2D Distributed Label Propagation on 400 GPUs

George M. Slota, Michael Mandulak, Ujwal Pandey, and Anthony Fabius
Rensselaer Polytechnic Institute
{slotag, mandum, pandeu, fabiua}@rpi.edu

***Abstract*—The label propagation algorithm (LPA) is widely used for community detection, graph partitioning, and unsupervised learning, among many other problems. While appearing simple on the surface, high performance implementations of LPA are deceptively challenging to compose, particularly targeting large-scale graphs on GPUs and in distributed memory environments. We present the first widely distributed multi-GPU implementation of LPA, utilizing a novel variant of 2D distributed graph processing. Our method computes label propagation on graphs an order-of-magnitude larger than has previously been done on GPUs, while offering the fastest time-to-solution in the literature for the 128 billion edge 2012 Web Data Commons web crawl graph, the largest current public graph dataset.**

***Index Terms*—label propagation, community detection, parallel algorithms**

I. Introduction

Graph-based data and its relevant computations have become increasingly prevalent within a variety of disciplines, such as computer science, math, and physics. While pervasive in usage, graph analytic methods face many challenges in application due to the sheer size and irregularity of real-world datasets, which often requires complex adaptations to mitigate. Among these analytics, the label propagation algorithm (LPA) [1] is one of the most well known and widely used. LPA is an iterative graph algorithm that propagates labels through a network based on neighborhood frequency, eventually converging to a stable labeling, given certain considerations. LPA has a broad application in areas such as community detection [1], machine learning [2, 3], and graph partitioning [4, 5, 6, 7].

Despite its deceptive simplicity, scaling LPA to massive, irregular datasets remains a challenge. LPA generally requires the use of a hash table to calculate neighborhood label frequency, which introduces substantial scaling issues in widely parallel or communication-intensive environments. Existing methods often introduce communication or computation-based tradeoffs for scalable performance [8, 9, 10, 11, 12]. In general, these methods use a variety of multi-node and single or multi-GPU distribution schemes as a baseline to accommodate the large scale of input data. These workflows, however, induce further additional challenges, such as workload imbalance on device, expensive communication costs, and cache inefficiencies, which limit performance. Running LPA efficiently across multiple GPUs requires all of the above considerations in implementation design, and it has only been accomplished at the single-node scale before this work.

A. Our Contribution

This work develops a widely distributed multi-GPU implementation of LPA. In this paper, we discuss our algorithm design that enables balanced and scalable computational and communication workloads. We improve upon the state-of-the-art in several ways.

1) **Input Scale**: The largest graph processed exclusively on GPUs we have found in prior literature is the 3.3 billion edge uk-2007-05 graph [9]. We run on the 128 billion edge Web Data Commons 2012 crawl (WDC12), which is well over an order-of-magnitude larger.
2) **System Scale**: The largest number of GPUs used for label propagation in the literature is two on a single node [8]. We strong scale to 400 GPUs on 67 nodes, expanding on existing art by over two orders-of-magnitude
3) **Speed at Scale**: The fastest prior time to run 10 iterations of label propagation in the literature for WDC12 is 40 seconds per iteration on 512 CPUs across 256 compute nodes [13]. Our code executes at under 10 seconds per iteration on our largest 400 GPU runs.

II. Background

A. Definitions

We consider a graph defined as $G = (V, E)$, where V represents the set of vertices and E represents the set of edges. The de facto computational representation of the graph uses an adjacency list representation, typically in the compressed sparse row (CSR) format or equivalent, where direct access to a vertex v's neighborhood $N(v)$ and degree $d(v)$ is accessible. We also implicitly consider a graph's adjacency matrix A for use with an edge-based 2D block distribution.

B. 1D and 2D Distribution methods

1D distribution schemes can be simply defined as the following: given p process ranks and $n = |V|$ vertices in a graph G, create a p-way disjoint subset of vertices and their incident edges. By doing this, each process p has roughly $\frac{n}{p}$ vertices and their immediate neighborhood, which enables vertex state-based computations. Communication is generally required across cut edges, which results in all-to-all communications with $O(p^2)$ messages per communication round. 2D distributions [14] consider the block partition of a graph – consider an adjacency matrix A, we distribute roughly a $(\sqrt{p} \times \sqrt{p})$ set of blocks to p processes. See Figure 1 (top) for an example with 8 ranks, where we have 2 *row groups* and 4 *column groups*. Communication occurs in two phases,

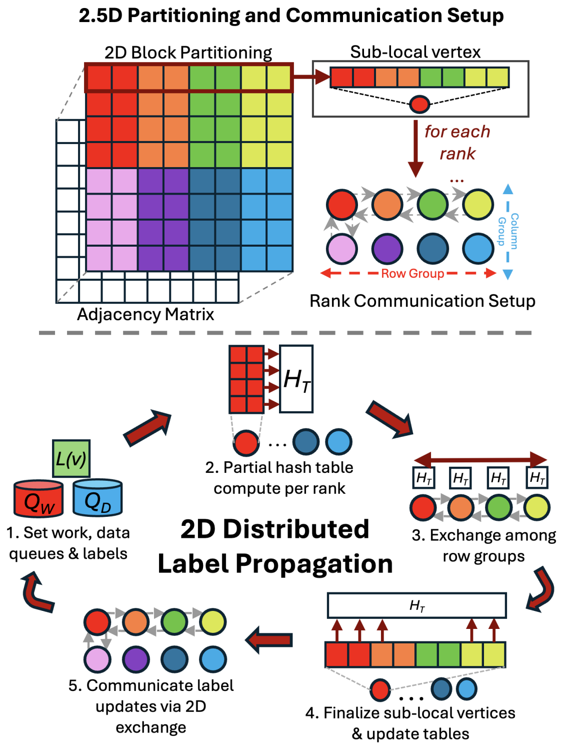
2.5D Partitioning and Communication Setup
2D Block Partitioning
Sub-local vertex
for each rank
Column Group
Row Group
Adjacency Matrix
Rank Communication Setup
H_T
$L(v)$
Q_W
Q_D
1. Set work, data queues & labels
2. Partial hash table compute per rank
2D Distributed Label Propagation
3. Exchange among row groups
4. Finalize sub-local vertices & update tables
5. Communicate label updates via 2D exchange

Fig. 1. (Top) The adjacency matrix breakdown of 2D distribution and the effective processing and communication of our 2.5D extension. (Bottom) The processing pattern of each iteration of our 2.5D distributed label propagation.

separately across row groups and column groups, resulting in only $O(p)$ messages per round. We use a 2D distribution for improved load balance across GPUs, especially for handling large degree vertices in very large graph inputs. 2D distributions also tend to scale out better to a larger number of ranks, due to the lower expected messages. Our implementation of LPA also requires an additional communication round, which we have previously termed as a *2.5D* distribution [15] – we will discuss this in our methods. We refer the reader to prior work [14, 15] for a more detailed description of the tradeoffs between 1D, 2D, and similar distributions.

C. LPA: Synchronous and Asynchronous

Algorithm 1 Basic Label Propagation Community Detection

```
1: procedure LabelPropagation(Graph $G(V, E)$)
2:     for all $v \in V(G)$ do
3:         $L(v) \leftarrow$vid($v$)                ▷ Initialize labels as vertex IDs
4:     for some number of iterations do
5:         for all $u \in V(G)$ do
6:             $H \leftarrow \emptyset$                ▷ Initialize $v$'s hash table
7:             for all $u, v \in E(G)$ do
8:                 Increment $H(L(v))$
9:                               ▷ $u$ assigns statistical mode of labels in $N(u)$
10:            $L(u) \leftarrow$ argmax($C$)
```

Given in Algorithm 1 is the serial LPA algorithm. Each vertex initializes their label to a unique value, usually its vertex identifier, and then all vertices update their label to the statistical mode of labels in their neighborhood, with ties broken randomly. This proceeds until *convergence*, when no labels are updated. Alternatively, many implementations simply run to a fixed number of iterations.

Updating can be done synchronously (all labels are updated at once at the end of an iteration) or asynchronously (labels are updated immediately as they are processed during an iteration) or semi-asynchronously (such as in a distributed-memory environment or a bipartite graph). Each approach has its tradeoffs: synchronous converges slower and can not converge at all, yet tends to produce higher quality and more stable results, while asynchronous tends to generate lower quality results yet converges with fewer iterations. Past work such as [16, 17] highlight these differences. Many LPA implementations tend to use asynchronous or semi-asynchronous implementations mainly for their performance advantages [1, 16, 18]. In this work, we adopt synchronous updates for practical purposes, as it is considerably simpler to implement with a 2D distribution.

D. Related Work

The foundational LPA was introduced by Raghavan et al. [1] as a near linear-time method for community detection in large networks. Following this work, existing research has focused on distributed and parallel computing paradigms to handle larger volumes of graph data. Many approaches, such as those detailed in [8, 10, 11, 12], utilize multi-node and GPU distribution schemes. These methods often introduce inherent trade-offs between communication overheads and computational efficiency. For instance, distributed community detection algorithms aim to reduce inter-node communication but suffer from synchronization costs. Similarly, GPU-accelerated implementations leverage massive parallelism, yet are constrained by memory bandwidth and data transfer limitations.

Beyond LPA, other community detection methods such as the Louvain method have seen significant work. Recent work includes a GPU-specific algorithm that parallelizes access to individual edges to improves load balance for networks with highly varying degrees, demonstrating substantial speedups over classical methods [19]. Furthermore, efforts have been extended to multi-GPU implementation of Louvain on exascale platforms, enabling the processing of graphs with billions of edges on up to 1024 GPUs in minutes [20].

III. Methodology: Implementation Details

Here we will describe our approach for implementing synchronous label propagation community detection under a 2D graph distribution. One primary consideration, in line with prior work, will be the handling of the hash table (or equivalent) that tracks labels (keys) and counts (values) over each vertex's neighborhood. In our case, we must implement this hash table in a distributed and hierarchical fashion in our 2D edge-based distribution.

We give a general overview of our method in Algorithm 2 and visualization of the primary loop in Figure 1 (bottom), and we will describe each listed function in the remainder of this section. In the broad scope, after basic initializations, we track a set of 'active' vertices in a work queue, which are the vertices whose labels we update on the current iteration. As neighborhood adjacency information is distributed across a row group, we require two phases of hash table calculation for these vertices. We describe this as a *2.5D* processing method. First, each rank computes partial hash tables for all local row-owned vertices using their rank-owned edges. Further, each rank is assigned a disjoint set of sub-local vertices as a subset of their row group-owned vertices, as shown in Figure 1 (top). The partial hash tables are communicated across the row to each rank owning the associated sub-local vertex. These hash tables are 'reduced' to a final hash table by the owning rank, where the label with the maximum count can be determined. If this label is different than the existing one for the given vertex, it is assigned to its associated vertex, and communication of this updated label is performed row-wise and then column-wise using standard 2D communications. We then build the next level work queue by using all neighbors of all vertices that were assigned an updated label. We iterate until convergence or for a fixed iteration count.

Algorithm 2 2D Distributed Label Propagation

```
1:  procedure LabelPropagation(Graph $G(V,E)$)
2:      $L \leftarrow$ InitLabels()
3:      $Q, D \leftarrow$ InitQueues()                ▷ Work and data queues
4:      $H \leftarrow$ InitHashTables()
5:      for some number of iterations do
6:          ComputeLocalRowTables($G, L, Q, H$)
7:          LocalRowExchange($D, H$)
8:          FinalizeSubLocalTables($L, D, H$)
9:          BroadcastUpdatedLabels($L, D$)
10:         ComputeNextQueue($G, Q, D$)
```

A. Implementation Details

We implement our methods in `C/C++` using `CUDA` for parallel computations on GPU and `NCCL` for distributed message passing. We utilize the graph structure described in [15], which includes methods for creating and accessing the distributed 2D adjacency lists, vertex degree information, and the process rank and group information needed for standard 2D communications. We rely on no other external libraries.

B. Initialization

Our initialization phase must first assign initial labels to each vertex, create communication and work queues, and initialize hash tables that track neighborhood label counts.

InitLabels(): As shown in Algorithm 1, it is standard to initialize a vertex's label as its unique vertex identifier. However, we note that in a synchronous label propagation implementation, a vertex will have a unique set of singular labels across its neighborhood during the first iteration. Hence, with tie breaking, a vertex effectively randomly selects some neighbor label to assign to itself. We explicitly implement this as our initialization to avoid unnecessary computations and communications during the first iteration. In practice, this allows us to use the theoretically smallest memory space required for hash tables to track neighborhood label counts, as we will show in our results.

InitQueues(): We initialize and maintain several queues. We have a dynamic work queue Q, which contains vertices whose labels we will update on the given iteration. As noted above, this queue will be updated to the neighbors of vertices which modify their labels on the current iteration. We also maintain communication queues respresented by D, which will be used to pass hash table information and updated label information. We initialize the work queue to contain all vertices, initially.

InitHashTables(): We allocate a single contiguous memory space H that is reused for both computing partial local hash tables as well as final sub-local tables. This hash table is allocated to be proportional in size to twice the maximum of the number of rank-owned edges in the standard 2D distribution OR the twice sum total of degrees of the owned sub-local vertices, by some factor f of 1 or larger. Effectively, this space is partitioned such that each vertex has a static space allocated of $2f\times$ the number of edges that will be involved in its statistical mode computation in the maximum case, where we can store $\langle label, count\rangle$ pairs. While prior work sets the factor $f = 2$ or larger, we set $f = 1$ to minimize memory usage. While the assigned space would theoretically allow the table to fill completely, our initial label assignment step greatly reduces the load factor in practice for large degree vertices, where such an impact would be noticeable for performance. We utilize linear, quadratic, or double probing, with a basic modulo hash function. We will show experimental load factors and compare against various probing methods in our results.

C. Main Loop

Algorithm 3 Compute Local Hash Tables for Row Vertices

```
1:  procedure ComputeLocalRowTables(
    Graph $G$, Labels $L$, Work Queue $Q$, Data Queue $D$)
2:      $W \leftarrow$ ComputeWorkloads($Q, G$)
3:      for $u, v \in W$ do in parallel
4:          $h \leftarrow H(u)$                   ▷ Hash table location for $u$
5:          $l \leftarrow L(v)$                              ▷ Label of $v$
6:          InsertOrIncrement($h, l$)
7:      SynchronizeThreads()
8:      for $u, v \in W$ do in parallel
9:          $h \leftarrow H(u)$
10:         $l \leftarrow L(v)$
11:         if $l$ is in $h$ then        ▷ Check if $l$ was cleared from $h$
12:             $D \leftarrow \langle u, l, h(l)\rangle$        ▷ Add tuple to comm. queue
13:             ClearTable($h, l$)              ▷ Clear label from table
```

ComputeLocalRowTables(): The first computation phase of our implementation is to compute partial hash tables on each rank. These partial hash tables are the local $\langle label, count\rangle$ pairs for all row group vertices in the queue for all edges owned on the rank. Given in Algorithm 3 is a condensed version of the kernel logic for computing the partial hash tables. First, we compute the workload distribution for the given thread block, which will enable us to assign each neighbor of each vertex in the work queue to a unique thread. We refer the reader to prior work [15, 21] for a more in-depth understanding of how we do this via the Manhattan Collapse technique. Using the assigned (vertex, neighbor) pair, we can determine the location of the hash table for the vertex in the globally allocated space and then insert the label of the neighbor into that hash table. If the label is not already in the table, we insert a value of 1; otherwise, we increment the existing value. For GPU thread-safe insertions and increments in the table, we adapt techniques utilized in prior work [22].

Once we have built the hash table, we copy the unique $\langle \mathit{label}, \mathit{count} \rangle$ pairs to a data queue as a $\langle \mathit{vertex}, \mathit{label}, \mathit{count} \rangle$ tuple, that we will communicate in the next phase. We finally reset the hash table, which will be reused with new offsets when we compute final values for sub-local vertices.

LocalRowExchange(): Every rank within a row group has an assigned disjoint sub-group of the row's vertex set, termed as the rank's sub-local vertices. The goal of this communication phase is to send, for a given sub-local vertex v owned by rank r, the partial hash tables computed by all the other ranks in the row group for vertex v to rank r. We do this through the equivalent of an `Alltoall`-style collective operation.

FinalizeSubLocalTables(): At the end of the preceding communication round, all ranks have the $\langle \mathit{vertex}, \mathit{label}, \mathit{count} \rangle$ data contained in all partial hash tables across the row group for their sub-local vertices in a receive buffer. This phase utilizes a similar approach as in Algorithm 3, where we process each tuple by once again placing them into hash tables contained within the prior allocated space. We are then able to compute the final $\langle \mathit{label}, \mathit{count} \rangle$ pairs for all sub-local vertices. After processing all tuples, we can then determine the maximum count and associated label for each vertex. If the label is different from the current label for a given vertex, we update the label locally and place $\langle \mathit{vertex}, \mathit{label} \rangle$ into a queue for communication to the row group.

BroadcastUpdatedLabels(): This communication follows a standard 2D pattern. We first communicate new labels across the row group by doing a set of broadcasts from each rank, as an `Allgather`-style communication. After each rank processes all received labels, we then communicate the updated labels via another set of broadcasts among the column group.

ComputeNextQueue(): We utilize an approach similar to that described in [15] to build the work queue for the next iteration, as needed. In the preceding steps, we track all vertices which have had their labels updated. The next work queue is built by placing all vertices adjacent to updated ones into the queue. As this is a relatively costly operation when there is a significant number of updates, we can instead place all vertices in the work queue for early iterations where most vertices update their labels. We switch between these *dense* vs. *sparse* work queues when the number of updated vertices is below some threshold, specifically when 25% or fewer vertices update their labels.

D. Complexity Analysis

We consider a distributed BSP-like CRCW PRAM model to analyze work, depth, and memory consumption. For 2D-based iterative vertex state programs without complex computational dependencies, one can assume $O(\frac{n}{\sqrt{p}} + \frac{m}{p})$ work and memory per rank and $O(1)$ parallel depth [15]. **Our method achieves these optimal bounds for work and memory**.

Computing and communicating the intermediate local table requires $O(\frac{n}{\sqrt{p}} + \frac{m}{p})$ work and memory and $O(1)$ depth, as offsets for each edge within the hash table can be independently calculated in parallel. Likewise, these complexities also hold for communicating the intermediate tables and computing final values for each rank-owned sub-local table, assuming the global vertex ordering assures a relative balance of $O(\frac{m}{p})$ edges owned by each rank within the 2D distributed and each sub-local vertex partitioning. Unlike with standard 2D communication which needs $O(p)$ messages overall, our two-phase table computation requires $O(p\sqrt{p})$ messages, as the `Alltoall` partial table exchange within each row group needs p messages for each of all $\sqrt{p}$ row groups.

Communicating updated labels requires $O(\frac{n}{\sqrt{p}})$ work and memory and $O(1)$ depth, while determining the next level queue once again requires $O(\frac{n}{\sqrt{p}} + \frac{m}{p})$ work and memory with $O(1)$ depth. We do not specifically analyze our approach in terms on iteration counts, as it will be equivalent to other synchronized implementations, but our overall depth will be a factor of the number of iterations.

IV. Experimental Setup

We perform experiments on our LPA implementation using the standard benchmark graphs listed in Table I. We emphasize, in particular, the 2012 Web Data Commons graph, as it is the largest publicly available graph dataset as of September 2025. We run our experiments on a RPI's AiMOS system, which contains nodes with 2× IBM Power-9 CPUs and 512 GB DRAM, 6× NVIDIA 32 GB V100 GPUs, connected via an EDR Infiniband network. On node, groups of 3 GPUs are connected via NVLink, with communications between groups traveling through PCIe. We run our scaling experiments on up to 400 GPUs. We compile with `-O3` using CUDA V11.8.89, NCCL 2.10.3, and OpenMPI 3.1.5, which were the most recent versions supported on system.

Name	Vertices	NNZ	d_{avg}	Source
USA-road-d	24 M	116 M	5	[23]
twitter-2010	41 M	2.9 B	72	[24]
com-friendster	65 M	3.6 B	55	[25]
MOLIERE_2016	30 M	6.7 B	223	[26]
web-ClueWeb09	1.7 B	16 B	9	[27]
gsh-2015	1.0 B	66 B	66	[28]
WDC12	3.6 B	257 B	73	[29]

TABLE I
GRAPH INPUT DATASETS FOR EXPERIMENTS WITH NUMBER OF VERTICES, NONZEROS IN THE IMPLICIT ADJACENCY MATRIX, AVERAGE DEGREE, AND ORIGINAL DATA SOURCE.

We use square 2D distributions in our experiments, where the number of row groups and column groups are equal. Vertex order is randomized. Unless otherwise specified, we set our hash table capacity factor to the theoretical minimum $f = 1$ and use quadratic double probing [10] for collision resolution.

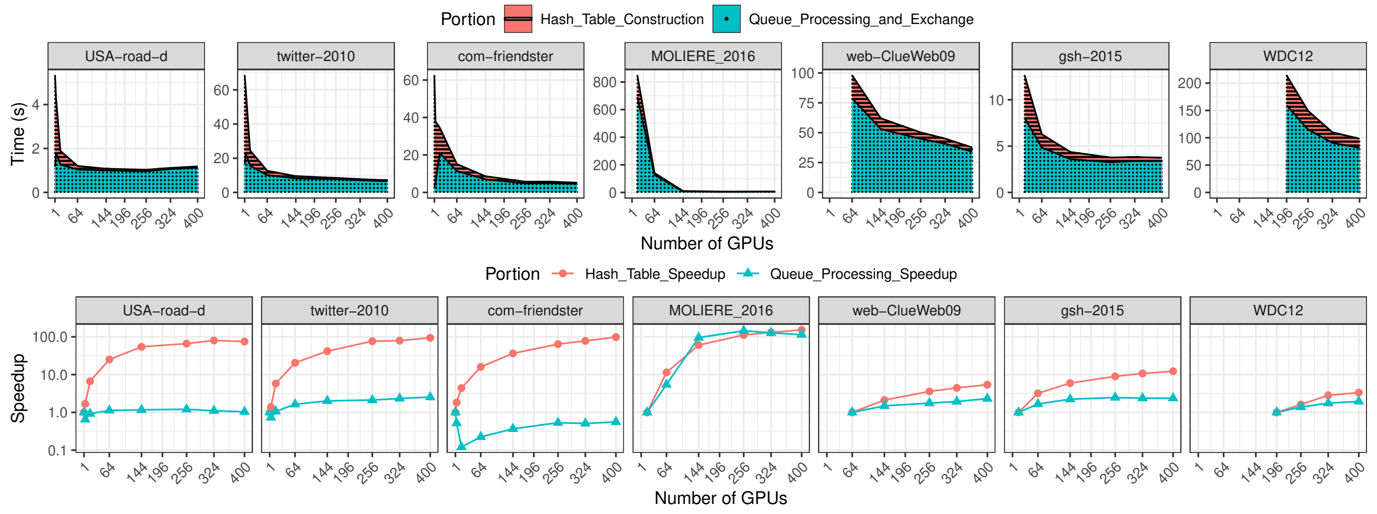

Fig. 2. Strong scaling on all inputs from 1 to 400 GPUs, with runs specifically on 1, 4, 16, 64, 144, 196, 256, 324, and 400 GPUs. We break down execution time into processing of the hash tables (primary computation cost) and processing of queues and exchange of hash tables and label updates (communication costs). We plot absolute timing values in seconds on top and speedups relative to the smallest GPU count run per graph on bottom.

V. Results

A. Strong Scaling

Given in Figure 2 is strong scaling performance from 1 to 400 GPUs for all inputs. We plot the absolute timings in the top plots and speedups in the bottom plots. We plot timing as broken into two primary phases: 1.) the computational kernels that construct and process the hash tables and label updates; 2.) the kernels that build and exchange update and work queues – essentially, the communication overhead introduced by distributed processing. Note, this communication overhead includes computation necessary for building and processing queues, so it is not just the network communication cost. Due to the large scale of our inputs, particularly for WDC12, many inputs required the memory of multiple GPUs to run.

We observe that all inputs except for the road network (the smallest input by far) fully strong scale in term of overall execution time to 400 GPUs. For most inputs, both the hash table computation and communication overhead scale, with the exception of a higher communication cost observed when running from 1 or 4 GPUs on 1 node to 16 GPUs across 3 nodes in certain instances, with com-friendster as an obvious example. The flattening of scaling of communication overheads occurs after 256 GPUs in multiple instances, as the network becomes the bottleneck. 400 GPUs is the maximum allocation possible we were able to consistently receive on AiMOS, hence we are approaching practical system limits.

With lower GPU counts, the hash table construction cost dominates execution time. However, communication overheads take a proportionally larger amount of time as we scale out. This is expected with the 2D model, given that hash table construction costs $O(\frac{m}{p})$ work and queue computation and communication overheads require an additional $O(\frac{n}{\sqrt{p}})$ work, as discussed in Section III-D. We approximately obverse this scaling behavior as visualized in the bottom plots of Figure 2.

B. Performance Breakdown

We further examine the breakdown in performance in Figure 4 by examining the timings of the computation and communication phases detailed in Section III. We plot the proportion of total execution time for each phase on 16, 64, and 256 GPUs with USD-road-d, com-friendster, twitter-2010, and gsh-2015 as representative instances. These represent graphs with a regular structure and flat degree distribution, social networks with semi-skewed degree distribution, very skewed social networks, and web crawls, respectively. We will use these test inputs for this current and subsequent results.

As observed in Figure 4, like with Figure 2, we again observe communication and associated overheads requiring a larger proportion of execution time as we scale out for all inputs. The largest input, gsh-2015, requires the largest time for computation of the sub-local hash table, due to the considerably larger vertex set, as would be expected.

C. Hash Table Performance

Other work has noted that hash table design is a key driver of performance [8, 10], particularly on GPUs. Conversely, our scaling experiments have demonstrated that, at the extremely large scale, hash table computation becomes a small fraction of overall execution time. Hence, it might be desirable to run as large an input on a given hardware configuration by minimizing hash table overhead.

As we noted in our experimental setup, we select a hash table capacity just large enough to contain the maximum possible label counts. This is considerably smaller than other work, which often allocated more than double the minimum. However, with our initialization, we note that we on average do not approach capacity. In Figure 5 we display the average load factor across all vertices' hash tables for varying inputs as a function of LPA iteration. These results are from 64 GPUs. We note that the less skewed social network, com-friendster, is the only input that has a load factor average over 50%. This

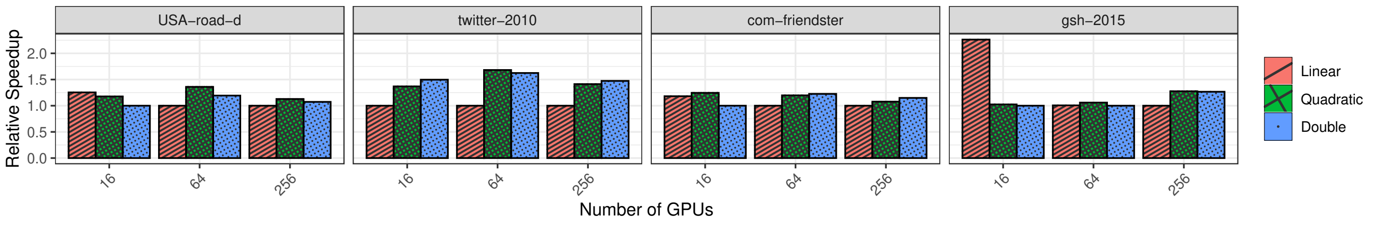

Fig. 3. Relative speedup in terms of execution time for hash function processing kernels using collisions resolution methods of of linear probing, quadratic probing, and quadratic-double probing. Speedups are computed relative to the slowest method for a given (graph, rank) combination.

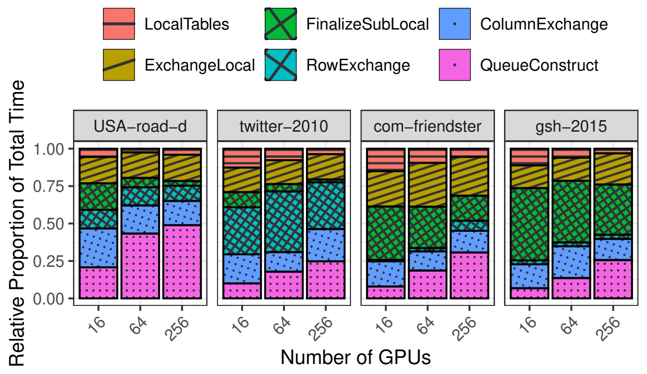

Fig. 4. Timing breakdown of algorithm phases for 16, 64, and 256 ranks.

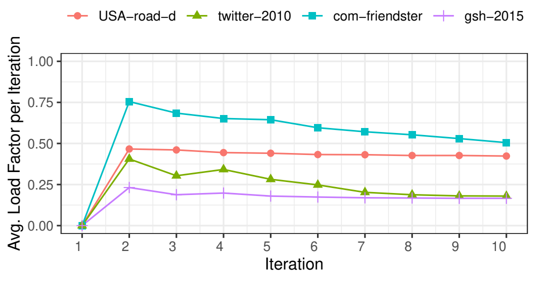

Fig. 5. Average hash table load factor across all active vertices per iteration.

is likely due to its large number of low degree vertices, which in practice have insignificant cost, regardless of load factor.

Other work has also noted that the collision resolution method can be a significant driver of performance [10]. In Figure 3, we give performance results of multiple choices of collision resolution, including linear probing, quadratic probing, and quadratic-double probing (from [10]) on 16, 64, and 256 GPUs. Contrary to the results in [10], we notice minimal difference in overall performance at our larger scale. With the exception of USA-road-d, all of our test instances are larger than the largest input used in that work, which primarily considered optimizing for a single GPU. In general, we note that quadratic double probing is generally most performant, mirroring the prior work. Linear probing on 16 ranks with gsh-2015 stands out as an exception, likely explainable by improved cache performance – 16 ranks is very near the absolute lower limit of memory capacity to process the input.

D. Prior Work Comparisons

Here we summarize and compare against other label propagation and related implementations. Due to how widespread LPA is in the literature, we limit our focus to the most recent work on GPU(s) and the largest work on distributed CPUs. At the largest scale, HPCGraph [13] presented results on WDC12 using 512 CPUs across 256 nodes with average iteration times of 40 seconds over 10 total iterations. In this paper, we achieved 10 seconds per iteration over 10 total iterations across 400 GPUs on 67 nodes. Giraph [30] gave performance on a "48B+ edge" Facebook network at 0.17 GPEPS (billion edges processed per second) over 2 iterations on 200 nodes. We achieve up to 12.8 GPEPS on WDC12.

We also compare against smaller-scale GPU works, while noting that the design decisions driving the development of our code were targeted at input dataset and system scales orders-of-magnitude larger. Even on a single GPU, we still call all kernels building communication queues and make NCCL calls. The only multi-GPU work we found is GLP [8], which gives an average iteration time of 0.80 seconds across 20 iterations for twitter-2010 on 2 NVIDIA 2080ti GPUs. We approximately match that time on 256 GPUs. However, they also require about 4 minutes of preprocessing time, while all of our I/O, preprocessing, and graph construction time totals under 7 seconds. On 1 GPU, we measure 6 seconds per iteration with about 1 minute of I/O and preprocessing. The other most recent GPU work is v-LPA, which processes the it-2004 web crawl at a rate of 3.0 GPEPS on a single NVIDIA A100. Our slowest performance on 400 GPUs with our web crawls is with web-ClueWeb09 at a rate of 4.2 GPEPS.

Other recent work implemented the Louvain algorithm for multi-GPU, scaling to an impressive 1024 NVIDIA V100 GPUs while processing a 4 billion edge random graph in 3 minutes. Lastly, a label propagation variant was implemented as part of the TeraPart graph partitioner for initial coarsening. They report 191 seconds total time for partitioning WDC12 on 256 CPUs across 128 nodes, though it is not exactly delineated what portion of that time is for LPA, specifically.

VI. Conclusions

This paper presented a novel distributed implementation for LPA. We discussed our design and optimization of both computation and communication subroutines, which enables scaling to 400 GPUs and graphs with hundreds of billions of edges. Future work can further refine our 2.5D communication routines, optimize hash table access and allocations, and address other current scaling bottlenecks.

VII. Acknowledgments

This work is supported by the National Science Foundation under Grant No. 2047821.

References

[1] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.76.036106

[2] Y. Fujiwara and G. Irie, "Efficient label propagation," in *International conference on machine learning*. PMLR, 2014, pp. 784–792.

[3] A. Iscen, G. Tolias, Y. Avrithis, and O. Chum, "Label propagation for deep semi-supervised learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 5070–5079.

[4] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 481–490.

[5] D. Salwasser, D. Seemaier, L. Gottesbüren, and P. Sanders, "Tera-scale multilevel graph partitioning," *arXiv preprint arXiv:2410.19119*, 2024.

[6] M. S. Gilbert, K. Madduri, E. G. Boman, and S. Rajamanickam, "Jet: Multilevel graph partitioning on graphics processing units," *SIAM Journal on Scientific Computing*, vol. 46, no. 5, pp. B700–B724, 2024.

[7] G. M. S. . S. R. K. D. K. Madduri, "Partitioning trillion-edge graphs in minutes," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 481–490.

[8] C. Ye, Y. Li, B. He, Z. Li, and J. Sun, "Large-scale graph label propagation on GPUs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 10, pp. 5234–5248, 2024.

[9] Y. Kozawa, T. Amagasa, and H. Kitagawa, "GPU-accelerated graph clustering via parallel label propagation," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 567–576.

[10] S. Sahu, "-LPA: Fast GPU-based label propagation algorithm (LPA) for community detection," *arXiv:2411.11468*, 2024.

[11] M. Ovelgönne, "Distributed community detection in web-scale networks," in *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ser. ASONAM '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 66–73. [Online]. Available: https://doi.org/10.1145/2492517.2492518

[12] J. Soman and A. Narang, "Fast community detection algorithm with GPUs and multicore architectures," in *2011 IEEE International Parallel Distributed Processing Symposium*, 2011, pp. 568–579.

[13] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 293–302.

[14] E. G. B. K. D. D. S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2D graph partitioning," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013.

[15] G. M. Slota and M. Mandulak, "Scaling distributed graph processing to hundreds of GPUs," in *International Conference on Parallel Processing (ICPP)*, 2025.

[16] G. Cordasco and L. Gargano, "Community detection via semi-synchronous label propagation algorithms," in *2010 IEEE international workshop on: business applications of social network analysis (BASNA)*. IEEE, 2010, pp. 1–8.

[17] P. L. Ian X. Y. Leung*, Pan Hui† and J. Crowcroft, "Towards real-time community detection in large networks," 2009.

[18] C. L. Staudt and H. Meyerhenke, "Engineering parallel algorithms for community detection in massive networks," in *2016 IEEE Transactions on Parallel and Distributed Systems, Volume 27, Issue 1)*. IEEE, 2016, pp. 171 – 184.

[19] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo, "Community detection on the GPU," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 625–634.

[20] N. S. Sattar, H. Lu, F. Wang, and M. Halappanavar, "Distributed multi-GPU community detection on exascale computing platforms," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2024, pp. 815–824.

[21] G. M. Slota, S. Rajamanickam, and K. Madduri, "High-performance graph analytics on manycore processors," in *International Parallel & Distributed Processing Symposium (IPDPS)*, 2015.

[22] G. M. Slota and C. Brissette, "A constant-memory framework for graph coarsening," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2024.

[23] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[24] H. P. S. M. Haewoon Kwak, Changhyun Lee, "What is Twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 591–600.

[25] K. P. G. P. D. B. B. Alan Mislove, Massimiliano Marcon, "Measurement and analysis of online social networks," in *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 29–42.

[26] I. S. Justin Sybrandt, Michael Shtutman, "MOLIERE: automatic biomedical hypothesis generation system," in *KDD '17: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data*

Mining. ACM, 2017, pp. 1633–1642.

[27] C. L. Clarke, N. Craswell, and I. Soboroff, “Overview of the trec 2009 web track,” DTIC Document, Tech. Rep., 2009.

[28] P. Boldi, A. Marino, M. Santini, and S. Vigna, “BUbiNG: Massive crawling for the masses,” in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2014, pp. 227–228.

[29] C. B. S. V. Robert Meusel, Oliver Lehmberg, “Web data commons - hyperlink graphs,” 2012.

[30] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at facebook-scale,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.