SAND2020-9530C

Distributed Memory Graph Coloring Algorithms for Multiple GPUs

Ian Bogle
Rensselaer Polytechnic Institute
Troy, NY
boglei@rpi.edu

Erik G. Boman, Karen Devine, Sivasankaran Rajamanickam
Sandia National Laboratories
Albuquerque, NM
{egboman,kddevin,srajama}@sandia.gov

George M. Slota
Rensselaer Polytechnic Institute
Troy, NY
slotag@rpi.edu

***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 hybrid MPI+GPU algorithms have been unexplored until this work, to the best of our knowledge. We present several MPI+GPU coloring approaches that use implementations of 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 solve for distance-2 coloring, giving the first known distributed and multi-GPU algorithm for this problem. In addition, we propose novel methods to reduce communication 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.**

***Index Terms*—graph coloring; distributed algorithms; GPU;**

I. Introduction

We present new multi-GPU, distributed memory implementations of distance-1 and distrance-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. 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 [5].More recently, parallel algorithms [4], [9] 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], [9]; 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 [13] and to identify short circuits in printed circuit designs [12]. 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. [9] 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 [14]. For such applications, assembling the associated graphs on a single node to run a sequential coloring algorithm may not be feasible [4]. As such, we focus on running our algorithms on the parallel architectures used by the underlying applications. These architectures typically are highly distributed, with multiple CPUs and/or GPUs per node. Therefore, we specifically consider coloring algorithms that can use the “MPI+X” paradigm, where “X” is multicore CPU or GPU acceleration.

A. Contributions

We present and examine two MPI+X implementations of distance-1 coloring as well as one MPI+X implementation of distance-2 coloring. In order to run on a wide variety of architectures, we use the Kokkos performance portability framework [1], [11] for on-node parallelism and Trilinos [17] 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. However, 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 see up to 28x speedup on 128 GPUs compared to a single GPU, and only a 7.54% increase in colors on average. For distance-2 coloring, our algorithm sees up to 27.6x speedup, and a 4.9% increase in colors in the worst case. We also demonstrate good weak scaling behavior up to 128 GPUs on graphs up to 13 billion vertices and 76.7 billion edges in size. **added this last sentence. I think it'd be good to err on over-emphasizing our ability to process graphs much much larger than can fit in a single node. What's the largest graph colored before in distributed memory?**

II. Background

A. Coloring Problem

While there exist many definitions of the “graph coloring problem,” we specifically consider variants of distance-1 and

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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-NA0003525.

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 so 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. 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.

B. Coloring Background

While minimizing the number of colors is NP-hard, serial coloring algorithms using greedy heuristics have been effective for many applications [15]. 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 $G = (V, E)$)
    $C(\forall v \in V) \leftarrow 0$        ▷ Initialize all colors as null
    for all $v \in V$ in some order do
        $c \leftarrow$ the smallest color not used by a neighbor of $v$
        $C(v) \leftarrow c$
```

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, 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.

C. Parallel Coloring Algorithms

There are two popular approaches to parallel graph coloring. The first is to concurrently find independent sets of vertices and then concurrently color all of the vertices in each set; this approach was used by Jones and Plassmann [19]. The second, referred to as "speculate and iterate" [7], is to color as many vertices as possible in parallel and then iteratively fix conflicts in the resulting pseudo-coloring until no conflicts remain. Çatalyürek et al. [7] and Rokos et al. [20] present shared-memory implementations based on the speculate and iterate approach. Distributed-memory and hybrid CPU+GPU algorithms such as those in [4], [16], [22] use the speculate and iterate approach. Bozdağ et al. [4] showed that, in distributed memory, this approach is more scalable than methods based on the independent set approach of Jones and Plassmann.

D. 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, edges between local vertices, and E_g, edges containing at least one ghost vertex as an endpoint. Bozdağ et al. [4] 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).

III. METHODS

We present three hybrid MPI+GPU algorithms, called Distance-1 (D1), Distance-1 Two Ghost Layer (D1-2GL) and Distance-2 (D2). D1 and D1-2GL solve the distance-1 coloring problem and D2 does distance-2 coloring. We leverage Trilinos [17] for distributed MPI-based parallelism and Kokkos [11] for on-node parallelism. KokkosKernels [1] provides baseline implementations of distance-1 and distance-2 coloring algorithms that we use and modify for our local coloring and recoloring subroutines.

Our three proposed algorithms follow the same basic framework, which builds upon that of Bozdağ et al. [4]. Bozdağ et al. observe that interior vertices can be properly colored independently on each process without creating conflicts or requiring communication. They propose first coloring interior vertices, and then coloring boundary vertices in small batches over multiple rounds involving communication between processes. This approach can reduce the occurrence of conflicts, which in turn reduces the amount of communication necessary to properly color the boundary. In our approach, we color all *local* vertices first. Then we fix all conflicts after communication of boundary vertices' colors. 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, and it allowed us to use existing parallel coloring routines in KokkosKernels without substantial modification.

Algorithm 2 demonstrates the general approach for our three speculative distributed algorithms. 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

Algorithm 2 Distributed-Memory Speculative Coloring

needed to modify this algorithm so it fits with alg 3 and the text

```
procedure PARALLEL-COLOR(Graph $G = (V, E)$)
    Color all local vertices
    Communicate colors of boundary vertices
    do
        Detect conflicts
        Recolor conflicting vertices
        Communicate updated boundary colors
    while Conflicts exist
```

of conflicted vertices. Finally, processes locally recolor all uncolored vertices, communicate updates, detect conflicts, and repeat until no conflicts are found.

A. Distance-1 Coloring (D1)

a) Local Coloring: Our D1 method begins by independently coloring all local vertices on each process using GPU-enabled algorithms by Deveci et al. [9]. These algorithms include VB_BIT and EB_BIT in KokkosKernels [1]. VB_BIT uses vertex-based parallelism; each vertex is colored by a single thread. VB_BIT 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. This cutoff was selected experimentally.

b) Conflict Detection: After the initial coloring, only boundary vertices can be in conflict with one another[1]. We perform a full exchange of boundary vertices' colors using Trilinos [17]. Specifically, we use the FEMultiVector class of Tpetra [18] to communicate the colors of boundary vertices to their ghost copies on other processes via an all-to-all exchange. After each process receives its ghosts' colors, it detects conflicts by checking every owned vertex's color against the colors of its neighbors. Our conflict-finding process uses vertex-based parallelism and is parallelized 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. Our conflict resolution approach is illustrated in Algorithm 3, which takes place in the inner loop of Algorithm 2. Note that this algorithm runs on each process using its owned local graph G_l.

[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 3 Distance-1 conflict resolution and recoloring

```
procedure RESOLVE-CONFLICTS(
  Local Graph $G_l = (V_l + V_g, E_l + E_g)$, colors, GID)
    $gc \leftarrow$ current colors of all ghosts
    conflicts $\leftarrow 0$
    for all $v \in V_l$ do in parallel
        for all $\langle v, n \rangle \in (E_l + E_g)$ do        ▷ Neighbors of $v$
            if colors[$v$] = colors[$n$] then
                conflicts $\leftarrow$ conflicts + 1
                if rand(GID[$v$]) > rand(GID[$n$]) then
                    colors[$v$] $\leftarrow 0$
                    break
                else if rand(GID[$v$]) < rand(GID[$n$]) then
                    colors[$n$] $\leftarrow 0$
                else
                    if GID[$v$] > GID[$n$] then
                        colors[$v$] $\leftarrow 0$
                        break
                    else
                        colors[$n$] $\leftarrow 0$
    Do the following changes make sense?
    Allreduce(conflicts, SUM)          ▷ Get global conflicts
    if conflicts > 0 then
        colors = Color($G_l$, colors)          ▷ Recolor vertices
        Replace ghost colors with $gc$
        Communicate recolored vertices to ghost copies
    return conflicts
```

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. We adopt the random conflict resolution scheme of Bozdağ et al. We use a random number generator (given as the "rand" function in Algorithm 3) seeded by the GID of each conflicted vertex, as this produces a consistent set of random numbers across processes without communication. In a conflict, the vertex with the larger random number is chosen for recoloring. For the rare case in which both random numbers are equal, the tie is broken based on GID. Using random numbers instead of simply using GIDs helps balance recoloring workload across processes.

c) Recoloring: 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 3) on a given process. Then we give color zero to all vertices that will be recolored; KokkosKernels interprets color zero as uncolored.

To prevent KokkosKernals 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.

B. Two Ghost Layers Coloring (D1-2GL)

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.

We use to reduce the total amount of communication relative to D1 for certain inputs by reducing the total number of recoloring rounds needed. In particular, for mesh or otherwise regular graph inputs, the second ghost layer is primarily made up on 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 can be more expensive, because a larger boundary from each process is communicated. Also, in irregular graph inputs, 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. For the extra ghost layer to pay off, it must reduce the number of rounds of communications enough to make up for the increased cost of each communication. We discuss this tradeoff in our results.

To construct the second ghost layer on each process, processes exchange the adjacency lists of their boundary vertices; this step is needed only once. After the ghosts' connectivity information is added, we use the same coloring approach as in D1. However, we optimize our conflict detection by looking through only the ghost vertices' adjacencies, as they neighbor all local boundary vertices. By keeping the new ghost adjacency information separate from the local graph, we can detect all conflicts by examining only the edges between ghosts and their neighbors. **could we have done this optimization in D1? would it have been faster than looping over the local graph? don't necessarily need to discuss here, but KDD is curious about the answer**

C. 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 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. [24] 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. However, to find a distance-2 conflict for a given vertex, its entire two-hop neighborhood must be checked for potentially conflicting colors.

Algorithm 4 Distance-2 conflict detection

```
procedure DETECT-DISTANCE2-CONFLICTS(
  Local Graph $G_l = (V_l + V_g, E_l + E_g)$, colors, GID)
  for all $v \in V_l$ do in parallel
    for all $\langle v,n \rangle \in (E_l + E_g)$ do        ▷ Neighbors of $v$
      if colors[$v$] = colors[$n$] then
        if rand(GID[$v$]) > rand(GID[$n$]) then
          colors[$v$] ← 0
          break
        else if rand(GID[$n$]) > rand(GID[$v$]) then
          colors[$n$] ← 0
        else
          if GID[$v$] > GID[$n$] then
            colors[$v$] ← 0
            break
          else
            colors[$n$] ← 0
      for all $\langle n,m \rangle \in (E_l + E_g)$ do    ▷ Neighbors of $n$
GMS - Ian should check this. I just modified notation but left algorithm alone.
        if colors[$v$] = colors[$m$] then are there any issues with race conditions here? m could be a boundary vertex, not a ghost; does setting its color to zero cause correctness problems if it is a boundary vertex? could it cause excessive recoloring (m set to zero by one thread while another thread is setting m's neighbors to zero)?
          if rand(GID[$v$]) > rand(GID[$m$]) then
            colors[$v$] ← 0
            break
          else if rand(GID[$m$]) > rand(GID[$v$]) then
            colors[$m$] ← 0
          else
            if GID[$v$] > GID[$m$] then
              colors[$v$] ← 0
              break
            else
              colors[$m$] ← 0
```

TABLE I: Summary of input graphs. δ_{avg} refers to average degree and δ_{max} refers to maximum degree. Numeric 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}
ldoor	PDE Problem	0.9 M	21 M	45	77
Audikw_1	PDE Problem	0.9 M	39 M	81	345
Bump_2911	PDE Problem	2.9 M	63 M	43	194
Queen_4147	PDE Problem	4.1 M	163 M	78	89
hollywood-2009	Social Network	1.1 M	57 M	99	12 k
soc-LiveJournal1	Social Network	4.8 M	43 M	18	20 k
com-Friendster	Social Network	66 M	1.8 B	55	5.2 k
europe_osm	Road Network	51 M	54 M	2.1	13
indochina-2004	Web Graph	7.4 M	194 M	26	**add value here**
MOLIERE_2016	Document Mining Network	30 M	3.3 B	80	**add value here**
rgg_n_2_24_s0	Random Graph	17 M	133 M	15	40
mycielskian19	Random Graph	393 k	452 M	2.3 k	**add value here**
mycielskian20	Random Graph	786 k	1.4 B	3.4 k	**add value here**
hexahedral	Weak Scaling Tests	12.5 M – 13 B	75 M – 77 B	6	6

What about the twitter graph? I saw 13 mentioned as number of graphs. Was twitter one of those? Which one didn't we use from above?

Algorithm 4 shows the straightforward way in which we detect conflicts 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. As with distance-1 conflict detection, we identify all local conflicts and use a random number generator to ensure that vertices to be recolored are chosen consistently across processes. The iterative recoloring of D1 then also works for D2; we recolor all conflicts, replace the old ghost colors, and then communicate local changes.

D. 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 [23] to partition our input 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 input 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.

IV. Experimental Setup

We performed scaling experiments on the AiMOS supercomputer housed at Rensselaer Polytechnic Institute. The system has 268 nodes, each equipped with 2 IBM Power 9 processors clocked at 3.15 GHz, 4 NVIDIA Tesla V100 GPUs with 16 GB of memory each, 512 GB of RAM, and 1.6 TB Samsung NVMe Flash memory, and connected together with a Mellanox Infiniband interconnect.

The input graphs we used are listed in Table I. We used graphs from the SuiteSparse Matrix Collection (formerly UFL Sparse Matrix Collection [8]). 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 at most δ_{max} colors [6]. 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 [14], which is a target application for graph coloring algorithms. We also include social network graphs to demonstrate scaling of our methods on irregular real-world datasets. We preprocessed all graphs to remove multi-edges and self-loops.

We compare our implementation against the distributed distance-1 and distance-2 coloring in the Zoltan [10] package of Trilinos. Zoltan's implementations are based directly on Bozdağ et al. [4]. For our results, we ran Zoltan and our approaches with four MPI ranks per node on AiMOS, and used the same partitioning method across all of our comparisons. Our methods D1, D1-2GL, and D2 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. We restrict Zoltan to four MPI ranks per node, 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.

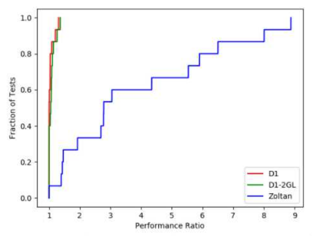
V. Results

For our experiments, we compare the performance of our methods to Zoltan for distance-1 and distance-2 coloring. Our performance metrics include execution time, parallel scaling, number of colors used, and the number of communication rounds. We do not include the partitioning time for XtraPuLP; we assume target applications partition and distribute their graphs. Each of the results reported represent an average of five runs.

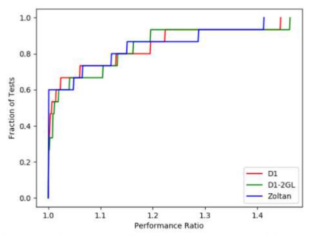
A. Distance-1 Performance

We summarize the performance of our algorithms relative to Zoltan using performance profiles. Performance profiles plot the number 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 for a given problem. In these plots, the line that is the highest is the best performing algorithm.

Fig. 1: Performance profiles comparing D1 and D1-2GL 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 graphs in Table I



(a) Runtime performance profile **plot needs axis labels**



(b) Color performance profile **plot needs axis labels**

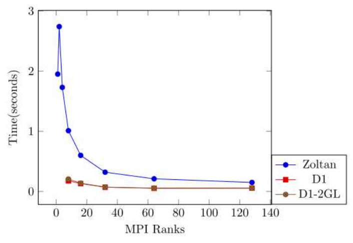
We ran D1, D1-2GL, and Zoltan with 128 MPI ranks to color all of the input graphs in Table I. D1 and D1-2GL used MPI plus 128 Tesla V100 GPUs, while Zoltan used MPI on 128 Power9 CPU cores across 32 nodes. Figure 1a shows that both D1 and D1-2GL outperform Zoltan in terms of execution time in these experiments. The D1 method is the fastest in roughly 50% of the cases, D1-2GL is fastest in roughly 40%, and Zoltan outperforms both in only a single instance. D1 has at most a 8.8x speedup over Zoltan (with the europe_osm graph) and at worst a 20% slowdown relative to Zoltan (with Audikw_1). D1-2GL shows similar speedups over Zoltan: an 8.8x relative speedup with europe_osm, and at worst an 8% slowdown with Audikw_1. D1 and D1-2GL are close in terms of execution times; the largest differences are with indochina-2004, which D1 is 36% faster, and with twitter7, which D1-2GL is 30% faster. In general, D1 and D1-2GL have performance differences of approximately 10%; it is difficult to say which approach will perform better on a given graph.

Figure 1b shows that Zoltan outperforms D1 and D1-2GL in terms of color usage. Zoltan uses fewer colors in over 60% of our experiments. In most cases, however, D1 and D1-2GL use no more than 20% more colors than Zoltan. With the Twitter7 graph, Zoltan uses 45% fewer colors than D1 and D1-2GL, but with Mycielskian20, D1 and D1-2GL use 41% fewer colors than Zoltan. On average, D1 uses 6.8% more colors than Zoltan. D1 and D1-2GL differ by 1% on average. The differences in the number of colors used exist because KokkosKernels uses different local coloring algorithms from Zoltan.

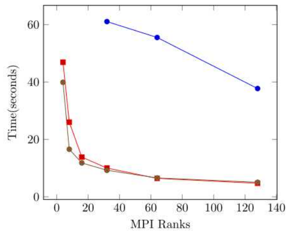
B. Distance-1 Strong Scaling

Figure 2 shows strong scaling times for Queen_4147 and com-Friendster. The D1 method scales better on the com-Friendster graph than on Queen_4147, as the GPUs can be more fully utilized with the larger com-Friendster graph. For Queen_4147, D1 is at least 2.7x faster than Zoltan for each run, and D1 uses 12% fewer colors than Zoltan in the 128 rank run. For com-Friendster, D1 is roughly 7x faster than Zoltan in the 128 rank run, but D1 uses 26% more colors than Zoltan in that case. For smaller rank runs, D1-2GL

Fig. 2: Zoltan, D1, and D1-2GL strong scaling



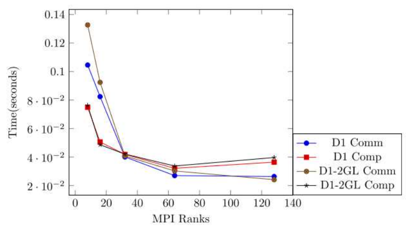
(a) Queen_4147



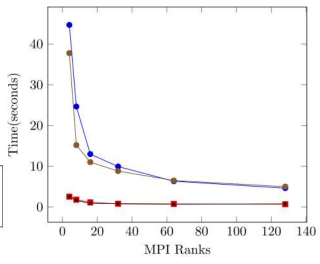
(b) com-Friendster

In general, it is difficult for a multi-GPU approach to show good strong scaling over a single GPU run due to communication overhead. Graphs that can fit into a single GPU do not provide sufficient work and parallelism for large numbers of GPUs, and multi-GPU execution incurs communication overhead. However, on average **average over what? all graphs? all rank configurations for these two graphs?**, D1 shows a 5.35x speedup over the single GPU run on 128 GPUs, while the D1-2GL approach sees an average speedup of 4.5x. On small or highly skewed graphs that fit on a single GPU we do not see speedup from a single GPU, due to communication overhead or underutilizing the GPUs.
what does "on average" mean in the next sentence? average over all graphs using 128 ranks? average on these two graphs over all rank configurations? On average, D1 sees a 60% increase in the number of colors from a single GPU run, while D1-2GL sees an average 61% increase. **what about Zoltan?** 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 our approaches and the Zoltan implementation have trouble coloring these graphs, but our D1 and D1-2GL implementations color these graphs in fewer colors than Zoltan. Without these two outliers, D1 sees an average 7.54% increase in color usage, while D1-2GL sees a 7.24% increase. **increase over what? single GPU?**

Fig. 3: D1 and D1-2GL communication time (comm) and computation time (comp)



(a) Queen_4147



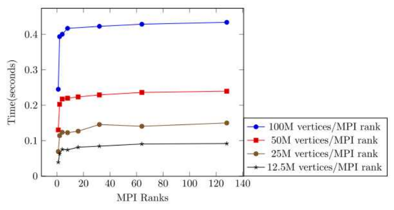
(b) com-Friendster

Figure 3 shows the total communication and computation time associated with each run. For the Queen_4147 graph, the computation dominates the total execution time for larger numbers of ranks. As the number of ranks increases, there is no computational benefit to adding more GPUs to the problem. **the previous two sentences seem contradictory. If computation time is dominating, adding GPUs should help, right? Is computation time really dominating, or is it kernel launch time? with large numbers of ranks, are we doing as well as is possible given the device latency?** The com-Friendster graph also shows computational scaling, but in this case communication is the dominant factor of the execution time.

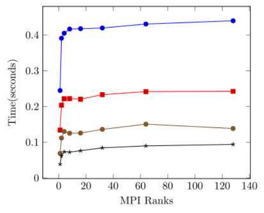
C. Distance-1 Weak Scaling

This weak-scaling study was conducted with uniform 3D hexahedral meshes. The meshes are partitioned with block partitioning along a single axis, resulting in the mesh being partitioned into slabs. Larger meshes were generated by doubling the number of elements in a single dimension in order to keep the per-process communication and computational workload the same.

Fig. 4: Distance-1 weak scaling on 3D mesh graphs



(a) D1 approach



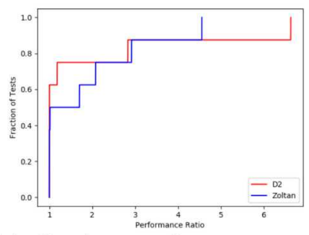
(b) D1-2GL approach

Figures 4a and 4b show that the weak scaling behavior for both D1 and D1-2GL is very consistent. For both methods, there is an overall increase of roughly 0.04 seconds for each workload. There is no clear-cut winner between the two methods here; both are essentially the same. For these graphs, D1-2GL does reduce the number of rounds of communication **how much?**, but the extra communication overhead offsets this savings such that the execution time is very similar to D1. **are we sure that it is extra communication overhead and not something else? do we have a comm vs comp breakdown for D1-2GL?** Additionally, due to the regular structure of these graphs, D1 does not use many rounds of communication, so any time savings are minimal. **this discussion is weak; can we back it up with numbers?**

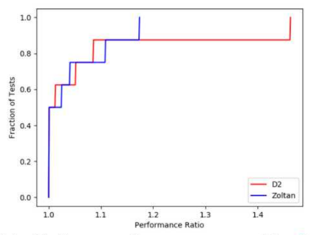
In general, the results section doesn't do a good job of comparing D1-2GL to D1. We don't show the reduction in number of rounds that we hypothesized. We don't characterize how much additional communication volume is incurred by the extra ghost layer. We don't show that the performance similarity is due to extra communication volume rather than additional ghost traversal. The only detailed comparison is done in this weak-scaling section on graphs that don't necessarily need many rounds of communication (somewhat nullifying the effect of the extra ghost layer).

D. Distance-2 Performance

Fig. 5: 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 graphs in Table I **axes need labels**



(a) Runtime performance profile **legend says D1; should be D2**



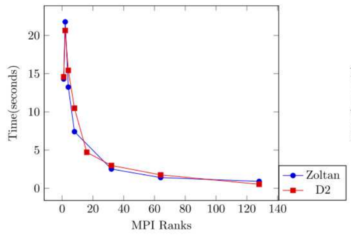
(b) Color performance profile **legend says D1; should be D2**

We compare our D2 method on 128 Tesla V100 GPUs with Zoltan's distance-2 coloring method on 128 Power9 cores using eight graphs from Table I: Bump_2911, Queen_4147, hollywood-2009, europe_osm, rgg_n_2_24_s0, ldoor, Audikw_1, and soc-LiveJournal1. Figure 5a shows that D2 compares well against Zoltan in terms of execution time. Zoltan's distributed algorithm uses local distance-2 coloring and a conflict detection scheme that requires only a single ghost layer. Additionally, to reduce conflicts, the boundary vertices are colored in small batches. The Zoltan is faster than D2 on three graphs: Audikw_1 (1.18x), hollywood-2009 (2.82x) and soc-LiveJournal1 (6.6x). In the best case, we see a 4.5x speedup over Zoltan on the europe_osm graph. Figure 5b shows that D2 has similar color usage to Zoltan. D2 and Zoltan each produce the lowest number of colors in half of the experiments. In all but one of the cases in which Zoltan uses fewer colors, D2 uses no more than 10% more colors. D2 uses 46% more colors with the soc-LiveJournal1 graph than Zoltan. These color usage differences are due to the differences in the D2 approach and Zoltan's distance-2 approach. We use vertex-based conflict detection, and because of that we require two ghost layers. Additionally, we do not color boundary vertices in rounds, so we expect D2 to have to resolve more conflicts. **the previous three sentences are weak. Sentence 1: "the usage differences are due to the approach differences" doesn't say anything. Sentence 2: Vertex-based conflict detection doesn't require two ghost layers; the additional two-hop check requires two ghost layers, right? But we don't show that having two ghost layers increases the number of colors; if anything, I thought it might reduce the number of colors as each processor has more info with which to make decisions. Sentence 3: we haven't shown that D2 requires more conflict resolution, nor have we shown that more conflict resolution results in more colors.**

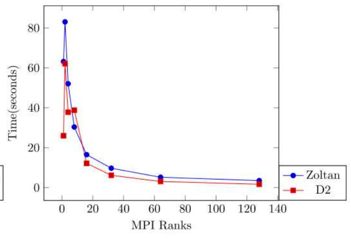
For D1, we argued that KokkosKernels gave more colors in its local coloring than Zoltan did; is D2 different?

E. Distance-2 Strong Scaling

Fig. 6: Zoltan and D2 strong scaling



(a) europe_osm

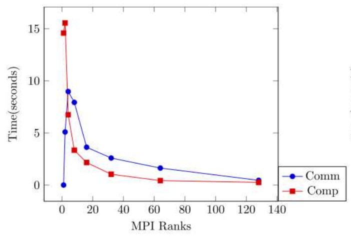


(b) Queen_4147

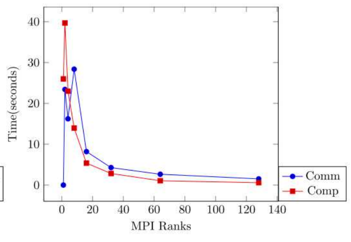
Figures 6a and 6b show the strong scaling behavior of D2 on europe_osm and Queen_4147. These graphs were chosen to highlight features of D2's scaling behavior. For both graphs, D2 has a spike in the execution time at 8 ranks, for two different reasons. For europe_osm, the spike is primarily caused by our vertex-based distance-2 conflict detection. **why? what's happening?** For Queen_4147 the spike is due to communication overhead, which is likely due to the two ghost layers that D2 requires. **"likely"? what's happening?** After the spikes, both D2 and Zoltan scale similarly, and both implementations use a similar number of colors.

D2 on average **overall all graphs? and MPI rank configurations? same edits needed as with D1** exhibits 9.32x speedup over a single GPU, and on average **ditto** D2 uses 2.7% more colors than the single GPU run. The speedup is likely greater with D2 than D1 because the distance-2 coloring problem is more computationally intensive, and thus benefits more from adding GPUs.

Fig. 7: D2 communication time (comm) and computation time (comp)



(a) europe_osm



(b) Queen_4147

Figures 7a and 7b show the source of the spikes in the execution time. For europe_osm, the primary factor in the spike is conflict detection. We use a vertex-based conflict detection scheme in D2 which is similar to our conflict detection in D1 and D1-2GL. For this graph, there is an imbalance in the eight-rank run that causes conflict detection to happen very slowly. The Queen_4147 graph has a similar spike, but it is caused by communication on eight ranks. The D2 method currently requires a second ghost layer, which increases communication overhead. **this explanation is weak; isn't the overhead higher on more ranks? why is the eight-rank case special?**

F. Distance-2 Weak Scaling

Fig. 8: Distance-2 weak scaling on mesh graphs

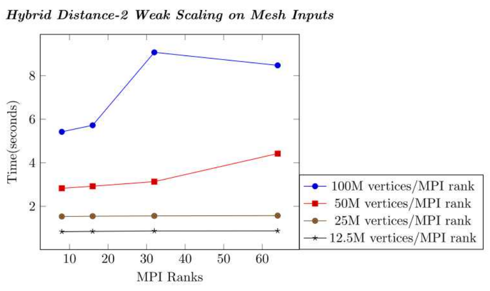


Figure 8 shows the weak scaling behavior for D2. The same hexahedral mesh graphs were used as in the D1 weak scaling experiments. **discussion needed. why does weak scaling break down for the larger meshes? we did not see that behavior with D1 or D1-2GL.**

VI. Future work

We plan to extend our distance-2 coloring to partial distance-2 coloring to support automatic differentiation applications. In partial distance-2 coloring, coloring criteria are applied only to vertices that are two hops apart. Since the colors of adjacent vertices are not considered, a proper partial distance-2 coloring may not be a proper distance-2 or even a proper distance-1 coloring. Our goal is to deliver a complete suite of MPI+X algorithms for distance-1, distance-2, and partial distance-2 coloring in the Zoltan2 package of Trilinos. This work's target application is the optimization of the computation of sparse Jacobian [21] and Hessian matrices [14], both of which are used in automatic differentiation and other computational problems [3].

VII. 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 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] "Kokkos Kernels," 2017. [Online]. Available: https://github.com/kokkos/kokkos-kernels

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

[3] D. Bozdağ, Ü. V. Çatalyürek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner, "Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation," *SIAM Journal on Scientific Computing*, vol. 32, no. 4, pp. 2418–2446, 2010.

[4] D. Bozdağ, A. H. Gebremedhin, F. Manne, E. G. Boman, and U. V. Catalyurek, "A framework for scalable greedy coloring on distributed-memory parallel computers," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 515–535, 2008.

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

[6] R. L. Brooks, "On colouring the nodes of a network," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 37, no. 2. Cambridge University Press, 1941, pp. 194–197.

[7] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Computing*, vol. 38, no. 10-11, pp. 576–594, 2012.

[8] 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. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

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

[10] 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 Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

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

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

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

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

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

[16] 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, 2011.

[17] 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)*, vol. 31, no. 3, pp. 397–423, 2005.

[18] M. F. Hoemmen, "Tpetra project overview." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.

[19] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.

[20] G. Rokos, G. Gorman, and 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.

[21] M. A. Rostami and H. M. Bücker, "Preconditioning Jacobian systems by superimposing diagonal blocks," in *International Conference on Computational Science*. Springer, 2020, pp. 101–115.

[22] A. E. Sariyüce, E. Saule, and 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.

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

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