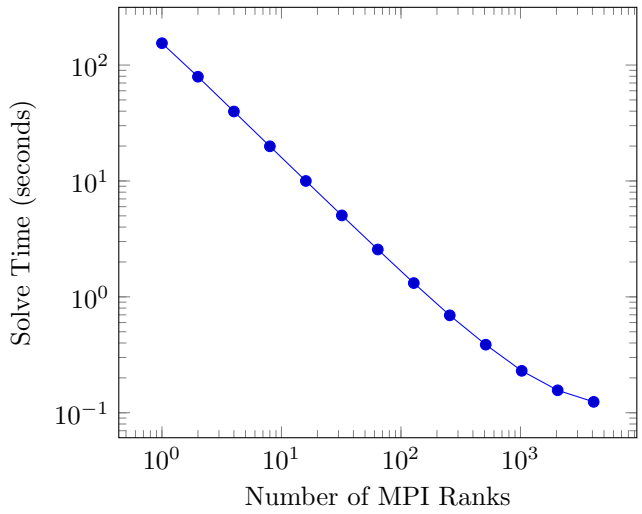

Figure 3.5: Strong scaling: runtime time for the largest synthetic ice sheet mesh, *syn-largest*.

The largest synthetic mesh, *syn-largest*, was slightly larger than our largest real case at 16 million elements. The results in Figure 3.5 show good strong scaling as well. There is little reduction in speedup as the number of MPI ranks approaches 4096; the extra vertices may aid the scaling, or our mesh generator may not perfectly replicate the features of the

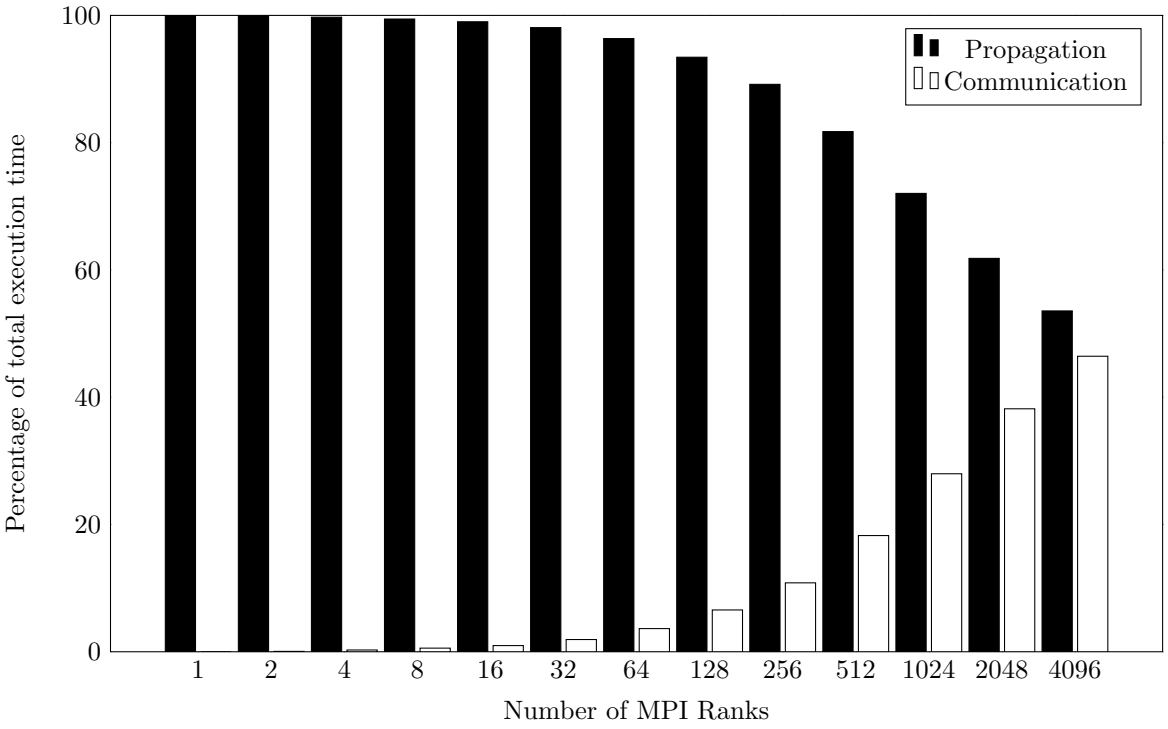

Figure 3.6: Strong scaling: percentage of runtime for propagation and communication with synthetic mesh *syn-largest*.

real ice sheet mesh. Figure 3.6 shows how much time our algorithm spends propagating, and how much time our algorithm spends communicating while solving a synthetic test case. The time spent communicating remains nearly constant as the number of processors increases, and the computation time reduces nearly by half when we double the number of processors.

3.8.1.2 Weak Scaling

The availability of real ice sheet meshes with varying resolutions enables effective weak-scaling experimentation. As mesh resolution is halved in each dimension, the number of elements increases by roughly four (see Table 3.1, top). Our weak-scaling tests on the real meshes *16km, 8km, 4km, 2km and 1km* assigned roughly 52K vertices per MPI rank on AMOS. Figure 3.7 shows that our algorithm's weak scaling is good on real data.

Our weak-scaling tests on synthetic meshes assigned roughly 3.9K vertices per MPI rank. Figure 3.8 shows our algorithm has good weak scaling on the synthetic data, even with a small workload.

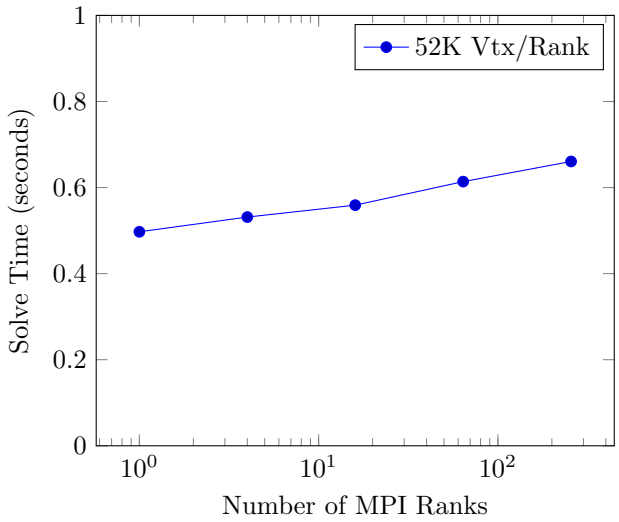

Figure 3.7: Weak scaling: runtime time for real meshes *16km, 8km, 4km, 2km, and 1km*.

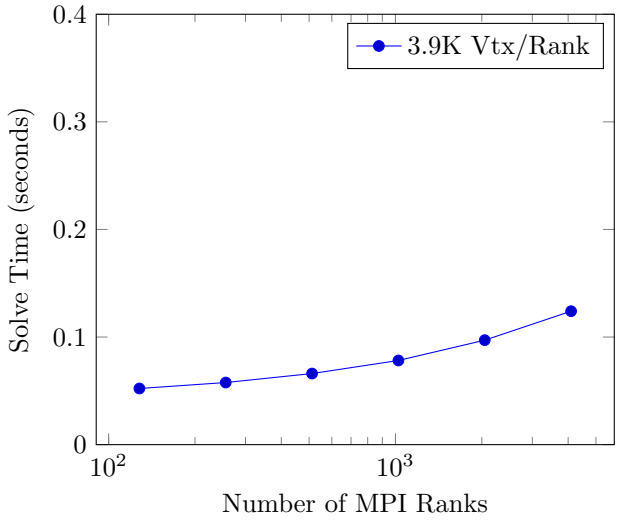

Figure 3.8: Weak scaling: runtime time for synthetic meshes.

3.8.1.3 Scaling vs. Mesh Complexity

The sizes and numbers of mesh features and grounded vertices can impact the performance of our algorithm. To study this impact, we modified the synthetic 2km mesh to vary its characteristics.

First, in the *ground(2km)* mesh, we varied the number of vertices in the mesh that were initially grounded, from the extreme case of only one initially grounded vertex to the case typical in real ice sheet meshes where 89% of vertices are initially grounded. Figure 3.9 shows that as the number of initially grounded vertices increases, the runtime of our algo-

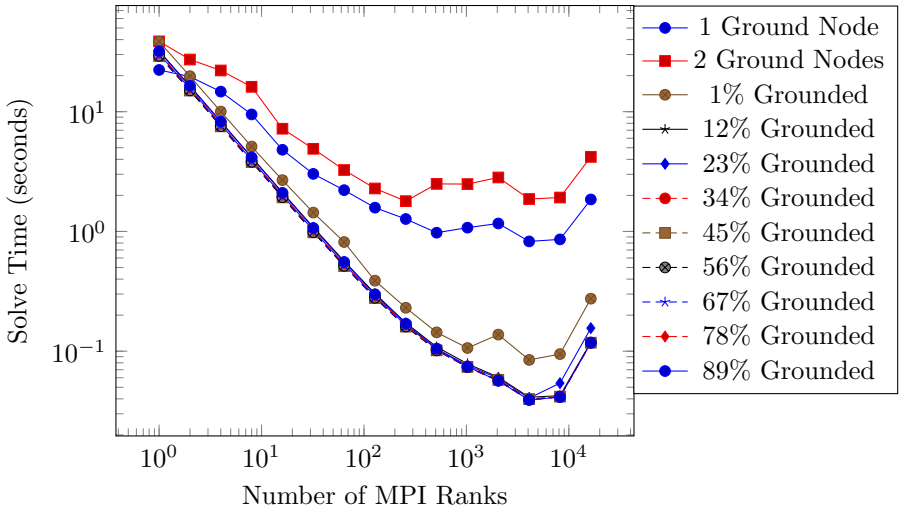

Figure 3.9: Scaling with different numbers of initial grounded vertices with synthetic mesh *grounded(2km)*.

rithm decreases, as fewer vertices' grounding state needs to be determined and grounding information is available across more processors initially. In the extreme cases with very few initially grounded vertices, strong scaling is poor because few processor have grounding information initially; most grounding information needs to propagate via communication. However, with 12% or more initially grounded vertices, all tests exhibited good strong scaling similar to the realistic 89% grounding case.

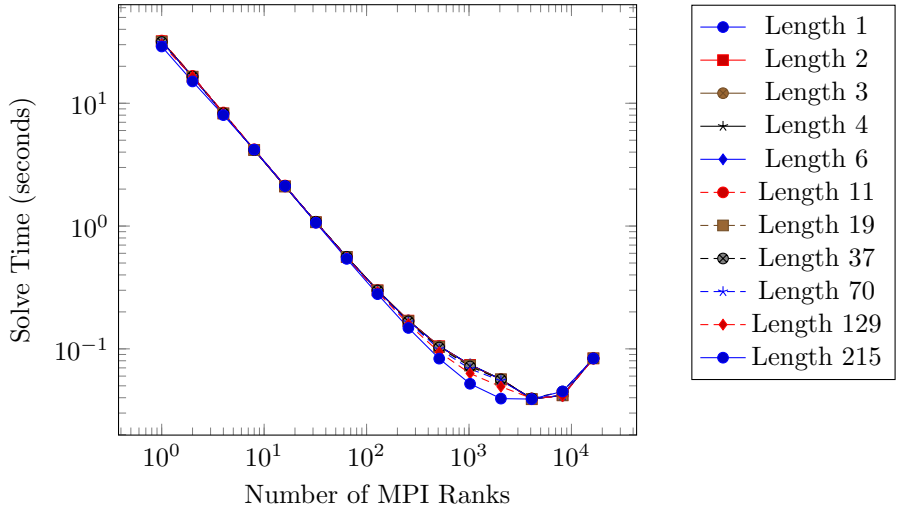

Figure 3.10: Scaling with different lengths of complex features using synthetic mesh *longcomplex*.

Figure 3.10 shows our algorithm's behavior as the length of complex features increases.

Varying the length of complex features from one to 215 elements shows little effect on the algorithm's runtime. Since vertices in complex features may be initially grounded, propagation through complex features is equivalent to propagation in the central ice mass. We do not measure the typical lengths of complex features in real meshes, but we can use the number of potential articulation points to get an idea. The real *2km* mesh has 389 potential articulation points; the synthetic mesh has 373 potential articulation points with a complex feature length of two.

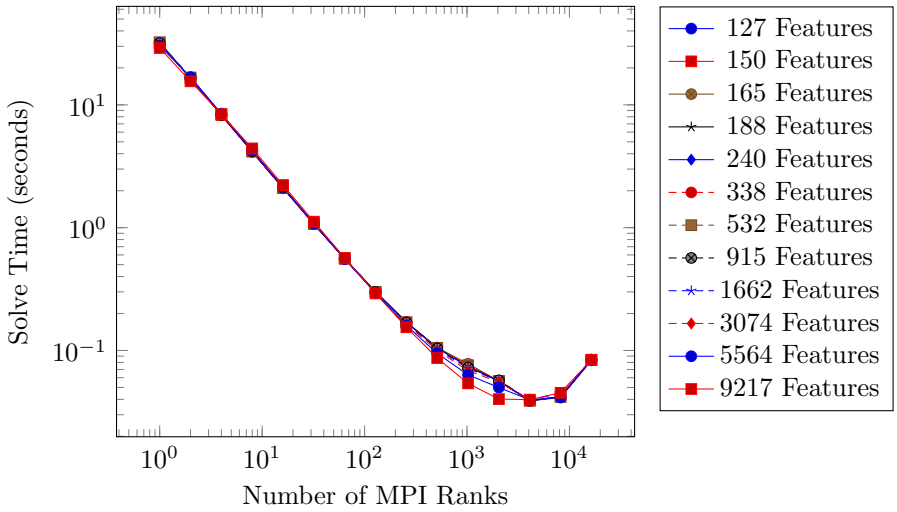

Figure 3.11: Scaling with different numbers of complex features with mesh *numcomplex*.

Next, with mesh *numcomplex*, we varied the number of complex features in a 2km mesh from 127 to 9217. The results, shown in Figure 3.11, are similar to Figure 3.10. Adding complex features does not adversely affect our algorithm's running time, as the algorithm's runtime does not depend on the number of biconnected components. Similarly to above, we can use the number of potential articulation points to get an idea of how many complex features are realistic. There are 366 potential articulation points in the synthetic mesh that has 188 complex features, which is as close as we get to the 389 potential articulation points of the *2km* mesh.

While the lengths and number of complex features do not impact the algorithm's runtime, the length of degenerate features can have a dramatic impact. Results varying the degenerate feature length from 1 to 3000 in a 2km mesh are shown in Figure 3.12. For small degenerate lengths typical of real ice sheet meshes, the length of degenerate features has modest impact. But as the length grows to 3000 and above, the algorithm loses scalability.

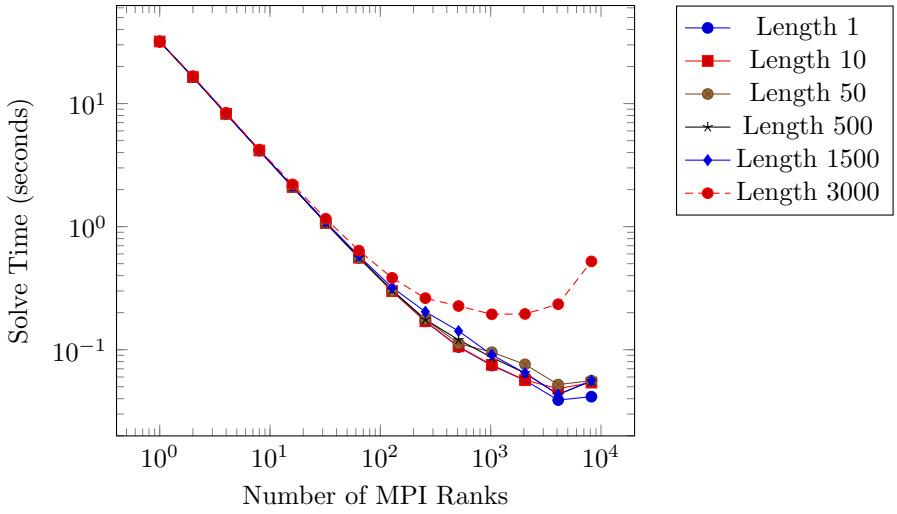

Figure 3.12: Scaling with different lengths of degenerate features with mesh *longdegen*.

Since degenerate features do not have grounded vertices, the algorithm makes slower progress on features split over processor boundaries.

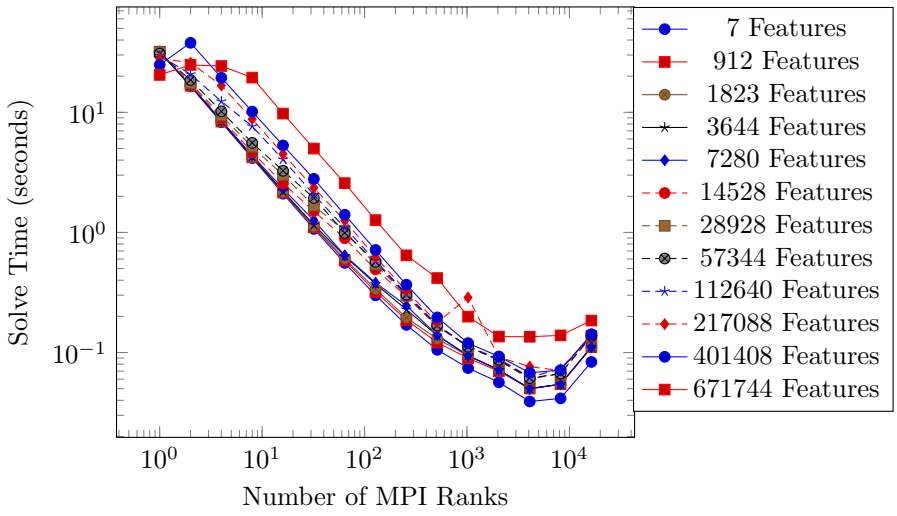

Figure 3.13: Scaling with different numbers of degenerate features with mesh *numdegen*.

The number of degenerate features has a less dramatic effect on runtime, but as Figure 3.13 shows, the more degenerate features there are, the more slowly our algorithm runs. While scalability is unaffected by the number of degenerate features, the overall runtime increases with the number of degenerate features. We observe that the number of degenerate features in ice sheets typically is small. In the *1km* mesh, only 65 vertices are removed, so

numbers of degenerate features over 100 are atypical of our real mesh data.

3.8.2 Application Results

We demonstrate the performance of our algorithm by incorporating it into the Zoltan2 graph algorithm package [54] of the Trilinos solver framework [61] and calling it from the Albany-LI component of the MPAS-Albany Land Ice (MALI) simulation code [49]. MALI is a high-fidelity, variable-resolution ice sheet model developed as part of the U.S. Department of Energy's Energy Exascale Earth System Model (E3SM). Albany-LI uses a conjugate gradient solver with a semicoarsening algebraic multigrid preconditioner [50]. The iterative solver is sensitive to degenerate features in ice sheet meshes. Thus, developers preprocessed their meshes with a degenerate-feature removal algorithm that was implemented in Matlab and run in serial before the simulation began; dynamic degerate-feature removal as the ice evolved during a simulation was not possible.

Table 3.2: Execution time for our distributed method in Albany-LI, compared to the serial Matlab preprocessor.

Mesh	Distributed (MPI Ranks)	Serial Matlab	Speedup
16km (real)	0.0176 s (6)	1.04 s	59×
8km (real)	0.0217 s (24)	5.65 s	260×
4km (real)	0.0414 s (96)	34.6 s	835×
2km (real)	0.0407 s (384)	245 s	6019×
1km (real)	0.0561 s (1536)	2630 s	46880×

With our new algorithm, serial preprocessing in Albany-LI is no longer needed. The mesh can be read into parallel processors, and degenerate features can be detected quickly at runtime. Our distributed implementation enables dynamic degenerate-feature removal to capture changes in the ice over the course of a simulation.

Table 3.2 shows the striking difference in runtime between our approach and the Matlab preprocessing approach. The preprocessing approach was run on a workstation with an Intel Xeon Gold 6146 CPU (3.20 GHz). Our distributed code was run in Albany-LI model on NERSC's Edison Cray XC30 supercomputer on varying numbers of MPI ranks. Albany-LI uses geometric partitioning (recursive inertial bisection [63]) to assign mesh elements to processors; our algorithm then used the distribution from Albany-LI. We see roughly 46,000× speedup in the highest resolution case. Moreover, our algorithm takes at most 0.4% of the time of a single simulation step. Thus, our algorithm is fast enough to be used dynamically

in the simulation as needed.

3.9 Conclusions

This chapter presented a novel distributed memory graph algorithm that is capable of detecting degenerate mesh features in a land-ice simulation. Our approach improves on the previous approach by orders-of-magnitude and is also more convenient to use, saving more time than just the computational speedup. In the next chapter, we will show the logical conclusion of generalizing this ice sheet algorithm to solve graph biconnectivity in distributed memory.

CHAPTER 4
LCA-BASED DISTRIBUTED BICONNECTIVITY

4.1 Chapter Introduction

This chapter adapts the intuition of the previous chapter to implement a biconnectivity algorithm capable of operating on social networks in distributed memory. There are no distributed memory biconnectivity algorithms in the literature currently, so this represents one of the first distributed memory graph biconnectivity algorithms. It demonstrates good strong scaling on fairly large social networks, though it does not manage to be competitive with state-of-the-art shared memory approaches. We also present the first distributed memory implementation for distributed LCA traversals, to our knowledge.

4.2 Ice Sheet Extension Introduction

The general graph biconnectivity decomposition problem seeks to identify all maximal biconnected subgraphs (subgraphs that can't be disconnected with the removal of a single vertex) as well as all cut vertices (or *articulation points*) and cut edges (or *bridges*) within some graph.

Biconnectivity is a well studied problem, with a work optimal depth-first search (DFS) algorithm originally proposed by Hopcroft and Tarjan [57]. As DFS lacks good parallelisms, Tarjan and Vishkin [53] later presented a parallel algorithm for finding biconnected components in a concurrent-read, concurrent-write PRAM model. This parallel algorithm reduces the problem of biconnectivity to the problem of connectivity in an auxiliary graph. The auxiliary graph can be constructed without using DFS and its connectivity can then be found efficiently in shared memory.

More recently, Slota and Madduri presented shared-memory parallel algorithms for biconnectivity by utilizing simple graph subroutines such as breadth-first search (BFS) and color propagation [58]. Most recently, LCA-BiCC was proposed by Chaitanya and Kothapalli [59]. The use a BFS spanning tree along with lowest common ancestor (LCA) vertices of all non-tree edges to identify all bridges and then perform a recursive algorithm on the

This chapter is to appear as: I. Bogle, and G. M. Slota, "Distributed algorithms for the graph biconnectivity and least common ancestor problems," In *2022 IEEE Int. Parallel and Distrib. Process. Symp. Workshops (IPDPSW)*, 2022.

components resulting from the removal of these bridges. The *lowest common ancestor* w of two vertices u, v in a rooted tree is the *lowest* (i.e., farthest from the root) vertex that has both u and v as descendants.

Our Contribution: In this work, we present the first parallel biconnectivity algorithm for a distributed graph structure. As with the algorithm of Chaitanya and Kothapalli [59], we require the computation of LCA vertices. We therefore also propose the first fully distributed algorithm for computing LCAs. We observe strong scaling with our algorithms up to 128 MPI ranks with speedups up to 14.8× across a selection of large test instances.

4.3 Distributed Biconnectivity

Our algorithm builds upon the *degenerate mesh features* algorithm of Bogle et al. [64]. That algorithm specifically considered ice sheet meshes and degenerate features (e.g., *hinges* of ice connected by a single mesh vertex to an ice sheet) that prevented climate simulation convergence. The Bogle et al. algorithm used the notion of *two paths* – any non-degenerate mesh vertex will have at least two paths to the known stable part of the mesh. We generalize this basic approach to the biconnectivity via Whitney's 1932 Theorem, which shows that for all u, v vertices in a 2-connected subgraph there exists two edge-disjoint u, v-paths.

Algorithm 12 Generalized BiCC Algorithm

1: **procedure** BCC-Overview($G = (V, E)$)
2: *parents, levels* ← Dist-BFS(G)
3: *potential_artpts* ← LCA-Heuristic(G,*parents*, *levels*)
4: BCC-LR(G, *levels*, *potential_artpts*)

Our overall approach is given in Algorithm 12. First, we do a distributed BFS using subroutines from the HPCGraph Framework [43] to get a rooted spanning tree of the global graph defined by *parents* and *levels*. Then, an LCA-based heuristic uses the BFS tree to identify a set of potential articulation points (*potential_artpts*). This set is guaranteed to contain all actual articulation points [59]. Finally, our label propagation algorithm uses the set of potential articulation points and the BFS tree to uniquely label each biconnected component in the global graph.

4.3.1 LCA Heuristic

The LCA Heuristic algorithm starts *LCA traversals* for the endpoint pair of vertices of each non-tree edge of the BFS tree. Without explicit list construction, non-tree edges are easily identified using *parent* information. An LCA traversal replaces the lower-level (farthest from root) vertex with its parent, until both vertices in the traversal reach the same ancestor. In a distributed-memory context, we need to communicate traversal progress across processor boundaries. Additionally, we keep track of any edge that we have traversed, as the endpoints of unvisited edges are within our potential articulation point set.

In order to facilitate the distributed memory LCA traversals, we use three entries for each vertex in each queue *package* to track each LCA traversal: the global ID of the vertices, the global ID of the parents of the vertices, and the processes that owns the vertices. If the lower level vertex is owned by a remote process, all six values are sent to it in order for it to process the package. We process a single step of each traversal before communicating our queues in an all-to-all fashion. Note that after we flag a vertex as a potential articulation point, we also check to see if the vertex is owned on another process, ensuring that all flags are consistent across processes. After this procedure, the endpoints of any unvisited edge plus all LCA vertices make up the set of potential articulation points.

While there is likely room for further optimization in our procedure, this stage is a small portion of the total execution time. In addition, we claim that this is the first distributed LCA algorithm for labeling all non-tree vertex pairs. Distributed algorithms for LCA vertex-pair queries exist, but these (e.g., [65]) generally require extensive pre-processing in shared memory to construct some labeling that enables distributed LCA queries for some u, v vertex pair.

4.3.2 Label Propagation and Reduction

After we have the set of potential articulation points and the BFS tree, we can proceed with our label propagation algorithm. Our distributed memory approach is given in Algorithm 13. Our label propagation procedure uses two types of label, denoted as *LCA* and *Low*. LCA labels contain only global IDs of LCA vertices in the global graph. Low labels hold the lowest-level vertex that has a particular LCA label. As both of these labels can propagate across processor boundaries, they require special considerations in our communication function.

Algorithm 13 BiCC Label Propagation

```
1: procedure BCC-LR(G, levels, potential_artpts)
2:     for all v ∈ V do
3:         LCA_labels[v] ← null
4:         low_labels[v] ← v
5:     queue ← {all IDs of local potential_artpts}
6:     verts_to_send ← ∅
7:     labels_to_send ← ∅
8:     LCA_procs_to_send ← ∅
9:     while queue is not empty on some process do
10:        irreducible_queue ← ∅
11:        while queue is not empty on this process do
12:            curr_vtx ← queue.pop()
13:            if LCA_labels[curr_vtx].size() > 1 then
14:                Reduce-Labels(...)
15:                if LCA_labels[curr_vtx].size() = 1 then
16:                    for all neighbor n of curr_vtx do
17:                        Push-Low-Labels(...)
18:                if LCA_labels[curr_vtx] or
19:                        low_labels[curr_vtx] changed then
20:                    verts_to_send.push(curr_vtx)
21:                    labels_to_send.push(LCA_labels[curr_vtx])
22:                    labels_to_send.push(low_labels[curr_vtx])
23:            for all neighbor n of curr_vtx do
24:                Pass-Labels(curr_vtx, n, ...)
25:        Label-Comm(...)
26:        queue ← irreducible_queue
27:        irreducible_queue ← ∅
```

Initially, every vertex in the local graph starts with no LCA vertex, and its own ID as a Low label. Each local potential articulation point propagates its own ID as an LCA label to its lower-level neighbors. Propagation of the LCA vertices ensures that they only move down the tree, and when a vertex gets two or more LCA labels we *reduce* the labels. Label reduction involves applying a variation of LCA traversal to the multiple LCA labels. Algorithm 14 shows this procedure. We take the lowest-level LCA label from the set of multiple labels, and attempt to replace it, if possible. We abort this traversal if the lowest-level label has an *irreducible label*. A label set is irreducible for the current propagation sweep if it contains multiple labels and at least one is currently *irreducible*, meaning that the label traversal eventually leads to a remote LCA for which this process has not received

any LCA label information.

Algorithm 14 Label Reduction Procedure

```
1: procedure REDUCE-LABELS(
   G, curr_vtx, levels, LCA_labels, queue, irreducible_queue)
2:     if irreducible_queue contains curr then
3:         return
4:     while LCA_labels[curr_vtx].size() ≠ 1 do
5:         low = lowest level ID in LCA_labels[curr_vtx]
6:         if LCA_labels[low].size() = 1 then
7:             LCA_labels[curr_vtx].erase(low)
8:             LCA_labels[curr_vtx].insert(LCA_labels[low][0])
9:         else
10:            if irreducible_queue contains low then
11:                irreducible_queue.push(curr_vtx)
12:            else if LCA_labels[low].size() = 0 then
13:                irreducible_queue.push(curr_vtx)
14:            else
15:                queue.push(curr_vtx)
```

Algorithm 14 shows the procedure for reducing labels. Note that LCA labels are stored in a set, so once traversals converge on a common LCA label, the size of the LCA label set for that vertex will decrease.

The procedure that propagates Low labels is given in Algorithm 15. We pass Low labels between vertices that have the same LCA label, such that each vertex retains the Low label representing the vertex at the lowest level in the BFS tree with the lowest vertex ID. We also prevent Low labels from propagating up through potential articulation vertices.

Once the label propagation and reduction completes successfully, the sole LCA label for a vertex will point to an articulation vertex. If we perform reductions only when a vertex has two different labels, it has two vertex-disjoint paths to the final vertex represented by the LCA label. However, due to the parallel and distributed nature of our implementation, we need to restrict this baseline label propagation in order to prevent LCA labelings from forming cycles. These propagation rules are described in Algorithm 16.

Importantly, both potential articulation points and regular vertices propagate a given LCA label only to vertices lower in the tree than the label. This restriction prevents cyclical behavior in propagation of LCA labels. Additionally, we also start collecting information needed to send to remote processes during the propagation. A standard communication

Algorithm 15 Low Label Propagation

```
1: procedure PUSH-LOW-LABELS(
   curr_vtx, n, potential_artpts, levels, LCA_labels, Low)
2:     if potential_artpts[curr_vtx] = true then
3:         if LCA_labels[curr_vtx] = LCA_labels[n] and (levels[n] ≤ levels[curr_vtx] or
   LCA_labels[n][0] ≠ curr_vtx) then
4:             curr_low_lvl ← level[Low[curr_vtx]]
5:             n_low_lvl ← level[Low[n]]
6:             if (curr_low_lvl > n_low_lvl) or (curr_low_lvl = n_low_lvl and Low[curr_vtx] >
   Low[n]) then
7:                 Low[n] = Low[curr_vtx]
8:     else
9:         if LCA_labels[curr_vtx] = LCA_labels[n] then
10:            curr_low_lvl ← level[Low[curr_vtx]]
11:            n_low_lvl ← level[Low[n]]
12:            if curr_low_lvl > n_low_lvl or (curr_low_lvl = n_low_lvl and Low[curr_vtx] >
   Low[n]) then
13:                Low[n] = Low[curr_vtx]
```

Algorithm 16 Label Propagation Rules

```
1: procedure PASS-LABELS(G, curr_vtx, n, LCA_labels, low, levels, potential_artpts, queue,
   verts_to_send, labels_to_send, LCA_procs_to_send)
2:     if potential_artpts[curr_vtx] = true then
3:         if (levels[n] ≤ levels[curr_vtx] and
4:               curr_vtx was recently reduced) or
5:               (levels[n] = levels[curr_vtx] and
6:                     curr has a single label) then
7:             pass LCA labels that curr_vtx has to n
8:         else if levels[n] > levels[curr_vtx]
9:             pass ID of curr_vtx to n as an LCA label
10:    else
11:        if level[n] > level[curr_vtx] and
12:              curr_vtx has a reduced label then
13:            pass LCA labels that curr_vtx has to n
14:    Push-Low-Labels(curr_vtx, n, ...)
15:    if n's labels changed and n is ghosted then
16:        verts_to_send ← n
17:        labels_to_send ← LCA_labels[n], low[n]
18:        LCA_procs_to_send ← ranks with copy of n
19:        queue ← n
```

pattern of communicating owned vertex values to ghosted copies is not sufficient in this algorithm. We also need to send the set of LCA labels that the owned vertex has, so that

remote processes do not need to wait to be able to compute local label reductions.

4.4 Biconnectivity Correctness Proofs

In order to prove our biconnectivity algorithm correct, first we must show that when our label propagation and reduction algorithm completes, each vertex v has an LCA label l, representing the vertex with the highest level in the BFS tree (closest to the root) that is in the same biconnected component as v. This high-level vertex is either an articulation vertex and/or the root. We will use this fact to prove that two vertices have both the same LCA and Low labels if and only if they are in the same biconnected component.

Proposition 2: Under our label propagation and reduction rules, a vertex v will have an LCA label l that contains the ID of the highest level LCA vertex in the BFS tree that is in the same biconnected component as v.

Proof: Assume v has final LCA label l. We ask if there can exist a higher LCA vertex h in the same biconnected component as both v and l. Per Menger's Theorem (1927), h has at least two vertex disjoint paths to v. There exists at most one h, v-path that goes through l. We specifically consider the second h, v-path and two possible cases: either it contains LCA vertices or it does not.

Case 1 – The second h, v-path does not contain LCA vertices: By our propagation rules, which allow LCA labels to propagate freely down the BFS levels through non-LCA vertices, v would receive h as an LCA label. When v also gets l as an LCA label, the label reduction would leave only h, as h is the higher level LCA vertex. This is a contradiction, as v's final label is l, so v cannot have a path to h that does not have LCA vertices in it.

Case 2 – The second path between v and h does contain LCA vertices: By our propagation rules, v would receive a label from the lowest level LCA vertex in this path. This would result in v having two labels at some point (as v will at the very least also receive label l), and a reduction would traverse all of the LCA vertices in the h, v-path to h, eventually leaving only h in the label set. This is again a contradiction, so v cannot have a path to h that does have LCA vertices in it.

Thus, there cannot be two vertex disjoint paths between v and our hypothetical h, so they cannot be in the same biconnected component. Therefore, l is the highest level LCA

vertex in the same biconnected component as v and the LCA labeling is correct and complete for all vertices in the biconnected component.

Next, we will show that two vertices have the same labels (LCA *and* Low) if and only if they are in the same biconnected component.

Proposition 3: At the end of our label propagation and reduction, two vertices u and v will have the same labels $\iff$ they are in the same biconnected component.

Proof: Suppose that vertex u has LCA label l_u and Low label low_u, and vertex v has LCA label l_v and Low label low_v. We will first prove that if u and v have the same labels, these vertices must be in the same biconnected component.

Assume by contradiction that u and v have the same labels but are in different biconnected components. By our propagation rules, where LCA labels traverse down the tree, it is only possible for $l_u = l_v$ if u and v are in biconnected components that are rooted on the same articulation vertex a. However, by our propagation rules, it is impossible for a Low label to propagate over an articulation point, so $low_u = low_v$ implies the existence of a u, v-path that does not include a. This is a contradiction, because any path between some u and some v in separate biconnected components must pass through an articulation vertex.

Next, we will prove that if u and v are in the same biconnected component, then they must have the same labels. Again by contradiction, assume that u and v are in the same biconnected component, but do not have the same labels. We observe two possible cases.

Case 1 – $l_v \neq l_u$: Per Proposition 2, all vertices in the same biconnected component will have the same LCA label. If u and v are in the same biconnected component, it is a direct contradiction that they could have different LCA labels. Thus, this case is not possible.

Case 2 – $low_u \neq low_v$: Assume $l_u = l_v$ and $low_u \neq low_v$. Per our propagation rules, Low labels are propagated between vertices that have the same LCA label. Per Proposition 2 and *Case 1* above, we can assume all vertices in the biconnected component containing u and v will have the same LCA labels. $low_u \neq low_v$ implies that there is a lower level vertex w that, without loss of generality, u is able to reach that is unreachable from v. By the basic definition of biconnectivity, there must exist some w, v-path within this biconnected component. Thus, this case is also not possible.

Therefore, it is not possible for two vertices in the same biconnected component to

have different labels. Thus, we have proven that, under our propagation and reduction rules, two vertices will have the same labels iff they are in the same biconnected component.

4.4.1 Biconnectivity Complexity Discussion

Our biconnectivity algorithm has three phases. The first phase is a simple breadth-first search. This has known work complexity of $O(|E| + |V|)$ and optimal parallel time complexity of $O(\log |V|)$, though our simpler level-frontier implementation has time $O(d)$, where d is the graph diameter.

Our second phase is the LCA initialization. The maximum length of a traversal to find an LCA is d and the maximum number of LCA traversals is bounded above by $\sim |E| - |V|$. This would give a work complexity of $O(d|E|)$ and parallel time complexity of $O(d)$. However, in practice, the work bound is quite loose, as the average traversal length for an LCA is usually much smaller than the graph diameter for all observed real networks.

The final phase of our algorithm is the label propagation phase. This phase requires the labeling of all vertices with their LCA labels and the subsequent labeling of all vertices with their final Low labels. The maximum distance a vertex v is from their highest LCA vertex l is d and the maximum number of reductions is equivalent to the number of vertices along the path from v to l. Hence, to label all vertices we have the (loose) work complexity of $O(d^2|V|)$ and parallel time complexity of $O(d^2)$. For Low label labeling, we will propagate at most d steps for all $|V|$ vertices, giving us a work complexity of $O(d|V|)$ and parallel time complexity of $O(d)$. But, one should note that Low label propagation will occur concurrently with LCA label propagation.

Overall, we claim a work complexity of $O(d|E| + d^2|V|)$ and time complexity of $O(d^2)$. However, these bounds are somewhat deceiving, as we've experimentally observed an inverse relation between average LCA traversal distances and graph diameter. This can be explained through the fact that many high-diameter graphs (road networks, meshes, etc.) are also quite structured. Our experimental results readily corroborate this.

4.5 Results

4.5.1 Experimental Setup

Experiments for our distributed biconnectivity algorithm were performed on Rensselaer Polytechnic Institute's DRP machine. The DRP cluster has 64 nodes, each with two eight-core 2.6 GHz Intel Xeon E5-2650 processors and 256GB of system memory, connected via 56 Gb FDR Infiniband. Our experiments allocate a maximum of 16 ranks per node, with our largest experiments using 128 MPI ranks. We use edge-block partitioning, where we assign to each rank approximately $\frac{|E|}{R}$ edges, where R is the number of ranks.

Table 4.1: Graphs used in our experiments for biconnectivity.

Graph	Type	# Verts	# Edges	d_{max}	BiCCs
com-LiveJournal	Social	3.9M	69.3M	14.8K	594k
wiki-Talk	Social	2.3M	5M	100K	34k
roadNet-CA	Road	1.9M	5.5M	12	327k
roadNet-PA	Road	1.0M	3M	9	194k
web-Google	Web	0.9M	5.1M	6K	60k

Table 4.1 shows the details for the graphs we used to test our distributed biconnectivity algorithm. We selected social, web, and road networks from the SNAP database[3], as these are representative of the social networks and other graphs generally considering in connectivity analysis applications.

As it is typical for evaluation of k-connectivity algorithms [58], [59], [66] to run on the largest $(k-1)$-connected component of an input, we run our algorithm on the largest connected component of each graph to allow for relative comparison (trivially, any k-connectivity algorithm could be used on graphs with multiple $(k-1)$-connected components by first calling a $(k-1)$-connectivity algorithm and running on the extracted components in serial or parallel). It is important to note that in comparison to the other distributed biconnectivity algorithms studied in chapter 4, this algorithm did not prove to be competitive, and also does not have a strong guarantee of theoretical efficiency. For these reasons we opt not to explore further optimizations to this algorithm in this thesis.

[3] `http://snap.stanford.edu/snap/`

4.5.2 Biconnectivity Algorithm Performance

Our algorithm has three main parts: the BFS, the LCA heuristic, and the label propagation algorithm. Figure 4.1 shows the strong scaling of each part of our distributed biconnectivity algorithm on each of our test graphs. Due to the irregularity of the degree distribution of the wiki-Talk graph, 128 ranks failed due to memory errors. Explicit partitioning to balance vertices and edges would address this issue, but optimizing partitioning approaches is beyond the scope of this work. The overall approach is able to scale on the other graphs up to 128 ranks, and scales on wiki-Talk upto 64 ranks. In terms of overall speedup, we see the highest speedups of 12.8x and 14.8x on roadNet-CA and roadNet-PA, respectively. We see a 3.6x speedup on wiki-Talk, 3.1x speedup on web-Google, and a 2.8x speedup on com-LiveJournal.

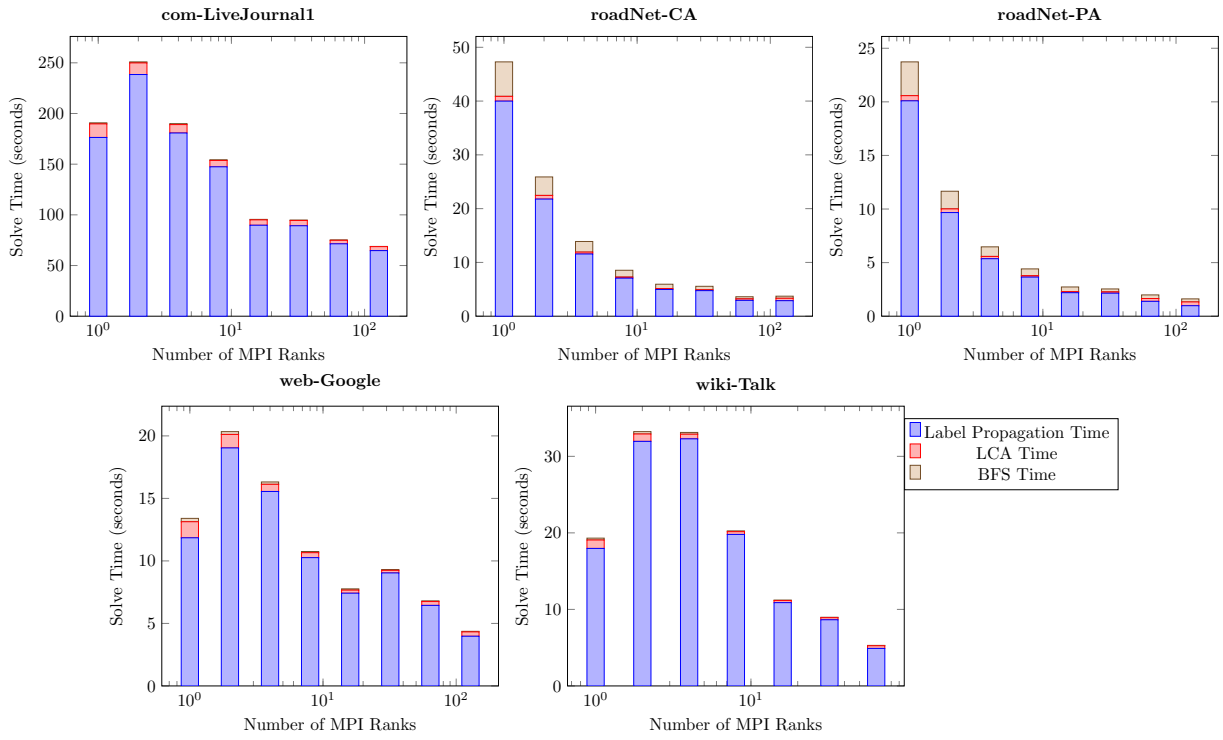

Figure 4.1: Strong scaling of our BiCC algorithm from 1 to 128 ranks.

From figure 4.1, we see that label propagation is currently the most time-consuming part of this approach. Though it takes the most time, the label propagation scaling does not tend to taper off in most of the graphs that we consider. Label propagation spends a majority of the time doing computation, with cumulative communication times never exceeding half a second. Our communication pattern is aimed to reduce both the amount of commu-

nication and maximize the number of label reductions which can make progress. However, the traversals that determine what to send can be expensive during initial processing of the algorithm. Additionally, at higher numbers of ranks, a greater number of reductions will often get serialized via our propagation rules, limiting scalability to a degree. In practice this algorithm sees poor scalability on graphs with high diameters. Interestingly, that means that this biconnectivity algorithm sees poor weak scaling on meshes such as ice sheet meshes, while the original algorithm that inspired this approach sees very good weak scaling across the different ice sheet meshes.

The label reductions used in the label propagation algorithm are nearly identical to the traversals done in the LCA algorithm. An optimization we will pursue in future work is converting the current communication pattern to one that is more like the LCA Heuristic. This should greatly accelerate label propagation, by essentially trading a small increase in communication overhead with a large improvement in load balance.

4.6 Conclusions

In this chapter we showed the logical conclusion of generalizing our ice sheet algorithm to solve graph biconnectivity in distributed memory. We show that our distributed LCA traversals see good strong scaling on our selected graphs. We show that the algorithm acheives good strong scaling on a selection of social networks, and discussed avenues for further optimizations. However, we found this implementation is not competitive with shared memory biconnectivity algorithms. As such, the next chapter explores the distributed memory implementation of two shared memory graph algorithms: one theoretically optimal algorithm, and an algorithm with state-of-the-art parallel performance.

CHAPTER 5
DISTRIBUTED BICONNECTIVITY

5.1 Chapter Introduction

The previous chapter explored the generalization of an algorithm to solve graph biconnectivity in distributed memory. In this chapter we use the problem of Graph Biconnectivity as a case study of the difficulties in porting shared memory graph algorithms to distributed memory. There are many shared memory parallel graph biconnectivity algorithms, but very few efficient distributed memory implementations. We find it is the case that proven theoretical efficency in shared memory algorithms does not necessarily translate to practical performance in a distributed memory implementation. We also explore efficient distributed memory implementations for popular subroutines used in shared memory biconnectivity algorithms, in order to aid future work in this area.

5.2 Graph Biconnectivity Introduction

The large scale of datasets representing real-world graphs and meshes for scientific computations often necessitates the use of distributed memory processing. Likewise, efficiently solving basic graph problems such as determining graph biconnectivity remains important to multiple applications, including climate modeling [67], network resilience [68], social network analysis [69], and other mesh-based scientific computing applications [10]. However, many graph algorithms, including biconnectivity, are not widely studied in distributed memory, and many do not have efficient distributed implementations described in the existing literature.

Graph biconnectivity is often described as identifying *cut vertices* in a connected graph – vertices that, when removed, will disconnect the graph. This can be considered a specific instance of a more general connectivity problem, k-vertex connectivity, where k corresponds to the minimum number of vertices that need to be removed from a graph in order to disconnect it. We note that there exists time-efficient shared memory algorithms for biconnectivity (i.e., 1-vertex connectivity), but as the value of k increases, algorithm complexity correspondingly does as well. Algorithms for general k-vertex connectivity often use flow-based approaches [70], while more specific algorithms for $k \leq 3$ connectivity use a wide variety of

subroutines, such as the approach proposed by Ramachandran [71].

We restrict this current study to biconnectivity, as current algorithms that solve the problem have constituent subroutines that we observed to be re-used across other k-connectivity algorithms. One such algorithm is the $O(\log n)$ parallel time shared memory algorithm of Tarjan and Vishkin [53], which itself includes interesting subroutines that have also not been widely studied in distributed memory. For example, we consider list ranking (specifically, preordering) of a spanning tree, subtree size computation, and other similar subroutines. Other representative biconnectivity algorithms include those of Slota and Madduri [8] and Chaitanya and Kothapalli [13], which themselves include subroutines for color propagation and least common ancestor (LCA) traversals.

Generally, studying how to efficiently implement such algorithms when moving from shared to distributed memory is a key motivator for this current work. We hope to establish baseline approaches for more efficient distributed memory implementations of graph k-connectivity algorithms and their subroutines in the future.

5.2.1 Our Contributions

We study the adaptation of two shared-memory biconnectivity algorithms towards a distributed memory setting. Specifically, we consider implementing one of the Slota-Madduri algorithms noted above along with the optimized version of the Tarjan-Vishkin algorithm as presented by Cong and Bader [12]. Cong and Bader utilize the edge filtering technique of Cheriyan and Thurimella [72] to greatly reduce the necessary number of edges to determine cut vertices in some input graph. Our specific contributions are noted below:

1. We implement the Slota-Madduri Color-BiCC algorithm, the Cong-Bader TV-Filter algorithm, and the Cheriyan-Thurimella edge filtering algorithm in distributed memory. These are the **first distributed-memory implementations** of these algorithms presented in the literature.

2. We experimentally evaluate the tradeoffs between theoretical efficiency and implementation simplicity for all of these algorithms and their constituent subroutines. Overall, we find that **simplicity is key** for performance for implementations in distributed memory.

3. As an extension of this study, we also develop novel distributed algorithms for spanning tree preordering and subtree size computation. Our relatively simple approach for preordering provides up to a 22× speedup relative to the more "theoretically efficient" algorithm on 16 MPI ranks with 8-way thread parallelism per rank.

4. Overall, our distributed implementation of the Color-BiCC algorithm demonstrates average speedups of 11× and 7.3× relative to the optimal serial Hopcroft-Tarjan algorithm [73] and the shared memory Color-BiCC algorithm, respectively, on 64 MPI ranks.

5.3 Background

Here we will discuss general distributed graph processing, explicitly define the graph biconnectivity problem, and present several prior and relevant works.

5.3.1 Parallel Computation Models

We consider the Parallel Random Access Memory (PRAM) and Bulk Synchronous Parallel (BSP) models of parallel computation. For our distributed algorithms, we also assume (and implement) task+thread multi-level parallelism, essentially combining a global BSP model with per-rank PRAM computations within each superstep. Note that for the majority of the subroutines and algorithms discussed below, the parallel time complexity in PRAM and BSP+PRAM ends up being equivalent, as each superstep can theoretically be done in constant time when assuming a sufficient number of PRAM processors per rank. The memory-update operations and subsequent dependencies expressed in the PRAM model are generally equivalent to the messages passed under the BSP model within each superstep, though obviously this would be assuming a costless network.

In this paper, we will be discussing what we refer to as "time-efficient" parallel algorithms. In this context, this refers to parallel algorithms that require $O(\log n)$ parallel time at most under the above models.

5.3.2 Distributed Graph Processing

We consider a graph $G = (V, E)$, where V is the vertex set and E is the edge set. In our discussions, we will use $n = |V|$ and $m = |E|$. For our distributed implemen-

tation, we explicitly consider a *1-dimensional* distributed graph processing model, where our input graph $G = (V, E)$ is broken into some number of vertex-disjoint subgraphs $G = (\{V_1, V_2, \ldots V_p\}, \{E_1, E_2, \ldots E_p\})$ that are distributed to p computational ranks. Each of these subgraphs contain a set of *owned* vertices and all of their incident edges. Consequently, these subgraphs also include all non-owned vertices (i.e., *ghosted vertices*) within the 1-hop neighborhood of owned vertices. Each rank holds vertex state information for both owned and ghosted vertices.

For all of our implementations, we enforce a $O(\frac{n+m}{p})$ memory bound. In other words, no single rank (assuming $p > 1$) can view *all* vertex or edge state information. The assumption being, under this distributed model, we can theoretically strong scale any input graph or process any input of arbitrarily large scale given a sufficient number of compute nodes. Note however, in practice with this type of *1-dimensional* distribution, each V_i, V_j and E_i, E_j are not going to be perfectly balanced, so there is a practical limit.

5.3.3 Biconnectivity Definitions

We consider the general problems of k-connectivity and the specific problem of biconnectivity. A graph G is said to be k-connected if the minimum size of a vertex separator (a subset of vertices $S \subseteq V$ such that $G - S$ is disconnected) is equal to k; i.e., $|S| = k$. Most computational formulations for k-connectivity seek to identify all such minimum separators.

Many graph biconnectivity algorithms are described as performing a *biconnectivity decomposition*, which is an edge-disjoint labeling of some connected G such that all maximal 2-connected subgraphs have edges with the same label. Biconnectivity algorithms often in practice simply seek to identify all edges e such that $G - e$ is disconnected (cut edges or *bridges*) and all vertices v such that $G - v$ is disconnected (cut vertices or *articulation vertices*). With articulation vertices and bridges denoted, producing the desired edge labeling is relatively trivial.

5.3.4 Prior Work

Due to their wide applicability across a broad number of fields, biconnectivity algorithms have been studied for many decades. Hopcroft and Tarjan provided the first optimal serial algorithm [73], which is based on depth-first search and requires a single traversal with work $O(n + m)$.

Later, Tarjan and Vishkin [53] described the first time-efficient parallel algorithm for biconnectivity. Their algorithm runs in $O(\log n)$ time on $O(n + m)$ processors. They use common subroutines found in many graph algorithms, including spanning tree construction, preorder labeling, subtree size enumeration, and connectivity checking. Cong and Bader [12] incorporated Cheriyan and Thurimella [72] edge filtering as a preprocessing step to provide an optimized implementation of the Tarjan-Vishkin algorithm, *TV-Filter*. Cong and Bader showed their TV-Filter implementation achieved a 2× speedup over an optimized parallel implementation of baseline Tarjan-Vishkin.

Slota and Madduri [8] studied simple shared memory approaches for biconnectivity by developing algorithms using breadth-first search (BFS) and color propagation as primary subroutines. These subroutines have been extensively studied in practice. Among many other works in the literature, Buluç et al. [74] examined BFS implementations for massive graphs in distributed memory, while Stergiou et al. [75] considered time-efficient distributed color propagation for connectivity. While time-efficient algorithms exist for these subroutines, many efficient implementations trade time-optimality for practical implementations and low work overheads. In this paper, we specifically consider Slota and Madduri's color propagation biconnectivity algorithm, *Color-BiCC*.

More recently, Chaitanya et al. [13] further developed the Slota-Madduri biconnectivity algorithms by using least common ancestor (LCA)traversals in shared memory and a more complex parallelization scheme. Most recently, Bogle et al. [67] provided distributed memory implementations for least common ancestor and general biconnectivity, as well as an efficient algorithm for a similar but more constrained version of biconnectivity for ice sheet simulations. However, their algorithm is constrained by graph diameter, so biconnectivity approaches based on this work tend to scale poorly with respect to input diameter.

5.4 TV-Filter Implementation

We will first describe our distributed implementation of the Cong-Bader TV-Filter algorithm. Algorithm 17 gives the general overview for this algorithm. There are two overall steps to TV-Filter, with the first being Cheriyan-Thurimella edge filtering and the second being the Tarjan-Vishkin biconnectivity algorithm. We will discuss these steps in more detail later in this section.

Algorithm 17 Cong-Bader TV-Filter Algorithm

procedure TV-FILTER($G = (V, E)$)
  $F \cup T =$ Filter(G)
  TV($F \cup T$)
  for all edge $e = (u, v) \in E(G - F \cup T)$ **do**
    Label e as in BiCC containing v and v's parent
  return edge labels

An overview of Tarjan-Vishkin's algorithm is shown in algorithm 18. As can be observed, this algorithm consists of seven primary phases. These phases include the construction of a rooted spanning tree (Tree), the preorder labeling of this tree (Preorder), the computation of all subtree sizes for all vertices in the tree (Descendants), the labeling of vertices based on preorder number and subtree size (HighLow), the construction of a secondary auxiliary graph based on these labels (AuxGraph), and then a connected components check (CC) on the auxiliary graph with a final labeling (Label). Most of these phases have possible time-efficient but nontrivial implementations in distributed memory. The biggest challenge for these implementations is in minimizing necessary communication, particularly for the auxiliary graph construction.

Algorithm 18 Tarjan-Vishkin Biconnectivity Algorithm

procedure TARJAN-VISHKIN($G = (V, E)$)
  Tree: Compute Spanning Tree T
  Preorder: Label $v \in V$ with preorder labels
  Descendants: For each $v \in V$ compute subtree size in T
  HighLow: Compute low and high labels for each vertex
  AugGraph: Construct AuxG using above computed values
  Components: Label connected components of AuxG
  Label: Map AuxG labels back to edges of G

As we will discuss, we initially attempted to implement a subset of theoretically time-efficient approaches for various phases in Algorithm 18. However, we noted considerable overheads in practice that severely impacted real-world performance. As such, we developed novel traversal-based approaches for several of these phases.

5.4.1 Cheriyan-Thurimella Edge Filtering

As mentioned, the Cong-Bader TV-Filter algorithm initially reduces the number of edges needed to determine biconnectivity on some G. This is possible via the interesting

result of Cheriyan and Thurimella [72], which states that the k-connectivity of some connected graph G is equivalent (in terms of separators) to the k-connectivity of some G' where $G' = T \cup F_1 \cup ... \cup F_k$. Here T is a spanning tree of G and F_i is a spanning forest of $G - T - F_1 - \ldots - F_{i-1}$. As biconnectivity decompositions seek to identify separators of size 1, we can simply construct $G' = T \cup F_1$. We observe that this procedure can greatly reduce the number of edges needing processed during a biconnectivity decomposition. In practice, we got reductions in the number of edges by a factor of 3-10 for the graphs we used for our experiments.

Algorithm 19 Cheriyan-Thurimella Edge Filtering

```
procedure FILTER(G = (V, E))
    T = BFS(G)
    C = ConnectedComponents(G − T)
    F = BFS(G, C)
    return T ∪ F
```

Our approach to distributed edge filtering is given in Algorithm 19. We use two primary subroutines, including a single/multi-source BFS and an implementation of connected components. For BFS, we reuse an efficient implementation from our prior work [43]. Our connected components implementation uses label propagation and is given in Algorithm 20. We use connected components to identify a single vertex per component of $G - T$ for a multi-source BFS to construct the spanning forest F. See the discussion below with respect to the preorder subroutine, as reasoning for why we use this approach in lieu of a more direct computation of the spanning forest.

Our connected components algorithm initializes component labels for all vertices to be their global vertex identifier. Until convergence, each vertex iteratively updates their label to the lowest label in its neighborhood. For this and many of our other algorithms, we use a queue-based approach, where we maintain an *active* set of vertices – these vertices had their or their neighbor's label update on the prior iteration. Likewise, many of our implementations utilize a multi-level queue, where each processing thread owns a small queue, which they push to a rank-level queue as it fills, while the rank-level queue is what is used during communication boundary exchanges with other ranks. We don't show this low-level of granularity in our algorithm listings for space considerations, but we note it here in text because it is an important performance optimization.

Algorithm 20 Connected Components

```
procedure ConnectedComponents(G = (V, E))
    for all v ∈ V(G) do in parallel
        C(v) ← VertexID(v)
    Q ← V(G), Q_n ← ∅
    while Q is not empty on some rank do
        for all v ∈ Q do in parallel
            for all (v, u) ∈ E(G) do
                if C(v) > C(u) then
                    C(v) ← C(u)                    ▷ Propagate lower labels
                    Q_n.insert(v)
            for all (v, u) ∈ E(G) do
                if C(v) changed then
                    Q_n.insert(u)
        Exchange C(v) for all boundary v ∈ Q_n
        Update C(v) for all received values
        Swap(Q, Q_n), Q_n ← ∅
    return C
```

5.4.2 Tree, Preorder and Descendants Computation

The original Tarjan-Vishkin paper describes the construction of a spanning tree (Tree, as listed in Algorithm 18), following by a preorder labeling of this tree (Preorder), and then a computation of the sizes of the subtrees rooted at each vertex (Descendants). Each of these routines have a $O(\log |V|)$ parallel time implementation in the PRAM model, using $O(n + m)$ processors. Generally, pointer jumping and similar techniques are used to reach this time bound for these subroutines [76].

We consider two such approaches for their implementation in distributed memory. First, we consider a pointer-jumping equivalent implementation, where each "pointer" is a communication package containing any needed information for the subroutine (originating vertex, next vertex to jump to, total jumps, etc.) that gets communicated in a bulk synchronous fashion. As the total number of jumps and therefore communication rounds is similarly bounded by $O(\log n)$, we have an equivalent time bound in the BSP model for such an approach. When assuming a sufficient number of PRAM processors per rank, each package can be processed independently. Our second approach for implementing these subroutines is a non-time-efficient use of traversals similar to BFS. We modify the BFS routine from Algorithm 19 for this. Note that most level-synchronous BFS implementations with

shared memory parallelization over vertices in the queue will have $O(D\Delta(G))$ time and $O(n+m)$ work bounds, where D is the diameter of the input graph and $\Delta(G)$ is the maximum degree. While many real graphs have a diameter of approximately $\log n$, we note that web crawls, information networks, and meshes can have diameters significantly larger than that.

5.4.2.1 A New Preorder Algorithm

We'll focus our discussion on the preordering subroutine specifically. For the pointer-jumping-based approach, we implemented the algorithm described by Chen-Das-Akl [77] to give us an algorithm which runs in $O(\log n)$ parallel time with $O(n)$ processors, and does $O(n \log n)$ work. However, preliminary experiments demonstrated that such an implementation in distributed memory can be described as having a high constant factor associated with these bounds, due to the necessity of having to communicate each of these pointer packages across most iterations. We'll show that in our results, this leads to an orders-of-magnitude performance difference in practice. As such, we've generally avoided any such approach for our other implementations, instead developing novel traversal-based implementations for these subroutines. Algorithm 21 shows our approach for preorder. Note that as input this algorithm takes in a rooted spanning tree, which we compute via BFS, and the number of descendants for each vertex, the algorithm for which we will discuss next.

Algorithm 21 uses the number of descendants for each vertex to enable the assignment of preorder labels independently at each level in the BFS tree. Starting at the root, we assign preorder labels to the children of each vertex in the tree, offsetting their label numbers by their numbers of descendants. Because preorder labels are strictly ordered, the preorder assignment are done serially for each vertex, but we can safely compute each vertex independently from one another in parallel. In serial it is straighforward to see that the work complexity is $O(n)$, as each vertex must be visited and assigned a label. In parallel, the number of supersteps would be bounded above by D, with shared memory computations in each superstep bounded by $\Delta(G)$, as child label assignment is inherently serialized in our implementation.

Algorithm 21 Distributed-Memory Preorder

```
procedure PREORDER-DIST(G = (V, E), Tree T, n_desc)
    for all v ∈ V do in parallel
        Preorders(v) ← null
    Q, Q_n ← ∅
    if rank(root) = this_rank then
        Q.insert({root, 1})
    while Q is not empty on some rank do
        for all package ∈ Q do in parallel
            {v, preorder} ← package
            Preorders(v) ← preorder
            child_label ← preorder + 1
            for all child of v in T do
                Q_n.insert({child, child_label})
                child_label = child_label + n_desc(child)
        Exchange all Q_n packages having non-local vertices
        Swap(Q, Q_n), Q_n ← ∅
    return Preorders
```

5.4.2.2 Descendant Count Computation

As noted, our preorder algorithm requires the size of the subtree rooted at each vertex. The subtree size is the total number of descendents from a vertex in the original spanning tree plus that vertex itself. Algorithm 22 shows our approach, which is based on passing *packages* up the tree via a level-synchronous processing queue.

Algorithm 22 show how we accumulates descendant counts by doing traversals from all leaves up the tree to the root. The traversals track how many unique vertices they have reached. When a vertex is reached during these traversals, it updates its descendant count by the amount of vertices enumerated by the given traversal and directs the traversal to its parent. Note that a vertex only adds itself to the count the first time its reached by a traversal. The time complexity of this algorithm is similar to our preorder algorithm, with the number of supersteps bounded by graph diameter and each superstep bounded by $\Delta(G)$, due to the need to atomically update descendant counts. An alternative parallel algorithm which runs in $O(\log n)$ time with $O(n)$ processors, uses pointer jumping via adjacency doubling, such as the method referred to by Tarjan and Vishkin [53], which is a similar processing approach as with Chen-Das-Akl preordering. However, as before, we sacrifice a better time complexity for a much simpler implementation with what is likely better performance in

Algorithm 22 Distributed-Memory Descendant Counting

```
procedure DESCENDANTS-DIST(G = (V, E), Tree T)
    for all v ∈ V do in parallel
        n_desc(v) ← 1
    Q, Q_n ← ∅
    for all leaf ∈ T do in parallel
        Q.insert({leaf, 1})
    while Q is not empty on some rank do
        for all package ∈ Q do in parallel
            {v, count} ← package
            ATOMIC CAPTURE:
            {desc ← n_desc(v); n_desc(v) ← n_desc(v) + count}
            if desc = 1 then
                count ← count + 1
            if v is not root in T then
                Q_n.insert({parent(v), count})
        Exchange all Q_n packages having non-local vertices
        Swap(Q, Q_n), Q_n ← ∅
    return n_desc
```

practice.

5.4.3 HighLow Computation

The HighLow computation determines the minimum (Low) and maximum (High) pre-order values of any descendant, or any non-tree edge neighbor of a descendant. As before, the optimal algorithm described by Tarjan-Vishkin uses pointer jumping via adjacency doubling. Similarly, we note that a straightforward implementation can follow a very similar pattern to our descendant counting computation. We start from the leaves of the tree and traverse towards the root passing High and Low descendant values via packages in a queue. Each vertex encountered by a package updates its High and/or Low value atomically, and then passes the package to its parents if the vertex updated its High or Low value. Our approach for calculating the High values is given in Algorithm 23, the computation of the Low values is similar.

In general, we note that a majority of these initial stages uses variations on BFS traversal, which might benefit from optimizations given in the thousands of papers in the literature about distributed BFS and spanning tree construction. However, we also note these

Algorithm 23 Distributed-Memory High Computation

```
procedure HIGH-DIST(G = (V, E), Tree T, preorder)
    for all v ∈ V do in parallel
        highs(v) ← max(preorders(neighbors(v)))
    Q, Q_n ← ∅
    for all v ∈ V do in parallel
        Q.insert({v, highs(v)})
    while Q is not empty on some rank do
        for all package ∈ Q do in parallel
            {v, value} ← package
            while value < highs(v) do
                ATOMIC CAPTURE:
                {high ← highs(v); highs(v) ← value}
                if high > value then
                    highs(v) ← high
                else if highs(v) == value then
                    Q_n.insert({parent(v), value})
        Exchange all Q_n packages having non-local vertices
        Swap(Q, Q_n), Q_n ← ∅
    return highs
```

algorithms do require inherently ordered and atomic operations, so certain optimizations might not be practical. Further exploring this is of interest for future work.

5.4.4 Auxiliary Graph Construction

After computing the descendant counts, preorder labels, and high and low values for each vertex, we proceed with the next step of the Tarjan-Vishkin algorithm, which is constructing the auxiliary graph. The auxiliary graph has vertices that correspond to tree edges in the original graph, and the auxiliary edges are added such that the connected components of the auxiliary graph correspond to biconnected components in the original graph.

It is important to note that, due to the auxiliary vertices corresponding to edges in the original graph, a consistent global identification of auxiliary vertices is nontrivial in distributed memory. This is due to the fact that a significant number of edges are shared across processor boundaries. Our approach to consistent global auxiliary vertex identification is to assign unique identifiers to each unique edge. We distribute the global identifiers along with the graph, ensuring that the mapping from global edge indicies to local edge indicies is complete and consistent. For auxiliary vertices that correspond to cut edges, we assign

ownership to the process that owns the endpoint with a higher global vertex ID.

Algorithm 24 Auxiliary Graph Construction

procedure AuxGraph(G, T, *Preorders*, *Lows*, *Highs*, *n_desc*)
    Count up edges to be added to auxiliary graph
    Request global IDs of remote auxiliary vertices
    Create and fill auxiliary edgelist
    Communicate remote edges and vertex owners
    Create CSR representation of auxiliary edgelist
    return Auxiliary graph

Algorithm 24 shows an overview of our distributed auxiliary graph construction procedure. To save space, we omit the cumbersome logic involved in counting or creating auxiliary edges, as we follow the same logic as Tarjan and Vishkin. Edge creation can be simply described as a function of preorder labels, high and low values, and descendant counts for each tree and non-tree edge. The crux is that, in distributed memory, the auxiliary endpoints of an edge may not be local to the process that determines the creation of the edge. In fact, the process that adds the auxiliary edge may not own either of the endpoints. This means that the auxiliary graph construction process requires complex remote lookups that incur significant communication costs, and this issue can not be mitigated by simply altering the auxiliary graph distribution.

We parallelize our implementation by giving each thread a set of vertices to process, and thus construct the auxiliary edge list in parallel. While this procedure can be theoretically completed in $O(1)$ time using $O(m)$ processors, we note the additional lookups and translations necessary in distributed memory makes the expected performance quite difficult to accurately predict in practice.

5.4.5 Connected Components

Once the auxiliary graph is created, we need to assign labels to each connected component of the auxiliary graph. To do so, we use the same distributed connected components routine as with our Cheriyan-Thurimella Filtering implementation, again noting our explicit tradeoff between good theoretical time complexity and simplicity of implementation.

5.4.6 Final Labeling

As mentioned, the connected components of the auxiliary graph are the biconnected components of the original graph. Thus, after the connected components of the auxiliary graph are labeled, we can translate the labels to the original graph to label biconnected components and articulation vertices. Again, translating between the original graph and the auxiliary graph is not necessarily straightforward in distributed memory, as processor boundaries are not consistent. Hence, while we can trivially parallelize over all labels that we need to translate from the auxiliary graph, some label translations need to be communicated. In other words, while a theoretical work complexity for this subroutine is linear in graph size and parallel time is constant given $O(n + m)$ processors, in practice the necessary implementation details for distributed memory incur significant overheads. As of now, we have no solution for this issue while still retaining true distributed graph representations with an $O(\frac{n+m}{p})$ memory bound. Investigating this in more detail is another avenue for future work.

5.4.7 Discussion of TV-Filter-Dist

In summary, we note that porting the original TV-Filter algorithm as originally described to a distributed memory environment is likely nonviable, as the stated time complexity for several of the subroutines is dependent on techniques that have high communication overheads in practice. We additionally observe that there are several challenges associated with the efficient construction and usage of the auxiliary graph, for which we currently have no immediate solution.

By comparison, the subroutine modifications we propose are designed with the distributed memory environment in mind, making them more straightforward to understand and implement. However, we also note that the worst case complexity of our subroutines can make them more susceptible to adversarial inputs (e.g., graphs with an extremely high diameter) than the original Tarjan-Vishkin algorithm. However, in practice, most real graphs do not exhibit such extreme topology.

5.5 Color-BiCC Implementation

As part of our study, and continuing along similar themes, we also considered the distributed implementation of a non-time-efficient shared memory biconnectivity algorithm that is built on simple and easy to optimize subroutines. Specifically, we distribute the Color-BiCC algorithm of Slota and Madduri [8]. We also considered their BFS-based algorithm and the algorithm of Chaitanya and Kothapalli [13]; however, the Slota-Madduri BFS algorithm uses a lot of concurrent thread-owned traversals which would be costly to distribute and the Chaitanya-Kothapalli algorithm uses task-based parallelism which also doesn't lend itself to an easy distributed formulation.

Algorithm 25 Slota-Madduri Color-BiCC Algorithm

```
procedure COLOR-BICC(G = (V, E))
    T = BFS(G)                          ▷ Compute spanning tree via BFS
    High, Low = LCA(G, T)               ▷ Initialize labels
    A = Color(G, High, Low, T)          ▷ Propagate labels
    return A
```

Given in Algorithm 25, the Color-BiCC algorithm has three primary phases. The first phase, similar to Tarjan-Vishkin, computes a rooted spanning tree. The second phase initializes per-vertex *High* and *Low* label values using least common ancestor (LCA) traversals of the tree. The final phase propagates these labels such that, at the conclusion of propagation, each vertex v will have a *High* label with a value of the nearest articulation vertex that will disconnect v from the root (or, the label of the root itself). The algorithm uses these labels to determine and then return a set of articulation vertices A. These articulation vertices can then be used to label a biconnectivity decomposition.

5.5.1 BFS and LCA Implementations

We re-use our breadth-first search implementations from TV-Filter to construct the rooted spanning tree T. T is implicitly created via *parents* and *levels* arrays that defines each vertex's parent and its distance from the root. The LCA phase uses these along with traversals up the tree to initialize *High* and *Low* values. Specifically, each vertex will have their *High* value initialized to the lowest-level ancestor that it has in common with one of its non-tree neighbors. Here, "lowest level" refers to being closer to the root. *Low* values for

each vertex are set to the lowest numeric vertex identifier among that vertex and all of its non-tree neighbors.

Algorithm 26 Distributed Memory LCA Phase of Color-BiCC

```
1: procedure LCA(G, T = (Parents, Levels))
2:     Q ← ∅, Q_n ← ∅                                          ▷ Traversal queues
3:     for all (v, u) ∉ T do in parallel                        ▷ Non-tree edges
4:         Q ← {Parents(u), u, Levels(u), Parents(v), v, Levels(v)}
5:     for all v ∈ V(G) do in parallel
6:         High(v) ← Parents(v)
7:         Low(v) ← lowest ID u ∈ N(v), where(v, u) ∉ T
8:     while Q is not empty on some process do
9:         for all package ∈ Q do in parallel
10:            high ← higher level Parents in package
11:            low ← lower or equal level Parents in package
12:            u, v ← vertices in package
13:            level_u, level_v ← levels in package
14:            if level_u = null then                           ▷ Received final value
15:                Update High(u)
16:            else if low = high then                          ▷ Common ancestor found
17:                if rank(u) ≠ this_rank then
18:                    package ← {u, high, null, ...}
19:                    Q_n.insert(package)
20:                else                                         ▷ Can set final value without comm.
21:                    Update High(u)
22:                if rank(v) ≠ this_rank then
23:                    package ← {v, high, null, ...}
24:                    Q_n.insert(package)
25:                else
26:                    Update High(v)
27:            else                                             ▷ Take next step in traversal
28:                package ← {Parents(high), u, level_u − 1,
29:                           Parents(low), v, level_v − 1}
30:                Q_n.insert(package)
31:        Exchange Q_n
32:        Swap(Q, Q_n), Q_n ← ∅
33:    return High, Low
```

Our distributed LCA implementation is given in Algorithm 26. As with our other implementations, we use a distributed queue-based approach. This queue is comprised of *packages* that contain the two vertices that the LCA traversal originated from (u and v) and

the current location of this traversal (stored as *high* and *low* along with the current levels). During each superstep, packages are processed with traversals progressing by following the values in *Parents* arrays. When *high* and *low* are equal, it indicates a least common ancestor has been found. $High(u)$ and $High(v)$ can then be updated to these values if the current level for *high* is closer to the root than current vertices indicated in $High(u)$, $High(v)$. This update can happen immediately if u, v are local to the processing rank; otherwise, a final communication occurs to transmit the result to the owning ranks.

5.5.2 Color Propagation Implementation

The final phase of Color-BiCC is the iterative propagation of *High* and *Low* values following a set of propagation rules, demonstrated in Algorithm 27. We modify the original algorithm of Slota and Madduri, which utilized a "push" style of propagation, where vertex v would modify its neighbor u's values. Within a distributed context having much wider parallelism, we use a "pull" style of propagation, where vertex v only modifies its own values based on its neighbors values. Otherwise, our propagation rules are consistent with the original algorithm.

As with our other implementations, we utilize a queue-based strategy with a boundary exchange on each iteration. We track which vertices are currently in the queue to avoid adding the same vertex multiple times. For our boundary exchange, we pass each owned vertex v along with $High(v)$ and $Low(v)$. The receiving ranks of v will have v as a ghost vertex. In addition to updating their local values for v, the receiving rank will also place all owned neighbors of v in the queue for processing. As such, the active set of vertices for processing in a given iteration are only the vertices who have had some value change in their neighborhood on the previous iteration. This optimization is particularly important for graphs such as web graphs, which can have a high diameter and long tails of iterations where only a few updates are processed per iteration. The final set of articulation vertices A are all unique vertex values stored in $High(v)$ across all $v \in V(G)$.

5.5.3 Discussion of Color-BiCC

We note that none of the distributed implementations we developed for Color-BiCC are theoretically time-efficient. In fact, all of these three subroutines are theoretically dependent on graph diameter within the BSP model. However, as one will be able to observe in our

Algorithm 27 Distributed Memory Coloring Phase of Color-BiCC

```
1: procedure COLOR(G, High, Low, T = (Parents, Levels))
2:     Q ← V(G), Q_n ← ∅
3:     while Q is not empty on some process do
4:         for all v ∈ Q do in parallel
5:             for all (v, u) ∈ E(G) do
6:                 if High(v) = v then
7:                     continue
8:                 if Levels(High(u)) > Levels(High(v)) then
9:                     High(v) ← High(u)
10:                if Levels(High(u)) = Levels(High(v)) then
11:                    if High(u) > High(v) then
12:                        High(v) ← High(u)
13:                if High(u) = High(v) then
14:                    if Low(u) < Low(v) then
15:                        Low(v) ← Low(u)
16:                if High(v) or Low(v) changed and v ∉ Q then
17:                    Q_n.insert(u)
18:        Exchange High, Low for all v ∈ Q_n
19:        Swap(Q, Q_n), Q_n ← ∅
20:        for all v ∈ Q do in parallel
21:            for all v, u ∈ E(G) do
22:                if u ∉ Q then
23:                    Q.insert(u)
24:    A ← all unique values in High
25:    return A
```

results, this is not restrictive on performance in practice on real graphs. In fact, we observe consistent speedups across all tests for our implementation and very fast performance relative to distributed TV-Filter and the optimal serial algorithm. This alludes to our primary takeaway for this work: in real-world environments, implementation simplicity can often trump theoretic efficiency by a considerable degree.

5.6 Experimental Setup

We run our primary experiments on well-known datasets with a variety of scales and topologies. The datasets are given in Table 5.1. We consider the underlying graphs for any directed graphs. We also preprocess all networks to extract the largest connected component and remove duplicate edges, as the TV-Filter algorithm assumes a simple and connected

input. Our graphs primarily come from the Stanford Large Network Dataset Collection[4], the Network Data Repository[5], and the Koblenz Network Collection[6]. We additionally include a random scale-25 R-MAT graph generated with parameters $\{A = 0.45, B = 0.15, C = 0.15, D = 0.25\}$.

Table 5.1: Graphs used for experiments and their properties in terms of the number of vertices $|V|$ and edges $|E|$ after preprocessing as well as the approximate diameter D and number of biconnected components (#BiCCs).

Graph Name	Type	$\|V\|$	$\|E\|$	D	#BiCCs	Ref.
soc-LiveJournal1	Social Net.	4.8 M	43 M	46	76 K	[78]
com-Friendster	Social Net.	52 M	1.1 B	35	5.5 M	[79]
web-Google	Web Graph	855 K	4.3 M	25	60 K	[78]
web-ClueWeb09	Web Graph	225 M	1.0 B	40	15 M	[80]
dbpedia-link	Info. Net.	18 M	127 M	13	2.8 M	[81]
wikipedia_link_en	Info. Net.	14 M	335 M	12	1.9 M	[81]
RMAT_25	Random	34 M	537 M	11	174 K	[82]

We implement all of our methods in `C++` using MPI and OpenMP for distributed and shared-memory parallelism. For direct comparison to the Slota-Madduri shared memory algorithm, we run the code provided on the author's website[7]. For additional comparisons, we implemented our own version of the serial Hopcroft-Tarjan algorithm.

For experimentation, we consider runs on two clusters at RPI, DRP and AiMOS. DRP consists of nodes that have 2x 8-core 2.6 GHz Intel Xeon E5-2650 CPUs and 256 GB DDR which are connected by 56 Gb FDR Infiniband. AiMOS has nodes with 2x 20-core 3.15 GHz IBM Power 9 CPUs and 512 GB DDR which are connected by 100 Gb EDR Inifiniband. Our build and execution environment on DRP includes OpenMPI 3.1.6 with GCC 10.2.0 running on CentOS 7.9.2009, while our environment on AiMOS includes Spectrum MPI 10.4 and XL 16.1.1 running on RHEL 8.4.

To better analyze distributed scaling behavior on a limited number of nodes, we run one MPI rank per socket on each of these systems for our experiments. We've generally observed the fastest execution times with only 1 thread per code on both systems for all algorithms, so we fix our thread count per rank equal to the core count per socket for all

[4] `http://snap.stanford.edu/data/index.html`
[5] `https://networkrepository.com/index.php`
[6] `http://konect.cc/`
[7] `https://www.cs.rpi.edu/~slotag/soft/BiCC-HiPC14.tar`

experiments.

5.7 Results

For our experiments, we are primarily concerned with performance in terms of strong scaling and overall execution time. We consider a contribution of this work an experimental study that examines the performance tradeoffs between our various implementations, as we extend them to distributed memory.

5.7.1 Distributed Edge Filtering

As noted, both our TV-Filter and Color-BiCC algorithms first preprocess the input graph via Cheriyan-Thurimella edge filtering. Given in Figure 5.1 is strong scaling on both test systems for our 6 real input graphs.

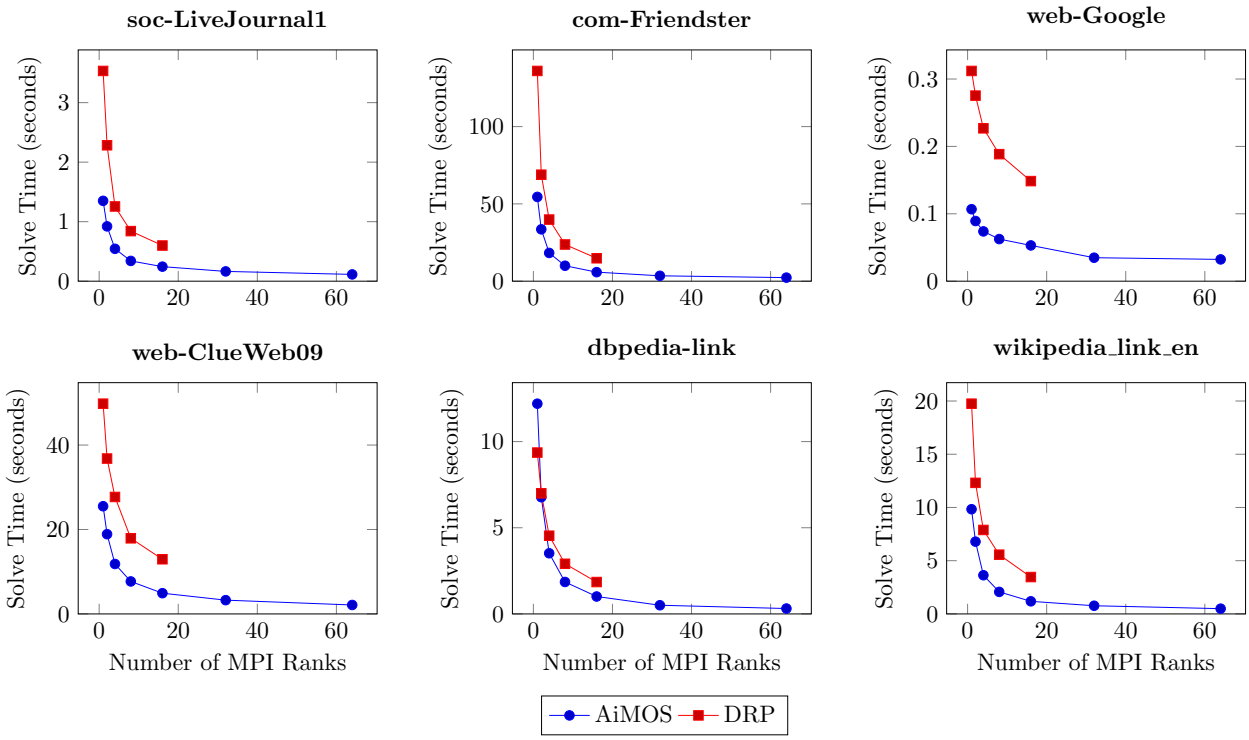

Figure 5.1: Scaling of our implementation of the Cheriyan-Thurimella Filter algorithm used as a preliminary step for both of our implementations of TV-Filter and BCC-Color-Dist from 1 to 64 ranks on AiMOS and 1 to 16 ranks on DRP.

We notice consistent scaling across both systems and all inputs. We observe an av-

erage speedup of 4.7× from 1 to 16 ranks on DRP with a maximum speedup of 9.2× on com-Friendster. Likewise, we observe an average speedup of 16× from 1 to 64 ranks on AiMOS with a maximum speedup of 24×, again on com-Friendster. Overall, while our implementation does not provide perfect speedup, it scales all the way up to 64 ranks on even the smallest inputs. We otherwise observe no major scalability bottlenecks in these experiments.

5.7.2 Distributed TV-Filter

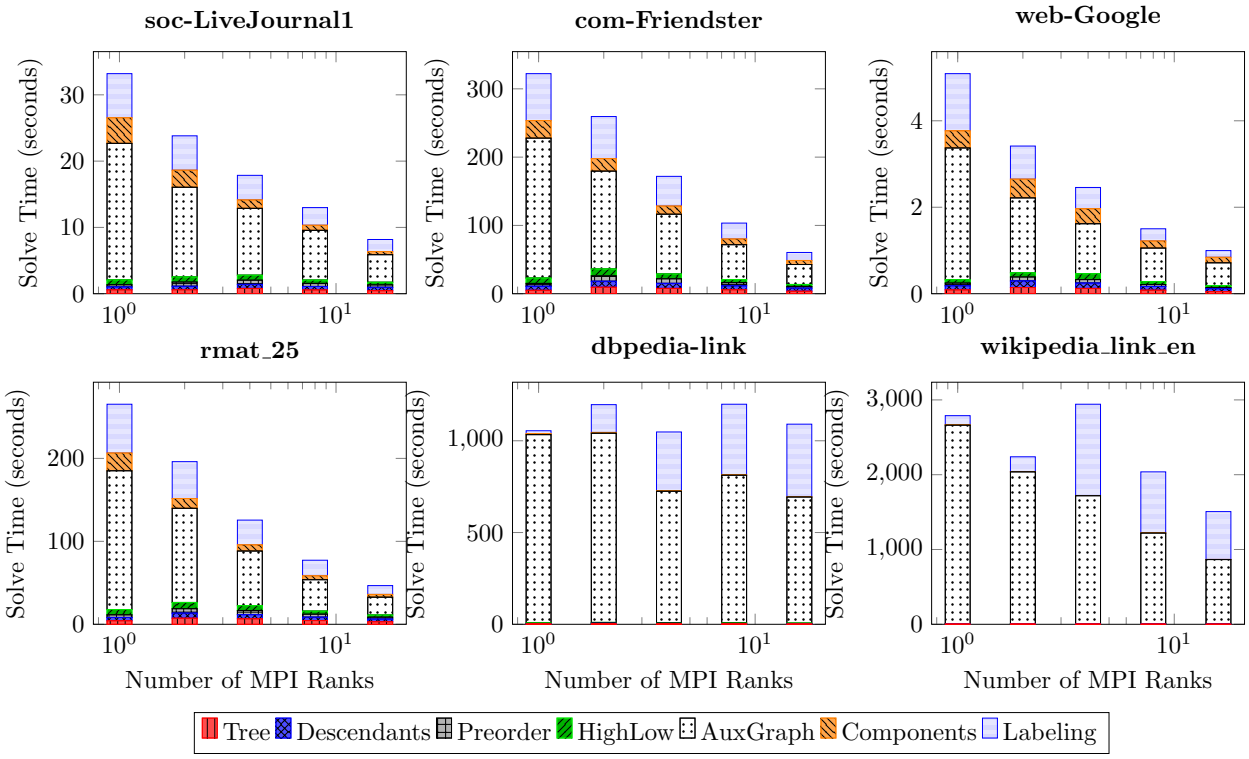

Figure 5.2: Strong scaling of TV-Filter with a breakdown of proportional execution times for all constituent subroutines from 1 to 16 ranks on DRP.

Figure 5.2 shows the performance of the distributed TV-Filter on up to 16 ranks of DRP processing a selection of our inputs. On smaller inputs we see relatively good scaling, with up to a 5× speedup over the single rank run. However, the performance of this approach is worse on large inputs such as wikipedia_link_en and dbpedia-link. We were unable to get web-ClueWeb09 to process successfully for all rank counts, and instead show scaling for a scale-25 R-MAT graph. The primary bottlenecks are the creation of the

auxiliary graph and the final labeling of the vertices. As noted previously, both of these subroutines require expensive (in distributed memory) lookups for non-local auxiliary edges, incurring substantial communication overheads. These performance results were relatively unsurprising, and were a major motivating factor for us to seek out simpler-to-implement algorithms for distributed biconnectivity. However, we still observed good speedup and performance for several of the subroutines, which we'll discuss next.

5.7.3 TV-Filter Subroutines

One of our most surprising findings of this study is not that time-efficient PRAM algorithms which have an equivalent time-efficient BSP formulation are slower relative to simpler but less theoretically efficient algorithms. What was most surprising was just *how much slower* these algorithms ended up being in a practical setting. A representative example is given in Figure 5.3, which compares the time-efficient algorithm of Chen-Das-Akl and our traversal-based algorithm for computing preorder labels of a rooted spanning tree. Note that our implementation of the Chen-Das-Akl algorithm utilizes the same communication framework and queue-based optimizations that we use for several other subroutines (specifically, connected components and LCA).

We observe in Figure 5.3 over an order-of-magnitude difference in performance across all test inputs on up to 16 ranks of DRP. On limited scaling experiments across larger rank counts on AiMOS, we still observe a considerable performance difference. We do note that the Chen-Das-Akl algorithm shows better strong scaling behavior; however, we have not observed a single test configuration where the performance difference is less than a factor of 10×. It is possible that this gap might close on a considerably higher rank and core count than we currently can run.

Figure 5.4 shows the speedups of our "fast" TV-Filter subroutines on AiMOS from 1 to 64 ranks, including BFS spanning tree construction, descendants counting, preorder labeling, and the high-low computation. We observe the best scaling in the high-low calculation, seeing up to a 7.3× speedup from a single rank and a 5.26× speedup on average. Descendants counting appears to be the least scalable subroutine, which is interesting, as its computational patterns is quite similar to high-low. This difference can likely be explained by the necessary use of more atomic operations in the descendants subroutine.

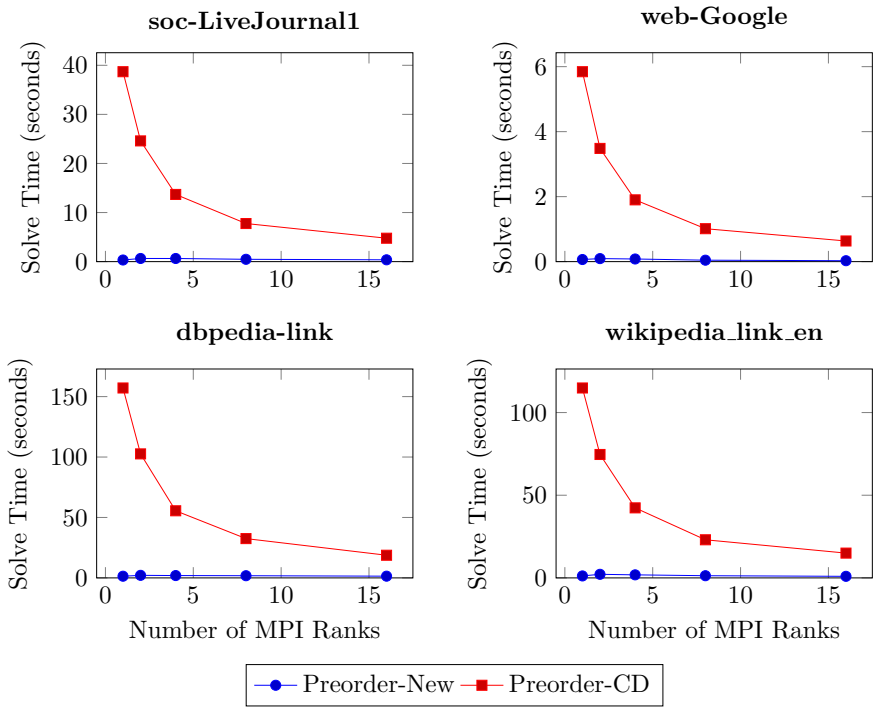

Figure 5.3: Execution time comparison between our new preorder algorithm (Preorder-New) and the Chen-Das-Akl algorithm (Preorder-CDA) from 1 to 16 ranks on DRP.

5.7.4 Distributed Color-BiCC

Figure 5.5 gives the strong scaling performance of our Color-BiCC implementation from 1 to 64 ranks on AiMOS. We give the summed time of Color-BiCC and edge filtering (Color-BiCC-Dist), the time of Color-BiCC without edge filtering (Color-BiCC-NoFilter), the time of the serial Hopcroft-Tarjan algorithm (HT-Serial), and the time of Slota and Madduri's shared memory code running on a single socket (Color-BiCC-SM). We run on only a single socket to enable relative comparisons to a single rank of the distributed implementation.

For Color-BiCC with edge filtering, we measure an average speedup of 15× from 1 to 64 ranks. Relative to the serial and shared memory algorithms, we measure average speedups of 11× and 7.3×, respectively. For all inputs except for web-Google, we note that the sum time for Color-BiCC and filtering is lesser than the time for Color-BiCC to process the unfiltered input. On a single rank and 20 threads, we note that our distributed Color-BiCC implementation is 1.9× slower on average than the Slota and Madduri code. However, we believe that this difference is quite reasonable in terms of overhead, considering that the communication routines and queue structures are still being utilized on single rank runs. For

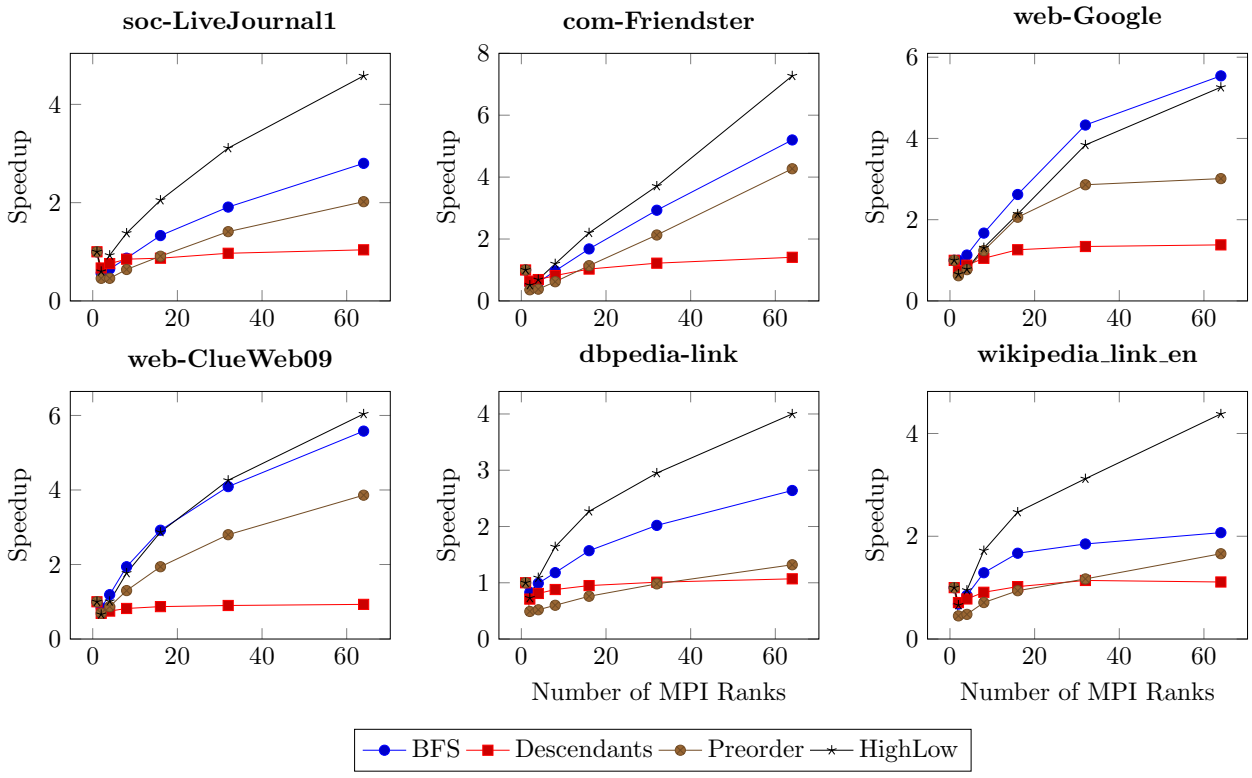

Figure 5.4: Parallel speedups of the fastest 4 subroutines from TV-Filter (BFS, Descendents, Preorder, HighLow) from 1 to 64 ranks on AiMOS.

the largest inputs of com-Friendster and web-ClueWeb09, we observe scaling past the shared memory performance in as little as 2 or 4 ranks. For a distributed implementation operating on irregular graph inputs, we believe this performance to be in-line with the state-of-the-art.

Table 5.2: Comparison of Hopcroft-Tarjan (HT) on a single thread, the Slota-Madduri code (SM) on 20 threads, the sum of the "fast" Tarjan-Vishkin subroutines (TV) on 64 ranks, our distributed implementation of Color-BiCC without filtering (CBNF) on 64 ranks, our distributed implementation of Color-BiCC with filtering (CBD) on 64 ranks, and the speedup of Color-BiCC with filtering from 1 to 64 ranks.

Graph	HT	SM	TV-Sum	CBNF	CBD	Speedup
soc-LiveJournal1	2.2	0.80	0 .36	0.36	**0.23**	10×
com-Friendster	61	33	3.1	5.6	**2.2**	30×
web-Google	0.21	0.098	0.051	**0.047**	0.060	3.7×
web-ClueWeb09	30	38	14	7.3	**4.9**	8.9×
dbpedia-link	6.5	6.6	1.8	0.97	**0.72**	22×
wikipedia_link_en	9.3	6.6	1.3	1.5	**1.0**	13×

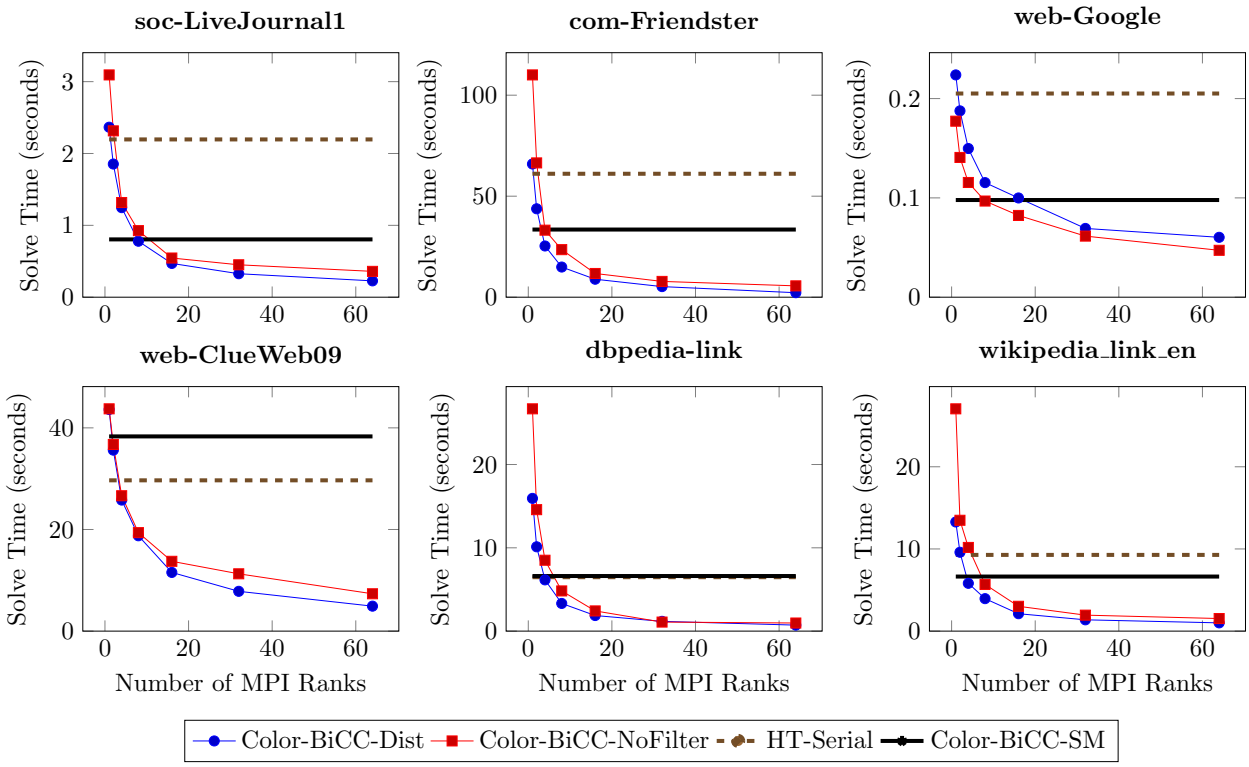

Figure 5.5: Strong scaling of our Color-BiCC implementation from 1 to 64 ranks on AiMOS with (Color-BiCC-Dist) and without filtering (Color-BiCC-NoFilter) relative to the shared memory Slota-Madduri algorithm (Color-BiCC-SM) on 20 threads and the optimal serial Hopcroft-Tarjan algorithm (HT-Serial) on a single thread.

Table 5.2 summarizes these performance results. In addition, we include the summed time for the four "fast" subroutines of TV-Filter shown in Figure 5.4. We note that the summed times for TV-Filter is consistently slower than those for Color-BiCC, despite the fact that several TV-Filter subroutines are omitted along with times for edge filtering. We believe this to be relatively firm evidence of how algorithms comprised of simple but non-efficient subroutines can outperform more theoretically time-efficient but considerably more complex algorithms in practice.

For our final analysis of results, we consider the proportional execution time breakdown of Color-BiCC with filtering on 64 ranks of AiMOS. This is given in Figure 5.6. We note that the connected components routine of the filtering algorithm is on average the largest proportion of execution time, followed by the color propagation phase of Color-BiCC. These results are relatively unsurprising, given that the filtering algorithm is running on the entire

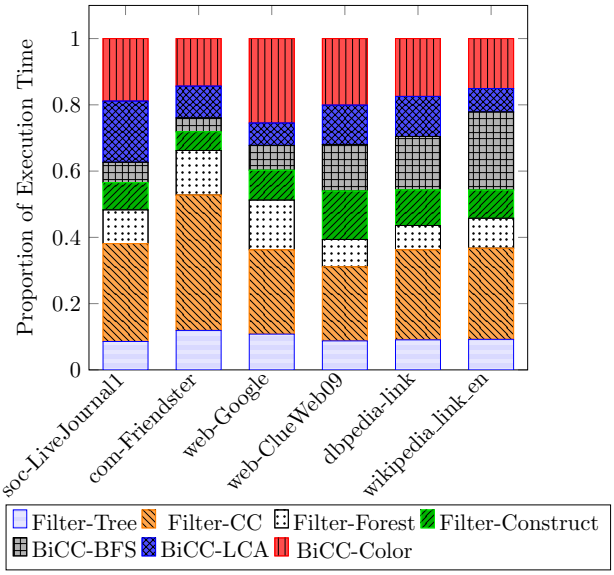

Figure 5.6: Proportion of execution total time of BCC-Color-Dist for each of the seven primary stages: Spanning tree, connected components, spanning forest, and graph construction for the filtering algorithm; as well as the BFS, LCA, and coloring stages of the biconnected components algorithm.

unfiltered graph, while the color propagation routine usually requires the greatest number of supersteps out of all of our implemented procedures.

5.8 Conclusions

This chapter explored difficulties in porting shared-memory parallel algorithms to distributed memory. We also presented efficient distributed memory implementations of popular shared memory subroutines, to better enable porting shared memory algorithms to distributed memory.

CHAPTER 6
CONCLUSION

6.1 Conclusion

This thesis has explored the implementation and practical applications of graph algorithms on HPC systems. Graphs are a flexible means of representing complex systems, and algorithms which operate on them have wide applications to any number of computational fields. As the power of HPC systems grows, so too will the scale of the data involved in scientific computations. This growth necessitates the exploration of efficient distributed graph algorithms, especially where none exist presently.

We have presented a tailored algorithm that greatly benefits land ice climate simulations, and also provides a new way of looking at the problem of graph biconnectivity. We built the first multi-GPU graph coloring framework that supports multiple different variants of the graph coloring problem that benefit many different scientific computing applications. Finally, we ported shared memory graph biconnectivity algorithms to establish efficient distributed implementations of popular subroutines, as well as implement the first efficient distributed memory graph biconnectivity approach in an HPC environment.

6.2 Future Work

As HPC platforms are ever-evolving, so too is the work of implementing algorithms which target them. This can be especially difficult for graph algorithms, as much of the literature focuses on optimal shared memory solutions that may not port efficiently to distributed memory. As such, expanding the literature exploring efficient distributed memory graph algorithms is the general subject of our future work.

Broadly, simulations which operate on meshes may benefit from distributed graph algorithms. These applications may be difficult to find as they are highly interdisciplinary, and a solution from the field of graph algorithms may not occur to those involved. However, graph algorithms can be used to identify particular features, as we have shown, and also calculate analytics as simulations progress. As the scale of simulations increase visual inspection of meshes becomes more and more costly, while distributed graph algorithms may be able to solve these problems without ever leaving the computational environment in which

the simulation runs.

Graph coloring is an important subroutine largely used for speeding up scientific computing applications by identifying concurrency in matrix data. While we were able to show state-of-the-art performance with our multi-GPU framework, there exist many avenues for improving our performance further. Primarily, the Zoltan framework implements many optimizations which were impractical for us, but may nonetheless provide speedup. Additionally, there are shared memory ideas that may improve our distributed memory recoloring, such as the Net-Based distance-2 approach of Taç et al. Finally, our partial distance-2 may be made more performant through allowing it to only color one set of vertices, though this change is not as trivial as it may sound.

Graph Biconnectivity presents a unique opportunity to study the best ways to port shared memory graph algorithms to distributed memory. We have presented the first efficient distributed memory implementations of many popular shared memory subroutines. However, there is more study that can be done. For instance, our work on BCC-LR implies that distributed pointer jumping may be performant if implemented correctly. If an efficient, general, distributed memory pointer jumping implementation could be found, it would enable the efficient implementation of many graph algorithms in distributed memory.

REFERENCES

[1] S. Fortunato, "Community detection in graphs," *Phys. Rep.*, vol. 486, no. 3-5, pp. 75–174, Feb. 2010.

[2] E. Larour, H. Seroussi, M. Morlighem, and E. Rignot, "Continental scale, high order, high spatial resolution, ice sheet modeling using the ice sheet system model (issm)," *J. of Geophysical Res.: Earth Surf.*, vol. 117, no. F1, Mar. 2012. DOI: `https://doi.org/10.1029/2011JF002140`.

[3] I. K. Tezaur, M. Perego, A. G. Salinger, R. S. Tuminaro, and S. F. Price, "Albany/felix: A parallel, scalable and robust, finite element, first-order stokes approximation ice sheet solver built for advanced analysis," *Geosci. Model Develop.*, vol. 8, no. 4, pp. 1197–1220, Apr. 2015.

[4] A. H. Gebremedhin and A. Walther, "An introduction to algorithmic differentiation," *Wiley Interdisciplinary Rev.: Data Mining and Knowl. Discovery*, vol. 10, no. 1, p. e1334, Jan. 2020.

[5] "Darpa mathematical challenges," DARPA, Arlington, VA, USA, Tech. Rep. BAA-07-68, 2008.

[6] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Process. Lett.*, vol. 17, no. 01, pp. 5–20, Mar. 2007.

[7] J. J. Dongarra, H. W. Meuer, and E. Strohmaier, "Top500 supercomputer sites," *Supercomputer*, vol. 13, pp. 89–111, Nov. 1997.

[8] G. M. Slota and K. Madduri, "Simple parallel biconnectivity algorithms for multicore platforms," in *2014 21st Int. Conf. on High Perform. Comput. (HiPC)*, Dec. 2014, pp. 1–10.

[9] R. E. Moraes and C. C. Ribeiro, "Power optimization in ad hoc wireless network topology control with biconnectivity requirements," *Comput. & Operations Res.*, vol. 40, no. 12, pp. 3188–3196, Dec. 2013.

[10] D. Day, M. Bhardwaj, G. Reese, and J. Peery, "Mechanism free domain decomposition," *Comput. Methods in Appl. Mechanics and Eng.*, vol. 192, no. 7-8, pp. 763–776, Feb. 2003.

[11] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. on Comput.*, vol. 14, no. 4, pp. 862–874, Nov. 1985.

[12] G. Cong and D. A. Bader, "An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps)," in *19th IEEE Int. Parallel and Distrib. Process. Symp.*, 2005, 9–pp.

[13] M. Chaitanya and K. Kothapalli, "Efficient multicore algorithms for identifying biconnected components," *Int. J. of Netw. and Comput.*, vol. 6, no. 1, pp. 87–106, Jan. 2016.

[14] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency: Pract. and Experience*, vol. 12, no. 12, pp. 1131–1146, Oct. 2000.

[15] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures," in *2016 IEEE Int. Parallel and Distrib. Process. Symp. (IPDPS)*, 2016, pp. 892–901.

[16] J. Allwright, R Bordawekar, P. Coddington, K Dincer, and C. Martin, "A comparison of parallel graph coloring algorithms," *SCCS-666*, pp. 1–19, 1995.

[17] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, "Colpack: Software for graph coloring and related problems in scientific computing," *ACM Trans. on Math. Softw. (TOMS)*, vol. 40, no. 1, p. 1, Oct. 2013.

[18] G. J. Chaitin, "Register allocation & spilling via graph coloring," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 98–101, Jun. 1982.

[19] L. Li, H. Feng, and J. Xue, "Compiler-directed scratchpad memory management via graph coloring," *ACM Trans. on Architecture and Code Optim. (TACO)*, vol. 6, no. 3, pp. 1–17, Oct. 2009.

[20] M. Garey, D. Johnson, and H. So, "An application of graph coloring to printed circuit testing," *IEEE Trans. on Circuits and Syst.*, vol. 23, no. 10, pp. 591–599, Oct. 1976.

[21] D. Brélaz, "New methods to color the vertices of a graph," *Commun. of the ACM*, vol. 22, no. 4, pp. 251–256, Apr. 1979.

[22] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM J. on Sci. Comput.*, vol. 14, no. 3, pp. 654–669, 1993.

[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 Comput.*, vol. 38, no. 10-11, pp. 576–594, Oct. 2012.

[24] G. Rokos, G. Gorman, and P. H. Kelly, "A fast and scalable graph coloring algorithm for multi-core and many-core architectures," in *Eur. Conf. on Parallel Process.*, 2015, pp. 414–425.

[25] 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," *J. of Parallel and Distrib. Comput.*, vol. 68, no. 4, pp. 515–535, Apr. 2008.

[26] A. E. Sariyüce, E. Saule, and Ü. V. Çatalyürek, "Scalable hybrid implementation of graph coloring using MPI and OpenMP," in *2012 IEEE 26th Int. Parallel and Distrib. Process. Symp. Workshops & PhD Forum*, IEEE, 2012, pp. 1744–1753.

[27] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating graph coloring on GPUs," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 297–298, Aug. 2011.

[28] A. H. Gebremedhin, F. Manne, and A. Pothen, "What color is your Jacobian? graph coloring for derivatives.," *SIAM Rev.*, vol. 47, no. 4, 629—705, 2005.

[29] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. of Parallel and Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, Dec. 2014.

[30] *Kokkos Kernels*, Accessed: Jun, 6, 2020, 2017. [Online]. Available: `https://github.com/kokkos/kokkos-kernels`.

[31] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, *et al.*, "An overview of the Trilinos project," *ACM Trans. on Math. Softw. (TOMS)*, vol. 31, no. 3, pp. 397–423, Sep. 2005.

[32] I. Bogle, E. G. Boman, K. Devine, S. Rajamanickam, and G. M. Slota, "Distributed memory graph coloring algorithms for multiple gpus," in *2020 IEEE/ACM 10th Workshop on Irregular Appl.: Architectures and Algorithms (IA3)*, Nov. 2020, pp. 54–62.

[33] M. Besta, A. Carigiet, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, and T. Hoefler, "High-performance parallel graph coloring with strong guarantees on work, depth, and quality," in *SC20: Int. Conf. for High Perf. Comput., Netw., Storage and Anal.*, Nov. 2020, pp. 1–17.

[34] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proc. of the 26th ACM Symp. on Parallelism in Algorithms and Architectures*, Jun. 2014, 166–177.

[35] M. Osama, M. Truong, C. Yang, A. Buluç, and J. D. Owens, "Graph coloring on the GPU," in *GrAPL: Workshop on Graphs, Architectures, Program., and Learn. (IPDPSW)*, 2019, pp. 231–240.

[36] S. Sallinen, K. Iwabuchi, S. Poudel, M. Gokhale, M. Ripeanu, and R. Pearce, "Graph colouring as a challenge problem for dynamic graph processing on distributed systems," in *Proc. of the Int. Conf. for High Perf. Comput., Netw., Storage and Anal.*, Jun. 2016, pp. 347–358.

[37] M. K. Taş, K. Kaya, and E. Saule, "Greed is good: Optimistic algorithms for bipartite-graph partial coloring on multicore architectures," 2017, *arXiv: 0902.0885*.

[38] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *2017 IEEE Int. Parallel and Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 646–655.

[39] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus," in *2013 42nd Int. Conf. on Parallel Process.*, Oct. 2013, pp. 80–89. DOI: `10.1109/ICPP.2013.17`.

[40] A. Venkatesh, K. Hamidouche, H. Subramoni, and D. K. Panda, "Offloaded gpu collectives using core-direct and cuda capabilities on infiniband clusters," in *2015 IEEE 22nd Int. Conf. on High Perf. Comput. (HiPC)*, Dec. 2015, pp. 234–243. DOI: `10.1109/HiPC.2015.50`.

[41] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. DOI: 10.1145/2049662.2049663.

[42] R. L. Brooks, "On colouring the nodes of a network," in *Math. Proc. of the Cambridge Philosophical Soc.*, vol. 37, Apr. 1941, pp. 194–197.

[43] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *2016 IEEE Int. Parallel and Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 293–302.

[44] K. D. Devine, E. G. Boman, L. A. Riesen, U. V. Catalyurek, and C. Chevalier, "Getting started with Zoltan: A short tutorial," in *Dagstuhl Seminar Proc.*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[45] M. Naumov, P. Castonguay, and J. Cohen, "Parallel graph coloring with applications to the incomplete-lu factorization on the gpu," NVidia White Paper, Santa Clara, CA, USA, Tech. Rep., 2015.

[46] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," in *Cray Users Group (CUG)*, vol. 19, May 2010, pp. 45–74.

[47] J. Church, P. Clark, A. Cazenave, J. Gregory, S. Jevrejeva, A. Levermann, M. Merrifield, G. Milne, R. Nerem, P. Nunn, A. Payne, W. Pfeffer, D. Stammer, and A. Unnikrishnan, "Sea level change," in *Climate Change 2013: The Physical Science Basis. Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change*, T. Stocker, D. Qin, G.-K. Plattner, M. Tignor, S. Allen, J. Boschung, A. Nauels, Y. Xia, V. Bex, and P. Midgley, Eds. Cambridge, UK and New York, NY, USA: Cambridge University Press, 2013, ch. 13, pp. 1137–1216.

[48] S. Cornford, D. Martin, D. Graves, D. Ranken, A. L. Brocq, R. Gladstone, A. Payne, E. Ng, and W. Lipscomb, "Adaptive mesh, finite volume modeling of marine ice sheets," *J. Comp. Physics*, vol. 232, no. 1, pp. 529–549, Jan. 2013.

[49] M. J. Hoffman, M. Perego, S. F. Price, W. H. Lipscomb, T. Zhang, D. Jacobsen, I. Tezaur, A. G. Salinger, R. Tuminaro, and L. Bertagna, "Mpas-albany land ice (mali): A variable-resolution ice sheet model for earth system modeling using voronoi grids," *Geosci. Model Develop.*, vol. 11, no. 9, pp. 3747–3780, Sep. 2018.

[50] R Tuminaro, M. Perego, I Tezaur, A Salinger, and S. Price, "A matrix dependent/algebraic multigrid approach for extruded meshes with applications to ice sheet modeling," *SIAM J. on Sci. Comput.*, vol. 38, no. 5, pp. C504–C532, 2016.

[51] X. Zou, K. Wu, D. A. Boyuka, D. F. Martin, S. Byna, H. Tang, K. Bansal, T. J. Ligocki, H. Johansen, and N. F. Samatova, "Parallel in situ detection of connected components in adaptive mesh refinement data," in *2015 15th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Comput.*, May 2015, pp. 302–312.

[52] D. Day, M. Bhardwaj, G. Reese, and J. Peery, "Mechanism free domain decomposition," *Comput. Methods in Appl. Mechanics and Eng.*, vol. 192, no. 7-8, pp. 763–776, Feb. 2003.

[53] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. on Comput.*, vol. 14, no. 4, pp. 862–874, Nov. 1985.

[54] E. Boman, K. Devine, V. Leung, S. Rajamanickam, L. Riesen, M. Deveci, and U. Catalyurek, "Zoltan2: Next-generation combinatorial toolkit.," Sandia National Laboratories, Albuquerque, NM, USA, Tech. Rep. SAND2012-9373C, 2012.

[55] K. Wu, E. Otoo, and K. Suzuki, "Optimizing two-pass connected-component labeling algorithms," *Pattern Anal. and Appl.*, vol. 12, no. 2, pp. 117–135, Jun. 2009.

[56] C. Harrison, J. Weiler, R. Bleile, K. Gaither, and H. Childs, "A distributed-memory algorithm for connected components labeling of simulation data," in *Topological and Statistical Methods for Complex Data*, J. Bennett, F. Vivodtzev, and V. Pascucci, Eds., Berlin, Germany: Springer, 2015, pp. 3–19.

[57] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973.

[58] G. M. Slota and K. Madduri, "Simple parallel biconnectivity algorithms for multicore platforms," in *2014 21st Int. Conf. on High Perform. Comput. (HiPC)*, Dec. 2014, pp. 1–10.

[59] M. Chaitanya and K. Kothapalli, "Efficient multicore algorithms for identifying biconnected components," *Int. J. of Netw. and Comput.*, vol. 6, no. 1, pp. 87–106, Jan. 2016.

[60] C. G. Baker and M. A. Heroux, "Tpetra, and the use of generic programming in scientific computing," *Sci. Program.*, vol. 20, no. 2, pp. 115–128, Jan. 2012.

[61] M. A. Heroux and J. M. Willenbring, "A new overview of the trilinos project," *Sci. Program.*, vol. 20, no. 2, pp. 83–88, Jan. 2012.

[62] *ProSPect software library*, Accessed: Aug, 18, 2019. [Online]. Available: `https://doe-prospect.github.io`.

[63] V. E. Taylor and B. Nour-Omid, "A study of the factorization fill-in for a parallel implementation of the finite element method," *Int. J. for Numer. Methods in Eng.*, vol. 37, no. 22, pp. 3809–3823, Nov. 1994.

[64] I. Bogle, K. Devine, M. Perego, S. Rajamanickam, and G. M. Slota, "A parallel graph algorithm for detecting mesh singularities in distributed memory ice sheet simulations," in *Proc. of the 48th Int. Conf. on Parallel Process.*, Aug. 2019, pp. 1–10.

[65] I. D. Scherson and C.-K. Chien, "Least common ancestor networks," *VLSI Des.*, vol. 2, no. 4, pp. 353–364, Jan. 1995.

[66] K. Kothapalli and M. Wadwekar, "Expediting parallel graph connectivity algorithms," in *2018 IEEE 25th Int. Conf. on High Perf. Comput. (HiPC)*, Dec. 2018, pp. 72–81.

[67] I. Bogle, K. Devine, M. Perego, S. Rajamanickam, and G. M. Slota, "A parallel graph algorithm for detecting mesh singularities in distributed memory ice sheet simulations," in *Proc. of the 48th Int. Conf. on Parallel Process.*, 2019, pp. 1–10.

[68] R. E. Moraes and C. C. Ribeiro, "Power optimization in ad hoc wireless network topology control with biconnectivity requirements," *Comput. & Operations Res.*, vol. 40, no. 12, pp. 3188–3196, Dec. 2013.

[69] C. A. R. Pinheiro, *Social Network Analysis in Telecommunications*. John Wiley & Sons, 2011, vol. 37.

[70] A. Frank, "Connectivity and network flows," in *Handbook of Combinatorics*. Amsterdam, The Netherlands and New York, NY, USA: Elsevier, 1995, vol. 1, pp. 111–177.

[71] V. Ramachandran, *Parallel Open Ear Decomposition with Applications to Graph Biconnectivity and Triconnectivity*. Princeton, NJ, USA: Citeseer, 1992.

[72] J. Cheriyan and R. Thurimella, "Algorithms for parallel k-vertex connectivity and sparse certificates," in *Proc. of the Twenty-Third Annual ACM Symp. on Theory of Comput.*, 1991, pp. 391–401.

[73] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. of the ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973.

[74] A. Buluç, S. Beamer, K. Madduri, K. Asanovic, and D. Patterson, "Distributed-memory breadth-first search on massive graphs," 2017, *arXiv: 1705.04590*.

[75] S. Stergiou, D. Rughwani, and K. Tsioutsiouliklis, "Shortcutting label propagation for distributed connected components," in *Proc. of the Eleventh ACM Int. Conf. on Web Search and Data Mining*, 2018, pp. 540–546.

[76] P. B. Gibbons, "A more practical PRAM model," in *Proc. of the First Annu. ACM Symp. on Parallel Algorithms and Architectures*, 1989, pp. 158–168.

[77] C. C.-Y. Chen, S. K. Das, and S. G. Akl, "A unified approach to parallel depth-first traversals of general trees," *Inf. Process. Letters*, vol. 38, no. 1, pp. 49–55, Apr. 1991.

[78] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Math.*, vol. 6, no. 1, pp. 29–123, Jan. 2009.

[79] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. and Inf. Syst.*, vol. 42, no. 1, pp. 181–213, 2015.

[80] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Twenty-ninth AAAI Conf. on Artif. Intell.*, Austin, Texas USA, Mar. 2015.

[81] J. Kunegis, "Konect: The koblenz network collection," in *Proc. of the 22nd Int. Conf. on World Wide Web*, 2013, pp. 1343–1350.

[82] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. of the 2004 SIAM Int. Conf. on Data Mining*, 2004, pp. 442–446.