Scalable Breadth-First Search on a GPU Cluster

Yuechao Pan\textsuperscript{1,2}, Roger Pearce\textsuperscript{2}, and John D. Owens\textsuperscript{1}
\textsuperscript{1}University of California, Davis; \textsuperscript{2}Lawrence Livermore National Laboratory
Email: ychpan@ucdavis.edu, rpearce@llnl.gov, jowens@ucdavis.edu

Abstract—On a GPU cluster, the ratio of high computing power to communication bandwidth makes scaling breadth-first search (BFS) on a scale-free graph extremely challenging. By separating high and low out-degree vertices, we present an implementation with scalable computation and a model for scalable communication for BFS and direction-optimized BFS. Our communication model uses global reduction for high-degree vertices, and point-to-point transmission for low-degree vertices. Leveraging the characteristics of degree separation, we reduce the graph size to one third of the conventional edge list representation. With several other optimizations, we observe linear weak scaling as we increase the number of GPUs, and achieve 259.8 GTEPS on a Scale-33 Graph500 RMAT graph with 124 GPUs on the latest CORAL early access system.

Keywords—multi GPU; distributed graph processing; BFS

I. INTRODUCTION

Breath-First Search (BFS) on graphs is a fundamental and important problem that draws attention from a wide range of research communities. It is a building block of more advanced algorithms that involve graph traversals, such as betweenness centrality and community detection. Traversals can be highly parallelizable; however, achieving good performance is challenging, especially on scale-free graphs with wide ranges of degree distribution. This is due in part to low arithmetic computation density and irregular memory access patterns caused by the algorithm and graph topology. When running on distributed memory systems, high communication cost adds additional challenges to achieve good performance. Because of the importance and challenging nature of BFS at large scale, the High Performance Computing (HPC) graph community chose BFS as the first benchmark in the Graph500 \cite{graph500}. In addition to testing hardware capability of HPC machines, the Graph500 has been a catalyst for a series of algorithmic innovations \cite{ray1,ray2,ray3} for HPC graph analytics.

Graphics processing units (GPUs) provide more computing power and memory bandwidth than CPUs, and thus may be a good candidate for BFS. A fast BFS on GPUs is a challenge, however; irregular memory access patterns and the workload imbalance caused by widely different neighbor list lengths require optimizations to utilize the GPU hardware. Another challenge is the low per-processor memory size of the GPU (16 GB for the largest NVIDIA GPUs), much smaller than the CPU’s. Processing graphs larger than one GPU’s memory requires multiple GPUs and a distributed-memory implementation.

On the algorithms side, Beamer et al. \cite{ray4} introduced Direction-Optimizing (DO) BFS that significantly reduces traversal workload on power-law graphs, such as those used by Graph500 and social-network graphs. DOBFS’s workload reduction exacerbates the imbalance between highly efficient local GPU computation and the relatively limited communication bandwidth in and out of GPUs: a DOBFS implemented across multiple GPUs using existing techniques will almost surely be limited completely by communication bandwidth and will fail to scale. Our previous work \cite{pan1} shows DOBFS is the most challenging algorithm (among the five we tried) to scale even on multiple GPUs connected by high-speed PCIe bus. Targeting a multi-node GPU cluster, with its lower inter-node bandwidth, will be even more difficult. Existing work on GPU clusters does not target DOBFS because of these challenges.

Our work targets the growing trend of multiple-GPUs per compute node on HPC systems. CORAL/Sierra \cite{coral} will be Lawrence Livermore National Lab’s newest supercomputer. This system will contain only a few thousand compute nodes, compared to 10\times that amount in previous supercomputers. However, each node will feature more local computing power, mainly from four Volta GPUs, and more memory. This change further raises the computing power vs. communication bandwidth ratio. From a BFS perspective, the graph partition on each GPU will be larger, while the communication bandwidth for each GPU may not increase. Thus, the available bandwidth per unit graph size decreases significantly, and makes scaling on such systems harder.

In short, the challenges of a scalable (DO)BFS on GPU clusters are: 1) limited GPU memory, small per-GPU graphs will not be sufficient to utilize the computing power of latest GPUs; 2) irregular memory access patterns and unbalanced workload, which limits the local traversal performance; 3) high computing power to limited communication bandwidth ratio, making scaling difficult.

Our work in this paper \cite{pan1} targets an scalable implementation of (DO)BFS for the CORAL early access system at LLNL called Ray \cite{ray7} that can utilize the latest hardware. Our implementation makes no CORAL-specific optimizations but instead aims for generality to address any GPU cluster.

\textsuperscript{1}This work was performed, in part, under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Experiments were performed at the Livermore Computing facility.
We achieve scalable performance up to a scale-33 RMAT graph on this machine. The key idea allowing us to achieve scalable performance is that by separating high- and low-degree vertices [8], we design and implement a scalable computation and communication model that, for the first time, achieves scalable DOBFS on GPU clusters. We make the following contributions:

- Scalable BFS and DOBFS traversal results, reaching 260 GTEPS on a Scale-33 Graph500 RMAT graph with 124 GPUs, which is 18.5× better weak-scaled performance than best known GPU cluster work on the Graph500 list [1];
- An efficient graph representation that uses about half the memory as conventional CSR format;
- Fast and scalable local traversal on GPUs;
- A scalable communication model for DOBFS; and
- Several design decisions that may be useful for other programmers on similar systems.

II. RELATED WORK

A. Terminology

For the ease of discussion, we define the following terms, and use them in the latter sections of the paper.

graph A graph \( G(V, E) \) is defined by its vertices \( V \) and edges \( E \). In this paper, to study DOBFS scalability without doubling the storage size, we assume the graph is symmetric.

\( n = |V| \), the number of vertices in the graph.

\( m = |E| \), the number of edges in the graph.

\( p_{\text{rank}} \) the number of MPI ranks.

\( p_{\text{gpu}} \) the number of GPUs per MPI rank.

\( p = p_{\text{rank}} \cdot p_{\text{gpu}} \) the number of GPUs used.

\( g \) the inverse of inter-node communication bandwidth.

TH the degree separation threshold (section III-A).

delegates vertices with out-degree larger than \( TH \).

normal vertices vertices with out-degree at most \( TH \).

\( E_{nn}, E_{nd}, E_{dn}, E_{dd} \) normal \( \rightarrow \) normal, normal \( \rightarrow \) delegate, delegate \( \rightarrow \) normal, and delegate \( \rightarrow \) delegate edges.

\( d \) the number of delegates in the graph.

\( S \) the number of iterations (i.e., super-steps) of running BFS on the graph, bounded by the diameter of the graph.

B. Challenges with Scaling Directional Optimization

Directional optimization (DO) is a widely adopted optimization used in high performance BFS implementations. First described by Beamer et al. [4], it switches from the conventional forward-push (i.e. top-down) direction to the backward-pull (i.e. bottom-up) direction, when the workload of visiting all neighbors of the newly discovered vertices from the previous iteration is greater than trying to find only one previously visited parent of the unvisited vertices. The workload savings from skipping a vertex’s parent list once a valid one is found can be huge, and it is very efficient for graphs with small diameters, and dense cores, for example, social networks and RMAT graphs.

However, conventional DOBFS implementations face scaling issues in a cluster environment. When running in the backward-pull direction, each active (unvisited) vertex must know the status of all its possible parents. This information comes with a high communication cost. If the graph is 1D-partitioned, it forces broadcasting the newly visited vertices to all the peers that host their neighbors. In practice, this often results in broadcasting the newly visited vertices to every peer, which is \( 8m \) bytes in communication volume, and \( 8m/p \cdot g \) in communication time. If the graph is 2D-partitioned [10], it takes 2 hops to propagate the visiting status of vertices: one reduction across the row direction, and one broadcast across the column direction. Using \( n_t \) to indicate the number of vertices visited in the forward-push iterations, and \( S_b \) to indicate the number of iterations in the backward-pull direction, the total communication volume for the forward direction is \( 8n_t \sqrt{p} \log \sqrt{p} \) bytes, and it is \( 2n_tS_b(\log \sqrt{p})/8 \) bytes for the backward direction using compressed bit masks. The communication time is \( (4n_t + n_tS_b)/(\log \sqrt{p}) \cdot g. \)

When the graph size and the number of nodes increase at the same rate (weak scaling), the above communication cost will increase in the order of \( \sqrt{p} \), and this limits the scalability on large systems. There are also increases in the computation workload: instead of finding only one valid parent for each unvisited vertex, the 2D partitioned case tries to finds \( \sqrt{p} \) valid parents, one in each of the \( \sqrt{p} \) row-partitions of an unvisited vertex. When running on large clusters, i.e. \( \sqrt{p} \) is large, this workload increase defeats the workload saving purpose of DO. In summary, either a 1D or 2D partitioning within a cluster on a DOBFS presents significant scalability challenges.

Previous work on large-scale BFS falls into three categories. Single-node projects, either CPU or GPU, generally sustain the highest throughputs per processor but are limited by storage or compute to relatively modest graph scales. The largest CPU clusters (tens of thousands of nodes) have addressed the largest graph scales (\( \leq 36 \)), whereas smaller-sized GPU clusters (thousands of nodes) have not yet reached that scale. As a gross generalization, CPU implementations are limited in scalability by computation (they must add nodes to have more compute resources to process larger graphs), whereas the GPU ones are limited by memory size (they must add nodes to have more memory to store larger graphs). We summarize this work in figure [1].

C. BFS within Single Node

Using GPUs in the same node for BFS yields impressive per-node performance [5], [11], [12], but because all their...
communication is within a node and thus faster than within a cluster, their per-node performance is superior to cluster-based solutions. However, their graphs must fit into one node’s memory (GPU or CPU), and this inherently limits the maximum size of a processed graph.

To break this memory limitation, other researchers have used a shared memory architecture [9] or high-speed local storage [13]. The shared memory architecture is essentially multiple nodes with unified memory space, and it is less common than distributed memory architectures. Using fast local storage can help to process huge graphs with limited hardware resources, but moving large amounts of graph data limits overall traversal performance.

D. BFS on CPU Clusters

The Graph500 list is mostly CPU cluster implementations [14]–[16], which use a large amount of processors, primarily more than 10k, to reach the reported performance.

These implementations tend to use very specific graph representations [2], [14], which may not be GPU-friendly, because their complex memory access patterns bring extra irregularity and more branching conditions, both of which reduce achievable parallelism on GPUs. We instead choose a standard graph representation (CSR). We expect our BFS implementation will be used as a component of a complex workflow with many components that use standard formats for passing data between them. Using non-standard graph representations requires such a workflow to incur an additional cost of format conversion, to duplicate graphs, or to redesign other components, none of which are desirable.

These implementations also generally use 2D partitioning to distribute the graph across processors. 2D partitioning may introduce a high communication cost (Section IV-B). As subgraph sizes on each processor increase (to make full usage of more capable nodes as the number of nodes decreases), the data transmitted per node will increase together with the graph size, but the bisection network bandwidth will be lower as the network shrinks. Machine-specific network optimization could help, but this direction may make the implementation less applicable to other systems.

The recent implementation by Yasui et al. [9] shows a significant improvement in per-processor performance, using a shared memory system with 128 processors. In this work, the subgraph size on each processor is considerably larger than previous BFS work on CPU clusters. With upcoming supercomputers featuring a smaller number of nodes with more resources per node, using larger sub-graphs per processor may be more suitable for upcoming machines.

E. BFS on GPU Clusters

BFS on GPU clusters is a relatively recent topic of study [1], [17], [18], with some recent work focusing on a smaller number of GPUs [19]–[21]. None of these work demonstrates competitive per-node performance to single-node work, and none shows the combination of scalability and performance per node that we demonstrate in this work.

III. GRAPH REPRESENTATION

The key to a scalable DOBFS on a GPU cluster is to (a) maximize the fraction of the graph that can be stored on one GPU, thus allowing fast computation with no communication on that portion of the graph, and (b) optimize the communication between GPUs, which would otherwise limit scalability, even within a single node [5].

A. Separation of Vertices

Our design to accomplish these goals starts from a simple but powerful idea that we have pursued in previous work.
on CPU clusters [8]: separate the vertices into two sets by out-degrees, and treat them differently. The separation point between the two, the threshold out-degree \( TH \), is an important tuning parameter, and we will show how it affects the overall performance in upcoming sections. We call vertices with more than \( TH \) direct neighbors the delegates, and the rest normal vertices.

The intuition behind this design choice is that in local traversal, vertices at different ends of degree distribution should have different load balancing strategies; and in communication, vertices that almost every GPU touches should not be treated the same as those needed by very few GPUs. By separating vertices into different sets, we can pursue different strategies in graph representation, local traversal, and communication on those sets, which we describe below.

### B. Distribution of Edges

**Algorithm 1** Edge Distributor

Let \( P(v) = v \% p_{\text{rank}} \) and \( G(v) = (v / p_{\text{rank}}) \% p_{\text{gpu}} \)

1. for each edge \((u \to v)\) do:
2.  if \( u \) is normal then: to rank \( P(u), \text{GPU } G(u) \)
3.  else if \( v \) is normal then: to rank \( P(v), \text{GPU } G(v) \)
4.  else if \((\text{OutDegree}(u) < \text{OutDegree}(v))\) then:
5.    to rank \( P(u), \text{GPU } G(u) \)
6.  else if \((\text{OutDegree}(u) > \text{OutDegree}(v))\) then:
7.    to rank \( P(v), \text{GPU } G(v) \)
8.  else: to rank \( P(\text{min}(u, v)), \text{GPU } G(\text{min}(u, v)) \)
9. end if
10. end for

On scale-free graphs, most storage is devoted to edges, not vertices. We distribute edges to individual MPI ranks, and then to individual GPUs within the same rank, using the distributor described in Algorithm [1]. In it, we divide edges into four categories depending on the type of their source and destination vertices (normal or delegate). Our edge distributor has the following advantages:

**Simple** The location of an edge can be easily computed from its index locally without table lookup or remote query.

**Symmetric** Except for normal to normal edges, subgraphs on individual GPUs are symmetric. Since we make an edge pair of opposite directions for an undirected edge, they need to be on the same GPU to preserve the correctness of DOBFS without global traversal direction. Otherwise if the traversal directions of the edge pair are opposite to their respective directions, both edges will be ignored.

**Bounded size** The number of possible destination vertices for non-(normal to normal) edges are bounded: the number of normal vertices is at most \( n/p \), and the number of delegates is at most \( d \). Thus vertex indices for these edges can be represented as 32-bit numbers locally, and converted back to 64-bit when necessary for communication. This allows us to store more of the graph in a fixed-size memory.

**Balanced** This distribution prioritizes placement of vertices with lower out-degrees. Neighbor lists of high-degree vertices are distributed according to the destination vertices, and scattered across the entire cluster. The number of edges in the partitioned subgraphs on individual GPUs are very close to each other, giving each GPU a balanced workload.

Figure 2 illustrates our vertex separation and edge distribution strategies. Vertices 7 and 8 have out-degree more than \( TH \), which is 5 in this example, and they are converted to delegates 0 and 1, respectively. All partitions keep a copy of the delegates, and all edges involving the delegates are changed to the local copies. After this operation, only edges between normal vertices require communication across GPUs; all other edges are between two vertices on the same GPU. This is the right choice because normal vertices are the ones with the fewest neighbors and thus the least communication. Any delegate-related communication is performed using global reductions (details in section [V]).

### C. Efficient Graph Storage

The bounded-size feature of our edge distributor is critical for processing huge graphs within limited GPU device memory, and makes processing larger graphs using the same number of GPUs possible. When the number of local normal vertices and the number of delegates are bounded by \( n/p \), with the exception of destinations of \( nn \) edges, we can use
32-bit values to store local normal vertex and global delegate ids, instead of 64-bit values. This provides significant saving on memory storage footprint of the graph. As listed in Table I, the total memory usage for all subgraphs on the GPUs is \(8n + 8d \cdot p + 4m + 4|E_{nn}|\) bytes. In practice, when using suitable values of \(TH\) (Section VI-B), while still using CSR format for each sub-graphs, the above memory usage is only about one third of the conventional edge list format (16\(m\) bytes), and a little more than half of CSR format (8\(n\) + 8\(m\)) without the degree separation.

It is possible to utilize GPU memory and handle graphs larger than GPU memory, with different techniques [22], [23]. However, the current latency and bandwidth differences between GPU memory and the GPU-CPU connection would imposes a high performance penalty. This decision could be revisited when CORAL is fully equipped with NVLink2, which doubles CPU-GPU bandwidth, in the near future. In this paper, we only focus on graphs that fit in GPU memory.

IV. LOCAL COMPUTATION

On each GPU, we now have 4 different subgraphs: normal to normal (nn), normal to delegate (nd), delegate to normal (dn) and delegate to delegate (dd). While we could apply the exact same strategies to each of them, we note that their different characteristics motivate different load-balancing strategies for traversal (section V-A), different direction switching conditions for DObFS (Section IV-B), and different input to output to the communication model (section V). Because subgraphs can be processed in parallel, we can achieve some overlap between computation and communication in our processing pipeline (Fig. 3).

At a high level, we separate local traversal on the four subgraphs into a delegate stream and a normal stream, depending on the destination type of the edge, as two cudaStreams. Each stream begins with a “previsit” kernel, used to preprocess the inputs. This includes marking level labels for input vertices, filtering out duplicates and zero-out-degree vertices, forming the queues of vertices to be visited by the visit kernels, and calculating the workload for these kernels, which is important for the direction decisions in DO. Then each stream spawns a “visit” kernel for the two edge types in the stream. The two streams run independently of each other, except when dependencies are established explicitly (Fig. 3).

A. Forward Traversal

The visited status of delegates are maintained by bitmasks, with each delegate only occupying 1 bit. This is an effective way to store and communicate the status of high out-degree vertices. We use advanced load balancing techniques for the visiting kernels: the delegate to delegate visit kernel uses merge-based workload partitioning [24], because the dd subgraph covers a wide range of degree distribution, and has large average out-degrees; the other visit kernels use thread-warp-block dynamic workload mapping [25], based on the fact that the out-degree range of dn, nd, and nn subgraphs are all limited, and the average out-degrees are low.

B. Directional Optimization

Not all subgraphs benefit from directional optimization. We do not use DO for normal \(\rightarrow\) normal visits, because the nn subgraph on each GPU is not symmetric, the range of destination vertices of nn edges are unbounded, and most importantly, DO is not efficient for the very low in-degree nn subgraphs. Without separating the graph, skipping the nn portion from DO is impossible.

On each GPU, we keep a source list of the normal to delegate subgraph, i.e., all the normal vertices that have edges pointing to delegates. These are exactly the potential destination vertices in the reverse subgraph, i.e., the delegate to normal subgraph. When running in the backward-pull direction for a delegate to normal visit, we use the normal to delegate subgraph, and start from its source list. For the same purpose, we keep source masks for the dd and dn subgraphs. Keeping source lists and masks avoids vertices that may not find local parents, and provides more accurate workload prediction.

The traversal direction is decided based on a workload comparison, computed on each step, between the forward and the back directions. The forward workload \(FV\) is calculated by the previsit kernels as the sum of neighbor list workload, calculated by the previsit kernels as the sum of neighbor list workload, calculating the backward workload \(BV\) is calculated using the estimated number of parents to check until finding the first visited one. Let \(U\) being unvisited sources in the reversed graph, \(q\) the input frontier length, \(s\) the number of unvisited sources in the forward graph, \(a\) the probability that a potential parent is newly visited, and \(od(u)\) the out degree of \(u\); then:

\[
BV = \sum_{u \in U} ((1 - a)^{od(u)} + \sum_{i=0}^{od(u)-1} a(1 - a)^i)
\]

\[
= \frac{1 - (1 - a)^{od(u)}}{a}
\]

\[
\approx |U|/a; \text{ assuming } od(u) \text{ is large, and } a \text{ not too small}
\]

\[
= |U|q/(q + s)
\]
AllReduce
Frontier
Normal
Delegate
Masks
is large, because it is in the order of $O(p)$ whenever any update occurs in an iteration. The reduction is
GPU, or consumed by any GPU. We thus use a global
processing than ever before. Our scalable communication
Neighborhood is concentrated cores, the delegate updates will finish faster
that only the outputs from nn edge visits may result in
different GPUs). The communication time is $|E_{nn}| / p \cdot g$. Note that only the outputs from nn edge visits may result in
direct remote normal vertex updates: the results from dd and
nn edge visits are communicated via global delegate mask
reduction, and the updates from dn visits are always local,
as a result of our edge distributor (Algorithm 1). When
setting $TH$, the degree threshold, to an optimal value, the
nn edges are only a small portion of the graph, and the
resulting normal communication is a lot less than $m$.

The normal vertex exchange requires some extra local
computation, such as binning (group vertices need to be sent
to the same GPU together) and vertex number conversion
(from the 64-bit global ids used in nn edge destinations
to 32-bit local ids at destination GPUs). These computa-
tions are done on GPUs. The workload is in the order of
$O(|E_{nn}| / p)$ on each GPU for all iterations combined. This
is a small cost compared to the traversal workload, and does
not affect the scalability of our BFS implementation.

V. COMMUNICATION

Because local computation performance is increasing
more quickly than interconnect bandwidth, designing for
scalable communication is more important for graph pro-
cessing than ever before. Our scalable communication model—shown in Fig. 4—adopts different strategies for
delegates and normal vertices.

A. Communication for Delegates

The visited status of delegates may be updated by any
GPU, or consumed by any GPU. We thus use a global
reduction to gather and distribute the delegate mask updates
whenever any update occurs in a iteration. The reduction is
done in two phases: locally across peer GPUs, and globally
across different MPI ranks. During the local phase, all
GPUs in the same MPI rank push their updated masks to
GPU0, and GPU0 performs the reduction in parallel. During
the global phase, only GPU0 (more accurately, the CPU
thread that controls GPU0) participates, and all GPUs in
the same MPI rank consume the resulted masks for the next
iteration. We utilize fast GPU-GPU data channels and the
GPU’s parallel computing capability for the local phase, and
efficient MPI (I)AllReduce calls for the global phase.

The cost of this delegate communication is small. For each
iteration that has updates to the delegate masks, the commu-
nication volume is $2d_{p_{ranks}}/8$ bytes, and the communication
time is $d \log p_{ranks}/4 \cdot g$. assuming the global reduction is
done in a tree-like manner. The delegate reduction might run
on every iteration, which gives the total communication
cost as $d \log p_{ranks}/4 \cdot g$. However, for graphs with more
concentrated cores, the delegate updates will finish faster
than normal vertices, which reduces the number of iterations
that require delegate communication. In practice, we keep $d$
(the number of delegates) low, so that the size of delegate
masks, $d/8$, is under the limit of several tens of MBs.

For BFS, each delegate only needs 1 bit for the visited
status, indicating whether that delegate been visited or not;
as a result, we can use compact bit masks. For other graph
algorithms that require more bits of state (i.e., ranking scores
for PageRank), the communication volume for delegates will
increase accordingly. Our delegate communication should
still be scalable, except for computation that involves updates
to the delegate status for too many iterations.

B. Communication for Normal Vertices

The basic communication model for normal vertices is
point-to-point transmission via MPI_Isend and MPI_Irecv.
We use a non-blocking version to keep the pipeline running
and take advantage of possible workload overlaps. The total
communication volume is $4|E_{nn}|$ bytes, assuming each nn
edge is a cutting edge (i.e., a edge with end points on two
different GPUs). The communication time is $4|E_{nn}| / p \cdot g$.

Figure 4: Communication flow

Start from the forward-push direction, with two
direction-switching factors $factor0$ and $factor1$, the visiting
direction is decided as:

- if current direction is forward, and $FV > factor0 \cdot BV$,
  then switch to backward;
- if current direction is backward, and $FV < factor1 \cdot BV$,
  then switch to forward;
- otherwise keep current direction.

No matter which direction a visiting kernel takes, it only
affects the kernel itself, and the input and the output are the
same. The three visiting kernels with DO have three sets of
direction-switching factors. This allows the kernels to switch
for their own optimized conditions.

Our strategy for DOBFS results in a smaller workload
than a 2D partitioning strategy. In our strategy, for normal
vertices, only one GPU must do the reverse visiting for
each individual normal vertices. Only the delegates may
need to have more than one GPUs visiting their parents,
and moreover, the delegates are only a small portion of
all vertices. Let $m'$ be the number of edges the DOBFS
algorithm would need to visit if the graph was traversed
by a single processor, then the workload of our DOBFS
implementation would be bounded by $m' + dp \cdot b$, where $b$
is the average number of parents a delegate must search on
each GPU before finding a visited one. While keeping $d$ in
the order of $O(n/p)$, the term $dp \cdot b$ is scalable even when
$p$ is large, because it is in the order of $O(nb)$ and $b$ is not
a large number—only delegates with very large out-degrees
are distributed on large number of nodes, and delegates
with large out-degrees tend to be close to portions of the
graph with high connectivity, which reduces the number of
neighbors to try before finding a visited one.

V. COMMUNICATION

Because local computation performance is increasing
more quickly than interconnect bandwidth, designing for
scalable communication is more important for graph pro-
cessing than ever before. Our scalable communication model—shown in Fig. 4—adopts different strategies for
delegates and normal vertices.

A. Communication for Delegates

The visited status of delegates may be updated by any
GPU, or consumed by any GPU. We thus use a global
reduction to gather and distribute the delegate mask updates
whenever any update occurs in a iteration. The reduction is
done in two phases: locally across peer GPUs, and globally
across different MPI ranks. During the local phase, all
GPUs in the same MPI rank push their updated masks to
GPU0, and GPU0 performs the reduction in parallel. During
the global phase, only GPU0 (more accurately, the CPU
thread that controls GPU0) participates, and all GPUs in
the same MPI rank consume the resulted masks for the next
iteration. We utilize fast GPU-GPU data channels and the
GPU’s parallel computing capability for the local phase, and
efficient MPI (I)AllReduce calls for the global phase.

The cost of this delegate communication is small. For each
iteration that has updates to the delegate masks, the commu-
nication volume is $2d_{p_{ranks}}/8$ bytes, and the communication
time is $d \log p_{ranks}/4 \cdot g$. assuming the global reduction is
done in a tree-like manner. The delegate reduction might run
on every iteration, which gives the total communication
cost as $d \log p_{ranks}/4 \cdot g$. However, for graphs with more
concentrated cores, the delegate updates will finish faster
than normal vertices, which reduces the number of iterations
that require delegate communication. In practice, we keep $d$
(the number of delegates) low, so that the size of delegate
masks, $d/8$, is under the limit of several tens of MBs.

For BFS, each delegate only needs 1 bit for the visited
status, indicating whether that delegate been visited or not;
as a result, we can use compact bit masks. For other graph
algorithms that require more bits of state (i.e., ranking scores
for PageRank), the communication volume for delegates will
increase accordingly. Our delegate communication should
still be scalable, except for computation that involves updates
to the delegate status for too many iterations.

B. Communication for Normal Vertices

The basic communication model for normal vertices is
point-to-point transmission via MPI_Isend and MPI_Irecv.
We use a non-blocking version to keep the pipeline running
and take advantage of possible workload overlaps. The total
communication volume is $4|E_{nn}|$ bytes, assuming each nn
edge is a cutting edge (i.e., a edge with end points on two
different GPUs). The communication time is $4|E_{nn}| / p \cdot g$.
Note that only the outputs from nn edge visits may result in
direct remote normal vertex updates: the results from dd and
nd edge visits are communicated via global delegate mask
reduction, and the updates from dn visits are always local,
as a result of our edge distributor (Algorithm 1). When
setting $TH$, the degree threshold, to an optimal value, the
nn edges are only a small portion of the graph, and the
resulting normal communication is a lot less than $m$.

The normal vertex exchange requires some extra local
computation, such as binning (group vertices need to be sent
to the same GPU together) and vertex number conversion
(from the 64-bit global ids used in nn edge destinations
to 32-bit local ids at destination GPUs). These computa-
tions are done on GPUs. The workload is in the order of
$O(|E_{nn}| / p)$ on each GPU for all iterations combined. This
is a small cost compared to the traversal workload, and does
not affect the scalability of our BFS implementation.
We also tried two optimizations to reduce communication cost. The first one is called Local All2all: prior to the remote vertex exchange, we first run a local exchange to gather vertices going to GPU ranks on the local GPU. As a result, normal vertex exchanges only occur among GPU ranks, among GPU ranks, etc., but never between GPU ranks and GPU ranks, etc. This reduces the number of communication pairs from \( \frac{p^2}{p_{g}} \) to \( \frac{p^2}{p_{g}} \), each of which has more vertices to send. In turn, this allows a second optimization, unification, which removes duplicated vertices going to the same GPU. However, because relatively few individual nn edge end vertices are on a given GPU or node, with the expected value capped by \( TH/p_{\text{rank}} \), the chance to find duplications is small, and may not be sufficient to overcome the extra computation. We show our findings in the next section.

Combining the communication for delegates and normal vertices together, we have a model that has at most \( dp_{\text{ranks}}/4 \cdot S + 4|E_{\text{nn}}| + d \log p_{\text{ranks}}/4 \cdot S + 4|E_{\text{nn}}|/p \cdot g \) communication cost. For graphs that have a small number of vertices covering a large portion of edges, the number of iterations \( S_{g} \) that need delegate masks exchange, is less than \( S \); for the graphs we tested, \( S' \) is about half of \( S \). With suitable values of \( TH \) (Section VI-B), we saw delegate mask reduction and normal vertex exchange taking roughly the same amount of time. Under these conditions, we approximate our communication cost to \( d \log p_{\text{rank}}/4 \cdot S_{g} \). We also keep \( d \) in the same scale of \( n/p \), more accurately, under the value of \( 4n/p \) in practice. As a result, the communication cost is \( n \log p_{\text{rank}}/p \cdot S_{g} \). It starts from \( n \cdot S_{g} \) on single node, and grows in the order of \( \log p_{\text{rank}} \) when \( n \) and \( m \) increase at the same rate as \( p \) (weak scaling). This growth is slow, and more scalable than the \( \sqrt{p} \) growth order of conventional 2D partitioning methods. Thus, we argue that our communication model is more scalable.

VI. Results

A. Testing Environment

1) Hardware: Our implementation targets an early access system (Ray) of LLNL’s upcoming CORAL/Sierra supercomputer. The current system has more than 40 compute nodes, each features two 10-core IBM Power8+ CPUs at 2.06 GHz and 256 GB CPU memory. Each CPU has two NVIDIA Tesla P100 GPUs; the two GPUs and the CPU are connected by high-speed NVLink with 40 GB/s bandwidth in each direction. Each socket has an EDR 100 Gbps InfiniBand connection to a network with FatTree topology.

Because the interconnection speed is higher than a conventional cluster, and because GPUs achieve their best performance only when fully occupied by a sufficient workload, we first test how the network performs with different message sizes. In an experiment, we use 32 nodes, each with one MPI rank, and 4 CPU threads, and each thread sending MB-sized data to all threads on other nodes, to simulate a scenario where each of the 128 GPUs sends out data to the 124 GPUs on other nodes. After sweeping through the message size from 128 kB to 16 MB, we found that the optimal message size is about 4 MB for data larger than 2 MB. While this is much larger than normal MPI usage, it is the best fit for the GPUs. With smaller data (under 2 MB), the network appears to do a better job with caching, and the differences between message sizes are not that significant.

2) Software: The cluster runs on 64b Linux with GPU driver version 384.59. The compilation toolchain includes gcc 4.9.3, cuda 9.0.167, and spectrum-mpi 2017.08.24 with OpenMP support. nvcc options are -O3 -std=c++11 –expt-extended-lambda. The GPU target is set to the hardware’s SM version (i.e. 6.0 for P100 GPUs).

We note the following current limitations with Ray. Hopefully they will be addressed in the full system.

- No NIC-GPU RDMA support; instead all NIC-GPU traffic goes through CPU memory.
- No asynchronous GPU memory copy support by the MPI implementation; this forces our implementation to copy data from GPU memory with appropriate cudaMemcpyAsync calls to CPU memory, then issue MPI calls from the CPU memory, and then copy from CPU to GPU on the receiving end.
- Random delays of \( \sim 100 \text{ ms} \) when consuming data on CPU right after receiving them from unblocking MPI calls; as a result, the CPUs are only used as GPU workload schedulers and data movement controllers.
- Degraded data movement performance between CPUs and GPUs on some nodes; only up to 31 nodes with 124 GPUs are used to avoid this issue.

3) Reporting: We use RMAT graphs for testing our BFS implementation. The RMAT graph generator is a distributed GPU implementation conforming to the Graph500 specifications [1]. The edge factor is 16, and the RMAT parameters are \( A, B, C, D = 0.57, 0.19, 0.19, 0.05 \). For a given RMAT graph at scale \( N \), the number of vertices \( n \) is \( 2^{N/2} \); the number of edges \( m \), after making the graph undirected by edge doubling, is \( 2^{N} \cdot 32 \). However, following the Graph500 specification [1], we only use \( m/2 = 2^{N} \cdot 16 \) to calculate the edge traversal rate. Vertex numbers are randomized using a deterministic hashing function after edge generation.

Our implementation outputs the hop-distances from the source vertex, instead of the BFS tree required by Graph500. The cost of building such tree should be low in our implementation: only the destination vertices of nn edges, without possible delegate parents, would need to communicate their parent information at the end of BFS; vertices visited by dd, dn, and nd kernels can get the parent information locally, with almost no extra cost to the local computation.

For each reported data point, we executed 140 BFS runs with randomly generated sources; only the ones that executed for more than 1 iteration are considered. We report the geometric mean (i.e. the harmonic mean) of edge traversal rates (in the unit of Giga Traversal Edges Per Second,
GTEPS) or elapsed times (in the unit of milliseconds, ms). We use number of nodes × number of MPI ranks per node × number of GPUs per MPI rank to denote the hardware for our experiments and for prior work. For example, 4 × 1 × 2 means 4 nodes with 1 MPI rank per node, and 2 GPUs per MPI rank, 8 GPUs in total.

B. Parameter Settings

Our implementation has several parameters and options that can be used to tune performance. The single most important parameter is the degree threshold $TH$. By changing $TH$, we are balancing the percentages of delegates and nn edges. Generally we want $d$ to be in the same order of number of local vertices $n/p$; in our experiment, we keep $d$ under $4n/p$. It would also be desirable to keep the nn edge percentage under 10%. Figure 5 shows how $TH$ changes the distributions of vertices and edges on the RMAT scale 30 graph. Any $TH$ in the range of $[16, 512]$ will satisfy our goal. We sweep this range to see the resulted performance, as shown in Figure 6. The actual range that gives the best performance for both BFS and DOBFS is quite wide, from 45 to 90; we use 64 in our experiments.

With a similar experiment, we suggest degree thresholds for a wide range of graph scales (Fig. 7). The optimal $TH$ increases at the rate of about $\sqrt{2}$ per scale. For scales up to 33, the delegate percentage is well below the $4n/p$ line; at scale 33, the delegate percentage is 1.75%, and the $4n/p$ line is at 3.23%. The nn edge percentages increases slightly, to 6.3% at scale 33, which is still a small and acceptable percentage. For larger scales that may lead to insufficient GPU memory caused by a large number of delegates or nn edges, the following options may be considered: 1) increase $TH$ to decrease the number of delegates, as a range of values yield similar performance; 2) increase $p$ to reduce the memory usage per GPU, as there is no limitation on how many GPUs can be used, provided that the GPU memory is sufficient. With these two options, we believe our method could continue to scale on larger GPU clusters.

We can use our implementation with several options: directional optimization (DO), local all2all (L), uniquify (U), blocking global mask reduction (BR) using $MPI_{Allreduce}$ or unblocking reduction (IR) using $MPI_{Iallreduce}$, and hardware configuration (e.g., $a \times 2 \times 2$ or $a \times 1 \times 4$ setup). Figure 8 shows how different options affect the timings of different parts of the BFS runtime. DO cuts the computation time by a factor of three, even when the workload is distributed on 64 GPUs. L and U add a small amount of time to local data exchange, but do not have a significant impact on the global communication time, mainly

---

### Table: Degree Thresholds and Edges Percentages

<table>
<thead>
<tr>
<th>Scale</th>
<th>Delegate Percentage</th>
<th>nn Edge Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Figure 5: Distribution of different kinds of edges and delegates, as a function of degree threshold, for a scale-30 RMAT graph.

---

### Figure 6: Traversal rates vs. degree threshold, for a scale-30 RMAT graph with $4 \times 1 \times 4$ GPUs.

---

### Figure 7: Suggested degree thresholds for different RMAT scales, with the resulting delegate and nn edges percentages. The $4/2^{N-26}$ line is the percentage of $4n/p$ vertices when using scale 26 RMAT graph per GPU.

---

### Figure 8: Effect of different options on performance. DO stands for directional optimization; L for local-all2all; U for uniquify; IR for unblocking delegate mask reduction; and BR for blocking reduction. The graph is RMAT Scale 32 with degree threshold 128, with a $16 \times 2 \times 2$ hardware configuration on the left and $16 \times 1 \times 4$ on the right.
because the degree threshold $TH$ is so low that we see few duplications in the normal vertex exchange. BR significantly reduces the communication time in this example, although the actual volumes of communication are the same. This may be a consequence of an unoptimized implementation of $MPI_{\text{Allreduce}}$, a newly available feature on this machine.

When running on fewer than 8 nodes, the communication time of IR is less than that of BR, we hope the same applies to larger number of nodes so that the advantage of workload overlapping can be fully explored. The sum of all parts in one column is more than the elapsed time of BFS, because different parts may overlap. For example, visiting from the delegates can start once the delegate masks are received without waiting for the normal vertices. For this particular experiment, the overlaps reduce the running time by about 10% on average when compared to the sum of all parts.

For each of the three subgraphs that apply DO, our implementation has two direction switching factors that decide when to change traversal direction. For RMAT, once the traversal switches to the backward direction, it does not need to change back; as a result, we only have three factors to decide. After scanning these factors from $10^{-8}$ to 10 for the best performance, we found out that all three factors have a wide range of near-optimal values; in fact, the same range $(0.5, 0.05, 1 \times 10^{-7})$ for dd, dn, and nd subgraphs) applies to almost all configurations that follow the weak scaling curve and the suggested $TH$ values. From our experience, these selections are similar for the same type of graphs.

C. Overall Results and Comparisons

Figure 9 shows overall weak scaling curves, with ~scale-26 RMAT graphs on each GPU up to 124 GPUs. In this range it is mostly linear, peaking at 259.8 GTEPS for RMAT scale 33 on 124 GPUs. From 16 GPUs to 32 GPUs, we switch from $MPI_{\text{Allreduce}}$ to $MPI_{\text{Allreduce}}$, as discussed earlier in this section, which introduces performance increases higher than the average.

Figure 10 shows detailed timing for DOBFS and BFS at different scales. Local visiting time grows slowly, only $4 \times$ over 7 scales for DOBFS as the graph size and the number of GPUs increase to $128 \times$. The BFS computation time increases to $3 \times$ for the same range. This shows the computation is scaling as expected. The communication grows slightly faster than the computation, especially from scale 32 to 33. This may be caused by the increases in number of delegates and edges, as shown previously in Figure 5 and Section VI-B, or it may be traffic conditions in the network, as about 70% of the nodes in the cluster are actively transmitting large amount of data. Because our implementation overlaps communication and computation, we mitigate the effects of this increase in communication cost. Both computation and communication appear to successfully scale throughout this range of RMAT sizes.

We compare our results with previous efforts in Table I. When compared against single-node multi-GPU Gunrock [5], this work is a little slower when using the same graphs, which may be the effect of more optimizations in Gunrock’s traversal kernels. As we add more GPUs in this work, we see the gap in performance is narrowing, which indicates better scalability; and the memory size improvements we made in this paper allows us to process larger graphs on one node, up to scale 28 on 4 GPUs, than any other GPU-based previous work.

Compared to Bernaschi et al. [18], our work achieves about 31% of their performance with only 3% the number of GPUs. Although the GPUs they used are not as new as ours, the $10 \times$ per-GPU performance shows our efficient computation and communication. Compared to Krajecki et al. [20], we achieve $4 \times$ the performance using only one eighth the number of GPUs.

Yasui et al.’s flagship CPU implementation [9] uses a similar number of processors; we obtained $1.49 \times$ the performance of their work, which we believe is partially because of the performance advantages of the GPU. We also demonstrate slightly better performance than Buluç et al. [16] despite their $8.4 \times$ more processors.
VII. CONCLUSIONS

Base on the idea of separating vertices by out-degrees, we implemented a scalable BFS, consisting of an efficient graph representation, scalable and fast local computation kernels, and a scalable communication model. With 124 P100 GPUs on the CORAL EA system, we achieved 259.8 GTEPS on the scale 33 RMAT graph. The close to linear weak scaling on the CORAL EA system, we achieved 259.8 GTEPS on 124 P100 GPUs in the range of thousands, when they are available, could bring us more findings and potentially more solutions.

We believe our work provides a better alternative to conventional 2D partitioning methods for scaling DOBFS, and it is more aligned with the latest trend of supercomputers and large systems. Further exploration using even more GPUs in the range of thousands, when they are available, could bring us more findings and potentially more solutions for the scalability problem.

It will also worth the efforts to investigate graph applications beyond BFS. These applications need more local computation than just neighborhood queries, more communication than just 1-bit visited status, and more attributes on vertices and edges than a single label. Most techniques and optimizations for BFS could still be applicable, but studies in the components of graph representation, local computation and remote communication, under more complex application situation will lead to higher impacts of the research works.

REFERENCES