LLNL-CONF-658291

Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates

R. Pearce, M. Gokhale, N. M. Amato

August 6, 2014

Supercomputing
New Orleans, LA, United States
October 16, 2014 through October 21, 2014

Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates

Roger Pearce, Maya Gokhale
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory; Livermore, CA
{rpearce, maya}@llnl.gov

Nancy M. Amato
Parasol Laboratory; Dept. of Computer Science and Engr.
Texas A&M University; College Station, TX
amato@cse.tamu.edu

***Abstract*—At extreme scale, irregularities in the structure of scale-free graphs such as social network graphs limit our ability to analyze these important and growing datasets. A key challenge is the presence of high-degree vertices (hubs), that leads to parallel workload and storage imbalances. The imbalances occur because existing partitioning techniques are not able to effectively partition high-degree vertices.**

We present techniques to distribute storage, computation, and communication of hubs for extreme scale graphs in distributed memory supercomputers. To balance the hub processing workload, we distribute hub data structures and related computation among a set of *delegates*. The delegates coordinate using highly optimized, yet portable, asynchronous broadcast and reduction operations. We demonstrate scalability of our new algorithmic technique using Breadth-First Search (BFS), Single Source Shortest Path (SSSP), K-Core Decomposition, and Page-Rank on synthetically generated scale-free graphs. Our results show excellent scalability on large scale-free graphs up to 131K cores of the IBM BG/P, and outperform the best known Graph500 performance on BG/P Intrepid by 15%.

I. Introduction

Analyzing large scale-free graphs such as social networks is a challenging and important problem. Graphs are used in a wide range of fields including computer science, biology, chemistry, and the social sciences. These graphs may model complex relationships between individuals, proteins, chemical compounds, etc. Graph datasets from many important domains can be classified as *scale-free*, in which the vertex degree-distribution asymptotically follows a power law distribution. In scale-free graphs, the presence of high-degree vertices (hubs) can create significant challenges for balancing storage, processing, and communication of parallel and distributed algorithms. The imbalances occur because existing partitioning techniques are not able to effectively partition high-degree vertices.

In this work, we present a new graph partitioning technique and computation model that distributes the storage, computation, and communication for hubs in large scale-free graphs. To balance the processing workload, we distribute hub vertex data structures and related computation among a set of delegates. An illustration of a graph before and after partitioning the hub is shown in Figure 1. Each partition containing a portion of the hub is assigned a local representative of the hub. One representative is distinguished as the *controller*, and the others are the *delegates*. The controller and its delegates coordinate using asynchronous broadcast and reduction operations rooted at the controller.

Our delegate technique leads to significant communication reduction through the use of asynchronous broadcast and reduction operations. For hubs whose degree is greater than the number of processing cores, p, using delegates reduces the required volume of communication. This reduction occurs because a broadcast, rooted at the controller, requires only $O(p)$ communication, while without delegates the volume of communication is proportional to the hubs' degree.

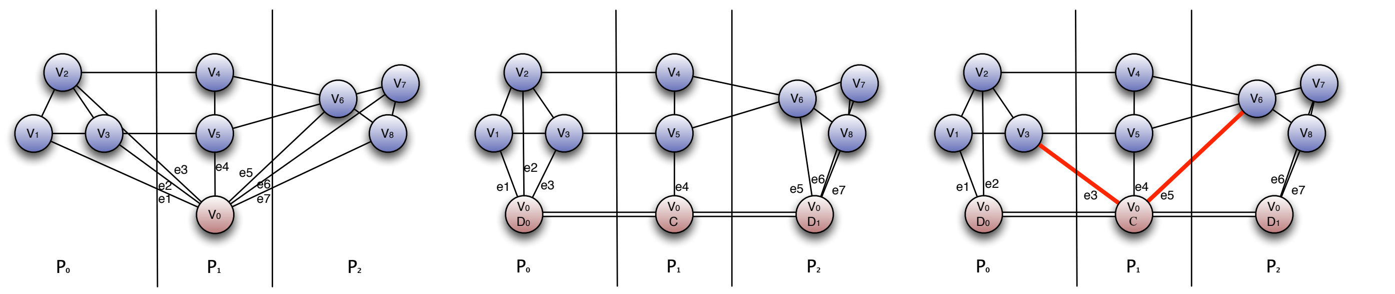
We develop the delegate partitioning and computation model by extending our asynchronous visitor model [1], [2]. Using the visitor computation model, the controller may broadcast visitors to all its delegates. Similarly, the delegates may participate in an asynchronous reduction rooted at the controller.

We demonstrate the approach and evaluate performance and scalability using Breadth-First Search (BFS), Single Source Shortest Path (SSSP), K-Core Decomposition, and PageRank on synthetically generated scale-free graphs. The data-intensive community has identified BFS as a key challenge for the field and established it as the first kernel for the Graph500 benchmark [3]. We demonstrate scalability up to 131K cores using the IBM BG/P supercomputer, and show portability on a standard HPC Linux cluster. We compare our work to existing approaches for processing scale-free graphs in distributed memory, most notably 2D graph partitioning [4], [5] by comparing our algorithm to the best known Graph500 performance on IBM BG/P Intrepid supercomputer at Argonne [6].

Summary of our contributions:

- We present a new algorithmic technique, called vertex delegates, to load balance the computation, communication, and storage associated with high-degree vertices. It uses asynchronous broadcast and reduction operations to significantly reduce communication associated with high-degree vertices.
- We demonstrate our delegate techniques using Breadth-First Search (BFS), Single Source Shortest Path (SSSP), K-Core Decomposition, and Page-Rank.
- We demonstrate excellent scalability up to 131K cores on BG/P Intrepid, and portability on a standard HPC Linux cluster. Our algorithm improves the best known Graph500 results for BG/P Intrepid, a custom BG/P implementation, by 15%.

SC14, November 16-21, 2014, New Orleans



(a) 1D Partitioning

(b) Distributed Delegates Partitioning. P_1 contains the *controller*.

(c) Distributed Delegates Partitioning after balancing.

Fig. 1: Comparison of 1D partitioning vs. distributed delegates partitioning for the same graph. In 1D partitioning (a), V_0 is a high-degree vertex that maps to a single partition and may lead to imbalances. In distributed delegates partitioning (b), V_0 is distributed across multiple partitions while low-degree vertices remain 1D partitioned. V_0C is the *controller* assigned to P_1. Delegates V_0D_0 and V_0D_1 are assigned to P_0 and P_1. After balancing (c), edges e_3 and e_5 have been relocated to balance the partitions, and are now delegate cut edges.

II. Preliminaries

A. Graphs, Properties & Algorithms

In a graph, relationships are represented using vertices and edges. A vertex may denote an object or concept, and the relationships between vertices are expressed by edges. Graphs provide a framework to analyze complex relationships among data, and are used in a wide variety of fields.

Many real-world graphs can be classified as *scale-free*, where the distribution of vertex degrees follows a scale-free power-law [7]. In a scale-gree graph, the majority of vertices have small degree, while a select few have a very large degree, following the power-law distribution. These high-degree vertices are called *hubs*, and create multiple scaling issues for parallel algorithms, discussed further in Section II-C.

We demonstrate our techniques using Breadth-First Search, Single Source Shortest Path, PageRank, and K-Core Decomposition.

1) Breadth-First Search (BFS): BFS is a simple traversal that begins from a starting vertex and explores all neighboring vertices in a level-by-level manner. Taking the starting vertex as belonging to level 0, level 1 is filled with all unvisited neighbors of level 0. Level $i+1$ is filled with all previously unvisited neighbors of level i; this continues until all neighbors of level i have been visited. The Graph500 benchmark, established in 2010, selected BFS as the initial benchmark kernel using synthetic scale-free graphs.

2) Single Source Shortest Path (SSSP): A SSSP algorithm computes the shortest paths in a weighted graph from a single source vertex to every other vertex. In this work, we only address non-negatively weighted graphs. Our approach to the SSSP problem can be viewed as a hybrid between Bellman-Ford [8] and Dijkstra's [9] SSSP algorithms.

3) PageRank: PageRank is a network analysis tool that ranks vertices by their relative importance [10]. Designed to rank pages on the Web, PageRank models a random web surfer that randomly follows links with random restart. It is often iteratively computed as a stochastic random walk with restart, where the starting distribution is a uniform distribution across all vertices.

4) K-Core Decomposition: The k-core of a graph is the largest subgraph where every vertex is connected to at least k other vertices in the subgraph. The k-core subgraph can be found by recursively removing vertices with less than degree k. K-Core has been used in a variety of fields including the social sciences [11].

B. Synthetic scale-free graph models

We use two synthetic scale-free graph models for our scaling studies. All graphs are undirected, and generated with a number of vertices and edges that is a power of two. The graphs are sparse, with an average degree fixed at 16. After graph generation, all vertex labels are uniformly permuted to destroy any locality artifacts from the generators. This methodology was chosen to conform closely with the Graph500 benchmark.

1) Graph500 RMAT: Generates scale-free graphs based on a recursive matrix model [12]; we follow the Graph500 V1.2 specification for generator parameters. We used the open source RMAT implementation provided by the Boost Graph Library [13].

2) Preferential Attachment (PA): Generates scale-free graphs based on the Barabási-Albert model [7]. We used a generalized PA model by Móri [14], where the probability of connecting to a vertex of degree d is proportional to $d + \beta$, where $\beta > -16$. By varying the value of β, we can control the rate in which hubs grow. For our studies, we chose β values of *-12, -13,* and *-14*. The β value of *-12* was used to roughly match Graph500 RMAT's hub growth, and the β values of *-13* and *-14* were chosen to increase hub growth and stress the delegate approach. We parallelize the generation of large PA graphs using similar techniques developed by Machta [15].

The growth of the largest hub vertex for the graph models in our study is shown in Figure 2(a).

C. Challenges created by high-degree vertices

An imbalanced partitioning of edges leads to communication and work imbalance between the partitions, inhibiting overall performance and scalability. In scale-free graphs, the

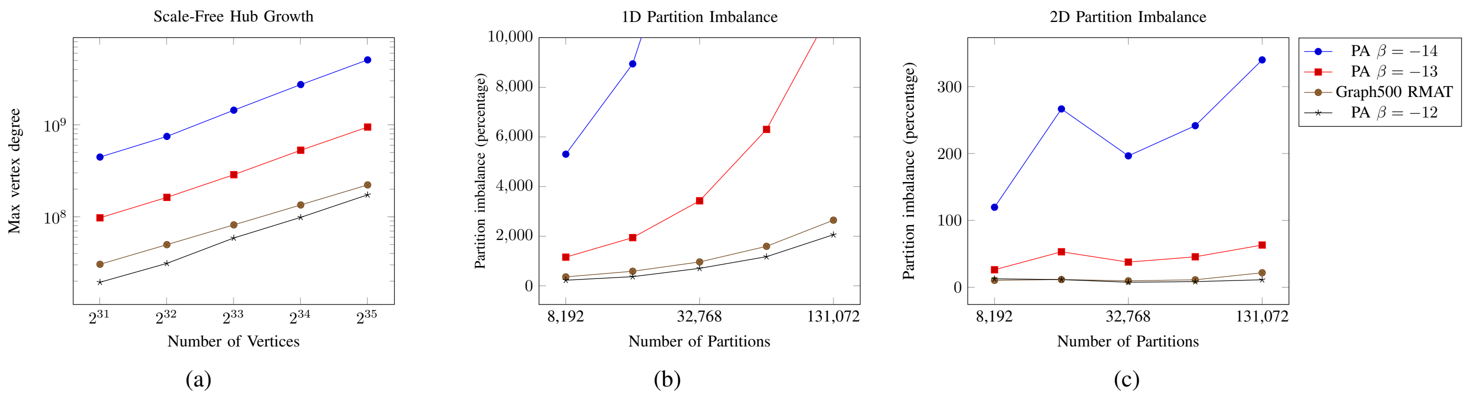


Fig. 2: Hub growth (a) for scale-free RMAT and preferential attachment graphs. Weak scaling of partition imbalance for 1D (b) and 2D (c) partitioning; imbalance computed for the distribution of edges per partition. Weak scaled using 262,144 vertices per partition. The number of vertices per partition matches the experiments on BG/P Intrepid shown in Sections VI-B and VI-E. Both 1D and 2D partitioning produce imbalanced partitioning, with the increased imbalance when the graph has greater hub size (e.g. PA $\beta = -14$). 2D partitioning is significantly better than 1D for all graphs in our studies; however, our distributed delegates partitioning produces perfectly balanced partitions for these weak scaled graphs.

core challenge involves creating a balanced partitioning of edges in the presence of power-law vertex degree distributions.

The simplest partitioning is 1D, where the entire adjacency list of a vertex resides on a single partition; this is commonly called *row wise* when the graph is considered as a sparse matrix. When scale-free graphs are 1D partitioned, the partitions having high-degree vertices may contain significantly more edges than the average. This can overwhelm a single partition, limiting performance or overflowing memory.

In 2D partitioning, the graph is partitioned according to a checkerboard pattern of the graph's adjacency matrix so that the adjacency lists of all vertices are split over $O(\sqrt(p))$ partitions, which greatly improves the partition balance. Recent work using 2D graph partitioning has shown the best results for large scale HPC systems [4], [5], [16]. However, 2D partitioning has serious disadvantages at scale. When processing sparse graphs, each 2D block may become hypersparse, i.e., fewer edges than vertices per partition [17], which will ultimately hit a scaling wall where the amount of local algorithm state per partition exceeds the capacity of the compute node. Recently, Boman, et al. have combined 1D graph partitioning (e.g., ParMETIS) with a 2D layout to achieve better partition balance than traditional 2D block partitioning [18]. The amount of algorithm state (e.g., BFS level) stored per partition scales with $O(\frac{V}{\sqrt{p}})$, where V is the total number of vertices and p is the number of partitions. We compare our approach to the best known implementation for BG/P Intrepid, which uses a 2D graph partitioning [6].

A comparison of partition imbalances for 1D and 2D partitioning is shown in Figure 2, where imbalance is calculated as the percent difference in the number of edges of the largest partition versus the average. The comparison shows the scale-free graphs used in our studies weak-scaled with the number of partitions in the x-axis. The weak-scaling matches the scaling studies shown in Sections VI-B and VI-E. Both 1D and 2D partitioning produce imbalanced partitioning, with the increased imbalance when the graph has greater hub size (e.g. PA $\beta = -14$). 2D partitioning is significantly better than 1D for all graphs in our studies. However, 2D partitioning is still 11% imbalanced for the Graph500 RMAT graph at 131k partitions and 2^{35} vertices. Our distributed delegates partitioning produces well balanced partitions for these weak scaled graphs.

D. Asynchronous Visitor Traversals

Our framework uses an *asynchronous visitor queue* abstraction [2]. The visitor queue provides the parallelism and creates a data-driven flow of computation. Parallel algorithms using this framework have been developed for breadth-first search, single source shortest paths, connected components, k-core, page-rank, and triangle counting [1], [2]. Traversal algorithm developers define vertex-centric procedures to execute on traversed vertices. Vertex visitors have the ability to pass visitor state to other vertices. The visitor pattern is discussed in Section IV-A.

Each asynchronous traversal begins with an initial set of visitors, which may create additional visitors dynamically depending on the algorithm and graph topology. All visitors are asynchronously transmitted, scheduled, and executed. When the visitors execute on a vertex, they are guaranteed exclusive access to the vertex's data. The traversal completes when all visitors have completed, and the distributed queue is globally empty.

III. Distributed Delegates

To balance the storage, computation, and communication of high-degree hubs, we distribute hub data structures and related computation amongst many partitions. Each partition containing a portion of the hub data structure is assigned a local representative; one representative is distinguished as the *controller*, and the others are designated as *delegates*. An illustration of a delegate partitioned graph is shown in Figure 1(b). The controller and its delegates communicate via asynchronous broadcast and reduction operations rooted at the controller.

The delegates maintain a copy of the state for the vertex and a portion of the adjacency list of the vertex. Because a delegate only contains a subset of a vertex's edges, the operations performed may need to be coordinated across multiple delegates.

A. Delegate Partitioning in Visitor Framework

We have integrated delegate partitioning into our asynchronous visitor framework. Vertices with degree greater than d_{high} are distributed and assigned delegates, while vertices with low-degree are left in a basic 1D partitioning. When a visitor visits a delegate, it only operates on the subset of adjacent edges managed by the delegate; it does not operate on the entire distributed adjacency list.

Ideally, the outgoing edges of high-degree vertices are stored at the edges' target vertex location. Such edges are called *co-located* because their delegate and target vertex reside on the same partition. Co-located edges do not require additional communication beyond the delegates' broadcast and reduction communication, so having multiple co-located edges per individual delegate leads to an overall reduction in communication.

This technique alone is not sufficient to produce balanced partitions. In some cases, including our experiments, simply storing edges of high-degree vertices at the edges' target vertex location can lead to imbalance amongst the delegates. To balance partitions, the delegated edges belonging to high-degree vertices can be moved to *any* partition at the cost of additional communication for the non-co-located edges.

B. Distributed Delegate Partitioning

In this section, we describe a simple technique to partition an input graph using distributed delegates. A distributed input graph, $G(V, E)$ with vertex set V and edge set E, is partitioned into p partitions in three steps. First, the high degree vertices in G are identified. Second, the edges in E belonging to low-degree vertices are 1D partitioned such that all of a low-degree vertex's edges reside on a single partition. The edges in E belonging to high-degree hubs are partitioned according to the partition of the edge target vertex. Finally, the partitions are balanced by offloading delegate edges from partitions with an above average number of edges.

The input distributed edge set, E, is assumed to be unordered, and is distributed over the p partitions. Undirected edges are represented by creating directed forward and backward edges that may reside on different partitions.

First, the high-degree vertices with degree larger than a threshold, d_{high}, are identified. Then the distributed edge set, E, is partitioned into two distributed edge sets: E_{high} for edges whose source vertex is high-degree, and E_{low} for edges whose source vertex's degree is less than d_{high}. Delegates are created on all partitions for high-degree vertices. The degree of every vertex must be accumulated to identify the high-degree vertices, which may require an all-to-all exchange amongst the partitions.

The second step uses a simple vertex-to-partition mapping (e.g., round-robin) to define a 1D partitioning. The edges in E_{low} are distributed according to the partition mapping of the source vertex of each edge. The edges in E_{high} are distributed according to the target vertex partition mapping of each edge. In the worst case, every edge will need to be relocated to a new partition which may require an all-to-all exchange amongst the partitions.

The third step corrects partition imbalances. The number of edges locally assigned to each partition (both E_{high} and E_{low}) can be imbalanced. An edge in E_{high} may be reassigned to any partition, because the edge's source is a delegated vertex. A new distributed edge set $E_{overflow}$ is created and filled with edges of E_{high} from partitions with greater than $\frac{|E|}{p}$ edges. The edges in $E_{overflow}$ are distributed such that the local partitions' sum of edges $|E_{low}| + |E_{high}| + |E_{overflow}| = \frac{|E|}{p}$. For performance reasons, minimizing the size of $E_{overflow}$ is desirable, because the edges are located on different partitions than their targets. An illustration of a delegate partitioned graph after edge balance is shown in Figure 1. Here, edges e_3 and e_5 have been relocated to partition p_1 to balance the partitions, and edges e_3 and e_5 are now delegate cut edges. In the worst case, each partition will either send or receive overflow edges and may require an all-to-all exchange amongst the partitions.

The complete partitioning can be accomplished in three parallel operations over the edges, $O(\frac{|E|}{p})$. In the worst case, each step may require all-to-all communication, $O(p^2)$. This partitioning cost is asymptotically the same as partitioning an unorganized edge set using 1D or 2D partitioning.

C. Benefits of Delegate Partitioning

Delegate partitioning leads to significant communication reduction by co-locating delegate edges. The reduction in communication occurs because a broadcast or reduction, rooted at the controller, requires only $O(p)$ communication, while without delegates the volume of communication is proportional to the degree of the hub. Experimentally, we show the percentage of co-located edges in Section VI-A, Figure 4(c).

Delegate partitioning also creates a balanced number of edges in each partition. There always exists a degree threshold, d_{high}, that will produce a balanced partitioning. Consider $d_{high} = 0$, every vertex with edges is delegated, and all edges are eligible to be reassigned to any partition to ensure a balanced partitioning. Setting $d_{high} < p$ could result in increased communication, so it is advisable to place a lower bound at p. Experimentally, we show the partition imbalance as d_{high} is swept in Section VI-A, Figure 4(c). In our weak-scaling experiments shown in section VI, we set $d_{high} = p$ and delegate partitioning produced evenly balanced partitions.

D. Comparison of partitioning techniques

Delegate partitioning is similar to *vertex splits* in PowerGraph [19]; however, PowerGraph's approach to distributing high-degree vertices is different in an important way. PowerGraph attempts to distribute a high-degree vertex over a minimal number of workers, in contrast to our approach which distributes across all edge target partitions O(p). By storing edges directly on an edge's vertex target location, our approach creates co-located edges that when combined with efficient broadcasts and reductions can reduce overall communication. PowerGraph has been designed for the cloud computing environment (e.g., Amazon EC2), and has not been tuned for the HPC environment.

TABLE I: Delegate Visitor Behaviors

Behavior	Description	Complexity	Examples
pre_visit_parent	Visitor is sent to parent delegate and executes *pre_visit*. If *pre_visit* returns *true*, visitor continues to visit parents until the *controller* is reached.	$O(h_{tree})$	BFS, SSSP
lazy_merge_parent	Lazily merges visitors using an asynchronous reduction tree. Merges visitors locally, and sends to parent in reduction tree when local visitor queue is idle. When *controller* is reached, normal visitation proceeds. Requires that visitors provide a *merge* function.	$O(h_{tree})$	k-core
post_merge	Visitors are merged into parent reduction tree after traversal completes. Requires that visitors provide a *merge* function.	$O(h_{tree})$	PageRank

TABLE II: Controller Visitor Commands

Behavior	Description	Complexity
bcast_delegates	Controller broadcasts the current visitor to all delegates.	$O(h_{tree})$
terminate_visit	Controller terminates the current visitor without sending to delegates.	$\Theta(1)$

TABLE III: Visitor Procedures and State

Required	Description
pre_visit( )	Performs a preliminary evaluation of the state and returns $true$ if the visitation should proceed, this can be applied to *delegate* vertices.
visit( )	Main visitor procedures.
operator<( )	Less than comparison used to locally prioritize the visitors in a *min heap* priority queue.
vertex	Stored state representing the vertex to be visited.
delegate_behavior	Desired delegate visitation behavior, see Table I.
merge(visitor_a, visitor_b)	Returns the merge of two visitors. Used for *lazy_merge_parent* and *post_merge* behaviors.

Our previous work developed an *edge list partitioning* (ELP) technique that creates balanced partitions for scale-free graphs [2]. Ghost verticies were used to reduce communication; however, they could only be applied to a small set of algorithms such as BFS, not PageRank or K-Core. ELP did not parallelize the processing and communication of hubs, or attempt to create co-located edges. The reductions in communication achieved by delegate partitioning exceeds ELP and supports a richer set of graph algorithms.

1D and 2D are previously discussed in Section II-C. 2D partitioning is the state of the art for leadership class supercomputers. Our comparison to the best known Graph500 results for BG/P Intrepid in Section VI-E used 2D partitioning [6].

IV. Asynchronous Visitor Queue

The driver of our graph traversal is the *distributed asynchronous visitor queue* [2]; it provides the parallelism and creates a data-driven flow of computation. Traversal algorithms are created using a visitor abstraction, which allows an algorithm designer to define vertex-centric procedures to execute on traversed vertices with the ability to pass visitor state to other vertices.

A. Visitor Abstraction

In our previous work, we used an asynchronous visitor pattern to compute Breadth-First Search, Single Source Shortest Path, Connected Components, k-core, and triangle counting in shared, distributed and external memory. We used *edge-list partitioning* and *ghosts* to address the scaling challenges created by high-degree vertices [2]. We showed these techniques to be useful; however, the application of *ghosts* was limited to simple traversals such as BFS.

In this work, we build on the asynchronous visitor pattern and introduce new techniques designed to handle distributed delegates. The coordination of the controller and its delegates must be considered when designing a visitor for an algorithm. The algorithm developer must specify a *delegate behavior* for each visitor, and *controller commands* must be specified at the return of the visitor's procedure. A list of delegate behaviors is described in Table I, and a list of controller commands is described in Table II. There are three types of reduction operations, *pre_visit_parent*, *lazy_merge_parent*, and *post_merge*, that allow algorithms to distribute computation amongst the delegates. For the controller, there is a broadcast operation, *bcast_delegates*, that broadcasts a visitor to all the delegates of the controller. The visitor procedures required by our asynchronous visitor queue framework are summarized in Table III.

B. Visitor Queue Interface

The *visitor queue* has the following functionality that may be used by a visitor or initiating algorithm:

- *push(visitor)* – pushes a new visitor into the distributed queue.
- *do_traversal()* – initializes and runs the asynchronous traversal to completion. This is used by the initiating algorithm.

When an algorithm needs to dynamically create new visitors, they are *pushed* onto the visitor queue using the *push()* procedure. When an algorithm begins, an initial set of visitors are pushed onto the queue, then the *do_traversal()* procedure is invoked which runs the asynchronous traversal to completion.

To support efficient broadcast and reduction operations, the distributed delegates for a vertex are arranged in a tree structure (a *delegate tree*) with the root of the tree defined as the *controller*. The height of the delegate tree is denoted by h_{tree}, and the value of h_{tree} for our experiments is discussed in Section IV-D.

C. Controller and Delegate Coordination

Operations on the controller and its delegates are coordinated through asynchronous broadcast and reduction operations. The return value of the *visit* procedure notifies the framework which controller action it is required to perform. A controller can broadcast commands to all delegates of a vertex by returning *bcast_delegates* from the *visit* procedure. The controller may choose to not broadcast a visitor by returning *terminate_visit* from the *visit* procedure.

Delegates can lazily participate in reductions by using the *lazy_merge_parent* behavior. This instructs the visitor framework to locally merge visitors, and send a merged visitor to the parent in the reduction tree when local visitor queue is idle. We show K-Core decomposition as an example algorithm using this behavior. To fully reach the controller, requires $O(h_{tree})$ visits.

Asynchronous filtering can be performed using the *pre_visit_parent* behavior. This tells the framework to immediately send the visitor to the delegate's parent where the *pre_visit* procedure will be executed. If the *pre_visit* returns *true* the visit will proceed up the delegate tree. We show Breadth-First Search as an example algorithm using this behavior.

Post-traversal reductions are performed when the visitor's behavior is set to *post_merge*. This tells the framework to merge the visitors into the parent reduction tree after the traversal completes. PageRank is an algorithm using this behavior.

D. Routed point-to-point communication

In our previous work, we applied communication routing and aggregation through a synthetic network to reduce dense communication requirements [2]. For dense communication patterns, where every processor needs to send messages to all p other processors, we route the messages through a topology that partitions the communication. Figure 3 illustrates a 2D routing topology that reduces the number of communicating channels a processor requires to $O(\sqrt{p})$. This reduction in the number of communicating pairs comes at the expense of message latency because messages require two hops to reach their destination. In addition to reducing the number of communicating pairs, 2D routing increases the amount of message aggregation possible by $O(\sqrt{p})$.

In this work, we embed the *delegate tree* into the synthetic routed communication topology, as illustrated in Figure 3. In this example, delegates residing on *Rank 11* are assigned *delegate parents* on *Rank 9* when the controller is on *Rank 5*.

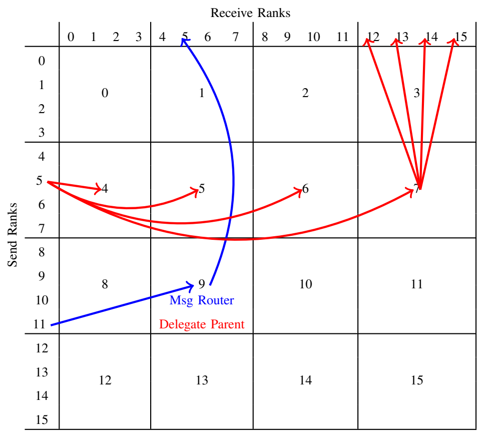


Fig. 3: Illustration of 2D communicator routing of 16 ranks. As an example, when *Rank 11* sends to *Rank 5*, the message is first aggregated and routed through *Rank 9*. Delegate tree operations are also embedded onto this topology. In this example, delegates residing on *Rank 11* are assigned *delegate parents* on *Rank 9* when the controller is on *Rank 5*. A *pre_visit_parent* originating on *Rank 11* is sent to the parent on *Rank 9* before being sent to the controller on *Rank 5*. An illustration of a broadcast tree is also shown for *Rank 5*. When *Rank 5* broadcasts, it first sends to the first level {*4,5,6,7*}. The second level of the broadcast is illustrated for *Rank 7*, which sends to {*12,13,14,15*}.

A *pre_visit_parent* originating on *Rank 11* is sent to the parent on *Rank 9* before being sent to the controller on *Rank 5*. An illustration of a broadcast tree is also shown for *Rank 5*. When *Rank 5* broadcasts, it first sends to the first level {*4,5,6,7*}. The second level of the broadcast is illustrated for *Rank 7*, which sends to {*12,13,14,15*}. The value of h_{tree} is 2 when using 2D partitioning; with 3D it is 3.

Scaling to hundreds of thousands of cores requires additional reductions in communication channels. Our experiments on IBM BG/P use a 3D routing topology that is very similar to the 2D illustrated in Figure 3, and on the BG/P, our routing is designed to mirror the 3D torus interconnect topology.

E. Asymptotic Effects on Hub Workload

When high degree vertices are delegated, their storage, computation, and communication are parallelized and load balanced. The algorithmic effects are:

- High-degree storage reduces from $O(d_{max})$ to $O(\frac{d_{max}}{p})$. The storage of high-degree vertices is now evenly stored across the partitions. This enables all partitions to participate in the computation and communication of high-degree vertices.
- High-degree computation reduces from $O(d_{max})$ to $O(\frac{d_{max}}{p})$. The computation for high-degree vertices is now evenly distributed across the partitions.
- High-degree communication performed through the delegate tree reduces from $O(d_{max})$ to $O(p)$ communication and $O(h_{tree})$ steps. The communication of high-degree vertices is performed using tree based broadcasts and reductions.

The effect on performance of these optimizations is shown in Section VI-A.

Algorithm 1 BFS & SSSP Visitor

```
1: visitor state: vertex ← vertex to be visited
2: visitor state: length ← path length
3: visitor state: parent ← path parent

4: delegate behavior: pre_visit_parent

5: procedure PRE_VISIT($vertex\_data$)
6:     if $length < vertex\_data.length$ then
7:         $vertex\_data.length \leftarrow length$
8:         $vertex\_data.parent \leftarrow parent$
9:         return $true$
10:    end if
11:    return $false$
12: end procedure

13: procedure VISIT($graph$, $visitor\_queue$)
14:    if $length == graph[vertex].length$ then
15:        for all $vi \in out\_edges(g, vertex)$ do
                                        ▷ Creates and queues new visitors
16:            $new\_len \leftarrow length + edge\_weight(g, vertex, vi)$
                                        ▷ $edge\_weight$ equals 1 for BFS
17:            $new\_vis \leftarrow bfs\_visitor(vi, new\_len, vertex)$
18:            $visitor\_queue.push(new\_vis)$
19:        end for
20:        return bcast_delegates
21:    else
22:        return terminate_visit
23:    end if
24: end procedure

25: procedure OPERATOR < ()($visitor\_a$, $visitor\_b$)
                                        ▷ Less than comparison, sorts by length
26:    return $visitor\_a.length < visitor\_b.length$
27: end procedure
```

Algorithm 2 Page-Rank Visitor

```
1: visitor state: vertex ← vertex to be visited
2: visitor state: rank ← partial Page-Rank value

3: delegate behavior: post_merge

4: procedure PRE_VISIT($vertex\_data$)
5:     $vertex\_data.sum$ += $rank$
6:     return $first\_visit()$
7: end procedure

8: procedure VISIT($graph$, $visitor\_queue$)
9:     for all $vi \in out\_edges(g, vertex)$ do
                                        ▷ Creates and queues new visitors
10:        $edge\_rank \leftarrow rank\ /\ out\_degree(g, vertex)$
11:        $new\_vis \leftarrow pr\_visitor(vi, edge\_rank)$
12:        $visitor\_queue.push(new\_vis)$
13:    end for
14:    return bcast_delegates
15: end procedure

16: procedure MERGE($visitor\_a$, $visitor\_b$)
17:    $visitor\_a.rank$ += $visitor\_b.rank$
18:    return $visitor\_a$
19: end procedure
                                        ▷ No visitor ordering required
```

V. Visitor Algorithms

A. Breadth-First Search & Single Source Shortest Path

The visitor used to compute the BFS level or SSSP for each vertex is shown in Algorithm 1. Before the traversal begins, each vertex initializes its $length$ to ∞; then a visitor is queued for the source vertex with $length = 0$.

When a visitor *pre_visits* a vertex, it checks if the visitor's path length is smaller than the vertex's current length (Alg. 1, line 14). If smaller, the *pre_visit* updates the level information and returns *true*, signaling that the main visit function may proceed. Then, the main *visit* function will send new visitors for each outgoing edge (Alg. 1, line 18). The *less than comparison* procedure orders the visitors in the queue by *length* (Alg. 1, line 26).

The *delegate behavior* is configured to *pre_visit_parent* (Alg. 1, line 4), which means that visitors of delegated vertices traverse up the *delegate tree* before reaching the *controller*. Forcing visitors to traverse up the delegate tree provides the opportunity to filter out visitors that are not part of the shortest path.

When a visitor successfully updates the controller's state, the controller broadcasts the visitor to all of its delegates (Alg. 1, line 20). If the visitor does not update the controller's state, then the visitor is terminated (Alg. 1, line 22).

B. PageRank

The visitor used to asynchronously compute the PageRank for each vertex is shown in Algorithm 2. For our experiments, we are concerned with the performance of a single PageRank iteration. Many iterations may be required for convergence, depending on the topology of the graph. Before the asynchronous PageRank begins, a temporary *sum* is initialized to 0 for all vertices, and a visitor containing the initial PageRank value is queued for every vertex.

When a visitor *pre_visits* a vertex, it simply increments the PageRank sum for the vertex (Alg. 2, line 4), and returns *true* if the vertex has not previously been visited. The visitor framework tracks if vertices have been previously visited, and visitors can query this by calling *first_visit()*. The *delegate behavior* is set to *post_merge* which requires a visitor merge function, that also simply returns a sum (Alg. 2, line 16). When every vertex is initially visited with the initial PageRank value, new visitors are queued for every outgoing edge (Alg. 2, line 12). When a controller is visited, it broadcasts the visitor to all its delegates (Alg. 2, line 14).

When the traversal completes, and the delegates have merged their visitors, the final PageRank value has been calculated for every vertex.

C. K-Core Decomposition

To compute the k-core decomposition of an undirected graph, we asynchronously remove vertices from the core whose degree is less than *k*. As vertices are removed, they may create a dynamic cascade of recursive removals as the core is decomposed.

The visitor used to compute the k-core decomposition of an undirected graph is shown in Algorithm 3. Before the traversal begins, each vertex initializes its *k-core* to *degree(v) + 1* and *alive* to *true*, then a visitor is queued for each vertex with *ntrim* set to 1.

Algorithm 3 K-Core Visitor

```
1: visitor state: vertex ← vertex to be visited
2: visitor state: ntrim ← count of edges trimmed
3: static parameter: k ← k-core requested

4: delegate behavior: lazy_parent_merge

5: procedure PRE_VISIT($vertex\_data$)
6:     if $vertex\_data.alive == true$ then
7:         $vertex\_data.kcore \leftarrow vertex\_data.kcore - ntrim$
8:         if $vertex\_data.kcore < k$ then
9:             $vertex\_data.alive \leftarrow false$
10:            return true
11:        end if
12:    end if
13:    return $false$
14: end procedure

15: procedure VISIT($graph$, $visitor\_queue$)
16:     for all $vi \in out\_edges(g, vertex)$ do
17:         $new\_visitor \leftarrow kcore\_visitor(vi, 1)$
18:         $visitor\_queue.push(new\_visitor)$
19:     end for
20:     return bcast_delegates
21: end procedure

22: procedure MERGE($visitor\_a$, $visitor\_b$)
23:     $visitor\_a.ntrim$ += $visitor\_b.ntrim$
24:     return $visitor\_a$
25: end procedure
                                          ▷ No visitor order required
```

The visitor's *pre_visit* procedure decrements the vertex's *k-core* number by *ntrim*, and checks if it is less than *k* (Alg. 3, line 8). If less, it sets *alive* to false and returns *true*, signaling that the visitors's main *visit* procedure should be executed (Alg. 3, line 10). The *visit* function notifies all neighbors of *vertex* that it has been removed from the k-core (Alg. 3, line 18). After the traversal completes, all vertices whose *alive* equals *true* are a member of the k-core.

The *delegate behavior* is configured to *lazy_merge_parent* (Alg. 3, line 4), which means that visitors of delegated vertices are lazy merged up the *delegate tree* before reaching the *controller*. Visitors are merged using the procedure shown in Alg. 3, line 23. Merging visitors before visiting the controller reduces the number of times the controller is required to execute the *pre_visit* procedure.

VI. Experiments

In this section we experimentally evaluate the performance and scalability of our approach. We use the IBM BG/P Intrepid supercomputer at Argonne National Laboratory [20] up to 131K processors to show scalability to large core count. We also use Cab [21] at Lawrence Livermore National Laboratory, which is a standard HPC Linux cluster with an Infiniband interconnect. We begin by exploring the effects of varying the *delegate degree threshold*. Next, we show a weak scaling study for Breadth-First Search, Single Source Shortest Path, K-Core Decomposition and PageRank, followed by comparisons to our previous *edge list partitioning* [2] and 1D partitioning. Finally, we compare performance to the best known Graph500 performance for Intrepid which uses a 2D partitioning approach [6].

For this experimental study, the only optimization specific to IBM BG/P is matching the routed communication topology to the 3D torus as discussed in Section IV-D. We use the Graph500 performance metric of Traversed Edges per Second (TEPS) for both BFS, SSSP and PageRank. Similar to TEPS, we used the rate of trimmed edges per second as the performance metric for K-Core Decomposition.

A. Effects of Delegate Degree Threshold

The delegate degree threshold (d_{high}) is the threshold at which vertices are selected to be delegated. Vertices whose degree is less than d_{high} are 1D partitioned, while those above the threshold are delegate partitioned.

We explore the scaling effects of d_{high} on overall performance, number of co-located edges, and partition imbalance, shown in Figure 4. For a fixed graph size of 2^{30} vertices, using 4096 cores, we demonstrate the performance effects of (a) BFS and (b) PageRank as d_{high} is scaled. The best performing degree threshold for both BFS and PageRank is 4096 (equal to the number of cores). Decreasing d_{high} results in a higher percentage of co-located edges (Fig. 4(c)). However, when the threshold decreases below 4096, the broadcasts to all partitions become wasteful as many delegates will have zero edges on some partitions. At large values of d_{high}, the partitioning reduces to a 1D partitioning with fewer vertices selected to become delegates. In addition to reducing overall performance, the partition imbalance increases when few delegates are created (Fig. 4(d)).

The optimal d_{high} is roughly equal to the number of cores (p), so for the remainder of our delegate experiments we set d_{high} equal to p. This means that d_{high} increases during our weak-scaling studies.

B. Weak Scaling of BFS and PageRank

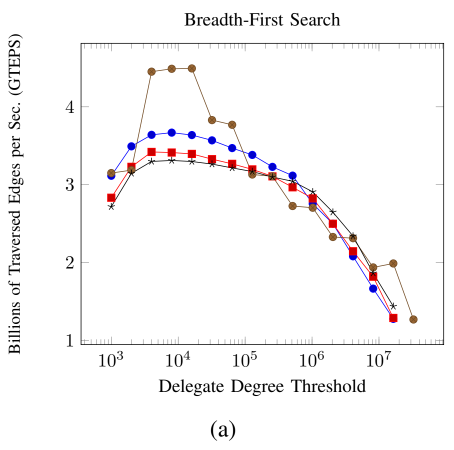
The weak scaled performance using distributed delegates on BG/P Intrepid is shown in Figures 5(a) and 5(b) for BFS and PageRank, respectively. The approach demonstrates excellent weak-scaling up to 131k cores with 2^{35} vertices. There are 2^{18} vertices per core, with the largest scale graph having 2^{35}.

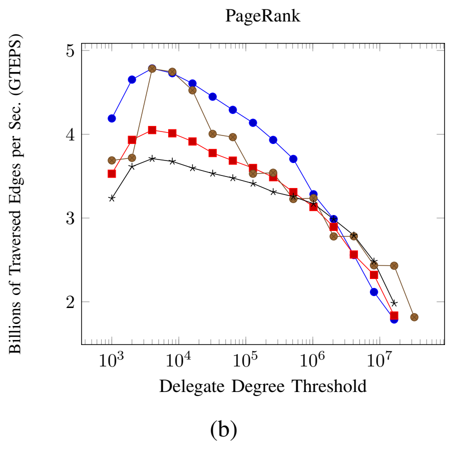
C. Weak Scaling of SSSP and K-Core Decomposition

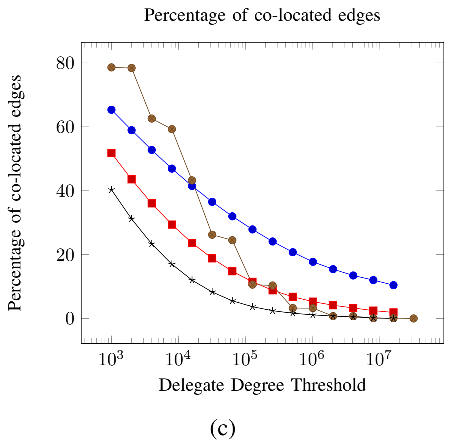
The weak scaled performance using distributed delegates on Cab at LLNL is shown in Figure 6 for SSSP and K-Core decomposition. In addition to good scaling, this demonstrates the portability of our approach to a broader class of HPC resources. For SSSP, edges are randomly weighted with integers ranging $[1, 2^{30})$.

D. Comparison to 1D and edge partitioning

We compare distributed delegate partitioning to our previous work on edge-list partitioning [2] and 1D partitioning in Figure 7. 1D partitioning is widely used by many graph libraries such as PBGL [22], and is used in these experiments as a baseline. For this experiment, the number of vertices per core have been reduced to prevent 1D partitioning from exhausting local partition memory due to imbalance. Also, the experiments are limited to 4096 cores due to increasing hub growth causing additional imbalance. At 4096 cores, our delegate partitioning is 42% faster than edge-list partitioning and 2.3x faster than 1D. PBGL was not able to run with more







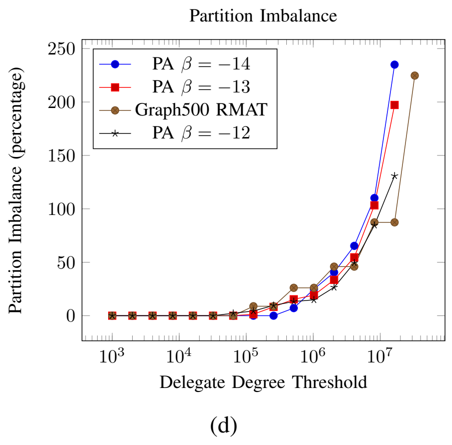
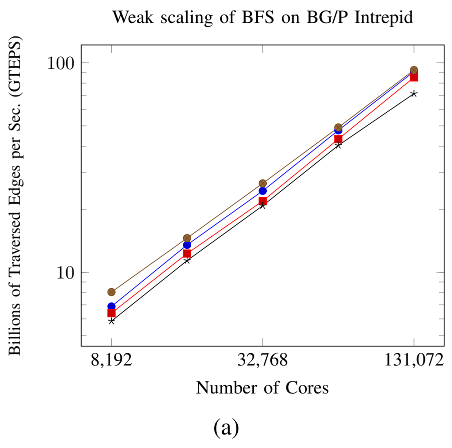


Fig. 4: Effects of delegate degree threshold (d_{high}) using 4096 cores on graphs with 2^{30} vertices. The performance effects of (a) BFS and (b) PageRank, (c) the effects on the percentage of co-located edges, (d) partition imbalance are shown.



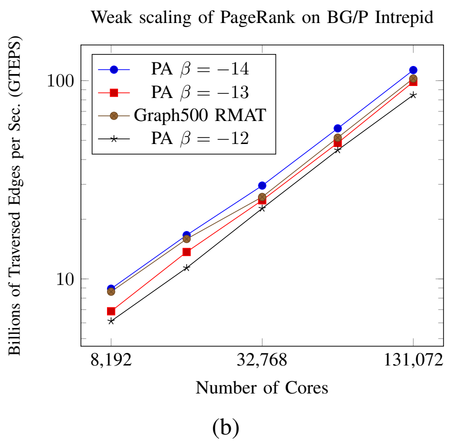
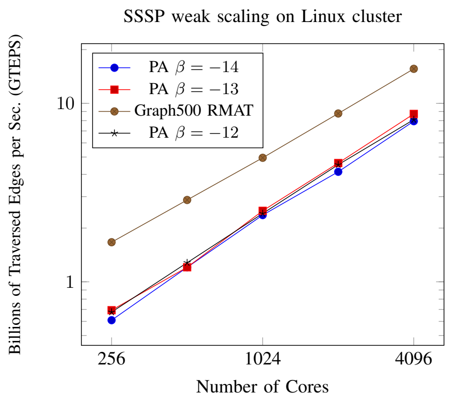
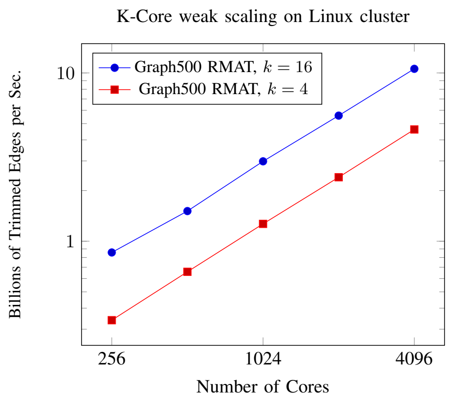


Fig. 5: Weak scaling of (a) BFS and (b) PageRank on BG/P Intrepid. There are 2^{18} vertices per core, with the largest scale graph having 2^{35} vertices.



(a)



(b)

Fig. 6: Weak scaling of delegate partitioned SSSP (a) and K-Core (b) on Cab Linux cluster at LLNL. There are 2^{20} vertices per core, with the largest scale graph having 2^{32} vertices. For SSSP, edges are randomly weighted with integers ranging $[1, 2^{30})$.

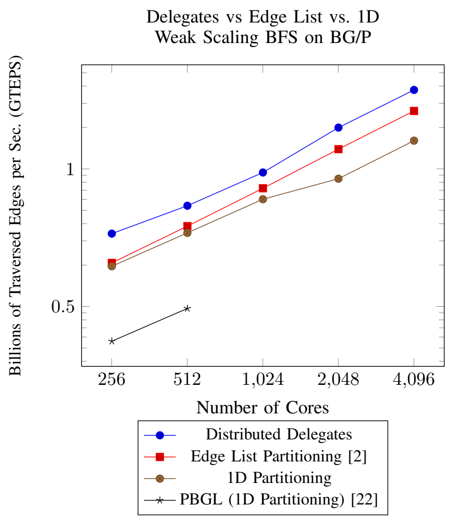


Fig. 7: Comparison of *distributed delegates* vs. edge list partitioning [2], 1D partitioning, and PBGL [22]. Performance of BFS on RMAT graphs shown on BG/P. Important note: the graph sizes are reduced to prevent 1D from running out of memory. There are 2^{17} vertices and 2^{21} undirected edges per core.

than 512 processors without exhausting available memory. At 512 cores, our delegate partitioning is 5.6x faster than PBGL.

E. Comparison to previous Graph500 results

We compare distributed delegates to the best known performance for Intrepid [6] on the Graph500 list in Figure 8. Our approach demonstrates excellent weak scaling, and achieves 93.1 GTEPS on a Scale 35 Graph500 input using 131k cores. The delegates approach outperforms the current best known Graph500 performance for Intrepid by 15%.

VII. Related Work

Willcock, et al. have developed an active message model related to our routed communication [23]. Active messages are routed through a synthetic *hypercube* network to improve dense communication scalability. A key difference from our

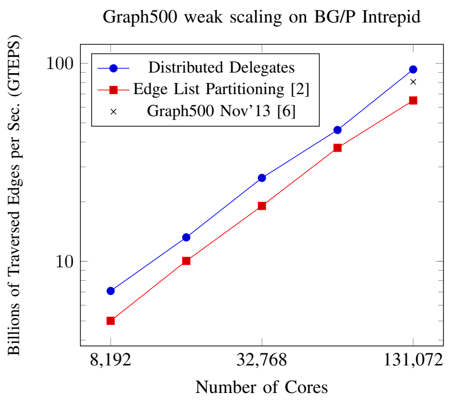


Fig. 8: Weak scaling of delegate partitioned BFS on BG/P Intrepid. Compared to Intrepid BFS performance from the Graph500 list, delegate partitioning is 15% faster than best results published for Intrepid on the Graph500 list. There are 2^{18} vertices per core, with the largest scale graph having 2^{35} vertices.

work is that their approach has been designed for the Bulk Synchronous Parallel (BSP) model.

The Pregel graph library [24] also uses a vertex-centric computation model and is designed to be Bulk Synchronous Parallel (BSP). The STAPL Graph Library [25], [26] provides a framework that abstracts the user from data-distribution and parallelism and supports asynchronous algorithms that are agnostic to the graph partitioning.

Techniques to partition high-degree vertices have also been explored by Kuhlemann and Vassilevski [27], which uses a disaggregation technique to break up hubs for mat-vec operations. The mat-vec operations for the original graph are performed via a factored triple matrix vector product involving an embedding graph. Unlike our work, this technique employs a 1D partitioning.

VIII. Conclusion

In this work, we present a novel technique to parallelize the storage, processing, and communication of high-degree vertices in large scale-free graphs. To balance the processing workload, we distribute hub data structures and related computation among a set of delegates. Computation is coordinated between the delegates and their *controller* through a set of commands and behaviors.

Our delegate technique leads to significant communication reduction through the use of asynchronous broadcast and reduction operations. For hubs whose degree is greater than the number of processing cores, p, using delegates reduces the required volume of communication.

We demonstrate the approach and evaluate performance and scalability using Breadth-First Search (BFS), Single Source Shortest Path (SSSP), K-Core Decomposition, and PageRank on synthetically generated scale-free graphs. We demonstrate scalability up to 131K cores using the IBM BG/P supercomputer, and show portability on a typical HPC linux cluster. Our algorithm improves the best known Graph500 results for BG/P Intrepid, a custom BG/P implementation, by 15%.

IX. Acknowledgments

This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-658291). Funding was partially provided by LDRD 13-ERD-025. Portions of experiments were performed at the Livermore Computing facility. This research used resources of the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. ALCF resources provided through an INCITE 2012 award for the Fault-Oblivious Exascale Computing Environment project. This research supported in part by NSF awards CNS-0551685, CCF-0833199, CCF-0830753, IIS-0917266, by DOE awards DE-AC02-06CH11357, DE-NA0002376 B575363, by Samsung, by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). Pearce was supported in part by a Lawrence Scholar fellowship at LLNL.

References

[1] R. Pearce, M. Gokhale, and N. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Supercomputing*, 2010, pp. 1–11.

[2] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[3] "The Graph500 benchmark," in *www.graph500.org*.

[4] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Supercomputing*, 2011.

[5] A. Yoo, A. Baker, R. Pearce, and V. Henson, "A scalable eigensolver for large scale-free graphs using 2D graph partitioning," in *Supercomputing*, 2011, pp. 1–11.

[6] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in *Supercomputing*, 2012.

[7] A. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[8] T. Cormen, *Introduction to algorithms*. The MIT press, 2001.

[9] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[10] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[11] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269 – 287, 1983.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the Fourth SIAM Int. Conf. on Data Mining*. Society for Industrial Mathematics, 2004, p. 442.

[13] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: user guide and reference manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[14] T. F. Móri, "On random trees," *Studia Scientiarum Mathematicarum Hungarica*, vol. 39, no. 1, pp. 143–155, 2002.

[15] B. Machta and J. Machta, "Parallel dynamics and computational complexity of network growth models," *Phys. Rev. E*, vol. 71, p. 026704, Feb 2005.

[16] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in *Supercomputing*, 2005.

[17] A. Buluç and J. Gilbert, "On the representation and multiplication of hypersparse matrices," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2008, pp. 1–11.

[18] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *Supercomputing*. New York, NY, USA: ACM, 2013, pp. 50:1–50:12.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 17–30.

[20] "IBM BG/P Intrepid," in *https://www.alcf.anl.gov/intrepid*.

[21] "Cab at LLNL," in *https://computing.llnl.gov/resources*.

[22] D. Gregor and A. Lumsdaine, "The parallel BGL: A generic library for distributed graph computations," in *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[23] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine, "Active pebbles: parallel programming for data-driven applications," in *Proceedings of the International Conference on Supercomputing*. ACM, 2011, pp. 235–244.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[25] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "The STAPL Parallel Graph Library," in *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2012.

[26] ——, "KLA: A new algorithmic paradigm for parallel graph computations," in *IEEE Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[27] V. Kuhlemann and P. S. Vassilevski, "Improving the communication pattern in mat-vec operations for large scale-free graphs by disaggregation," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-JRNL-564237, July 2012.