SAND2020-10993C

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 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. 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 [6]. More recently, parallel algorithms [5], [10] 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 [3], [10]; 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 [14] and to identify short circuits in printed circuit designs [13]. 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. [10] 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 [15]. For such applications, assembling the associated graphs on a single node to run a sequential coloring algorithm may not be feasible [5]. 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], [12] for on-node parallelism and Trilinos [18] 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 7.5% increase in colors on average. For distance-2 coloring, our algorithm also sees up to 28x speedup, and a 4.9% increase in colors in the *worst* case. We also demonstrate good weak scaling behavior on up to 128 GPUs on graphs with up to 12.8 billion vertices and 76.7 billion edges in size.

II. Background

A. 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 $\forall (u, v) \in E, C(u) \neq$

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.

$C(v)$. 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. 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 [16]. 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 concurrently finds independent sets of vertices and concurrently colors all of the vertices in each set; this approach was used by Jones and Plassmann [20]. Osama et al. [22] implement approaches based on finding independent sets on a single GPU and explore the impact of varying the baseline independent set algorithm.

The second approach, referred to as "speculate and iterate" [8], colors as many vertices as possible in parallel and then iteratively fixes conflicts in the resulting pseudo-coloring until no conflicts remain. Çatalyürek et al. [8] and Rokos et al. [23] present shared-memory implementations based on the speculate and iterate approach. Deveci et al. [10] present implementations based on the speculate and iterate approach that are scalable on GPUs. Distributed-memory algorithms such as those in [5], [25] use the speculate and iterate approach. Grosset et al. [17] 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. Bozdağ et al. [5] showed that, in distributed memory, the speculative approach is more scalable than methods based on the independent set approach of Jones and Plassmann. As such, we choose a speculative and iterative approach with our algorithms.

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. [5] 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 [18] for distributed MPI-based parallelism and Kokkos [12] 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. [5]. 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 Distributed-Memory Speculative Coloring

```
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
```

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 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)

Our D1 method begins by independently coloring all owned vertices on each process using the GPU-enabled algorithms by Deveci et al. [10] 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.

Algorithm 3 Algorithm to identify and resolve conflicts

```
procedure CHECK-CONFLICTS($v$, $u$, colors, GID)
    conflict $\leftarrow 0$
    if colors[$v$] $=$ colors[$u$] then
        if rand(GID[$v$]) $>$ rand(GID[$u$]) then
            colors[$v$] $\leftarrow 0$
        else if rand(GID[$u$]) $>$ rand(GID[$v$]) then
            colors[$u$] $\leftarrow 0$
        else
            if GID[$v$] $>$ GID[$n$] then
                colors[$v$] $\leftarrow 0$
            else
                colors[$u$] $\leftarrow 0$
        conflict $\leftarrow 1$
    return conflict
```

Algorithm 4 Distance-1 conflict resolution and recoloring

```
procedure RESOLVE-CONFLICTS(
    Local Graph $G_l = \{V_l + V_g, E_l + E_g\}$, colors, GID)
    conflicts $\leftarrow 0$
    for all $v \in V_g$ do in parallel
        for all $\langle v, u\rangle \in (E_g)$ do
            conflicts $\leftarrow$ conflicts $+$ Check-Conflicts($v, u, \ldots$)
            if colors[$v$] $= 0$ then
                break
    Allreduce(conflicts, SUM)            ▷ Get global conflicts
    $gc \leftarrow$ current colors of all ghosts
    if conflicts $> 0$ then
        colors = Color($G_l$, colors)        ▷ Recolor vertices
        Replace ghost colors with $gc$
        Communicate recolored vertices to ghost copies
    return conflicts
```

Algorithm 4 shows the conflict-resolution inner loop of Algorithm 2. This algorithm runs on each process using its owned local graph G_l. It detects conflicts across processor boundaries and recolors vertices to resolve the conflicts.

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 [18]. Specifically, we use the FEMultiVector class of Tpetra [19] to communicate the colors of boundary vertices to their ghost copies on other processes via an all-to-all exchange. After the initial all-to-all exchange, we only communicate the colors of boundary vertices which have been recolored. After each process receives its ghosts' colors, it detects conflicts by checking each owned vertex's color against the colors of its neighbor as in Algorithm 4. The conflict detection is done in parallel over the owned 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.

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.

Once we have identified all conflicts, we again use VB_BIT or EB_BIT to recolor the determined set of conflicting vertices.

[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.

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.

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.

This method can reduce the total amount of communication relative to D1 for certain graphs by reducing the total number of recoloring rounds needed. 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 can be more expensive, 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. 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 more 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 (E_g), 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.

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 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. [28]. 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 5 Distance-2 conflict detection

```
procedure DETECT-D2-CONFLICTS(
  Local Graph $G_l = \{V_l + V_g, E_l + E_g\}$, colors, GID)
  conflicts ← 0
  for all $v \in V_l$ do in parallel
    for all $\langle v, u \rangle \in (E_l + E_g)$ do
      conflicts ← conflicts + Check-Conflicts($v, u, \ldots$)
      if colors[$v$] = 0 then
        break
      for all $\langle u, x \rangle \in (E_l + E_g)$ do
                    ▷ $u$ is one hop and $x$ is two hops from $v$
        conflicts ← conflicts + Check-Conflicts($v, x, \ldots$)
        if colors[$v$] = 0 then
          break
      if colors[$v$] = 0 then
        break
  return conflicts
```

Algorithm 5 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 method 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 [27] to partition our graphs. Determining optimal partitions for

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}	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	8.7	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

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.

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, 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 communications uses a Mellanox Infiniband interconnect. We compile with xlC 16.1.1 and use Spectrum MPI with GPU-Direct communication disabled.

The input graphs we used are listed in Table I. We primarily used graphs from the SuiteSparse Matrix Collection [9]. 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 [7]. 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 [15], which is a target application for graph coloring algorithms. We also include social network graphs and a web crawl to demonstrate scaling of our methods on irregular real-world datasets. We preprocessed all graphs to remove multi-edges and self-loops, and we used subroutines from HPCGraph [26] for efficient I/O.

We compare our implementation against the distributed distance-1 and distance-2 coloring in the Zoltan [11] package of Trilinos. Zoltan's implementations are based directly on Bozdağ et al. [5]. 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. 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. For consistency, we set 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.

We omit direct comparison to single-node GPU coloring codes such as CuSPARSE [21], as we use subroutines for on-node coloring from Deveci et al. [10]. 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.

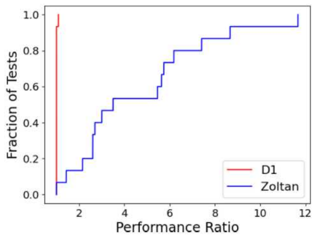
V. Results

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

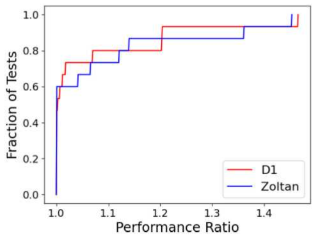
A. Distance-1 Performance

We summarize the performance of our algorithms relative to Zoltan using performance profiles. Performance profiles plot the 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 better approach's execution time (or colors used) for a given problem. In these plots, the line that is higher represents the better performing algorithm. The further to the right that an algorithm's profile is, the worse it is relative to the other algorithm.

Fig. 1: Performance profiles comparing D1 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 I.



(a) Runtime performance profile



(b) Color performance profile

We ran D1 and Zoltan with 128 MPI ranks to color the 15 SuiteSparse graphs in Table I. Some skewed graphs (e.g., twitter7) did not run on 128 ranks on Zoltan or D1; in those cases we use the largest run that completed for both approaches. D1 used MPI plus 128 Tesla V100 GPUs, while Zoltan used MPI on 128 Power9 CPU cores across 32 nodes (four MPI ranks per node). Figure 1a shows that D1 outperforms Zoltan in terms of execution time in these experiments. The D1 method is the fastest in roughly 95% of the cases; Zoltan outperforms D1 in only a single instance. D1 has at most a 11.6x speedup over Zoltan (with the europe_osm graph) and at worst an 8% slowdown relative to Zoltan (with Audikw_1).

Figure 1b shows that Zoltan outperforms D1 in terms of color usage. Zoltan uses fewer colors in over 60% of our experiments. However, in most cases, D1 uses no more than 5% more colors than Zoltan. With the twitter7 graph, Zoltan uses 45% fewer colors than D1, but with Mycielskian20, D1 uses 41% fewer colors than Zoltan. On average, D1 uses 6.8% more colors than Zoltan. These increases in the number of colors exist because of the higher concurrency used by D1 relative to Zoltan.

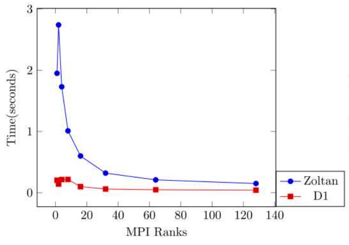
B. Distance-1 Strong Scaling

Figure 2 shows strong scaling times for Queen_4147 and com-Friendster. These graphs are selected for presentation because they are the largest graphs for their respective problem domains. Data points that are absent were the result of out-of-memory issues or execution times (including 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 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 8x faster than Zoltan in the 128 rank run, but D1 uses 26% more colors than Zoltan in that case.

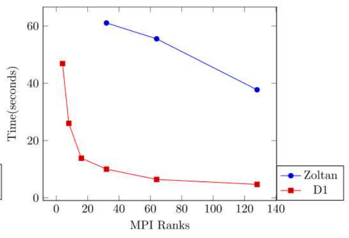
For graph processing in general, it is often difficult to demonstrate good strong scaling relative to single node runs. From the Graph500.org benchmark (June 2020 BFS results) [2], the relative per-node performance difference in the metric of "edges processed per second" between the fastest multi-node results and fastest single node results are well 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 average over all the graphs for which we have results, D1 still shows a 5.4x speedup over the single GPU run on 128 GPUs. On small or highly skewed graphs that fit on a single GPU, we do not see much speedup, due to the communication overheads and work imbalances that result from distribution even with relatively good partitioning.

On average over all our graphs, D1 sees a 47.2% increase in the number of colors from the single GPU run, while Zoltan sees an 53.6% increase in color use over 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 our approach and the Zoltan implementation have trouble coloring these graphs 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 3.15% for D1, and Zoltan decreases color usage by 0.1% on average. Zoltan's smaller observed increase is due to its inherently lower concurrency giving a better quality coloring.

Fig. 2: Zoltan and D1 strong scaling on select (a) PDE and (b) Social Network graphs.

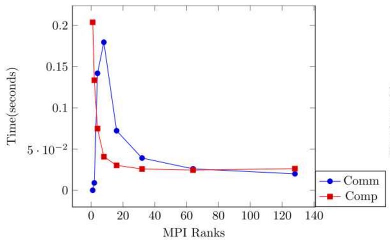


(a) Queen_4147

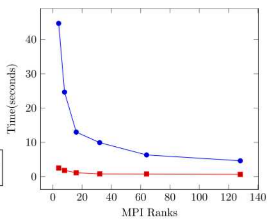


(b) com-Friendster

Fig. 3: D1 communication time (Comm) and computation time (Comp) from 1 to 128 GPUs.



(a) Queen_4147



(b) com-Friendster

Figure 3 shows the total communication and computation time associated with each run. For the Queen_4147 graph, computation time is the dominant factor in the larger rank runs. Figure 3a shows that we initially see computational scaling that levels off for large numbers of ranks. The computation time includes any computational imbalance and the time needed to launch GPU kernels. At high GPU counts, imbalance or kernel launches are likely the dominant component of the computation time for this graph, causing scaling to drop off above 64 GPUs. The com-Friendster graph shows computational scaling all the way to 128 GPUs, but, in this case, communication is the dominant factor of the execution time; computation scaling is not visible in the plot.

C. 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 with 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. 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 half a second**.

Fig. 4: Distance-1 weak scaling of D1 on 3D mesh graphs. Tests use 12.5, 25, 50, and 100 million vertices per GPU.

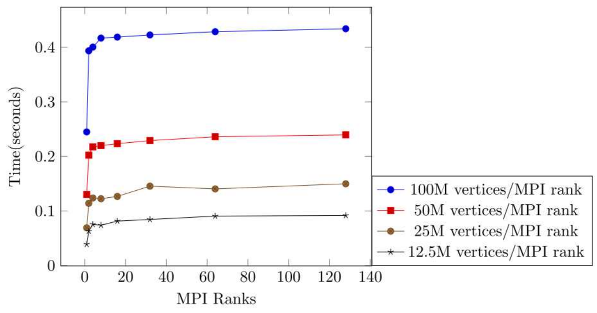
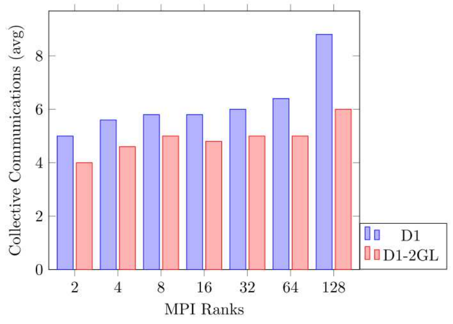


Figure 4 shows that the weak scaling behavior for D1 is very consistent. After a jump in execution time from one to two GPUs due to conflict resolution, the overall time increase from 2 to 128 GPUs is roughly 10% for each workload.

D. D1-2GL Performance

In general, D1-2GL does reduce the number of collective communications used in the distributed distance-1 coloring. Figure 5 compares the number of rounds for D1 and D1-2GL. Unfortunately, due to the increased cost of each communication round, D1-2GL does not generally achieve a speedup over D1. Additionally, second ghost layer vertices may be recolored if they are boundary vertices on another processor; this occurs often in dense inputs and incurs further communication costs. However, in distributed system with much higher latency, D1-2GL could be beneficial.

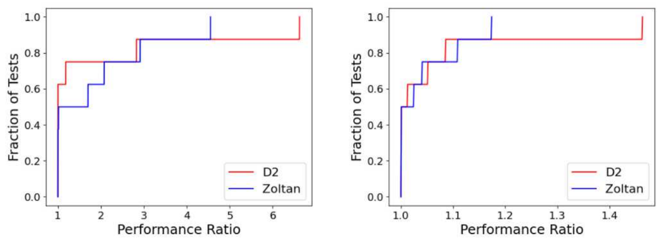
Fig. 5: Number of communication rounds for D1 and D1-2GL on Queen_4147 from 2 to 128 ranks.



E. Distance-2 Performance

We also compare our D2 method to Zoltan's distance-2 coloring 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. We use the same experimental setup as with the distance-1 performance comparison. Figure 6a shows that D2 compares well against Zoltan in terms of execution time, with D2 outperforming Zoltan on a majority of graphs. In the best case, we see a 4.5x speedup over Zoltan on the europe_osm graph.

Fig. 6: 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 I.



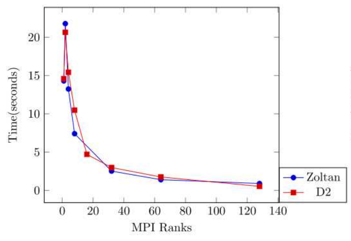
(a) Runtime performance profile (b) Color performance profile

Figure 6b shows that D2 has similar color usage as Zoltan. D2 and Zoltan each produce the lower 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. Interestingly, the number of colors used by D2 on the soc-LiveJournal1 graph is unchanged with one and 128 GPUs.

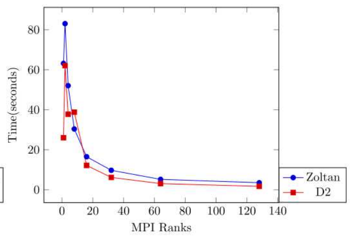
F. Distance-2 Strong Scaling

Figures 7a and 7b show the strong scaling behavior of D2 and Zoltan on Bump_2911 and Queen_4147. Bump_2911 shows similar scaling for both Zoltan and D2. For 128 ranks on Bump_2911, D2 uses 1.2% more colors than Zoltan, but runs 1.7x faster than Zoltan. With Queen_4147, D2 shows a brief scaling plateau from four to eight GPUs. This performance is an artifact of graph partitioning; the boundary size for eight ranks is the second largest out of all GPU counts except for 128 GPUs, resulting in a larger-than-expected communication cost. Zoltan is less sensitive to boundary sizes in the distance-2 case because it uses a more optimized communication pattern than our approach. After the eight-rank run, D2 scales slightly better than Zoltan up to 128 ranks. For the 128-rank run, D2 runs 2.1x faster than Zoltan, and uses 10% fewer colors.

Fig. 7: D2 and Zoltan strong scaling for distance-2 coloring.



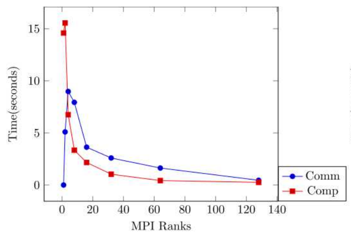
(a) Bump_2911



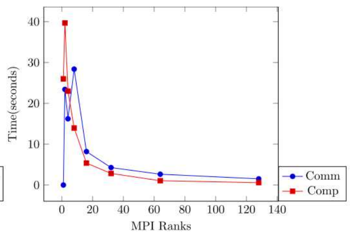
(b) Queen_4147

On average over the eight graphs, D2 exhibits 9.32x speedup on 128 GPUs over a single GPU, and uses 2.7% 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.

Fig. 8: D2 communication time (comm) and computation time (comp) from 1 to 128 GPUs.



(a) Bump_2911



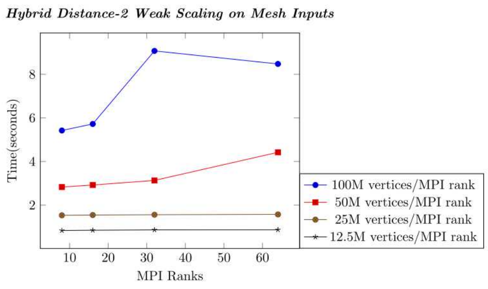
(b) Queen_4147

Figures 8a and 8b 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 2.7%. 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 8b shows that communication is the source of the runtime plateau shown in Figure 7b.

G. Distance-2 Weak Scaling

Figure 9 demonstrates the weak scaling behavior for D2. The same hexahedral mesh graphs were used as in the D1 weak scaling experiments. For small per-node workloads, the weak scaling is good. For larger per-node workloads, execution times increase slightly. For these larger tests, the communication time, conflict detection time and initial coloring time all stay relatively flat, as does the number of rounds of communication. However, we observe an increase in imbalance for recoloring times across GPUs in these instances. Identifying and correcting the source of this imbalance is future work.

Fig. 9: Distance-2 weak scaling of D2 on 3D mesh graphs.



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 [24] and Hessian matrices [15], both of which are used in automatic differentiation and other computational problems [4].

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 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] "Kokkos Kernels," 2017. [Online]. Available: https://github.com/kokkos/kokkos-kernels
[2] "Graph 500," 2020. [Online]. Available: https://graph500.org/
[3] J. Allwright, R. Bordawekar, P. Coddington, K. Dincer, and C. Martin, "A comparison of parallel graph coloring algorithms," *SCCS-666*, pp. 1–19, 1995.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.