Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory

Roger Pearce*†, Maya Gokhale†, Nancy M. Amato*
*Parasol Laboratory; Dept. of Computer Science and Engineering
Texas A&M University; College Station, TX
†Center for Applied Scientific Computing
Lawrence Livermore National Laboratory; Livermore, CA
{rpearce, maya}@llnl.gov {rpearce, amato}@cse.tamu.edu

***Abstract*—We present techniques to process large *scale-free* graphs in distributed memory. Our aim is to scale to trillions of edges, and our research is targeted at leadership class supercomputers and clusters with local non-volatile memory, e.g., NAND Flash.**

We apply an *edge list partitioning* technique, designed to accommodate high-degree vertices (hubs) that create scaling challenges when processing scale-free graphs. In addition to partitioning hubs, we use *ghost* vertices to represent the hubs to reduce communication hotspots.

We present a scaling study with three important graph algorithms: Breadth-First Search (BFS), K-Core decomposition, and Triangle Counting. We also demonstrate scalability on BG/P Intrepid by comparing to best known Graph500 results [1]. We show results on two clusters with local NVRAM storage that are capable of traversing trillion-edge scale-free graphs. By leveraging node-local NAND Flash, our approach can process thirty-two times larger datasets with only a 39% performance degradation in Traversed Edges Per Second (TEPS).

***Keywords*-parallel algorithms; graph algorithms; big data; distributed computing.**

I. Introduction

Efficiently storing and processing massive graph data sets is a challenging problem in *data intensive computing* as researchers seek to leverage "Big Data" to answer next-generation scientific questions. Graphs are used in a wide range of fields including Computer Science, Biology, Chemistry, and the Social Sciences. These graphs may represent complex relationships between individuals, proteins, chemical compounds, etc.

Graph datasets from many important domains can be classified as scale-free, where the vertex degree-distribution asymptotically follows a power law distribution. The power law distribution of vertex degree causes significant imbalances when processing using distributed memory. The growth of high-degree vertices, also known as *hubs*, provides significant challenges for balancing distributed storage and communication.

In this work, we identify key challenges to storing and traversing massive *scale-free* graphs using *distributed external memory*. We have developed a novel graph partitioning strategy that evenly partitions scale-free graphs for distributed memory, and overcomes the balance issues created by vertices with large out-degree. The growth of high-degree vertices also contributes to communication hotspots for hub vertices with large in-degree. To prevent communication hotspots we selectively use *ghost* vertices to represent hub vertices with high in-degree, and show that only a small number of ghosts are required to provide significant performance improvement.

We tackle three important graph algorithms and demonstrate scalability to large scale-free graphs: Breadth-First Search (BFS), K-Core decomposition, and Triangle Counting. 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 [2]. The Graph500 was designed to be an architecture benchmark used to rank supercomputers by their ability to traverse massive scale-free graph datasets. It is backed by a steering committee of over 50 international HPC experts from academia, industry, and national laboratories. While designed to be an *architecture* benchmark, recent work has shown that it is equally an *algorithmic* challenge [3].

Our aim is to scale to trillions of edges, and our research is targeted at leadership class supercomputers and clusters with local non-volatile memory (NVRAM). These environments are required to accommodate the data storage requirements of large graphs. NVRAM has significantly slower access speeds than main-memory (DRAM), however our previous work demonstrated that it can be a powerful storage media for graph applications [4], [5].

A. Summary of our contributions

- We identify key challenges for traversing scale-free graphs in distributed memory: high-degree hub vertices, and dense all-to-all processor-processor communication;
- We present a novel *edge list partitioning* strategy for large scale-free graphs in distributed memory, and show its application to three important graph algorithms: BFS, K-Core, and triangle counting;
- Using *ghost* vertices to represent high-degree hubs, we scale our approach on leadership class supercomputers by scaling up to 131K cores on BG/P Intrepid and compare it to best known Graph500 results;
- We show that by leveraging node-local NAND Flash, our approach can process thirty-two times larger datasets with only a 39% performance degradation in TEPS over a DRAM-only solution.

II. Background

A. Graphs, Properties & Algorithms

In a graph, relationships are stored using vertices and edges; a vertex may represent an object or concept, and the relationships between them are represented by edges. The power of graph data structures lies in the ability to encode complex relationships between data and provide a framework to analyze the impact of the relationships.

Many real-world graphs can be classified as *scale-free*, where vertex degree follows a scale-free power-law distribution [6]. The degree of a vertex is the count of the number of edges connecting to the vertex. A power-law vertex degree distribution means that the majority of vertices will have a low degree (fewer than 16 for Graph500), while a select few will have a very large degree, with degree following a power-law distribution. These high-degree vertices are called *hub* vertices, and create multiple scaling issues for parallel algorithms, as discussed further in Section III.

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 reachable vertices have been visited.

The Graph500 benchmark, established in 2010, selected BFS as the initial benchmark kernel, and the only one to date, using massive synthetic *scale-free* graphs.

2) 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 [7].

3) Triangle Counting: A triangle is a set of vertices A, B, C such that there are edges between $A - B$, $B - C$, and $A - C$. Triangle counting is a primitive for calculating important metrics such as *clustering coefficient* [8].

B. NVRAM for Data Intensive Computing

In our previous work, we addressed the challenge of achieving DRAM-like performance when some portion of the program state resides in I/O bus-connected NVRAM capable of low latency random access [5]. We identified that high levels of concurrent I/O are required to achieve optimal performance from NVRAM devices (e.g., NAND Flash); this is the underlying motivation for designing highly concurrent asynchronous graph traversals.

We also identified that the Linux I/O system software introduces many bottlenecks. This has led many application developers to use the `O_DIRECT` I/O flag to bypass Linux's default page cache system. For this work, we implemented a custom *page cache* that resides in user space and provides a POSIX I/O interface. Our custom page cache was designed to support a high level of current I/O requests, both for cache hits and misses, and interfaces with NVRAM using direct I/O. The design of our page cache is not the focus of this work, but was required to optimize performance from the NAND Flash devices used in our studies.

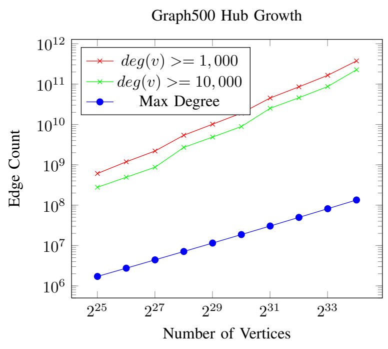

Figure 1. Hub growth for Graph500 graphs. The total count of edges belonging to the $MaxDegree$, and to vertices with $deg(v) >= 1,000$, $deg(v) >= 10,000$.

C. NVRAM in the HPC environment

Node-local or node-near NVRAM is gaining traction in the HPC environment, often motivated by improving the performance of checkpointing in traditional HPC applications [9]. Our work leverages the NVRAM for data-intensive applications. Examples of HPC systems with NVRAM include: Hyperion and Coastal at LLNL; Trestles and Flash Gordon at SDSC; TSUBAME2 at Tokyo Tech.

The architecture and configuration of NVRAM in supercomputing clusters is an active research topic. To our knowledge, our work is the first to integrate node-local NVRAM with distributed memory at extreme scale for important data intensive problems, helping to inform the design of future architectures.

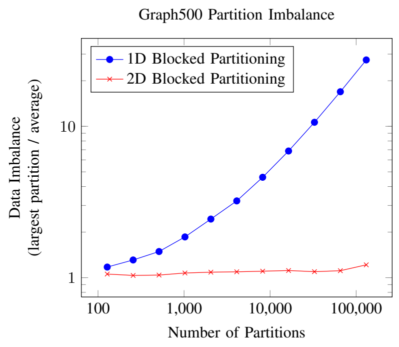

Figure 2. Weak scaling of Graph500 Partition Imbalance for 1D and 2D block partitioning; imbalance computed for the distribution of edges per partition. Weak scaled using 262,144 vertices per partition. The number of vertices per partition matches the experiments on BG/P Intrepid discussed in Section VII-B.

source	0	1	1	2	2	2	2	2	2	3	4	5	5	6	7	7
target	1	0	2	1	3	4	5	6	7	2	2	2	7	2	2	5
partition	p_0				p_1				p_2				p_3			

Figure 3. Example of *edge list partitioning* for a graph with 8 vertices and 16 edges, split into 4 partitions. The edge list is globally sorted by the *sources*, then evenly partitioned. In this example vertices 2 and 5 have adjacency lists that span multiple partitions. $min\_owner(2) = 0$, $max\_owner(2) = 2$, $min\_owner(5) = 2$, $max\_owner(5) = 3$.

III. Scale-free graphs in distributed memory

Parallel processing of unstructured *scale-free* and *small-world* graphs, $G(V, E)$ containing a set V of vertices and a set E of edges, using p processors can lead to significant scaling challenges. These challenges include high-degree hub vertices and dense processor-processor communication, and stem from unstructured locality and power-law distributions of the graph data.

A. High Degree Hubs

Underlying many of the challenges is the growth of *hub* vertices in the graph as the size of the graph increases. Hub vertices have degrees significantly above average. The hub growth for Graph500 scale-free graphs is shown in Figure 1. While the average degree is held constant at 16, the number of edges belonging to hubs of degree greater than 1,000 or 10,000 continue to grow as graph size increases. The *max degree hub* also continues to grow, and by the graph size of 2^{30} vertices, the max degree hub has already crossed 10 Million edges.

Graphs that contain vertices with above-average degree may have large communication imbalances. Specifically, when a graph is 1D partitioned into a set of P partitions, and $\max_{p_i \in P} (\sum_{v \in V_{p_i}} degree(v)) > \frac{|E|}{|P|}$, then at least one processor will process more than its fair share of visitors.

1) Edge List Partitioning: Partitioning the graph data amongst p distributed processes becomes challenging with the presence of *hub* vertices. The simplest partitioning is *1D*, where each partition receives an equal number of vertices and their associated adjacency list. In 1D, the adjacency list of a vertex is assigned to a single partition. This simple partitioning leads to significant data imbalance, shown in Figure 2, because a single *hub's* adjacency list can exceed the average edge count per partition. Data imbalance is computed from the distribution of edges per partition. Recent work has advocated the use of *2D* partitioning [3], [10], [11], where each partition receives a 2D block of the adjacency matrix. In effect, this partitions the *hub's* adjacency list across $O(\sqrt{p})$ partitions, and significantly improves data balance, also shown in Figure 2. 2D partitioning is discussed further in Section VIII-A.

To maintain a balance of edges across p partitions with ranks 0 to $p-1$, we designed a partitioning based on a sorted *edge list*. In this work, the graph's edge list is first sorted by the edges' source vertex, then evenly distributed. This causes many of the adjacency lists (including hubs) to be partitioned across multiple consecutive partitions. Our edge list partitioned graph supports the following partition-related operations:

- $min\_owner(v)$ – returns the minimum partition rank that contains source vertex v;
- $max\_owner(v)$ – returns the maximum partition rank that contains source vertex v.

These operations can be performed in constant time by preserving the rank owner information with the identifier v, or by a $O(lg(p))$ binary search. We choose to store the owner information as part of the identifier v. The underlying storage of each edge list partition is flexible; we choose to store each local partition as a *compressed sparse row*.

An example of edge list partitioning is illustrated in Figure 3. In this example, vertices 2 and 5 have adjacency lists that span multiple partitions: *min_owner(2) = 0, max_owner(2) = 2, min_owner(5) = 2, max_owner(5) = 3.*

Requiring the edge list to be globally sorted is an additional step that is not needed by 1D or 2D graph partitioning. This is not an onerous requirement, because there are numerous distributed memory and external memory sorting algorithms, and in many graph file formats the edge list is already sorted.

Each partition that contains v also contains the algorithm state for v (e.g., BFS level). This means that state is replicated for vertices whose adjacency list spans multiple partitions. The min_owner partition is the *master* partition with all others acting as *replicas*. The algorithms for controlling access to each replica are discussed in section V. The global number of partitioned adjacency lists is bounded by $O(p)$, where each partition contains at most two split adjacency lists.

2) Ghost vertices: The number of incoming edges to hub vertices in scale-free graphs can grow very large, significantly larger than the total number of edges per partition. To mitigate the communication hotspots created by hubs, we selectively use ghost information. Ghosts can be used to filter excess visitors, reducing the communication hotspots created by high in-degree hubs. Ghosts are discussed further in Section IV-B.

B. Dense processor-processor communication

Effectively partitioning many scale-free graphs is difficult, and often not feasible. Without good graph separators, parallel algorithms will require significant communication. Specifically, when the parallel partitioned graph contains $\Omega(|E|^{\alpha})$, where $0 < \alpha \leq 1$, cut edges, a polynomial number of graph edges will require communication between

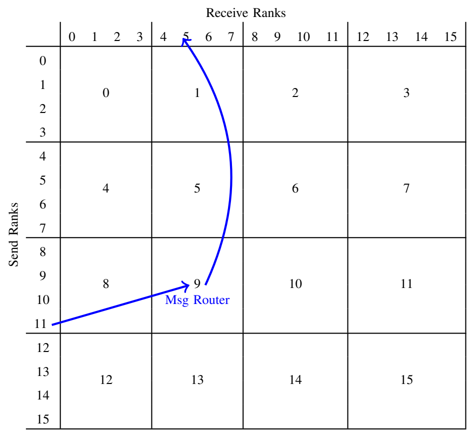

Figure 4. Illustration of 2D communicator routing of 16 ranks. As an example, when *Rank 11* sends to *Rank 5*, the message is first aggregated and routed through *Rank 9*.

processors. Additionally, *dense communication* occurs when $\Omega(p^{\alpha+1})$ pairs of processors share cut edges, in the worst case creating all-to-all communication.

We apply communication routing and aggregation through a synthetic network to reduce dense communication requirements. For dense communication patterns, where every process needs to send messages to all p other processes, we route the messages through a topology that partitions the communication. We have experimented with 2D and 3D routing topologies. Figure 4 illustrates a 2D routing topology that reduces the number of communicating channels a process 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 $O(\sqrt{p})$.

Scaling to hundreds of thousands of cores requires additional reductions in communication channels. Our experiments on BG/P use a 3D routing topology, that is very similar to the 2D illustrated in Figure 4, and is designed to mirror the BG/P 3D torus interconnect topology.

IV. Distributed Asynchronous Visitor Queue

The driver of our graph traversal is the *distributed asynchronous visitor queue*; 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. 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.

A. Visitor Abstraction

In our previous work [4], we used an asynchronous visitor pattern to compute Breadth-First Search, Single Source Shortest Path, and Connected Components in shared and external memory. We used multi-threaded *prioritized* visitor queues to perform the asynchronous traversal.

In this work, we build on the asynchronous visitor pattern with modifications to handle *edge list partitioning* and *ghost* vertices. The required visitor procedures and state are outlined in Table I.

Table I
Visitor Procedures and State

Required	Description
pre_visit( )	Performs a preliminary evaluation of the state and returns *true* if the visit should proceed; this can be applied to *ghost* 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.

B. Ghost Vertices

Ghost information replicates distributed state to avoid communication. By replicating the state of vertices with large in-degree, the communication hotspot associated with that vertex can be reduced. The ideal case is to reduce the communication from hundreds of millions down to $O(p)$, where each partition only requires communicating once per hub vertex.

The ghost information is never globally synchronized, and represents only the local partitions' view of remote hubs. Each partition locally identifies high-degree vertices from its edges' targets to become ghost vertices. Ghosts cannot be used for every algorithm, so each algorithm must explicitly declare ghost usage.

We investigate the useful number of ghosts in our experimental study in Section VII-E2. The number of ghosts required for scale-free graphs is small, because the number of high-degree vertices is small.

Ghosts can only be used as an imprecise filter for algorithms such as BFS, because the ghosts are not globally synchronized. Algorithms that require precise counts of events, such as k-core discussed in Section VI-B, cannot use ghosts.

C. Visitor Queue Interface

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

- $push(visitor)$ – pushes a visitor into the visitor queue.

- $do\_traversal()$ – initializes and runs the asynchronous traversal to completion. This is used by the initiating algorithm.

When an algorithm requires dynamic creation of new visitors, they are *pushed* into the visitor queue using the *push(visitor)* procedure. When an algorithm begins, an initial set of visitors is pushed into the queue, then the *do_traversal()* procedure is invoked and runs the asynchronous traversal to completion.

D. Example Traversal

To illustrate how a asynchronous traversal algorithm works, we will discuss Breadth-First Search (BFS) at a high level. The details of BFS are discussed in Section VI-A. The visitor is shown in Algorithm 2, and the initiating procedure is shown in Algorithm 3.

BFS begins by setting an initial path length for every vertex to ∞ (Alg. 3 line 4). A visitor is queued for the *source* vertex with path length 0, then the traversal begins (Alg. 3 lines 9–10). As the visitors proceed, if they contain a lower path length that is currently known for the vertex, they update the path information and queue new visitors for the outgoing edges (Alg. 2 lines 13–16).

V. Visitor Queue Design Details

In this section, we discuss the design of the *distributed visitor queue*. The visitor queue has the following functionality, which will be discussed in detail in the following subsections and is shown in Algorithm 1.

- $push(visitor)$ – pushes a visitor into the distributed queue;
- $check\_mailbox()$ – receives visitors from *mailbox* and queues them locally;
- $global\_empty()$ – returns $true$ if globally empty;
- $do\_traversal()$ – runs the asynchronous traversal.

mailbox: The communication occurs though a *mailbox* abstraction with the following functionality:

- $send(rank, data)$ – Sends $data$ to $rank$, using the routing and aggregation network;
- $receive()$ – Receives messages from any sender.

ghost information: Our graph has the following ghost related operations used by the distributed visitor queue:

- $has\_local\_ghost(v)$ returns true if local ghost information is stored for v;
- $local\_ghost(v)$ returns ghost information stored for v.

push(visitor): This function pushes newly created visitors into the distributed visitor queue; the details are shown in Algorithm 1 (line 5). If the ghost information about the visitor's vertex is stored locally, it is *pre_visited* (line 8). If the *pre_visit* returns *true* or no local ghost information is found, then the visitor is sent to *min_owner* using the mailbox (lines 11, 14). This functionality abstracts the knowledge of ghost vertex information from the visitor function. If local ghost information is found, the framework will apply the visitor to the ghost. The ghosts act as local filters, reducing unnecessary visitors sent to *hub* vertices.

Algorithm 1 Distributed Visitor Queue

```
 1: state: g ← input graph
 2: state: mb ← mailbox communication
 3: state: my_rank ← local partition rank
 4: state: local_queue ← local visitor priority queue

 5: procedure PUSH($visitor$)
 6:     $vertex = visitor.vertex$
 7:     $master\_partition = g.min\_owner(vertex)$
 8:     if $g.has\_local\_ghost(vertex)$ then
 9:         $ghost = g.local\_ghost(vertex)$
                                  ▷ pre_visit locally stored ghost
10:         if $visitor.pre\_visit(ghost)$ then
11:             $mb.send(master\_partition, visitor)$
12:         end if
13:     else
14:         $mb.send(master\_partition, visitor)$
15:     end if
16: end procedure

17: procedure CHECK_MAILBOX
18:     for all $visitor \in mb.receive()$ do
19:         $vertex = visitor.vertex$
20:         if $visitor.pre\_visit(g[vertex])$ then
21:             $local\_queue.push(visitor)$
22:             if $my\_rank < g.max\_rank(vertex)$ then
                                  ▷ forwards to next replica
23:                 $mb.send(my\_rank + 1, visitor)$
24:             end if
25:         end if
26:     end for
27: end procedure

28: procedure GLOBAL_EMPTY          ▷ quiescence detection [12]
29:     return $true$ if globally empty, else $false$
30: end procedure

31: procedure DO_TRAVERSAL($source\_visitor$)
32:     $push(source\_visitor)$
33:     while $!global\_empty()$ do
34:         $check\_mailbox()$
35:         if $!local\_queue.empty()$ then
36:             $next\_visitor = local\_queue.pop()$
37:             $next\_visitor.visit(g, this)$
38:         end if
39:     end while
40: end procedure
```

check_mailbox(): This function checks for incoming visitors from the *mailbox*, and forwards visitors to potential edge list partitioned *replicas*. The details are shown in Algorithm 1 (line 17). For all visitors received from the mailbox (line 18), if the visitor's *pre_visit* returns *true*, the visitor is queued locally (line 21). Additionally, if the vertex is owned by ranks larger than the current rank, the visitor is forwarded to the next replica (line 22). This forwarding chains together vertices whose adjacency lists span multiple edge list partitions. The replicas are kept loosely consistent because visitors are first sent to the master and then forwarded to the chain of replicas in an ordered manner.

global_empty(): This function checks if all global visitor queues are empty, returning *true* if all queues are empty, and is used for *termination detection*. It is implemented using a simple $O(lg(p))$ *quiescence detection* algorithm based on visitor counting [12]. The algorithm performs an asynchronous reduction of the global visitor send and receive count using non-blocking point-to-point MPI communication. It is important to note that to check for non-termination is an asynchronous event, and only becomes synchronous after the visitor queues are already empty.

do_traversal(): This is the driving loop behind the asynchronous traversal process and is shown in Algorithm 1 (line 31). The procedure assumes that a set of initial visitors has been previously queued, then the main traversal loops until all visitors globally have been processed (line 33). During the loop, it checks the *mailbox* for incoming visitors (line 34), and processes visitors already queued locally.

A. External Memory Locality Optimization

The *less than comparison* operation used for local visitor ordering is defined by the algorithm. When using external memory, if two visitors have equal order priority, then they are prioritized to improve locality. In our experiments, the graphs are stored in a *compressed sparse row* format. To improve page-level locality, we order visitors by their vertex identifier when the algorithm does not define an order for a set of visitors. This additional sorting by the vertex identifier improves page-level locality for the graph data stored in NVRAM.

VI. Asynchronous Algorithms

In this section we discuss three algorithms implemented using our distributed visitor queue framework: breadth-first search, k-core decomposition, and triangle counting. In Section VI-D, we describe an asymptotic analysis framework to express the complexity of the asynchronous traversal in terms of the number of visitors.

A. Breadth-First Search

The visitor used to compute the BFS level for each vertex is shown in Algorithms 2 and 3. 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 length is smaller than the vertex's current length (Alg. 2 line 13). 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 *bfs_visitors* for each outgoing edge (Alg. 2 line 16).

The *less than comparison* procedure orders the visitors in the queue by *length* (Alg. 2 line 21). When a set of visitors all contain equal *length*, then the BFS algorithm does not specify an order and the framework can order based on locality, discussed in Section V-A.

Algorithm 2 BFS Visitor

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

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

12: procedure VISIT($graph$, $visitor\_queue$)
13:     if $length == graph[vertex].length$ then
14:         for all $vi \in out\_edges(g, vertex)$ do
15:             $new\_vis \leftarrow bfs\_visitor(vi, length + 1, vertex)$
16:             $visitor\_queue.push(new\_vis)$
17:         end for
18:     end if
19: end procedure

20: procedure OPERATOR $<$ ()($visitor\_a$, $visitor\_b$)
                              ▷ Less than comparison, sorts by length
21:     return $visitor\_a.length < visitor\_b.length$
22: end procedure
```

Algorithm 3 BFS Traversal Initiator

```
 1: input: $graph$ ← input graph $G(V, E)$
 2: input: $source$ ← BFS traversal source vertex
 3: input: $vis\_queue$ ← Visitor queue

 4: for all $v \in vertices(graph)$ parallel do
 5:     $graph[v].length \leftarrow \infty$
 6:     $graph[v].parent \leftarrow \infty$
 7: end for
 8: $source\_visitor \leftarrow bfs\_visitor(source, 0, source)$
 9: $vis\_queue.push(source\_visitor)$
10: $vis\_queue.do\_traversal()$
```

B. 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 Algorithms 4 and 5. 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.

The visitor's *pre_visit* procedure decrements the vertex's *k-core* number and checks if it is less than *k* (Alg. 4 line 6). If less, it sets *alive* to false and returns *true*, signaling that the visitors's main *visit* procedure should be executed (Alg. 4 line 8). The *visit* function notifies all neighbors of *vertex* that it has been removed from the k-core (Alg. 4 line 16). After the traversal completes, all vertices whose *alive* equals *true* are a member of the k-core.

Algorithm 4 K-Core Visitor

```
1: visitor state: vertex ← vertex to be visited
2: static parameter: k ← k-core requested

3: procedure PRE_VISIT($vertex\_data$)
4:     if $vertex\_data.alive == true$ then
5:         $vertex\_data.kcore \leftarrow vertex\_data.kcore - 1$
6:         if $vertex\_data.kcore < k$ then
7:             $vertex\_data.alive \leftarrow false$
8:             return $true$
9:         end if
10:    end if
11:    return $false$
12: end procedure

13: procedure VISIT($graph$, $visitor\_queue$)
14:    for all $vi \in out\_edges(g, vertex)$ do
15:        $new\_visitor \leftarrow kcore\_visitor(vi)$
16:        $visitor\_queue.push(new\_visitor)$
17:    end for
18: end procedure
                                          ▷ No visitor order required
```

Algorithm 5 K-Core Traversal Initiator

```
1: input: $graph$ ← input graph $G(V, E)$
2: input: $k$ ← k-core requested
3: input: $vis\_queue$ ← Visitor queue

4: $kcore\_visitor :: k \leftarrow k$            ▷ Set static visitor parameter
5: for all $v \in vertices(graph)$ parallel do
6:     $graph[v].alive \leftarrow true$
7:     $graph[v].kcore \leftarrow degree(v, graph) + 1$
8: end for
9: for all $v \in vertices(graph)$ parallel do
10:    $vis\_queue.push(kcore\_visitor(v))$
11: end for
12: $vis\_queue.do\_traversal()$
```

C. Triangle Counting

The visitor used to count the triangles in an undirected graph is shown in Algorithms 6 and 7. Each vertex maintains the count of the number of triangles for which the vertex identifier is the largest member of, initialized to zero.

The visitor's *pre_visit* always returns true; every visitor will execute its *visit* procedure. The *visit* procedure (Alg. 6) has three main duties: first visit (line 8), length-2 path visit (line 15), and search for closing edge of length-3 cycle (line 22). At each step, the vertices of the triangle are visited in increasing order (lines 10, 17) to prevent the triangle from being counted multiple times. If the closing edge is found, *num_triangles* is incremented (line 24). The global number of triangles can be accumulated after the traversal completes (Alg. 7 line 14).

This algorithm can be extended to count the number of triangles amongst a subset of vertices, or for individual vertices. It can also be extended to use approximate sampling based triangle counting methods [13].

Algorithm 6 Triangle Count Visitor

```
1: visitor state: vertex ← vertex to be visited
2: visitor state: second ← initialized to ∞
3: visitor state: third ← initialized to ∞

4: procedure PRE_VISIT($vertex\_data$)
5:     return $true$
6: end procedure

7: procedure VISIT($graph$, $visitor\_queue$)
8:     if $second == \infty$ then                   ▷ Visiting first vertex
9:         for all $vi \in out\_edges(graph, vertex)$ do
10:            if $vi > vertex$ then
11:                $new\_vis \leftarrow tri\_count\_vis(vi, vertex)$
12:                $visitor\_queue.push(new\_vis)$
13:            end if
14:        end for
15:    else if $third == \infty$ then              ▷ Visiting second vertex
16:        for all $vi \in out\_edges(graph, vertex)$ do
17:            if $vi > vertex$ then
18:                $new\_vis \leftarrow tri\_count\_vis(vi, vertex, second)$
19:                $visitor\_queue.push(new\_vis)$
20:            end if
21:        end for
22:    else                                      ▷ Search for closing edge
23:        if $third \in out\_edges(graph, vertex)$ then
24:            $graph[vertex].num\_triangles += 1$
25:        end if
26:    end if
27: end procedure
                                          ▷ No visitor order required
```

Algorithm 7 Triangle Count Traversal Initiator

```
1: input: $graph$ ← input graph $G(V, E)$
2: input: $vis\_queue$ ← Visitor queue

3: for all $v \in vertices(graph)$ parallel do
4:     $graph[v].num\_triangles = 0$
5: end for
6: for all $v \in vertices(graph)$ parallel do
7:     $vis\_queue.push(tri\_count\_vis(v))$
8: end for
9: $vis\_queue.do\_traversal()$
10: $local\_count = 0$
11: for all $v \in vertices(graph)$ parallel do
12:    $local\_count += graph[v].num\_traingles$
13: end for
14: $global\_count = all\_reduce(local\_count, SUM)$
15: return $global\_count$
```

D. Asymptotic Analysis Framework

Here we analyze the upper bounds on the number of visitors required for each algorithm. We make the following optimistic assumptions to simplify the analysis, for a graph $G(V, E)$ using p processors:

Parallel Rounds. The asynchronous algorithm proceeds in synchronized parallel *rounds*, in which each processor executes at most one visitor. There is a single shared visitor queue that all p processors access without contention. At the end of each round, the visitor queue is updated with newly queued visitors as necessary. The transmission latency for newly queued visitors is instantaneous, occurring at the end of the round. During a parallel round only one visitor can be selected per vertex, guaranteeing the visitor private access to the vertex.

Graph Properties. The underlying graph properties may have a significant impact on the complexity of algorithms. We use the following list of graph properties to parameterize our analysis:

- D – The Graph's diameter;
- d^{out}_{max} – Maximum out-degree, $\max\limits_{v \in V}(out\text{-}degree(v))$;
- d^{in}_{max} – Maximum in-degree, $\max\limits_{v \in V}(in\text{-}degree(v))$.

The analysis proceeds by bounding the number of parallel *rounds* required by the algorithm.

1) Analysis of BFS: Each parallel round executes up to p visitors, however only one of the visitors is guaranteed to belong to a shortest or critical path. For a connected graph, the length of the shortest path, and also the number of required parallel rounds is proportional to the diameter of the graph. The total number of parallel rounds is bounded by $\Theta(D + \frac{|E|}{p} + d^{in}_{max})$ without the use of ghosts. When using ghosts, the term d^{in}_{max} decreases to p, because the ghosts filter the high-degree visitors to one per partition. With ghosts, the number of parallel rounds is bounded by $\Theta(D + \frac{|E|}{p} + p)$.

2) Analysis of K-Core: Similarly to BFS, each parallel round executes up to p visitors, however only one of the visitors is guaranteed to belong to the critical path. For a connected graph, the length of the critical path, and the number of required parallel rounds, is proportional to the diameter of the graph. Unlike BFS, k-core cannot use ghost vertices for filtering, therefore the largest hub will require processing d^{in}_{max} visitors The total number of parallel rounds is bounded by $\Theta(D + \frac{|E|}{p} + d^{in}_{max})$.

3) Analysis of Triangle Counting: The visitor for triangle counting performs three basic duties: first visit, length-2 path visit, and search for closing edge of length-3 cycle. The *first visit* duty is performed for every vertex in the graph, and these visitors create *length-2 path* visitors. Each edge in the graph will have a corresponding *length-2 path* visitor, and these visitors will create at most $O(d^{out}_{max})$ visitors to search for the enclosing *length-3 cycle*. Triangle counting cannot use ghost vertices for filtering, therefore the largest hub will require processing d^{in}_{max} visitors. The total number of parallel rounds is bounded by $O(\frac{|E| d^{out}_{max}}{p} + d^{in}_{max})$.

VII. Experiments

In this section we experimentally evaluate the scalability of our approach both in distributed memory and distributed external memory. We focus on scale-free graphs with challenging hub growth. We demonstrate that our approach is scalable in two dimensions: it is scalable to large processor count for leadership class supercomputers, and it is scalable to distributed external memory using emerging HPC clusters containing node-local NVRAM. We also demonstrate the effects of using *edge list partitioning* and *ghosts* to mitigate the effects of high-degree hubs.

Currently, large scale HPC clusters with node-local NVRAM are not readily available; therefore we demonstrate scalability using two sets of experiments. To show scalability to large core count, we performed experiments using IBM BG/P supercomputers up to 131K cores; these experiments do not use the external memory aspect of the algorithm. BG/P experiments were preformed using Intrepid at Argonne National Laboratory and uDawn at Lawrence Livermore National Laboratory. Next, to demonstrate external memory scalability on distributed memory clusters, we performed experiments on Hyperion-DIT at LLNL; Hyperion has node-local NVRAM.

A. Experimental Setup

We implemented our distributed visitor queue and routed mailbox in C++ using only non-blocking point-to-point MPI communication. For our external memory experiments, where the graph data is completely stored in NVRAM, we used our custom user-space *page cache* to interface with NVRAM, discussed in Section II-B.

We show experiments using three synthetic graph models. After graph generation, all vertex labels are uniformly permuted to destroy any locality artifacts from the generators:

- RMAT – Generates scale-free graphs [14], and we follow the Graph500 V1.2 specification for generator parameters. We used the open source implementation provided by the Boost Graph Library [15].
- Preferential Attachment (PA) – Generates scale-free graphs [16]. We added an optional random *rewire* step to interpolate between a random graph and a PA graph for some experiments.
- Small World (SW) – Generates graphs with uniform vertex degree and a controllable diameter [8]. SW graphs interpolate between a ring graph and a random graph using a *random rewire* step.

B. Scalability on BG/P Supercomputer

We demonstrate the scalability using IBM BG/Ps at the Argonne National Laboratory and Lawrence Livermore National Laboratory. Intrepid is ranked 5th on the November 2011 and 15th June 2012 Graph500 list, with an efficient high performance custom implementation of the benchmark.

1) BFS: We have scaled our BFS algorithm up to 131K cores using the 3D routed mailbox discussed in Section III-B. We achieved excellent weak scaling as shown in Figure 5. In addition to showing weak scalability, we demonstrate the efficiency of our implementation by comparing to the current best known Graph500 result for Intrepid from the June 2012 list [1]. Our approach, designed to use portable MPI and external memory, achieved 64.9 GTEPS with 2^{35} vertices, which is only 19% slower than the best known BG/P implementation.

We use this experiment to establish the scalability of our approach, and the efficiency of our implementation. This

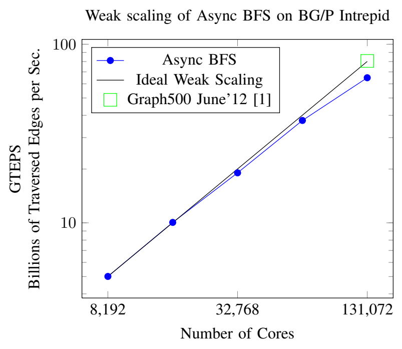

Figure 5. Weak scaling of Asynchronous BFS on BG/P Intrepid with all edge weights equal to 1. Compared to Intrepid BFS performance from the Graph500 list. There are 2^{18} vertices per core, with the largest scale graph having 2^{35}.

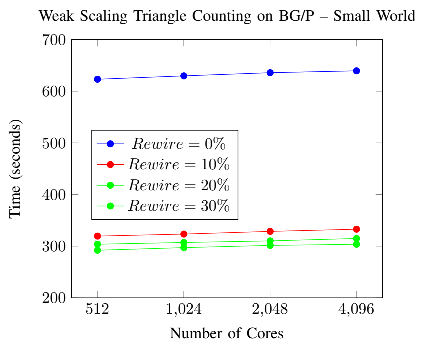

Figure 7. Weak scaling of triangle counting on BG/P using Small World graphs. Performance shown at with different small world rewire probabilities. There are 2^{18} vertices and 2^{22} undirected edges per core; at 4096 cores, the graph has 2^{30} vertices and 2^{34} edges.

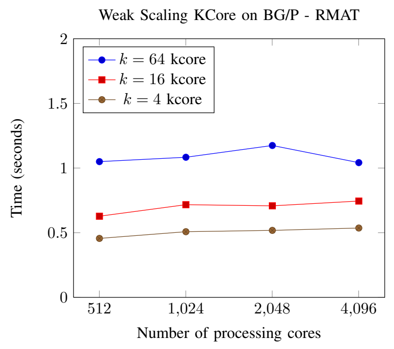

Figure 6. Weak Scaling of kth-core on BG/P using RMAT graphs. Time shown to compute cores 4, 16 and 64. There are 2^{18} vertices and 2^{22} undirected edges per core; at 4096 cores, the graph has 2^{30} vertices and 2^{34} edges.

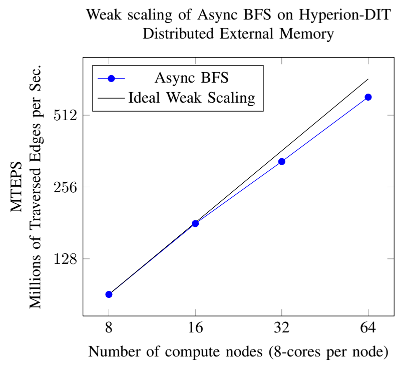

Figure 8. Weak scaling of distributed external memory BFS on Hyperion-DIT. Each compute node has 8-cores, 24GB DRAM, and is using 169GB NAND Flash to store graph data. There are 17B edges per compute node; the largest scale graph has over one-trillion edges and 2^{36} vertices.

experiment forms the basis for the NVRAM vs. DRAM experiments we performed on Hyperion-DIT.

2) K-Core Decomposition: We show weak scaling of k-core decomposition on BG/P up to 4096 cores using RMAT graphs in Figure 6. The time to compute the cores 4, 16 and 64 are shown for each graph size. Our techniques enable near linear weak scaling for computing k-core.

3) Triangle Counting: We show weak scaling of triangle counting on BG/P up to 4096 cores using Small World graphs in Figure 7. We show the time to count the triangles on small world graphs with rewire probabilities 0%, 10%, 20%, and 30%. The small world generator creates vertices with a uniform vertex degree (in this case 32). As will be discussed in Section VII-D, the performance of triangle counting is dependent on the maximum vertex degree of the graph. For this weak scaling study, we use small world graphs to isolate the effects of hub growth that would occur with PA or RMAT graphs.

C. Scalability of Distributed External Memory BFS

We demonstrate the distributed external memory scalability of our BFS algorithm on the Hyperion-DIT cluster at Lawrence Livermore National Laboratory (LLNL). The Hyperion-DIT is an 80-node subset of Hyperion that is equipped with node-local Fusion-io NAND Flash. Each compute node has 8 cores, 24 GB DRAM, and 600 GB NVRAM.

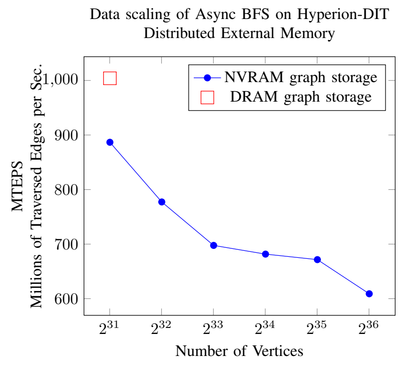

Figure 9. Effects of increasing external memory usage on 64 compute nodes of Hyperion-DIT. At 2^{36}, which is 32x larger data than DRAM-only, the NVRAM performance is only 39% slower than DRAM graph storage.

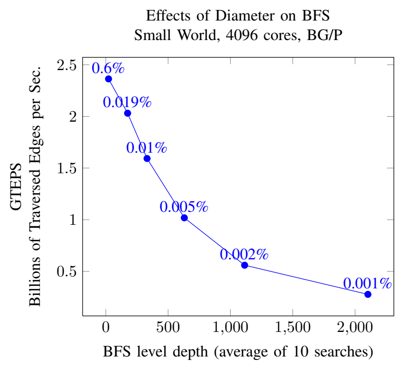

Figure 10. Effects of diameter on BFS performance. Small World graph model with varying *random rewire* probabilities (shown above each point). BFS level depth used for x-axis. Computational resources fixed at 4096 cores of BG/P. Graph size is fixed at 2^{30} vertices and 2^{34} undirected edges.

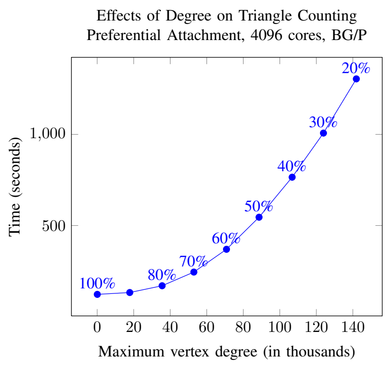

Figure 11. Effects of vertex degree on Triangle Counting performance. Preferential Attachment graph model with varying *random rewire* probabilities (shown above each datapoint). Maximum vertex degree used for x-axis. Computational resources fixed at 4096 cores of BG/P. Graph size is fixed at 2^{28} vertices and 2^{32} undirected edges.

We performed a weak scaling study, in which each compute node stores 17 billion edges on its local NVRAM, roughly 169GB in a compressed sparse row format. The results of the weak scaling are shown in Figure 8; at 64 compute nodes the graph has over one trillion undirected edges and 2^{36} vertices, twice the size as experiments on BG/P Intrepid. We expect our approach to continue scaling as larger HPC clusters with node-local NVRAM become available.

To experiment with the effects of increasing NVRAM usage per compute node, we performed an experiment where the computational resources are held constant while the data size increases. The results of our data scaling experiment are shown in Figure 9. As the data size increases, from 34 billion to 1 trillion edges (10.8 TB), the additional data we store on NVRAM results in only a moderate performance reduction. The overall performance reduction from DRAM-only to 32x larger graph stored in NVRAM is only 39%. This is a significant increase in data size with only a moderate performance degradation.

Results using our distributed external memory BFS for the Graph500 are shown in Table II. In addition to the Hyperion-DIT cluster, we performed experiments on Trestles at the San Diego Supercomputing Center (SDSC) and Leviathan (LLNL). Each compute node in Trestles has commodity SATA SSDs, and shows that our approach is not limited to enterprise class NVRAM. Leviathan is a single-node system using implementations from our previous multithreaded work [4]. Leviathan has 1TB of DRAM and 12TB of Fusion-io in a single host. These represent 3 of 8 total systems on the November 2011 Graph500 list traversing trillion-edge graphs.

D. Topological Effects on Performance

The performance of BFS and triangle counting are dependent on topological properties of the graph. For BFS, the performance is dependent on the diameter (the longest shortest-path) of the graph. Using the Small World graph

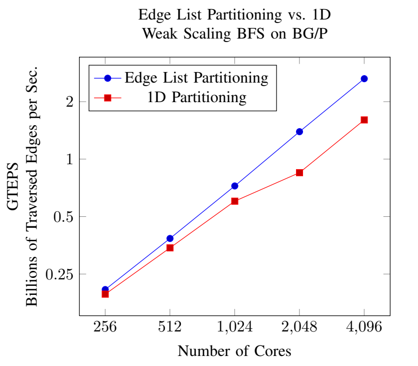

Figure 12. Comparison of *edge list partitioning* vs 1D. Performance of BFS on RMAT graphs shown on BG/P. Important note: the graph sizes are reduced to prevent 1D from running out of memory. There are 2^{17} vertices and 2^{21} undirected edges per core.

generator, we show the effects of increasing diameter on BFS in Figure 10. By decreasing the small-world random rewire probability while keeping the size of the graph constant, the diameter of the graph increases. The increasing diameter causes the performance of BFS to decrease. Similarly, the performance of triangle counting is dependent on the maximum vertex degree. Using the Preferential Attachment graph generator with an added step of random rewire, we show the effects of increasing hub degree while keeping the graph size constant in Figure 11. The topological effects can also be seen in the algorithm analysis in Sections VI-D1 and VI-D3.

E. Framework Effects on Performance

1) Edge List Partitioning vs 1D: To demonstrate the effects of *edge list partitioning* vs. traditional 1D partitioning we show the weak scaling of both with BFS on RMAT graphs on BG/P in Figure 12. Because 1D partitioning suffers from data imbalance, the graph sizes in the experiments were reduced to prevent 1D from running out of memory. The weak scaling of edge list partitioning is almost linear, while

Machine Name	Location	Machine Type	Graph Storage	Num Vertices	TEPS
Hyperion-DIT	LLNL	64 nodes, 512 cores	DRAM	2^{31}	1,004 MTEPS
			Fusion-io	2^{36}	609 MTEPS
Trestles	SDSC	144 nodes, 4608 cores	SATA SSD	2^{36}	242 MTEPS
Leviathan	LLNL	single node, 40 cores	Fusion-io	2^{36}	52 MTEPS

Table II
OUR NOVEMBER 2011 GRAPH500 RESULTS USING NAND FLASH. SCALE 36 IS A GRAPH WITH OVER 1 TRILLION EDGES.

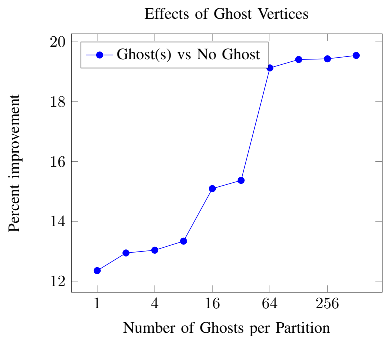

Figure 13. Experiment showing the percent improvement of ghost vertices vs. no ghost vertices. Results from 4096 cores of BG/P on a graph with 2^{30} vertices.

1D suffers slowdowns from the partition imbalance.

2) Use of Ghost Vertices: We experimented with the effects on BFS performance of choosing different size k on a graph with 2^{30} vertices on 4096 cores of BG/P. The results of this experiment are shown in Figure 13, where the percent improvement of k ghosts per partition vs. no ghosts is shown. Using a single ghost shows more than a 12% improvement, and 512 ghosts shows an 19.5% improvement. This improvement is highly dependent on the graph, and as hubs grow to larger sizes it may have a larger effect. For scale-free graphs, when $degree(v) > p$, there is an opportunity for ghosts to have a positive effect, because at least one partition will have multiple edges to v. All other BFS experiments in this work use 256 ghost vertices per partition.

VIII. RELATED WORK

Processing of large graphs is receiving increasing attention by the HPC community as datasets quickly grow past the memory capacity of commodity workstations. Significant challenges arise for traditional HPC solutions because of unstructured memory accesses and poor data locality in graph datasets [17], [18].

A. Distributed Memory Graphs

Recent work using 2D graph partitioning has shown the best results for traditional large scale HPC systems [3], [10], [11]. However, it has serious disadvantages at scale and with external memory. First, when processing sparse graphs, each 2D block may become hypersparse, i.e., fewer edges than vertices per partition [19]. Specifically, partitions become hypersparse when $O(\sqrt{p}) > degree(g)$, where p is the number of distributed partitions and g is the graph. For the sparse Graph500 datasets with average degree of 16, this may occur for as low as 256 partitions and is independent of graph size. Second, under weak scaling, 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. This will ultimately hit a scaling wall where the amount of local algorithm state per partition exceeds the capacity of the compute node. Finally, with respect to our desire to use semi-external memory where the vertex set is stored in in-memory and the edge set is stored in external memory, hypersparse partitions are poor candidates to apply semi-external memory techniques, because the in-memory requirements (proportional to the number of vertices) are larger than the external memory requirements (proportional to the number of edges). Our *edge list partitioning* does not create hypersparse partitions (assuming the original graph is not hypersparse) as 2D can.

Recent work related to our *routed mailbox* has been explored by Willcock [20]. Active messages are routed through a synthetic *hypercube* network to improve dense communication scalability. A key difference to our work is that their approach has been designed for the Bulk Synchronous Parallel (BSP) model and is not suitable for asynchronous graph traversals.

The STAPL Graph Library [21] provides a framework that abstracts the user from data-distribution and parallelism and supports asynchronous algorithms to be expressed.

B. Sequential External Memory Algorithms

To analyze the I/O complexity of algorithms using external storage, the Parallel Disk Model (PDM) [22] has been developed. When designing I/O efficient algorithms, the key principles are *locality of reference* and *parallel disk access*. Vitter has published an in-depth survey of external memory algorithms [23]; the majority of algorithms analyzed with the PDM model are sequential.

Graph traversals (e.g., BFS) are efficient when computing sequentially in-memory, but become impractical in external memory. Sequential in-memory BFS incurs $\Omega(V + E)$ I/Os when using external memory, and it has been reported that the in-memory BFS performs orders of magnitude slower when forced to use external memory [24]. Sequential external-memory algorithms for BFS for undirected graphs have been developed [25], [26], however it is considered impractical for general sparse directed graphs. Ajwani published an in-depth survey of external memory graph traversal algorithms [27].

IX. Conclusion

Our work focuses on an important *data intensive* problem of massive graph traversal. We aim to scale to trillion-edge graphs using both leadership class supercomputers and node-local NVRAM that is emerging in many new HPC systems.

We address scaling challenges, created by scale-free power-law degree distributions, by applying an *edge list partitioning* strategy, and show its application to three important graph algorithms: BFS, K-Core, and triangle counting.

We demonstrate the scalability of our approach on up to 131K cores of BG/P Intrepid, and we show that by leveraging node-local NAND Flash, our approach can process 32x larger datasets with only a 39% performance degradation in TEPS.

Our work breaks new ground for using NVRAM in the HPC environment for data intensive applications. We show that by exploiting both distributed memory processing and node-local NVRAM, significantly larger datasets can be processed than with either approach in isolation. Further, we demonstrate that our asynchronous approach mitigates the effects of both distributed and external memory latency. The architecture and configuration of NVRAM in supercomputing clusters is an active research topic. To our knowledge, our work is the first to integrate node-local NVRAM with distributed memory at extreme scale for important data intensive problems.

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-588232). Funding was partially provided by LDRD 11-ERD-008. Portions of experiments were performed at the Livermore Computing facility resources. 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-0615267, CCF-0833199, CCF-0830753, IIS-0917266, IIS-0916053, NSF/DNDO award 2008-DN-077-ARI018-02, by DOE awards DE-FC52-08NA28616, DE-AC02-06CH11357, B575363, B575366, by THECB NHARP award 000512-0097-2009, by Samsung, Chevron, IBM, Intel, Oracle/Sun and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). Pearce is supported in part by a Lawrence Scholar fellowship at LLNL.

References

[1] 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.

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

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

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

[5] B. V. Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of NVRAM in data-intensive architectures: an evaluation," in *International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

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

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

[8] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, jun 1998.

[9] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Supercomputing*, 2010.

[10] 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.

[11] 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.

[12] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, vol. 2, pp. 161–175, 1987.

[13] C. Seshadhri, A. Pinar, and T. G. Kolda, "Triadic measures on graphs: The power of wedge sampling," in *SIAM International Conference on Data Mining*, 2013.

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

[15] 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.

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

[17] B. Hendrickson and J. W. Berry, "Graph analysis with high-performance computing," *Computing in Science and Engineering*, vol. 10, no. 2, pp. 14–19, 2008.

[18] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[19] 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.

[20] 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.

[21] 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.

[22] J. S. Vitter and E. A. Shriver, "Algorithms for parallel memory I: Two-level memories," *Algorithmica*, vol. 12, no. 2-3, 1994.

[23] J. S. Vitter, "Algorithms and data structures for external memory," *Found. Trends Theor. Comput. Sci.*, vol. 2, no. 4, pp. 305–474, 2008.

[24] D. Ajwani, R. Dementiev, and U. Meyer, "A computational study of external-memory BFS algorithms," in *SODA '06: Proceedings of the seventeenth annual ACM-SIAM Symposium on Discrete Algorithms*, 2006, pp. 601–610.

[25] K. Munagala and A. Ranade, "I/O-complexity of graph algorithms," in *SODA '99: Proceedings of the tenth annual ACM-SIAM Symposium on Discrete algorithms*, 1999.

[26] K. Mehlhorn and U. Meyer, "External-memory breadth-first search with sublinear I/O," in *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, 2002.

[27] D. Ajwani and U. Meyer, "Design and engineering of external memory traversal algorithms for general graphs," *Algorithmics of Large and Complex Networks*, pp. 1–33, 2009.