Parallel Graph Coloring Algorithms for Distributed GPU Environments

Ian Bogle and George M. Slota

Department of Computer Science

Rensselaer Polytechnic Institute

{*boglei,slotag*}*@rpi.edu*

Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam

Scalable Algorithms Department

Sandia National Laboratories, New Mexico

{*egboman,kddevin,srajama*}*@sandia.gov*

Abstract

Graph coloring is often used in parallelizing scientific computations that run in distributed and multi-GPU environments; it identifies sets of independent data that can be updated in parallel. Many algorithms exist for graph coloring on a single GPU or in distributed memory, but to the best of our knowledge, hybrid MPI+GPU algorithms have been unexplored until this work. We present several MPI+GPU coloring approaches based on the distributed coloring algorithms of Gebremedhin et al. and the shared-memory algorithms of Deveci et al. The on-node parallel coloring uses implementations in KokkosKernels, which provide parallelization for both multicore CPUs and GPUs. We further extend our approaches to compute distance-2 and partial distance-2 colorings, giving the first known distributed, multi-GPU algorithm for these problems. In addition, we propose a novel heuristic to reduce communication for recoloring in distributed graph coloring. Our experiments show that our approaches operate efficiently on inputs too large to fit on a single GPU and scale up to graphs with 76.7 billion edges running on 128 GPUs.

Keywords: graph coloring; distributed algorithms; GPU; combinatorial scientific computing

1. 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 [1]. More recently, parallel algorithms [2, 3] 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 [2, 4]; all data sharing a color can 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 [5] and to identify short circuits in printed circuit designs [6]. Partial distance-2 coloring in particular is used to color sparse Jacobian matrices [7]. 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. [2] 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 [8]. 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 [3]. 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.

1.1. 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 [9, 10] for on-node parallelism and Trilinos [11] 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.

1.2. Extension of Prior Work

This work is an extension of our previous distributed-memory coloring work published at the 2020 IA^3 Workshop [12]. 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. Background

2.1. 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.

2.2. Coloring Background

While minimizing the number of colors is NP-hard, serial coloring algorithms using greedy heuristics have been effective for many applications [13]. 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 [14]. These orderings are highly sequential and do not allow much parallelism. However, relaxations of those orderings can allow some parallelism [15]. 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. 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 [16]. Osama et al. [17] 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" [13, 18], 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. [13], Çatalyürek et al. [18] and Rokos et al. [19] present shared-memory implementations based on the speculate and iterate approach. Deveci et al. [2] present implementations based on the speculate and

iterate approach that are scalable on a single GPU. Distributed-memory algorithms such as those in [3, 20] use the speculate and iterate approach. Grosset et al. [21] 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. [22] demonstrated how to color very large, dynamic graphs efficiently. Besta et al. [14] 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. [3] 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.

2.4. 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. [3] 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 interior 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.

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
```

3. 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 [11] for distributed MPI-based parallelism and Kokkos [9] for on-node parallelism. KokkosKernels [10] 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 aforementioned work of Bozdağ et al. [3]. In our approach, we color all *local* vertices first. 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 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
```

Algorithm 3 demonstrates the general approach for our three speculative distributed algorithms, while Figure 1 shows our approach in action for a small example graph with two processes.

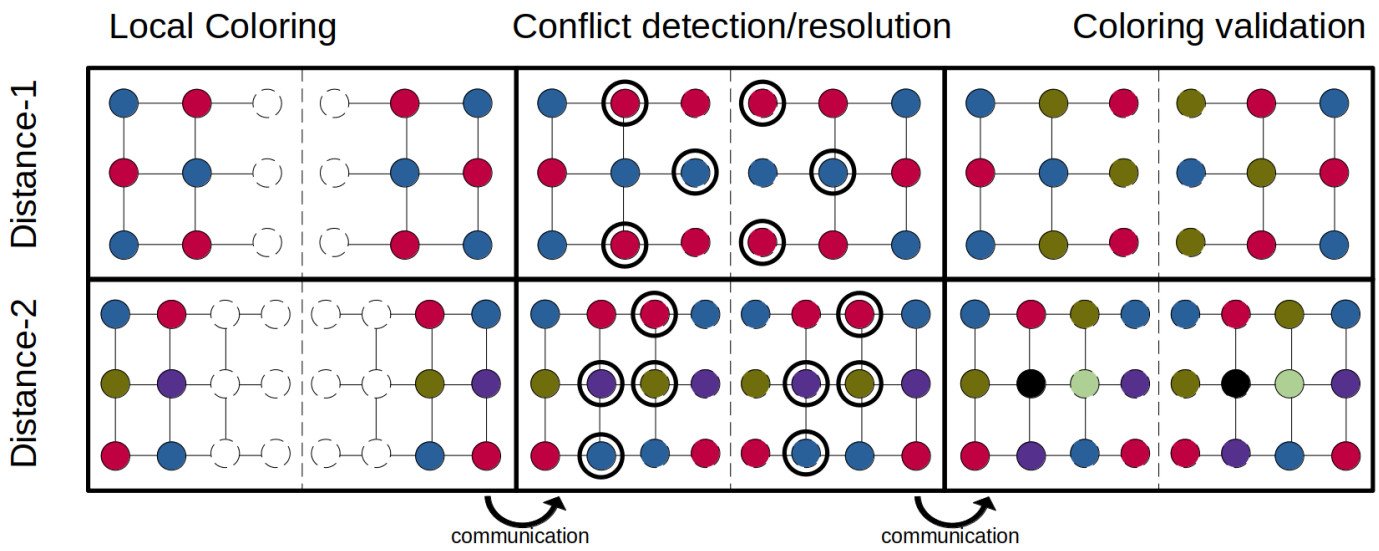


Figure 1: Example of our implementation of the distributed-memory coloring framework on a small graph with two processes. First, each process colors their local graph in shared-memory and communicates local colors to ghost copies. Second, we detect conflicts consistently and recolor the vertices local 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.

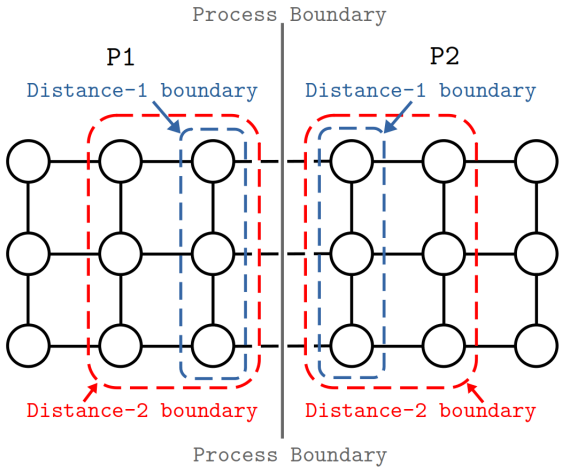


Figure 2: Definition of boundary vertex sets for different coloring instances

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.

3.1. Distributed Boundaries

Figure 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.

3.2. 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. [2] VB_BIT and EB_BIT in KokkosKernels [10]. 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 degree: we use EB_BIT for graphs with maximum degree greater than 6000; otherwise, we use VB_BIT.

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]. We perform a full exchange of

[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 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 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
```

boundary vertices' colors using collective communication functions implemented in the Zoltan2 package of Trilinos [11]. 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.

3.3. 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 [12, 3] 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.

3.4. 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.

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.

3.5. 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. [23]. 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), which we precompute. As with distance-1 conflict detection, we identify all 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

[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 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 Network	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 Tests	12.5 M – 12.8 B	75 M – 76.7 B	6	6	1.2 GB – 1.1 TB

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
```

D1 then also works for D2 — we recolor all conflicts, replace the old ghost colors, and then communicate local changes.

3.6. 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.

3.7. 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 [24] 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.

4. 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. [25] and Venkatesh et al. [26] 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 1. Most of the graphs are from the SuiteSparse Matrix Collection [27]. 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 [28]. 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 [8], 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 HPC-Graph [29] for efficient I/O.

We compare our implementation against distributed distance-1 and distance-2 coloring in the Zoltan [30] package of Trilinos. Zoltan's implementations are based directly on Bozdağ et al. [3]. 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 [31], as we use subroutines for on-node coloring from Deveci et al. [2]. 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.

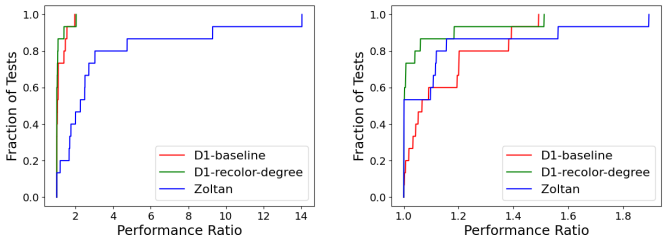
5. Results

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 Xtra-PuLP 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.

5.1. Distance-1 Performance

We summarize the performance of our algorithms relative to Zoltan using the performance profiles in Figure 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).

Figure 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 1.



(a) Runtime performance profile

(b) Color performance profile

We ran D1-baseline, D1-recolor-degree and Zoltan with 128 MPI ranks to color the 15 SuiteSparse graphs in Table 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 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 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 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).

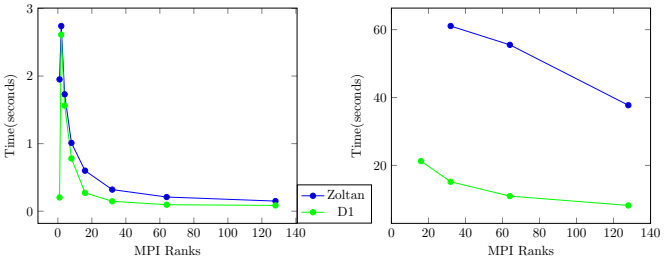
Because the performance of D1-recolor-degree is generally better than that of D1-baseline, all further distance-1 coloring results use D1-recolor-degree, and we refer to D1-recolor-degree as D1 going forward.

5.2. *Distance-1 Strong Scaling*

Figure 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.

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) [32], 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.

Figure 4: Zoltan and D1 strong scaling on select (a) PDE and (b) Social Network graphs.

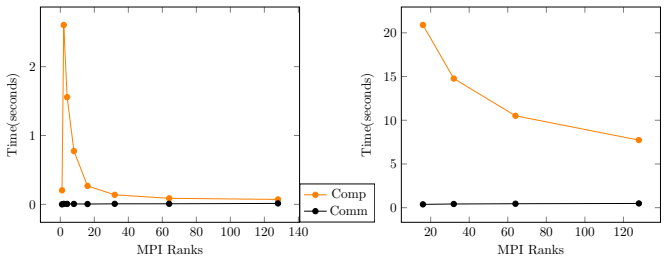


(a) Queen_4147 (b) com-Friendster

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 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 5b shows that D1 scales to more ranks on com-Friendster, primarily because of the graph's larger size.

Figure 5: D1 communication time (Comm) and computation time (Comp) from 1 to 128 GPUs.



(a) Queen_4147 (b) com-Friendster

5.3. 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 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 6: Weak scaling of D1 on 3D mesh graphs. Tests use 12.5, 25, 50, and 100 million vertices per GPU.

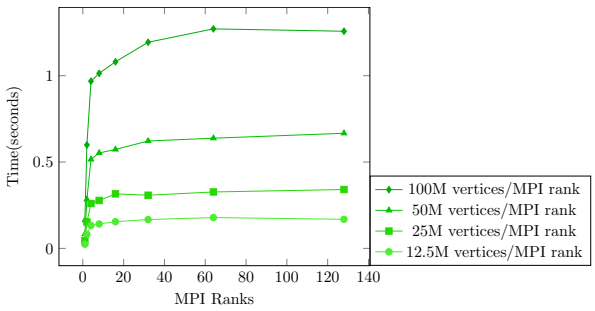


Figure 6 shows that the single rank runs for each workload are similar, indicating that communication and recoloring overhead are the dominant scaling factors for this study. In increasing the boundary size by a factor of two, we do not necessarily increase the number of distributed conflicts by two, 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.

5.4. D1-2GL Performance

In general, D1-2GL reduces the number of collective communications used in the distributed distance-1 coloring. Figure 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 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 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.

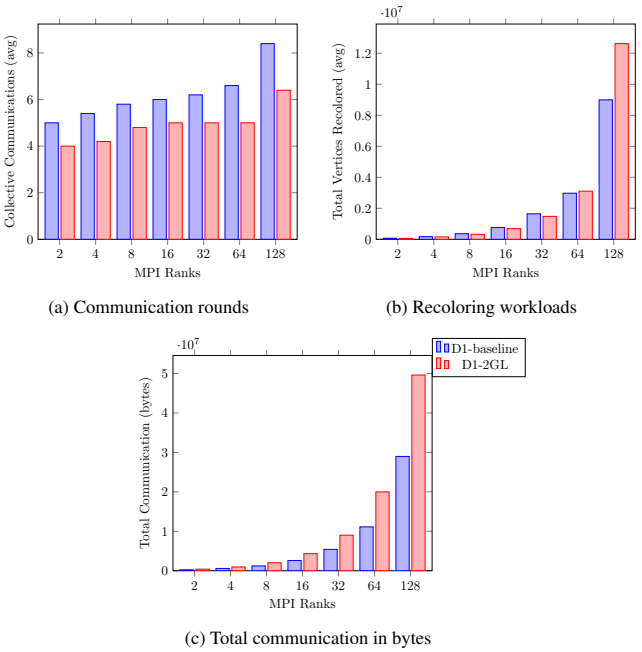
Despite the observed additional overhead, on Queen_4147 D1-2GL is able to see a speedup over D1-baseline due to the reduction in rounds, which reduces serialization of recoloring work. Unfortunately, for highly structured graphs such as those we used for our weak scaling experiments, D1-baseline does 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.

5.5. Distance-2 Performance

We compare our D2 method to Zoltan's distance-2 coloring using eight graphs from Table 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 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 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.

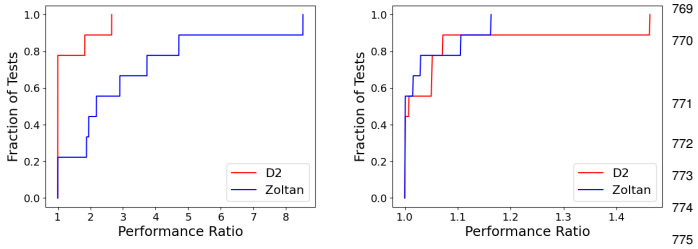
Figure 7: D1 vs D1-2GL comparisons



(a) Communication rounds (b) Recoloring workloads

(c) Total communication in bytes

Figure 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 1.



(a) Runtime performance profile (b) Color performance profile

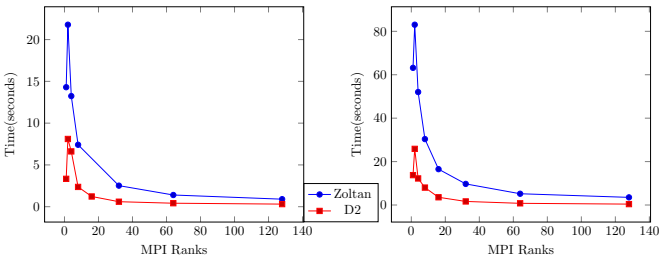
5.6. *Distance-2 Strong Scaling*

Figures 9a and 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.

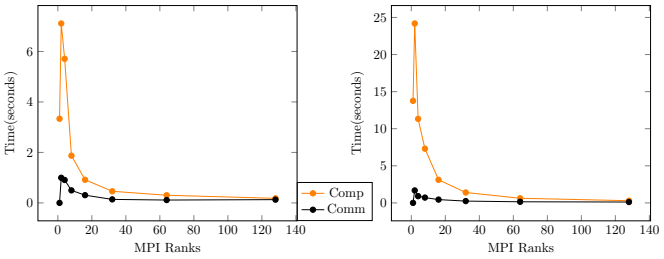
Figures 10a and 10b show the communication and computation breakdown of D2 on Bump_2911 and Queen_4147. Bump_2911 shows computation and communication scaling for up to 128 ranks, while color usage increases by only 0.6%. In general, the relative increase in color usage from a single rank for distance-2 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.

Figure 9: D2 and Zoltan strong scaling for distance-2 coloring.



(a) Bump_2911 (b) Queen_4147

Figure 10: D2 communication time (comm) and computation time (comm) from 1 to 128 GPUs.



(a) Bump_2911 (b) Queen_4147

5.7. *Distance-2 Weak Scaling*

Figure 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 11: Distance-2 weak scaling of D2 on 3D mesh graphs.

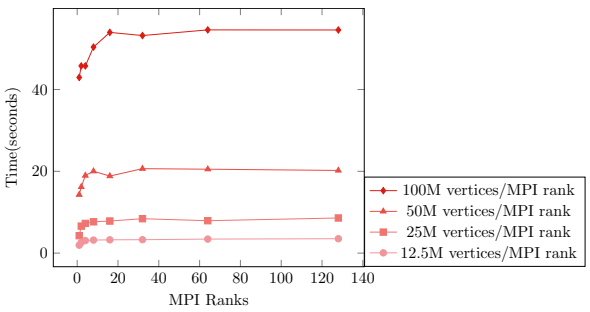


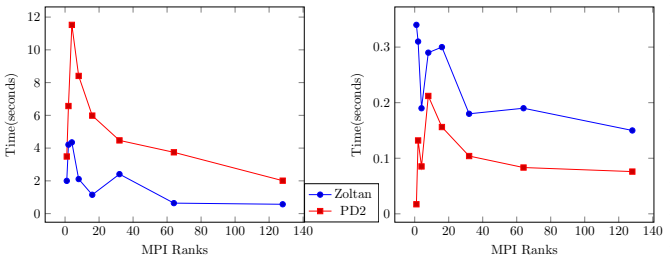
Table 2: Summary of the graphs used for PD2 tests. Statistics are for the bipartite representation of the graph (Section 3.6). δ_{avg} is average degree and δ_{max} is maximum degree. Numeric values listed are after preprocessing to remove multi-edges and self-loops. k = thousand, M = million

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

5.8. Partial Distance-2 Strong Scaling

Table 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 are representative of application use cases. We give metrics reported for the bipartite representation of the graph (as described in Section 3.6). 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 12: PD2 strong scaling for partial distance-2 coloring.

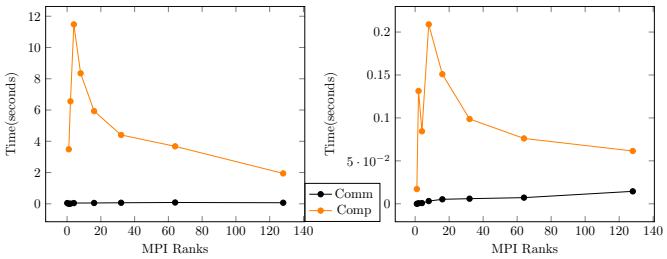


(a) patents (b) Hamrle3

Figure 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 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 13: PD2 communication time (comm) and computation time (comp) from 1 to 128 GPUs



(a) patents (b) Hamrle3

Figure 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 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 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.

6. Future work

We have presented new multi-GPU distributed memory implementations of distance-1, distance-2 and partial distance-2 graph coloring. These methods enable parallel graph coloring for graphs too large to fit into a single GPUs memory; our weak-scaling results demonstrate the distance-1 coloring of a graph with 12.8 billion vertices and 76.7 billion edges in less than two seconds. We introduced a new recoloring heuristic based on vertex degrees that reduces the amount of recoloring needed in parallel coloring methods. We showed that our approaches are scalable to 128 GPUs and produce colorings with quality similar to or better than Zoltan's distributed memory coloring algorithms.

Because our coloring algorithms use the Kokkos and KokkosKernels library for on-node performance portability, our MPI+X methods can also run on distributed-memory computers with multicore (CPU-based) nodes. In this work, we fo-

cused on GPU architectures; exploring multicore performance is future work.

We are currently integrating this code into the Zoltan2 package of Trilinos. Our goal is to deliver a complete suite of MPI+X algorithms for distance-1, distance-2, and partial distance-2 coloring in Zoltan2. We will modify PD2 to allow it to color only vertices of interest to the application as Zoltan does.

Additionally, it is possible that the distributed framework we use for graph coloring could be generalized to other graph problems as well, such as graph matching or label propagation. These problems all have similar local computation and communication patterns. Parallel graph matching in particular would require similar conflict resolution considerations, and it might similarly benefit from degree-based heuristics. Exploring these generalizations is reserved for future work.

We also will investigate further optimizations to increase performance. Zoltan's implementation currently uses asynchronous point-to-point communication and batched coloring on the boundary, which allows some overlap of communication and computation. Optimizing our coloring routines and distributed communications via incorportating GPU-Direct, CUDA streams, asynchronous operations, and non-blocking and/or point-to-point communications may offer additional benefit for us. These changes could increase performance for D1, D2, and PD2, as well as make D2 and PD2 more scalable on skewed graphs.

7. Acknowledgments

We thank the Center of Computational Innovations at RPI for maintaining the equipment used in this research, including the AiMOS supercomputer supported by the National Science Foundation under Grant No. 1828083. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] D. Brélaz, New methods to color the vertices of a graph, Communications of the ACM 22 (4) (1979) 251–256.

[2] M. Deveci, E. G. Boman, K. D. Devine, S. Rajamanickam, Parallel graph coloring for manycore architectures, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 892–901.

[3] D. Bozdağ, A. H. Gebremedhin, F. Manne, E. G. Boman, U. V. Catalyurek, A framework for scalable greedy coloring on distributed-memory parallel computers, Journal of Parallel and Distributed Computing 68 (4) (2008) 515–535.

[4] J. Allwright, R. Bordawekar, P. Coddington, K. Dincer, C. Martin, A comparison of parallel graph coloring algorithms, SCCS-666 (1995) 1–19.

[5] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, A. Pothen, Colpack: Software for graph coloring and related problems in scientific computing, ACM Transactions on Mathematical Software (TOMS) 40 (1) (2013) 1.

[6] M. Garey, D. Johnson, H. So, An application of graph coloring to printed circuit testing, IEEE Transactions on Circuits and Systems 23 (10) (1976) 591–599.

[7] A. H. Gebremedhin, F. Manne, A. Pothen, What color is your Jacobian? graph coloring for derivatives., SIAM Review 47 (4) (2005) 629—-705.

[8] A. H. Gebremedhin, A. Walther, An introduction to algorithmic differentiation, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 10 (1) (2020) e1334.

[9] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202–3216.

[10] Kokkos Kernels (2017).
URL `https://github.com/kokkos/kokkos-kernels`

[11] 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 Transactions on Mathematical Software (TOMS) 31 (3) (2005) 397–423.

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

[13] A. H. Gebremedhin, F. Manne, Scalable parallel graph coloring algorithms, Concurrency: Practice and Experience 12 (12) (2000) 1131–1146.

[14] M. Besta, A. Carigiet, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, T. Hoefler, High-performance parallel graph coloring with strong guarantees on work, depth, and quality, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2020, pp. 1–17.

[15] W. Hasenplaugh, T. Kaler, T. B. Schardl, C. E. Leiserson, Ordering heuristics for parallel graph coloring, in: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 166–177. `doi:10.1145/2612669.2612697`.
URL `https://doi.org/10.1145/2612669.2612697`

[16] M. T. Jones, P. E. Plassmann, A parallel graph coloring heuristic, SIAM Journal on Scientific Computing 14 (3) (1993) 654–669.

[17] M. Osama, M. Truong, C. Yang, A. Buluç, J. D. Owens, Graph coloring on the GPU, in: GrAPL: Workshop on Graphs, Architectures, Programming, and Learning (IPDPSW), 2019.
URL `http://eecs.berkeley.edu/~aydin/coloring.pdf`

[18] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, A. Pothen, Graph coloring algorithms for multi-core and massively multithreaded architectures, Parallel Computing 38 (10-11) (2012) 576–594.

[19] G. Rokos, G. Gorman, P. H. Kelly, A fast and scalable graph coloring algorithm for multi-core and many-core architectures, in: European Conference on Parallel Processing, Springer, 2015, pp. 414–425.

[20] A. E. Sariyüce, E. Saule, Ü. V. Çatalyürek, Scalable hybrid implementation of graph coloring using MPI and OpenMP, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE, 2012, pp. 1744–1753.

[21] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, M. Hall, Evaluating graph coloring on GPUs, ACM SIGPLAN Notices 46 (8) (2011) 297–298.

[22] S. Sallinen, K. Iwabuchi, S. Poudel, M. Gokhale, M. Ripeanu, R. Pearce, Graph colouring as a challenge problem for dynamic graph processing on distributed systems, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, IEEE Press, 2016.

[23] M. K. Taş, K. Kaya, E. Saule, Greed is good: Optimistic algorithms for bipartite-graph partial coloring on multicore architectures, arXiv preprint arXiv:1701.02628 (2017).

[24] G. M. Slota, S. Rajamanickam, K. Devine, K. Madduri, Partitioning trillion-edge graphs in minutes, in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 646–655.

[25] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, D. K. Panda, Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus, in: 2013 42nd International Conference on Parallel Processing, 2013, pp. 80–89. `doi:10.1109/ICPP.2013.17`.

[26] A. Venkatesh, K. Hamidouche, H. Subramoni, D. K. Panda, Offloaded gpu collectives using core-direct and cuda capabilities on infiniband clusters, in: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), 2015, pp. 234–243. `doi:10.1109/HiPC.2015.50`.

[27] T. A. Davis, Y. Hu, The university of florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1:1–1:25. `doi:10.1145/2049662.2049663`.
URL `http://doi.acm.org/10.1145/2049662.2049663`

[28] R. L. Brooks, On colouring the nodes of a network, in: Mathematical Proceedings of the Cambridge Philosophical Society, Vol. 37, Cambridge University Press, 1941, pp. 194–197.

[29] G. M. Slota, S. Rajamanickam, K. Madduri, A case study of complex graph analysis in distributed memory: Implementation and optimization, in: International Parallel & Distributed Processing Symposium (IPDPS), 2016.

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

[31] M. Naumov, P. Castonguay, J. Cohen, Parallel graph coloring with applications to the incomplete-lu factorization on the gpu, Tech. rep., NVidia White Paper (2015).

[32] Graph 500 (2021).
URL `https://graph500.org/`