Scaling Distributed Graph Processing to Hundreds of GPUs

George Slota
RPI
Troy, NY, USA
slotag@rpi.edu

Michael Mandulak
RPI
Troy, NY, USA
mandum@rpi.edu

Abstract

This paper presents HPCGraph-GPU, methods for optimized 2D communications for general graph computations on hundreds GPUs. While 2D methods have existed for decades, most prior efforts have focused on specialized benchmarks such as breadth-first search (BFS) or general processing at a modest scale. We extend the usage of 2D distributions to arbitrary and massive-scale graph computations, developing lightweight and sparse communication patterns, active vertex queues, and approaches for more complex reductions and communications. To demonstrate the efficacy of our approach, we implement a handful of the standard benchmark graph algorithms along with more complex routines, including label propagation, maximum weight matching, and pointer jumping. Our efforts approach the theoretical limits of strong and weak scaling for 2D methods, while greatly outperforming the scalability of prior related work. These efforts also offer the first generalized multi-GPU performance results on the largest publicly available dataset, the 128 billion edge 2012 Web Data Commons crawl. On this input, we observe performance from 26-123 billion edges processed per second on 400×V100 GPUs, depending on algorithm complexity.

ACM Reference Format:
George Slota and Michael Mandulak. 2025. Scaling Distributed Graph Processing to Hundreds of GPUs. In *54th International Conference on Parallel Processing (ICPP '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3754598.3754664

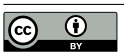
1 Introduction

Graph-structured data is pervasive, appearing in a broad range of fields, from math, physics, chemistry, and computer science, among many others. The study and analysis of such graph datasets is correspondingly widespread. However, there are several well-known challenges associated with graph analytics, primarily including the scale and irregularity of graph datasets and the complexity of efficient analytic algorithms to study these large datasets.

Many graph processing engines and techniques have been created in an attempt to address these challenges. Some have offered ease-of-use at the expense of overall performance [23] or are performant but limited in scale to shared-memory [29]. Others have only demonstrated performance for very specialized operations such as breadth-first search (BFS) [1] or relatively simple algorithms like PageRank [16]. A select handful have scaled to large datasets and impressive performance numbers [13, 27, 34]. For very large scale graph analytics or memory-intensive algorithms, distributing the graph structure across multiple nodes or GPUs (we will use the generic term 'rank' when referring to separate memory/compute spaces) is a requirement. However, this introduces several additional challenges, including communication overheads, load balance, and cache inefficiency. These problems are magnified on GPU, where the smaller device memory, lightweight SIMT execution, and higher latency communication (outside of specialized systems such as the DGX [16]) all compound to negatively impact performance [32]. The larger number of GPUs required to process the same dataset, relative to CPU, can also "blow up" the number of messages and communication volume when using common graph distributions.

The original and possibly still most common distribution type for multi-node graph processing assigns a subset of graph vertices and all associated adjacency information for those vertices to a single rank. This is referred to as a "1D distribution" in terms of the adjacency matrix, as each row is fully owned by some rank. This historically has been a reasonable and simple approach, as many graph algorithms tend to perform computations iteratively, updating some vertex (or edge) state information based on the immediate neighborhood. However, this can also limit scalability is two key ways: First, very large degree vertices in graphs with skewed power-law degree distributions are owned by a single rank, resulting in imbalance of computation and communication, especially when strong or weak scaling the number of ranks [17]. Secondly, the overheads for communication in general can become unwieldy. As it is generally necessary for each rank to communicate with every other rank, the number of total messages communicated scales quadratically with the number of ranks.

As such, within the past decade, so-called 1.5D or hybrid distributions [11] have emerged, where selected large degree vertices are shared among multiple ranks, vastly improving load balance for irregular graphs. A more generalized concept is edge-based distributions, where a rank owns some subset of edges for some subset of vertices. Continuing this trend is 2D distributions, where **all** vertices are shared among multiple ranks. In this, a graph's adjacency matrix is broken into rank-owned blocks, where communication occurs in stages among row and column groups of these blocks. Load balance and communication scaling can be vastly improved, and this is the de-facto standard approach for top-performers on benchmarks such as the Graph500 [1]. 2D-style distributions in particular have a long history in HPC, being employed by dense matrix multiplication methods for decades [12]. Their application to general graph analytics is less widespread [9], mostly being utilized for a few highly-studied benchmark algorithms, such as triangle counting [30] and PageRank [16].



ICPP '25, San Diego, CA, USA

ACM ISBN 979-8-4007-2074-1/25/09
https://doi.org/10.1145/3754598.3754664

Contributions: We implement a generalized methodology for distributed GPU-based 2D graph processing. We discuss several low-level implementation considerations, including graph data structures, dense/sparse 2D communication patterns, work queues, and workload balance. Our methods exhibit excellent scalability, strong scaling near theoretical limits up to 256 GPUs with graphs small enough to fit in a single device, as well as scaling to 400 GPUs with the largest publicly available graph dataset. We also demonstrate the generalizability of our approach, implementing several complex graph analytics beyond the standard benchmark algorithms.

2 Background

Basic Definitions: We consider in simplest terms a graph $G = (V, E)$, where V defines a set of vertices, E defines a set of edges, and $N = |V|, M = |E|$ are the number of vertices and edges, respectively. For computational purposes, this graph is generally represented as an adjacency-based data structure in memory, most often in a compressed sparse row (CSR) or similar format. In such a format, the *adjacencies* (or neighbors) of some vertex v can be directly accessed in some order. The *degree* of v is the number of adjacencies. For the purposes of discussion in this paper, we will also consider graph G implicitly represented as an adjacency matrix A. In this square $N \times N$ matrix A, nonzeros at $a_{i,j} \in A$ indicate a single edge or multi-edges between some vertex represented by row identifier i and some other vertex represented by j.

As discussed, this adjacency matrix can be partitioned or distributed across multiple ranks. We will term a rank *owns* a vertex if it is allocated some portion of the vertex's row of the adjacency matrix. For general 2D distributions, multiple ranks own the same vertex. If an owned vertex has an edge to a non-owned vertex, we consider the non-owned vertex as a "ghost". We generally need to maintain information about both owned and ghost vertices to perform most iterative graph computations.

2.1 Graph Processing

Generally, graph computations involve iterative updates to state values associated with vertices and/or edges within the given graph input. E.g., BFS will update *parent* or *level* state information as the traversal expands from a root. Commonly, the notion of pulling vs. pushing [3] of updates is considered in this context. Using BFS as an example again, an unvisited vertex can "pull' an update by examining the visitation status of its neighbors and marking itself as visited if it finds a visited neighbor. Conversely, a visited vertex can "push" an update of visitation status to all unvisited neighbors. Similarly, the notion of gather-apply-scatter [11] is used in these contexts, where a vertex gathers neighbors information, applies an update to its state, and scatters this update back to its neighbors.

When considering the distribution of a graph into distributed memory and a typical graph computation, the most important idea for the reader to recognize is that updates to vertex states generally require a vertex's adjacency information as well as the various states of each adjacent vertex. In 1D distributions, all of the 1-hop information is directly accessible for an owned vertex on a single rank. The updated state values of ghosted vertices are what is communicated during iteration computations, ensuring correctness and consistency. This is generally done using a bulk synchronous parallel (BSP) model. As noted above, these ghost updates can either be pushed to the ghost vertex's owner from a non-owning rank or pulled from the owning rank. Either way, this communication is typically done via an all-to-all exchange, requiring $O(p^2)$ messages for p processors.

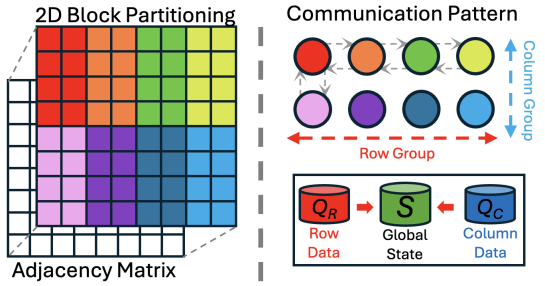


Figure 1: 2D block partitioning of an adjacency matrix with 2 row groups and 4 column groups (8 total ranks, designated with color). Communications occur along these row and column groups, with group-based updates (e.g., Q_R and Q_C) being committed to an implicit global state (S).

2.2 2D Graph Processing

See Figure 1 for an example of a 2×4 2D partitioning of an adjacency matrix into 8 ranks, consisting of row groups and column groups. Note that each row group exclusively owns the same set of vertices and each column group has the same set of ghosts, except for instances along the diagonal where the sets overlap on a single rank. 2D distributions have the added challenge of the fact that the full adjacency and ghost information of a given vertex is not considered to be owned by any single rank. Hence, an additional round of communication is necessary, to ensure consistent state values among all ranks that own that vertex.

If we first consider a pull update and associated communication, it is first necessary to perform some reduction over all vertices owned by the same row group to get consistent state values, before broadcasting these values along a column group to ranks that have these vertices as ghosts. Push updates are reversed, where a reduction update is first performed over the column group, before the updates are broadcast or otherwise communicated across a row group. Through these basic methods, all computations possible in a 1D distribution can be equivalently expressed in a 2D distribution.

The benefit of this approach is that each group-wise communication only requires $O(\sqrt{p})$ messages for p total ranks when the row and column group sizes are approximately equal. Since we have $O(\sqrt{p})$ groups in each direction, our total number of messages scales as $O(p)$ instead of $O(p^2)$ as in a 1D all-to-all exchange. However, the total communication volume for 2D distributions can be larger, as a rank can have to communicate up to $O(\frac{N}{\sqrt{p}})$ state updates relative to a maximum of $O(\frac{N}{p} + \frac{M}{p})$ for 1D. Correspondingly, 2D computational loads can also increase as $O(\frac{N}{\sqrt{p}} + \frac{M}{p})$. Because of this scaling behavior, 2D distributions are generally preferred in "weak scaling" scenarios with low communication volumes (e.g., BFS in the Graph500), where a larger N is considered along with a larger p. We will analyze and discuss this as part of our results.

2.3 Prior Work

Generalized distributed graph processing has existed for close to two decades now. Early methods utilized I/O intensive operations via MapReduce and similar frameworks [19]. Later, more HPC-based approaches were developed, first optimizing for user-friendly development interfaces [11], but offering low practical scalability [20]. Following those efforts, performance became more of a focus, with several works offering scalability to the largest available public datasets [8, 18, 27]. The most recent work at the very large scale has focused on optimizing partitioning through vertex placement to minimize communication latency and overall cost [10].

While much of the above work has focused on CPUs, GPUs have likewise received considerable attention. Several single GPU and single node multi-GPU frameworks have been developed [14, 15, 31], which address the challenges associated with GPU graph computations mentioned previously. Other work has sought to standardize graph computations via optimized algebraic backends [33], similar to BLAS in the linear algebra domain. While some true distributed multi-node GPU frameworks for general graph computations have been developed [14, 15], none of them have demonstrated fully generalized scalability to the largest current real graph dataset, the 128 billion edge Web Data Commons Crawl.

We know of only one other work besides ours that has utilized 2D distributions for general graph computations on GPU [9, 14]. However, that work utilizes a more generalized communication substrate which allows arbitrary distributions, instead of one specifically optimized for lightweight 2D communications. As demonstrated in our results, this can add overhead that limits performance and scalability in practice.

3 Methods: HPCGraph-GPU

To enable general and efficient graph computations in distributed memory, four important considerations are required.

(1) **Graph Structure:** A graph structure needs to be compact with minimal overheads to accessing adjacency information or state information.
(2) **State Communications:** Vertex (and edge) states need to be communicated in an efficient manner, with volume scaling proportional to the number of state updates.
(3) **Vertex Activation:** Many iterative graph computations have a long "tail" of small numbers of updates per iteration, with only a few vertices actively updating. There needs to be a way to track which vertices are relevant to the computation.
(4) **Load Balance:** Especially with BSP-style computations, a graph partitioning/distribution should enable computation or communication balance among ranks. Within GPUs, computational load balance should be ensured.

In this section, we detail the distributed structure of our graph data and describe the fundamental communication and computation patterns upon which we build our algorithm implementations. We extrapolate these base communication patterns to complex algorithms, developing more nuanced communication schemes. We call our codebase `HPCGraph-GPU` as an extension of our prior CPU-based `HPCGraph` [27], and we make our code available at the following repository: https://github.com/HPCGraphAnalysis/HPCGraph.

3.1 Implementation Details

We implement all of our methods in `C/C++` using `CUDA` for GPU programming and `NCCL` for communications. `NCCL` provided significant performance benefits relative to `CUDA`-aware `MPI`. Graph construction is done on CPU before being transferred to GPU using OpenMP and MPI. We use no other library dependencies, enabling broad portability and ease of compilation.

3.2 Graph Representation

Table 1: Primary variables used in our 2D graph structure.

Variable	Description
N	Global number of vertices
M	Global number of edges
Adj	Adjacency list for local CSR
Off	Offsets for local CSR
R	Number of ranks in each row group
C	Number of ranks in each column group
ID_R	Row group ID
ID_C	Column group ID
$Rank_R$	Row group rank
$Rank_C$	Column group rank
N_R	Local number of row group vertices
N_C	Local number of column group vertices
N_T	Total number of unique row+column group vertices
N_{Offset_R}	Starting global vertex ID for row group
N_{Offset_C}	Starting global vertex ID for column group
$Type$	How local vertex IDs are structured
C_{offset_R}	Starting local vertex ID for row group
C_{offset_C}	Starting local vertex ID for column group

For our 2D graph structure, we maintain data relative to our notion of row and column (ghost) ownership within our communication setup. We give an overview of the primary variables used for our representation in Table 1, maintained on each rank. Each rank individually tracks the total number of vertices in its row group N_R and its column group N_C, and the starting offsets in terms of global vertex IDs for its row N_{Offset_R} and column N_{Offset_C} vertices. Additionally, it has information about its row group and column group IDs (ID_R, ID_C) and a rank's group-rank within the row group and column group ($Rank_R$, $Rank_C$). Unlike in some matrix methods, we consider a rank as owning only a single block in the distribution.

Locally, vertices are remapped into local vertex IDs, mapping global identifiers from $[0 \dots N)$ to local identifiers in $[0 \dots N_T)$. This is a standard practice in distributed graph structures, done to accelerate local computations and simplify memory accesses. These local IDs are correspondingly used within our local CSR graph, defined by the adjacency array Adj and offsets array Off. As is standard, the adjacencies for a vertex v are listed between $\left[Adj[Off[v]] \dots Adj[Off[v+1]]\right)$ and the local degree of vertex v is calculated as $Off[v+1] - Off[v]$. Note that in a 2D data distribution, the local degree is not necessary the actual degree of v, though the actual degree will be equal to the summed local degrees across all ranks in the row group that owns v.

The way that the vertex IDs are mapped from global→local is dependent on what we term as the rank's *Type*. The possible *Type* of the mapping is internally denoted as an integer in $[0, 1, 2]$, and the

Table 2: Definitions for mapping global to local vertex IDs.

Type	Definition	Row and column vertex mapping
0	No overlap in row/column GIDs	Row LIDs = $[0 \dots N_R)$ Col LIDs = $[N_R \dots N_R + N_C)$
1	$N_{Offset_R} \leq N_{Offset_C}$	$diff = N_{Offset_C} - N_{Offset_R}$ Row LIDs = $[0 \dots N_R)$ Col LIDs = $[diff \dots diff + N_C)$
2	$N_{Offset_R} > N_{Offset_c}$	$diff = N_{Offset_R} - N_{Offset_C}$ Row LIDs = $[diff \dots diff + N_R)$ Col LIDs = $[0 \dots N_C)$

Type is defined based on the relative global IDs of row and column vertices. We give their definitions in Table 2. This explicit mapping is done for two reasons. Primarily, our communication methods, as described later, communicate via global IDs for consistency. Thus converting global→local IDs is a common operation. A structured definition can do this conversion via simple arithmetic, avoiding a hash table lookup that would otherwise be needed. In addition, this method also simplifies dense communications, as local IDs for row and column vertices are compacted in order, a communication of an array of vertex state values only requires the offset (C_{offset_R} or C_{offset_C}) within the local mapping and the number of vertices in the group, regardless of any row/column vertex overlap.

3.3 Communication Patterns

In this section, we detail the various types of communication patterns seen in algorithms built upon 2D communications. We classify these communications as being dense, sparse, or complex, depending on the method and application.

We will consider a generic vertex state algorithm as the basis for our descriptions, shown in Algorithm 1. In this generic algorithm, all vertex states are first initialized in some fashion. E.g., PageRanks are initialized to $\frac{1}{N}$ or BFS parent/level information is initialized to `MAX_INT`. For some fixed number of iterations until convergence, these vertex states are updated by looping through all (or some via a queue) vertices needed for the computation. States are generally propagated or updated via edge relations, as indicated in `update`$(S[v], S[u])$.

Algorithm 1 Basic State Update Algorithm

```
1: procedure StateAlgorithm(Graph G(V, E))
2:     S ← InitState(G)                          ▷ Initialize state of vertices
3:     for some number of iterations do
4:         UpdateState(G, S)
5: procedure UpdateState(G, S)
6:     for v ∈ V do
7:         for (v, u) ∈ E do
8:                                   ▷ State updates of u (push) and/or v (pull)
9:             update(S[v], S[u])
```

3.3.1 Dense Communications. We define dense communications as when *all* vertex state values are communicated along row and column groups, regardless of whether or not they were updated. For simple state update/reduction patterns, this requires no additional processing or handling of data queues or messages. In Algorithm 2, we show the typical patterns for a dense push and dense pull communication.

For a push operation, as visualized in Figure 2, we simply perform an AllReduce across each column group on the vertex state S array

Algorithm 2 Basic Dense Communication Pattern

```
1: procedure StateAlgorithm(Local Graph G(V, E))
2:     S ← InitState(G)
3:     for some number of iterations do
4:         UpdateState(G, S)
5:         // Assuming that N_R = N_C
6:         if PUSH
7:             AllReduce(S[C_offset_C], N_C, COL_GROUP_COMM)
8:             Broadcast(S[C_offset_R], N_R, ROW_GROUP_COMM)
9:         else if PULL
10:            AllReduce(S[C_offset_R], N_R, ROW_GROUP_COMM)
11:            Broadcast(S[C_offset_C], N_C, COL_GROUP_COMM)
```

starting at offset C_{Offset_R} through N_C values in the array. For PageRank, we would be performing a `SUM` reduction over these values. When $R = C$, we follow this with a row group Broadcast starting at $Offset_R$ through N_R state values with the root equal to the row group ID. When $R \neq C$, we used multiple grouped broadcasts via aggregated Group Calls in NCCL. We have found this approach most performant in practice compared to explicit Send/Recv or other gather-based operations. The roots will be all ranks that have an overlap between their row and column vertices, broadcasting up to a maximum of N_C values, each offset from C_{Offset_R} by multiples of N_C in the state values array.

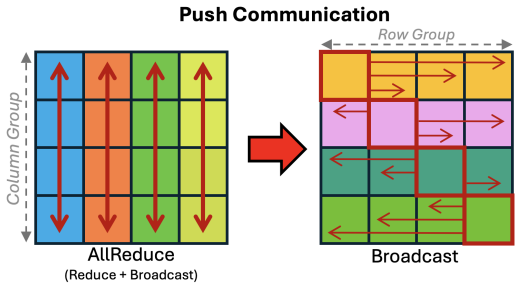


Figure 2: "Push" communication pattern (opposite "pull"), described as an AllReduce of the column group followed by a Broadcast of the row group, with 16 total ranks.

A pull operation first reduces over the row group and then broadcasts over the column group. We perform dense pull operations more-or-less the exact same as above, first performing an AllReduce over the row group and then a Broadcast or a group of Broadcasts over the column group. Note that some graph computations such as Label Propagation Community Detection might require a more complex reduction. In the case of Label Propagation, a vertex updates its state to the statistical 'mode' of labels in its neighborhood. In such instances, an approach similar to what we do for sparse communications (discussed next) would be required.

Because of the large communication volume, we reserve the usage of dense communications for instances when the amount being communicated is known to be large. For an algorithm such as PageRank, where vertices tend to update their PageRank values by a small amount on every iteration, dense communications are solely what is used. For many other graph algorithm (e.g., Label Propagation), the number of updates drastically decreases over multiple iterations, so dense communications become wasteful.

Hence, we therefore switch to sparse communications after a certain cutoff, defined by the proportion of state updates relative to the vertices in the graph. For our experiments, we make this switch when under $\frac{N}{\max(R,C)}$ of vertices in the graph have been updated, to ensure that communication volume is always being saved.

Algorithm 3 Basic Sparse Communication Pattern

```
1:  procedure StateAlgorithm(Local Graph G(V, E))
2:      S ← InitState(G)
3:      for some number of iterations do
4:          if PUSH
5:              Q ← UpdateState(G, S, Q, q_in)
6:              sbuf ← BuildQueue(G, S, Q, q_in)
7:              rbuf ← AllGatherv(sbuf, COL_GROUP_COMM)
8:              Q ← ReduceQueue(G, S, rbuf)
9:              sbuf ← BuildQueue(G, S, Q, q_in)
10:             rbuf ← Broadcast(sbuf, ROW_GROUP_COMM)
11:             Q ← ReduceQueue(G, S, rbuf)
12:         else if PULL
13:             ...        ▷ Same procedure with Row/Column comms swapped
```

3.3.2 Sparse Communications. Sparse communications trade additional computational overhead in building queues with reduced communication volume by only sending updated values. As we have noted, performant graph codes must utilize sparse communications in some fashion. The general communication pattern for push and pull updates are similar to those above, so we focus specifically on our methodology for efficient queue construction and reductions. See Algorithm 3 for an overview.

We first consider the case of push communications. During the `UpdateState()` kernel, we keep track of all unique column vertices with updates pushed to them, and place them in a queue Q. To do this is a thread-safe way, we use an `atomicExch()` on a boolean array q_{in} indexed based on local vertex IDs. A **true** at a given index signifies the vertex is already in the queue. Since a column vertex might have its state updated multiple times, we delay explicitly building the communication queue until launching the subsequent kernel `BuildQueue()`, given in Algorithm 4.

Algorithm 4 Build Communication Queue

```
1:  procedure BuildQueue(G, S, Q, q_in)
2:      sbuf = ∅, n_send = 0
3:      t_idx = unique offset in Q for each thread
4:      v = Q[t_idx]
5:      idx = atomicAdd(n_send, 2)
6:      sbuf[idx] = v              ▷ Place v and its state into the send queue
7:      sbuf[idx + 1] = S[v]
8:      q_in[v] = false
```

Here, we iterate through all vertices in the queue and build a communication buffer containing {vertex GID, state value} pairs, using the finalized state values for the current iteration. An AllGatherv-style communication (implemented as an NCCL AllGather followed by a group of broadcasts) is then performed to distribute all per rank updated values through the column group. A reduction kernel is then called, given in Algorithm 5, which utilizes similar logic to `UpdateState()` in order to update values. This logic can be relatively simple and accomplished via atomics, or it can be more complex logic, as discussed above. The reduction kernel also builds another queue to be used for updates among the row group.

Algorithm 5 Reduce Communication Queue

```
1:  procedure ReduceQueue(G, S, rbuf, q_in)
2:      Q = ∅, n_q = 0
3:      t_idx ← unique offset in rbuf for each thread, modulo 2
4:      v = rbuf[t_idx]
5:      val = rbuf[t_idx + 1]
6:      old = S[v]
7:      new = AtomicOp(S[v], val)        ▷ 'Op' can be MIN,ADD, etc
8:                                       ▷ Or can be a more complex routine
9:      if v ∈ Row Group and new ≠ old
10:         if atomicExch(q_in[v], true) == false      ▷ If v is not in queue
11:             idx = atomicAdd(n_q, 1)
12:             Q[idx] = v
```

To perform row communications, we first call a kernel for building a row group communication buffer. This kernel takes in the group of unique row-owned vertices that have been updated in Algorithm 5 and places their GID and most recent state value into a communication buffer. This buffer is then communicated via a broadcast or NCCL group of broadcasts (when $R \neq C$) among the row group. Finally, the row group performs a reduction by parsing the received communications and reducing the values as in Algorithm 5. The kernel utilized for queue building is essentially the same as with Algorithm 4. For the reduction with no further communications, we omit the final if-statement section in Algorithm 5.

As with dense communications, pull communications essentially mirror the above. We first build a queue of updates and perform a sparse reduction among the row group. These updates are then broadcast among the column groups for further reduction.

3.3.3 Complex Communications. Many graph algorithms do not directly follow the typical state update patterns given above. In fact, we have found that many non-trivial analytic algorithms we have implemented have some additional considerations. We highlight several noteworthy examples below:

Complex Reductions: As we have noted, certain algorithms require non-trivial reductions to compute updates. For such instances, one can solely use the sparse communications with custom reduction operations in `ReduceQueue()`. We implement a maximum weight matching algorithm that uses such a communication pattern.

2.5D Processing: For other algorithms like Label Propagation, the 'complex reduction' is very expensive. Determining the statistical mode of labels for a large neighborhood requires either compute and memory-intensive sorting or compute-intensive hash table construction. We implement this algorithm and opt for the latter using a space-efficient GPU hash-table adapted from prior work [24, 25]. For our approach, each rank in a row group first reduces neighborhood labels into a set of hash tables for all N_R vertices based on their locally-owned edges. Next, each row rank is set to hierarchically own $\frac{N_R}{R}$ unique *local* vertices by block partitioning the vertices owned by the row group. The initial hash tables are exchanged to the owner of the local vertex, which performs the final reduction. These final values are then broadcast back out to the row group, before being subsequently broadcast to the column group in the standard fashion. We refer to this as *2.5D Processing*, with similar examples of "beyond-2D" methods appearing in the literature for algebraic computations [28]. This method can also be used in general for dense communications requiring complex reductions, with

the tradeoff of building and communicating a group-wise set of *local* buffers in place of a possibly larger all-gather buffer.

Packet Swapping: In some applications, such as pointer jumping and least common ancestor traversal [4, 5], updates are not solely propagated along edges. For such applications, we use the notion of an information 'packet' that is communicated across row and column groups. We denote this communication pattern as "packet swapping". Generally, the packets contain owner, state, and send direction (e.g., an originating pointer and to which rank to jump), as well as other application-specific data. Communication of these packets can be similarly exchanged pairwise between any ranks via a single set of row and column group communications, as we do with the more structured communications patterns above.

3.4 Computation Patterns

3.4.1 Vertex Activation. We maintain active vertices by building a row-group-consistent queue on each iteration. For push updates, this approach requires little modification to our discussed kernels. In our `UpdateState()` kernel, we note if a vertex we are placing into Q is row-owned. If so, we also place that vertex into a separate 'active vertices' queue. Likewise, when we are performing the final row-wise reduction, we re-include the final if-statement logic from Algorithm 5, this time placing any updated but non-yet-queued vertices into the next active queue.

A vertex queue for pull updates is a bit more complex and expensive. The most important difference is that the active vertices on a subsequent pull iteration are **not** the vertices that updated their state on the current iteration – the active vertices must be set as the *neighbors* of these updated vertices. When we construct and reduce the state update queue along our row-group communication, we track all row vertices that have been updated. We then call a kernel which examines the adjacencies of these vertices and uniquely places them into an active vertex queue. This queue is then shared in a push-style sparse communication across the column groups and then the row groups.

3.4.2 Load Balance. One issue with the above approach is that vertices are not ordered within the queue, so GPU load balancing techniques such as sorting vertices by degree within a static CSR are ineffective. Other methods in the literature [21] utilize hierarchical processing methods via implicit or explicit queues, where a vertex's adjacencies are expanded via different methods based on degree. E.g., a low degree's adjacencies are all expanded by a single thread while a large degree vertex is processed by an entire thread block.

Manhattan Collapse: For the relevant adjacency expansion in the cases where we use a queue, we instead utilize the "Local Manhattan Collapse" loop collapse [26], detailed in Algorithm 6. This method collapses the nested [For $v \in V$] and [For $(v,u) \in E$] loops by performing a prefix sums over a portion of the active vertex queue assigned to a given GPU thread block. We assign one vertex per thread for each block for simplicity. The block then computes the prefix sums on vertex degrees to get total work and work offsets for each vertex. We then iterate over the total work bounds, assigning unique edges to each thread using the degree offsets and a binary search. These edges are used to perform an algorithm-dependent computation or reduction. This model of computation is agnostic to our application, and it allows us to effectively assign balanced edge work per thread in any arbitrary queue-based graph algorithms.

Algorithm 6 Local Manhattan Collapse per Rank

```
1:  procedure UpdateState(G, S, Q, q_in)
2:      t_id = thread ID
3:      b_id = block ID
4:      v = Q[t_id + b_id × BlockSize]                ▷ Thread-Queue Assign
5:      work = shared memory array of (BlockSize + 1)
6:      work[t_id + 1] = O[v + 1] − O[v]              ▷ Degree of v
7:      block_scan(work)                              ▷ Compute prefix sums
8:      for i = t_id to work[BlockSize] do
9:          j = binary_search(i, work)
10:         u = A[O[v + (i − work[j])]]
11:         update S[v]                               ▷ Algorithm Dependent
12:         if S[v] was updated
13:             if atomicExch(q_in[v], true) == false
14:                 q ← v                             ▷ Place v in communication queue
15:         i = i + BlockSize
```

While this can add overhead relative to hierarchical methods, algorithm implementation becomes much simpler and cleaner, computational load balance is almost fully optimized, and the overhead is small and easy to optimize, being near-negligible when the work per edge is high for algorithms such as Label Propagation. For certain graphs, where the maximum degree is close to N, a different approach would likely required, even with a 2D distribution. However, we know of no such graphs that exist at the large scale.

Vertex Distribution: We primarily use a 'striped' distribution for vertex to row group assignment when processing real data. Here, vertex with original GID 0 is assigned to the first row group, vertex 1 is assigned to the second group, with vertex n being assigned to the first row group after vertex $n-1$ is assigned to the n^{th} row group, continuing through all N vertices. We found that such a distribution offers comparable load balance to a random distribution without having varying group sizes, and it maintains some degree of memory locality of the original graph (whose vertices are often ordered using BFS or DFS traversals). Explicit 2D graph partitioning for smaller-scale graphs and matrices has been considered for decades [6, 7], though existing software lacks practical scalability to the massive inputs we consider. Future work might investigate communication-optimizing methods based on hardware network topology and similar methods [10].

4 Implemented Algorithms

We implement a handful of algorithms to test our methods, listed in Table 3. We consider BFS, PageRank and connected components as our primary 'benchmark' algorithms, with the rest highlighting the discussed complex communication patterns. The most important details for our experimental section are given in the table. We run PR and LP for fixed iteration counts, which is often standard for benchmarking purposes.

Breadth-First Search: We implement a standard push/pull-optimized BFS using sparse communications and static parameters from the original Beamer et. al paper [2].

PageRank: We implement the standard PageRank algorithm as a pull-based vertex state program with dense communications, instead of an optimized linear algebraic routine. We will compare against an optimized routine in our results.

Table 3: Algorithms, abbreviations, and experimental notes.

Algorithm Name	Abbr.	Notes
Breadth-first Search	BFS	Standard hybrid method.
PageRank	PR	We run for 20 iterations.
Connected Components	CC	We run until convergence.
Approx. Max Weight Matching	MWM	We run until convergence.
Label Propagation	LP	We run for 20 iterations.
Pointer Jumping	PJ	We run until convergence.

Connected Components: We implement the color propagation-based distributed algorithm for (weakly) connected components decompositions. We use this to study the effects of the described optimizations. We opt for this algorithm for connected components in place of a pointer-jumping based routine, as its simplicity and typical 'graph algorithmic' pattern enables us to generalize results to a broad class of algorithms. We implement both push and pull variants with both dense and sparse communications and with and without active vertex queues for comparison.

Label Propagation: We implement our previously-described '2.5D' variation of label propagation community detection as a pull update with sparse communications and vertex queues.

Approximate Maximum Weight Matching: We implement a distributed version of the Locally-Dominant approximate maximum weight matching (MWM) algorithm [22]. We define a MWM as follows: given our graph with respective edge weights w, a *matching* is a subset of edges, $M \subseteq E$, where every vertex of G has at most one endpoint in M. A MWM is a matching of maximum total weight among all possible matching. This method requires each vertex to scan neighbors and set a pointer along the highest weight, non-matching edge. Once set, mutually-pointing pairs are committed to the matching. Within the reduction pattern, we maintain a queue of unmatched vertices for pointers each iteration. All vertices in the queue consider reduced edge information among row/column groups with pointer updates marked and reduced. Mutual checks occur as a global state reduction for a finalized matching.

Pointer Jumping: We implement a pointer jumping algorithm that functions as a root-finding mechanism of a subtrees within a forest on our graph, similar to an approach used for connectivity. First, we instantiate a pointer for each vertex along an owned edge within a graph, creating a forest. We then iterate by communicating pointers up each subtree. Pointers are treated as packets of information that include the relevant jump, original global source, and destination IDs. These are communicated along either the row/column group, depending on the ownership of the source vertex, and correspondingly updated.

5 Results

We run two primary sets of experiments. First, we consider scaling experiments using standard benchmark algorithms (PR, CC, BFS) and a selection of standard benchmark graphs, given in Table 4. We consider graphs as undirected for consistency across algorithms, effectively symmetrizing the adjacency matrix. Our second set of experiments considers our complex algorithms. We put particular focus on the Web Data Commons 2012 web crawl, which as of May 2025 is still the largest publicly available real graph.

Our primary test system is AiMOS at RPI, which contains nodes with 2× IBM Power-9 CPUs and 512 GB DRAM, 6× NVIDIA 32 GB V100 GPUs, and an EDR Infiniband network. On node, a group of 3 GPUs per each CPU is interconnected with NVLink. Communications on node across GPU groups and communications across the network required movement though the CPU, which was likely our largest bottleneck for scaling. We ran our scaling experiments on up to 400 GPUs with the WDC12 graph, but limited ourselves to 256 GPUs for other inputs due to high turnaround times for larger allocations. Runs larger than 400 GPUs were not possible due to system constraints. We compiled with CUDA V11.8.89, NCCL 2.10.3, and OpenMPI 3.1.5. For smaller-scale comparisons to code we could not run on AiMOS, we additionally used a workstation with 4×A100 GPUs, referred to as `zepy`.

Table 4: Graph input datasets. The RMAT inputs are scale-*XX* with standard Graph500 parameters (*edgefactor* = *16*, $A = 0.57$, $B = 0.19$, $C = 0.19$). The RAND inputs have the same size and order as the RMAT graphs but are generated with an Erdös-Renyí $G(n, m)$ process.

Name	Abbr.	Vertices	Edges
twitter-2010	TW	41M	1.4B
com-friendster	FR	65M	1.8B
web-ClueWeb09	CW	1.7B	7.9B
gsh-2015	GSH	988M	33B
WDC12	WDC	3.5B	128B
RMAT*XX*	RMAT	2^{24}–2^{32}	2^{28}–2^{36}
RAND*XX*	RAND	2^{24}–2^{32}	2^{28}–2^{36}

5.1 Strong Scaling

Given in Figure 3 is strong scaling experiments using BFS, PR and CC. The top plot gives total execution times for 1-256 ranks (when possible) on our smaller inputs, reported as the maximum time over all ranks. We observe scaling on all inputs up to 256 GPUs. Given that TW and FR both fully fit within the memory of a single V100 GPU, we consider these results to be extremely notable. The middle plots of Figure 3 plot just the communication times. We note that both communication and computation times both continue to show reductions as we scale, with the exception of the smallest inputs with PageRank. However, we note communication times begin to dominate for all graphs and all applications at the largest scale. The jump between 4 and 16 ranks in some tests is due to all 4 rank runs being on a single node and not using the network.

The bottom plot gives our speedups from 16 ranks, the smallest number of ranks for which we were able to obtain results for all graph and algorithm combinations. As discussed, a downside of a 2D distribution is that communication volume and work per rank can scale by $O(\frac{N}{\sqrt{p}})$. Thus, when strong scaling a bandwidth-intensive application, one might expect speedups to be theoretically bounded at the extreme end as $p \rightarrow \infty$ by a factor of $\sqrt{p}$, which we also plot for comparison. We observe most speedup values from $16 \rightarrow 256$ GPUs being in the near-optimal range of 3-4× $\approx \sqrt{\frac{256}{16}}$. However, as noted, the primary benefit of a 2D distribution is with processing larger inputs, represented by our weak scaling experiments.

5.2 Weak Scaling

In Figure 5, we plot results from our weak scaling tests on RMAT and Erdös-Renyí random graphs along with times from 1 rank

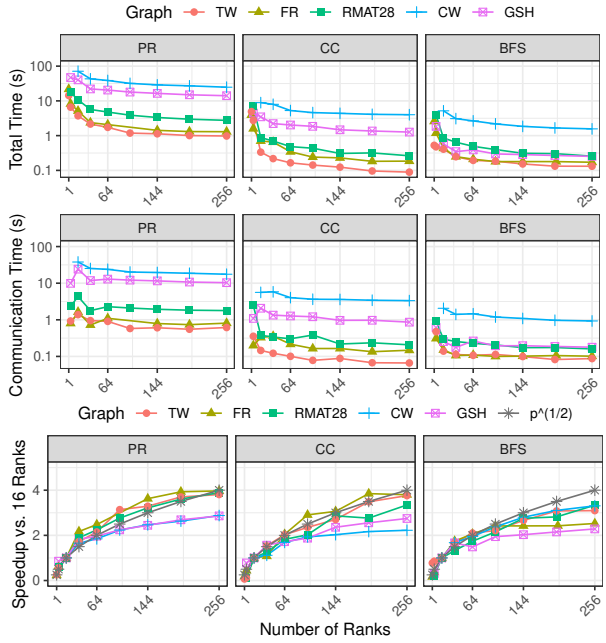


Figure 3: Strong scaling with total times (top), communication times (middle), and speedups (bottom) from 1 to 256 ranks on our benchmark tests.

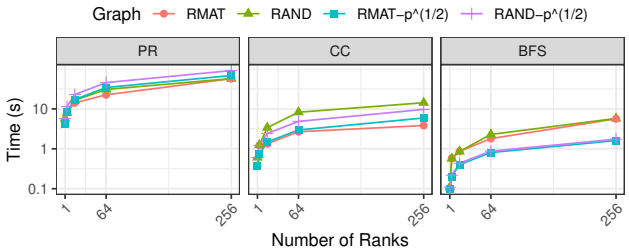


Figure 4: Weak scaling on RMAT and Random Graphs.

scaled by a $\sqrt{p}$ factor. We use graphs generated to have 2^{24} vertices and 2^{28} edges per rank. We observe that the timings follow the $\sqrt{p}$ factor discussed before, with all timings from all experiments just under doubling for every 4× increase in rank count, indicating that our methods are approaching the theoretical limits of efficiency. The exception is BFS, where the single GPU runs are correspondingly faster due to the relatively higher communication cost for the algorithm. Overall, these results are incredibly promising, and we hope to scale out even farther, pending access to larger test systems.

5.3 WDC Results

Figure 3 plots timing results from the same benchmark algorithms running on WDC from 100 to 400 ranks. Here, we consider the total time as the proportion of time spent in communication and computation phases. The maximum time over all ranks for each is reported. Again, we note that overall times scale nicely from 100 to 400 ranks, achieving speedups of about 2× for all algorithms,

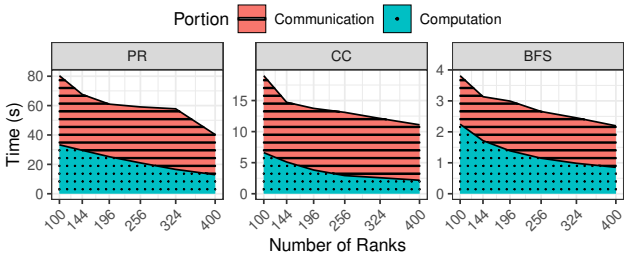


Figure 5: Computation and communication on WDC from 100 to 400 ranks.

matching the expected $O(\sqrt{p})$ factor again. We observe that computation and communication also scales for all algorithms, though the speedup is less for communication, as expected.

5.4 Sparse Communications and Vertex Queues

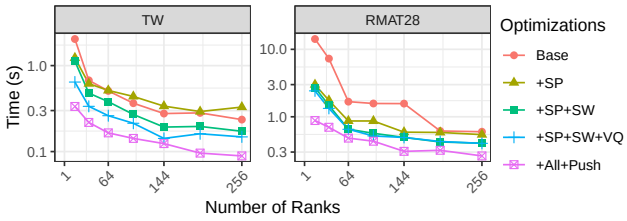


Figure 6: Effect of optimizations on Color Propagation CC performance.

Next, we use the CC algorithm to demonstrate the effects of our implemented communication and workload strategies. See Figure 6, where we compare pull updates with dense communications and no vertex queue (Base), fully using sparse communications (+SP), switching from dense to sparse communications (+SP+SW), adding in a vertex queue (+SP+SW+VQ), and then utilizing push updates with all strategies (+All+Push). We note the differences are significant, equating to an order of magnitude. While we can only show a couple inputs for space, we note that we observe these relative speedups consistently across the other inputs and that these speedups are also consistent with the other implemented algorithms that use them in part or in full (BFS, LP, MWM).

5.5 Non-square Distributions

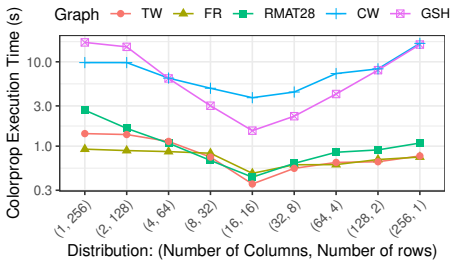


Figure 7: Non-square results with CC by varying C, R with 256 total ranks.

It is established that a 'square' 2D distribution (where $R = C$) minimizes communication. While the bulk of our tests use square distributions, we also study the effects of non-square distributions on performance. We plot results in Figure 7 of running CC on 256 ranks while varying R and C among all possible combinations. We first note the obvious in that a 16×16 distribution is optimal. However, performance does not significantly degrade right in the vicinity, especially for larger inputs, and particularly when decreasing the number columns. We observe about a 1.4× slowdown from (32,8) to (16,16). Note that CC is a push implementation, so the more expensive reduction communication is along the column group. These results suggest that performance scaling is still possible, even without a nice square number of ranks, and that one should bias towards minimizing the reduction direction of the implemented algorithm.

5.6 Complex Algorithms

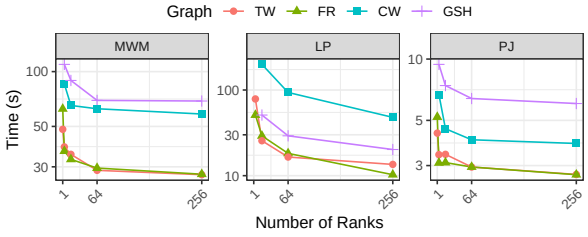


Figure 8: MWM (left), LP (middle), and PJ (right) strong scaling from 1 to 256 ranks on the real inputs.

We further collect some base strong scaling results using the implemented complex algorithms of MWM, LP, and PJ. We display these results in Figure 8. Overall, we observe strong scaling to 256 ranks for almost all methods and inputs. The overall execution times for MWM and PJ tend to plateau more significantly compared to our other algorithms, due to problem complexity and necessary communications to synchronize states. LP execution times exhibit better scaling trends under the 2.5D approach, having proportionally less communication cost and more computation. Nevertheless, the performance improvements are consistent across inputs, demonstrating the applicability of 2D distributions for algorithms with complex communication requirements.

5.7 Comparisons to Prior Art

We consider two direct comparisons to existing distributed GPU graph frameworks. Our first comparison is to Gluon-GPU [9, 14], part of the Galois library. While this code offers the fastest generalized distributed GPU performance we have found in the literature, we note the code has not been maintained for close to 6 years and has several outdated dependencies, requiring significant effort to run on our test systems. We ran comparisons on TW, FR, and RMAT28 using our three benchmark algorithms (BFS, CC, PR) on AiMOS. We were unable to successfully run GSH or CW due to memory allocation failures. We use their 2D CVC (cartesian vertex cut) partitioning with the push variants for BFS and CC and the pull variant for PR along with no thread binding and asynchronous communications, as suggested by the authors for best performance. We give our comparison in Figure 9.

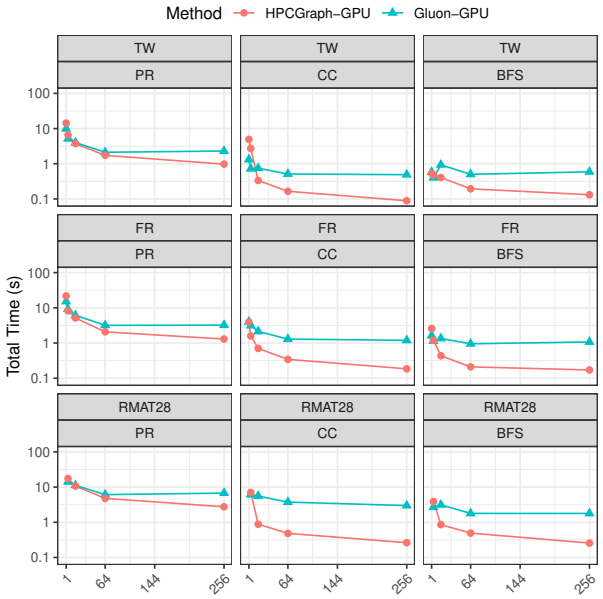


Figure 9: Our method and Gluon-GPU running from 1 to 256 ranks on TW (top), FR (middle), and RMAT28 (bottom) while processing PR (left), CC (middle), and BFS (right).

We note that performance of our method approximately matches Gluon-GPU on single rank and single node runs (ranks of 1 and 4), but Gluon-GPU suffers significant relative performance degradation when communicating across the network. Gluon-GPU does not scale at all past 64 ranks on the majority of tests. We note that 'Gluon', the communication layer, was built for general-purpose communications, and the 2D CVC distribution is built on top of it. This adds overhead relative to our optimized 2D communication methods, and it is likely the reason for the observed scaling results.

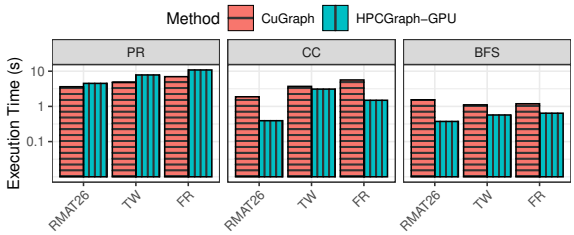


Figure 10: CuGraph comparison on PR, CC, and BFS.

We also compare to the latest CuGraph release (v25.04), which utilizes a 2D distribution methodology for PageRank [16]. However, as we were unable to get it successfully running on AiMOS after extensive efforts, we used 4×A100s on zepy for this comparison. We plot a comparison for our benchmark algorithms in Figure 10. We use RMAT26 as RMAT28 (and the other larger graphs) did not run on CuGraph on this system. We ran 20 iterations of PR for both and note an average slowdown of 1.47× for our code, likely due the fact that CuGraph uses optimized linear algebra routines, instead of a general purpose graph computational model, and computation

times dominate at the small single-node scale. We otherwise observe average speedups of 3.25× on CC and 2.64× on BFS for our code.

6 Concluding Remarks

In this paper, we introduce a 2D distributed GPU-based graph processing methodology for large scale inputs. We detail its implementation and show its efficacy on a wide range of benchmark and complex graph analytic algorithms. We leverage a variety of dense and sparse communication patterns, vertex queues, and workload balance techniques, and we show scaling near the theoretical limits of 2D methods on up to 400 GPUs, using graphs with up to billions of vertices and hundreds of billions of edges. This work significantly outperforms existing methods across a series of comparative tests.

Acknowledgments

This work is supported by the National Science Foundation under Grant No. 2047821 and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) Program through the FASTMath Institute under Contract No. DE-SC0021285 at the Rensselaer Polytechnic Institute, Troy NY.

References

[1] Junya Arai, Masahiro Nakao, Yuto Inoue, Kanto Teranishi, Koji Ueno, Keiichiro Yamamura, Mitsuhisa Sato, and Katsuki Fujisawa. 2024. Doubling Graph Traversal Efficiency to 198 TeraTEPS on the Supercomputer Fugaku. In *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*. IEEE Computer Society, 1616–1629.

[2] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–10.

[3] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 93–104.

[4] Ian Bogle and George M Slota. 2022. Achieving Speedups for Distributed Graph Biconnectivity. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[5] Ian Bogle and George M Slota. 2022. Distributed algorithms for the graph biconnectivity and least common ancestor problems. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1139–1142.

[6] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. 2013. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 50, 12 pages.

[7] Ümit V Çatalyürek and Cevdet Aykanat. 2001. A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices.. In *IPDPS*, Vol. 1. Citeseer, 118.

[8] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: graph processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1804–1815.

[9] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. 2019. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 15–28.

[10] Xinbiao Gan, Guang Wu, Shenghao Qiu, Feng Xiong, Jiaqi Si, Jianbin Fang, Dezun Dong, Chunye Gong, Tiejun Li, and Zheng Wang. 2024. GraphCube: Interconnection Hierarchy-aware Graph Processing. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 160–174.

[11] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[12] Bruce Hendrickson, Robert Leland, and Steve Plimpton. 1995. An efficient parallel algorithm for matrix-vector multiplication. *International Journal of High Speed Computing* 7, 01 (1995), 73–88.

[13] Vishwesh Jatala, Roshan Dathathri, Gurbinder Gill, Loc Hoang, V. Krishna Nandivada, and Keshav Pingali. 2020. A Study of Graph Analytics for Massive Datasets on Distributed GPUs. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

[14] Vishwesh Jatala, Roshan Dathathri, Gurbinder Gill, Loc Hoang, V. Krishna Nandivada, and Keshav Pingali. 2020. A Study of Graph Analytics for Massive Datasets on Distributed Multi-GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 84–94.

[15] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment* 11, 3 (2017), 297–310.

[16] Seunghwa Kang, Joseph Nke, and Brad Rees. 2022. Analyzing Multi-trillion Edge Graphs on Large GPU Clusters: A Case Study with PageRank. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[17] Mohsen Koohi Esfahani, Paolo Boldi, Hans Vandierendonck, Peter Kilpatrick, and Sebastiano Vigna. 2023. On Overcoming HPC Challenges of Trillion-Scale Real-World Graph Datasets. In *2023 IEEE International Conference on Big Data (BigData)*. 215–220.

[18] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, et al. 2018. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 706–716.

[19] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[20] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! but at what {COST}?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[21] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM Sigplan Notices* 47, 8 (2012), 117–128.

[22] Robert Preis. 1999. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 259–269.

[23] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 979–990.

[24] George M Slota, Jonathan W Berry, Simon D Hammond, Stephen L Olivier, Cynthia A Phillips, and Sivasankaran Rajamanickam. 2019. Scalable generation of graphs for benchmarking HPC community-detection algorithms. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[25] George M Slota and Christopher Brissette. 2024. Constant-Memory Graph Coarsening. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[26] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2015. High-performance graph analytics on manycore processors. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 17–27.

[27] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2016. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 293–302.

[28] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel Processing*. Springer, 90–109.

[29] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Network Science* 4, 4 (2016), 508–530.

[30] Ancy Sarah Tom and George Karypis. 2019. A 2D Parallel Triangle Counting Algorithm for Distributed-Memory Architectures. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP '19)*. Association for Computing Machinery, New York, NY, USA, Article 45, 10 pages.

[31] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[32] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. 2014. Graph processing on GPUs: Where are the bottlenecks?. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 140–149.

[33] Carl Yang, Aydın Buluç, and John D Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–51.

[34] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: a computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 301–316.