Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory

Roger Pearce†‡, Maya Gokhale‡, and Nancy M. Amato†
{rpearce, amato}@cse.tamu.edu, maya@llnl.gov
† Parasol Laboratory
Department of Computer Science and Engineering
Texas A&M University
‡ Lawrence Livermore National Laboratory

***Abstract*—Processing large graphs is becoming increasingly important for many domains such as social networks, bioinformatics, etc. Unfortunately, many algorithms and implementations do not scale with increasing graph sizes. As a result, researchers have attempted to meet the growing data demands using parallel and external memory techniques.**

We present a novel asynchronous approach to compute Breadth-First-Search (BFS), Single-Source-Shortest-Paths, and Connected Components for large graphs in shared memory. Our highly parallel asynchronous approach hides data latency due to both poor locality and delays in the underlying graph data storage. We present an experimental study applying our technique to both In-Memory and Semi-External Memory graphs utilizing multi-core processors and solid-state memory devices. Our experiments using synthetic and real-world datasets show that our asynchronous approach is able to overcome data latencies and provide significant speedup over alternative approaches. For example, on billion vertex graphs our asynchronous BFS scales up to 14x on 16-cores.

I. Introduction

The ability to store and efficiently search large-scale graphs is becoming increasingly important as we seek to model Internet-scale phenomena. Graphs are used in a wide range of fields including Computer Science, Biology, Chemistry, Homeland Security, and the Social Sciences. These graphs may represent complex relationships between individuals, proteins, chemical compounds, etc.

Fundamental to many graph search applications is graph traversal, a process of visiting all of the vertices and edges in the graph. In this work, we are concerned with three basic graph traversals: Breadth First Search (BFS), Single Source Shortest Path (SSSP), and Connected Components (CC). These traversals are important building blocks to many graph analysis algorithms and applications.

A. *Examples of Large Graphs*

Many real-world graphs are large and require significant computational and memory resources to search and store.

- World Wide Web (WWW) graph – Graphs that model the structure of the web often consist of vertices representing webpages and directed edges representing the hyperlinks between the webpages.
- Social Networks – Graphs naturally model the relationships established by social interactions. These interactions could be on-line friendships, call-networks, etc.
- Homeland Security – It has been estimated that graphs of interest to the Department of Homeland Security will reach 10^{15} entities [1], providing a significant challenge to analysts who wish to search them. Currently, External Memory [2], [3], [4] and Parallel [5], [6], [7] approaches provide tools capable of supporting large graphs, but systems capable of searching 10^{15} entities remain elusive.

B. *Properties of Real-world Graphs*

- Power Law – A common property of many real world graphs is a power law distribution of vertex degree. In literature, this property is often referred to as *scale-free* and can lead to significant load imbalance when processing such a graph in parallel. An effect of the power law degree distribution is that while the vast majority of vertices have a low degree, a select few vertices will have a very high degree. These high degree nodes are often referred to as *hub* vertices.
- Small Diameter – Although sparse, many graphs are connected into giant connected components with small diameters. The diameter of a graph is the longest shortest-path between two vertices.
- Community Structure – Vertices that group into interconnected clusters are called communities. In a cluster, there are more interconnected edges than outgoing edges. This property leads to clusters which are highly interconnected while having only few connections outside of the group.

C. *Processing Large Graphs*

While graph algorithms have received tremendous attention for the RAM computational model, many realistic datasets are too large to fit in the memory of a single computer. To address this, researchers have explored using External Memory (EM) and Distributed Memory (DM); these approaches are discussed in Section VI. Key challenges in processing large graphs come from the non-contiguous access to the data structure. Distributed Memory approaches suffer from poor load balancing due to the intrinsic nature of power-law distributions.

SC10 November 2010, New Orleans, Louisiana, USA

External Memory approaches suffer from poor data locality and unstructured memory accesses for which techniques such as prefetching, blocking, and pipelining generally provide little improvement. Some experiments have shown that BFS designed for the RAM computation model runs orders of magnitude slower when forced to use external memory [8].

In this work, we are interested in graphs in a Semi-External Memory (SEM) scenario. A graph is semi-external if there is enough main memory to store algorithmic information about the vertices but not edges. In our SEM work, the full graph structure is stored on the persistent storage device, and the visitor queues and the output of the algorithms are stored in main memory.

D. *Our Contribution*

We present a novel asynchronous approach for graph traversal and demonstrate it by performing Breadth First Search (BFS), Single Source Shortest Paths (SSSP), and Connected Components (CC) computations for large graphs in shared memory. Our approach allows the computation to proceed in an asynchronous manner, reducing the number of costly synchronizations. As clock speeds flatten and massive threading becomes mainstream, asynchronous approaches will become necessary to overcome the increasing cost of synchronization.

We present an experimental study applying our technique to both In-Memory (IM) and Semi-External Memory (SEM) graphs utilizing multi-core processors and solid-state memory devices (SSDs). We provide a quantitative study comparing our approach to existing implementations such as the Boost Graph Library (BGL) [9], the Parallel Boost Graph library (PBGL) [5], the Multithreaded Graph Library (MTGL) [6], and the Small-world Network Analysis and Partitioning library (SNAP) [7]. Our experimental study evaluates both synthetic and real-world datasets, and shows that our asynchronous approach is able to overcome data latencies and provide significant speedup over alternative approaches. Our In-Memory experiments show that our asynchronous BFS is 1.6-1.8x faster than MTGL's BFS and 2.3-5.3x faster than SNAP's BFS. Our asynchronous CC is 2.8-13x faster than MTGL's CC; the wide range is attributable to differences in the graph structure. Our Semi-External Memory experiments show that for moderate and fast SSDs, our asynchronous approach is consistently faster than a serial In-Memory alternative like BGL, with even the slowest SSD tested being competitive with BGL.

II. Preliminaries

In this work, we are concerned with three basic graph computations that are fundamental to many other areas of graph analysis: Breadth First Search (BFS), Single Source Shortest Paths (SSSP), and Connected Components (CC).

A. *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. BFS can be also computed using a SSSP algorithm with all edge weights equal to 1.

The parallel versions of BFS that we compare to are *level synchronous* [6], [7]. This means that all threads of execution working on level i must finish and synchronize before starting to work on level $i+1$. In some cases, additional work between the level synchronizations is needed to merge the level sets.

B. *Single Source Shortest Paths (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 Single Source Shortest Path (SSSP) can be viewed as a hybrid between Bellman-Ford [10] and Dijkstra's [11] SSSP. Bellman-Ford *label-correcting* computes SSSP by making $|V|-1$ loops over all vertices, *relaxing* the path length of each vertex. Dijkstra's SSSP algorithm also iteratively relaxes vertices, but proceeds in a greedy manner, *relaxing* only the shortest-path vertex at each iteration.

We show comparisons to a distributed implementation of the Δ-stepping SSSP algorithm [12] provided by PBGL [5]. This algorithm proceeds in *bulk-synchronous* steps, where vertices within a *delta* of the shortest path are relaxed together.

C. *Connected Components (CC)*

A connected component of an undirected graph is a subgraph in which all vertices can be connected to all other vertices through pathways in the graph. In other words, if two vertices are in the same connected component, then there exists a pathway between them in the graph.

The Shiloach-Vishkin parallel connectivity algorithm [13] is a well known PRAM algorithm for computing connected components. We show comparisons to MTGL's [6] parallel implementation, which is an iterative algorithm covered in detail by JáJá [14].

D. *Flash Memory Technologies*

Emerging technologies in persistent data storage will change the way External Memory algorithms are designed. Flash memory is a form of non-volatile persistent storage that has become a commodity product through widespread use in digital cameras, music players, phones, USB drives, etc. An overview of the characteristics and performance of flash memory (namely NAND Flash) with respect to algorithmic research is given in [15], [16]. The key differences with traditional rotating media can be summarized as:

- Significantly faster random access time than disk (microseconds instead of 100's of milliseconds).
- Asymmetric read/write performance (writes are more costly than reads).

An important characteristic of NAND Flash devices not covered by [15], [16] is the ability to service multiple concurrent I/O requests. To achieve maximum random I/O performance, multiple threads must queue I/O requests. This requires External Memory algorithms to be multithreaded to achieve maximum I/O performance. Figure 1 shows the multithreaded random read performance of the three NAND Flash configurations that we test in this paper. For all configurations tested, significant improvements in I/O per second (IOPS) are seen as an increasing number of threads issue read requests. The Flash configuration details are discussed in Section IV-C.

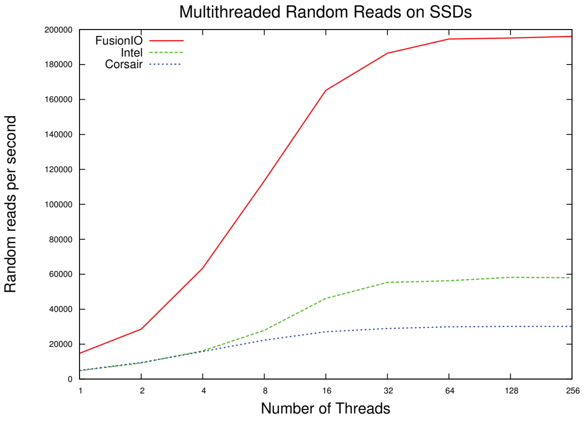


Fig. 1. Multithreaded random read I/O performance for three NAND Flash configurations. Configuration details discussed in Section IV-C.

III. Asynchronous Graph Traversal

Using currently accepted synchronous techniques, load imbalance may occur between the synchronization points, leading to performance loss. The motivation behind our asynchronous graph traversal is to overcome these performance issues caused by load imbalance and memory latencies. This section outlines the use of prioritized visitor queues to asynchronously compute Breadth First Search (BFS), Single Source Shortest Paths (SSSP), and Connected Components (CC).

A. Multithreaded Asynchronous Visitor Queue

The *vertex visitor* abstraction is a common way to abstract the process in which a graph traversal visits the vertices of a graph. Our approach uses the *vertex visitor* pattern with prioritized work queues to form a *visitor queue*.

As the graph traversal proceeds, vertices are visited and adjacent vertices to be visited (if needed) are queued into the visitor queue. The traversal is complete when the visitor queue is empty, and all visitors have completed.

In a multithreaded environment, the *visitor queue* can be implemented as a set of priority queues with a hash function controlling the selection of an individual queue. Using multiple queues with a hash function reduces lock contention when multiple threads are inserting or removing from the queues. In our implementation and experiments, each thread ‘owns’ a queue and the queue is selected based on a hash of the vertex identifier. This adds an additional guarantee that a visitor has exclusive access to a vertex when executing, removing the need for additional vertex-level locking when visiting a vertex. Additionally, a near-uniform hash function may improve load balance amongst the visitor queues as high-cost vertices will be uniformly distributed across the queues.

B. Single Source Shortest Path and BFS

Like Bellman-Ford, our approach relies on *label-correcting* to compute the traversal, and completes when all corrections are complete. Like Dijkstra’s SSSP, our approach traverses paths in a prioritized manner, *visiting* the shortest path possible at each visit. Our approach does not introduce synchronizations between steps; therefore, we cannot guarantee that the absolute shortest-path vertex is visited at each step, possibly requiring multiple visits per vertex. In this work, we compute a Breadth First Search (BFS) by applying our asynchronous SSSP algorithm with all edge weights equal to 1.

Algorithms 1 and 2 outline an asynchronous SSSP traversal of a graph g. The traversal starts at the source in Algorithm 1. For each vertex visited, an instance of Algorithm 2 decides if the current path length needs to be corrected, and queues the adjacent vertices if the path has been updated. A visitor corrects the path length if it represents a shorter pathway than is currently assigned to the vertex. The visitor queue is prioritized based on the visitors’ path length. After starting the traversal, Algorithm 1 waits for all queued visitors to complete.

Algorithm 1 Single Source Shortest Path – Main

```
INPUT: g ← input graph
INPUT: dist_array ← an array holding the path length to
    each vertex, initialized to $\infty$
INPUT: parent_array ← an array holding the path parent
    to each vertex, initialized to $\infty$
INPUT: start ← starting vertex for SSSP
pq_visit ← a multithreaded visitor queue, prioritized by
    SSSPVertexVisitor’s cur_dist
pq_visit.push( SSSPVertexVisitor(g, pq_visit, dist_array,
    parent_array, start, 0, start) )
pq_visit.wait() //wait for queued work to finish
```

1) Algorithmic Analysis: The performance of Algorithms 1 and 2 is highly dependent on the structure of the graph traversed. If the graph has multiple shortest-path pathways that can be independently traversed, then the algorithm will have the opportunity to proceed in parallel. However, without the independent pathways, the algorithm will traverse the graph in a serialized manner. Figure 2 is an example of a directed graph with poor parallelism when traversed starting from vertex 0.

More formally, the algorithm’s complexity can be represented by $O((|E|/p)log(|V|/p))$, where V and E are the number of vertices and edges in the graph, respectively, and p is the degree of parallelism the traversal can exploit. In the worst case, where the traversal is serialized ($p = 1$), the algorithm reduces to Dijkstra’s SSSP with a performance of $O(|E|log|V|)$. From our experiments with scale-free graphs and web-graphs, a significant amount of path parallelism exists in these real-world graphs, giving rise to performance approaching the best case.

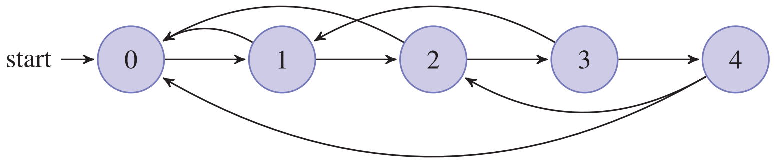
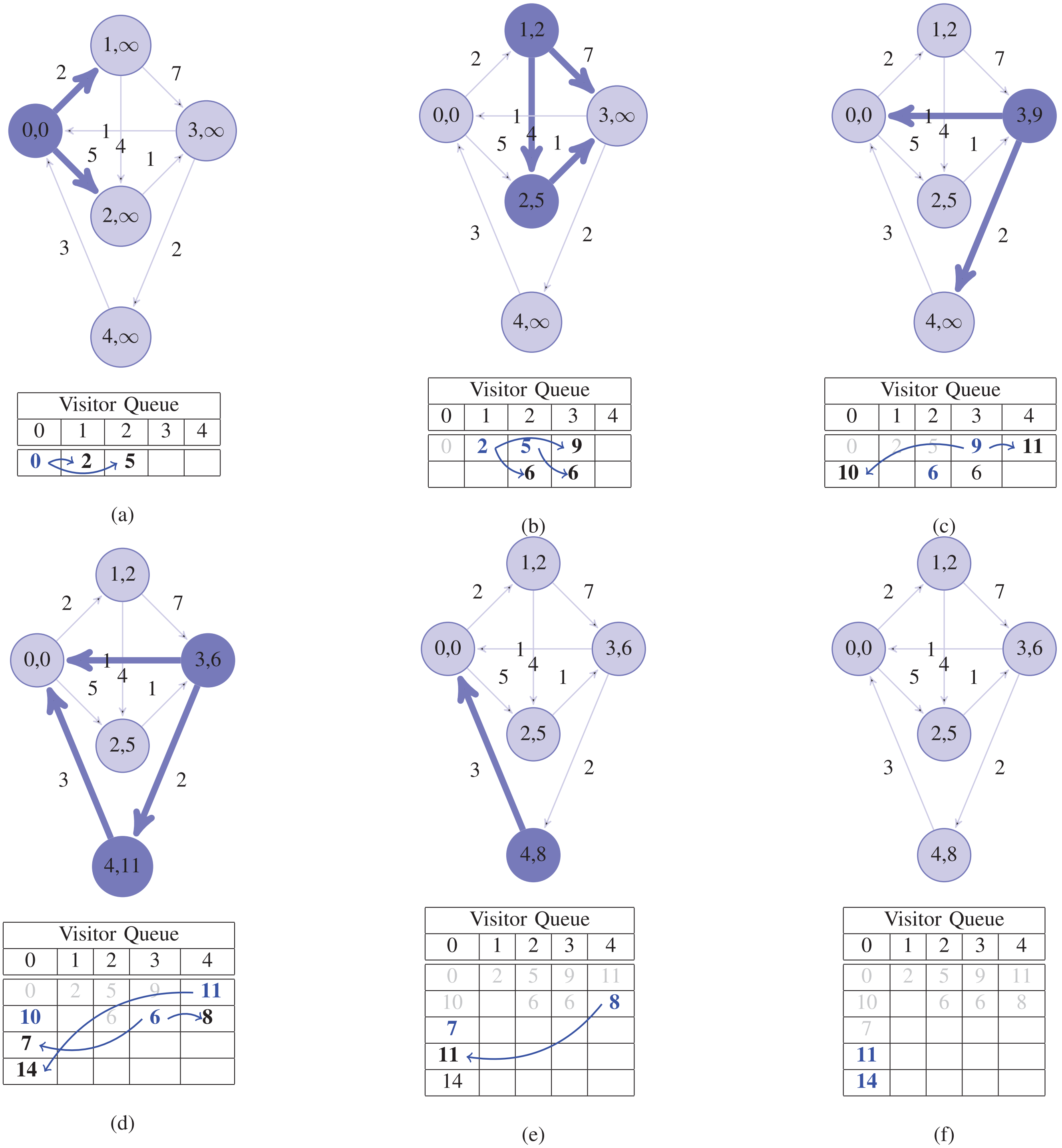


Fig. 2. An example directed graph with poor parallelism for BFS and SSSP



Visitor Queue				
0	1	2	3	4
0	2	5		

(a)

Visitor Queue				
0	1	2	3	4
0	2	5	9	
		6	6	

(b)

Visitor Queue				
0	1	2	3	4
0	2	5	9	11
10		6	6	

(c)

Visitor Queue				
0	1	2	3	4
0	2	5	9	11
10		6	6	8
7				
14				

(d)

Visitor Queue				
0	1	2	3	4
0	2	5	9	11
10		6	6	8
7				
11				
14				

(e)

Visitor Queue				
0	1	2	3	4
0	2	5	9	11
10		6	6	8
7				
11				
14				

(f)

Fig. 3. An example of an asynchronous Single Source Shortest Path (SSSP) traversal of a simple weighted directed graph. Section III-B2 discusses the details of this example.

Algorithm 2 Single Source Shortest Path – SSSPVertexVisitor

```
INPUT: g $\leftarrow$ input graph
INPUT: pq_visit $\leftarrow$ a multithreaded visitor queue
INPUT: dist_array $\leftarrow$ an array holding the path length
INPUT: parent_array $\leftarrow$ an array holding the path parent
INPUT: v $\leftarrow$ vertex to visit
INPUT: cur_dist $\leftarrow$ tentative bound on min path length
INPUT: cur_parent $\leftarrow$ tentative shortest path parent
if cur_dist $<$ dist_array[v] then
    dist_array[v] = cur_dist //relax vertex information
    parent_array[v] = cur_parent
    adj_list $\leftarrow$ g.adj_list( v )
    for all vj $\in$ adj_list do
        edge_weight = g.edge_weight(v,vj)
        pq_visit.push( SSSPVertexVisitor(g, pq_visit,
        dist_array, parent_array, vj, edge_weight, v ) )
    end for
end if
```

2) SSSP Traversal Example: To illustrate how the asynchronous computation proceeds we describe SSSP as seen in Algorithms 1 and 2. The computation is initialized by queuing a visitor at the *source* with *path_length* $= 0$. Upon visiting a vertex, each visitor evaluates if the current path length needs to be corrected. If the visitor updates the path, the visitor queues new visitors for the vertex's adjacent vertices.

Figure 3 gives a pictorial example for a simple weighted directed graph. In this example, the weights were purposefully selected to require multiple visits per vertex. In a real-world context, the weights may represent distances between locations, strength of association between agents, or any other domain-specific relationship information. For simplicity, the computation is represented by 6 steps; however it is important to note that no synchronization is introduced between the steps and the order of the visitor queues is not guaranteed. For clarity, a visitor queue is shown for each vertex; however in practice, a queue may represent a large subset of vertices. The computation in Figure 3 proceeds as follows:

(a) All vertices initialize their path length to ∞. *Vertex 0* initializes to a path length of 0 and queues a visitor to vert 1 with length 2, and a visitor to vert 2 with length 5.

(b) *Vertex 1* is visited with length 2, updates its path length to 2, and queues a visitor to vert 2 with length 6, and a visitor to vert 3 with length 9. *Vertex 2* is visited with length 5, updates its path length to 5, and queues a visitor to vert 3 with length 6.

(c) *Vertex 2* is visited with length 6; because length 6 is longer than its current length 5, it does not update its path length; no new visitors queued. *Vertex 3* has 2 visitors queued, however order is not guaranteed and it processes length 9 first, updates its path length to 9, and queues a visitor to vert 0 with length 10 and visitor to vert 4 with length 11.

(d) *Vertex 0* is visited with length 10, and does not update its path length; no new visitors queued. *Vertex 3* is visited with length 6, updates its path length to 6, and queues a visitor to vert 4 with length 8, and a visitor to vert 0 with length 7. *Vertex 4* is visited with length 11, updates its path length to 11, and queues a visitor to vert 0 with length 14.

(e) *Vertex 0* is visited with length 7, and does not update its path length; no new visitors queued. *Vertex 4* is visited with length 8, updates its path length to 8, and queues a visitor to vert 0 with length 11.

(f) *Vertex 0* is visited with length 11, and does not update its path length; no new visitors queued. For the subsequent time step, vertex 0 is visited with length 14, and does not update its path length; no new visitors queued. *End*: The computation terminates when all visitor queues are empty.

C. Undirected Connected Components

The asynchronous computation of Connected Components (CC) for undirected graphs is similar to that of SSSP. To compute CC, each vertex is labeled by the smallest vertex descriptor that is connectable. The computation is outlined in Algorithms 3 and 4; in Algorithm 3, a visitor for each vertex is queued in parallel with the vertex's descriptor as the starting component id. When a vertex is visited, if its component id can be updated to a smaller id, then it is updated and visitors for all adjacent vertices are queued with the updated component id. The computation is finished when all queued visitors complete.

Algorithm 3 Undirected Connected Components – Main

```
INPUT: g $\leftarrow$ input graph
INPUT: ccid_array $\leftarrow$ an array holding the cc id for each
vertex, initialized to $\infty$
pq_visit $\leftarrow$ a multithreaded visitor queue, prioritized by
UCCVertexVisitor's cur_ccid
for all v $\in$ g.vertex_list() parallel do
    pq_visit.push( UCCVertexVisitor(g, pq_visit,
    ccid_array, v, v) )
end for
pq_visit.wait() //wait for queued work to finish
```

As with SSSP, the parallel performance of Algorithms 3 and 4 is highly dependent on the underlying graph structure. A worst case graph with poor parallelism is similar to that of SSSP in Figure 2, only undirected.

Our approach to CC can be viewed as performing parallel BFS starting from every vertex. When two BFSs that started from different vertices merge, the BFS that started from the lowest vertex identifier takes over the remainder of both traversals. The end result is that all vertices in the graph are labeled with the smallest vertex identifier connectable to them.

IV. Implementation Details

We have created two similar implementations for In-Memory and Semi-External Memory graphs.

Algorithm 4 Undirected Components – UCCVertexVisitor

```
INPUT: g ← input graph
INPUT: pq_visit ← a multithreaded visitor queue
INPUT: ccid_array ← an array holding the cc id
INPUT: v ← vertex to visit
INPUT: cur_ccid ← tentative bound on min cc id
if cur_ccid $<$ ccid_array[v] then
    ccid_array[v] = cur_ccid //relax vertex information
    adj_list ← g.adj_list( v )
    for all vj $\in$ adj_list do
        pq_visit.push( UCCVertexVisitor(g, pq_visit, ccid_array, vj, cur_ccid) )
    end for
end if
```

A. Thread Oversubscription

Our implementation can benefit from using more threads than cores. Because there is a prioritized queue per thread, with an associated lock, having more threads/queues than cores reduces queue lock contention. From our experiments, using as many as 512 threads on 16 cores offers substantial benefit.

B. In-Memory Implementation

For In-Memory graph storage, we used Boost's Compressed Sparse Row C++ library [9]. For POSIX thread support, we used the Boost Thread library [17]. All code was compiled with g++ version 4.1.2 with -O3.

C. Semi-External Implementation

For Semi-External graph storage, we used a custom file-based storage implementing a *compressed sparse row* using explicit POSIX standard I/O access.

For POSIX thread support, we used the Boost Thread library [17]. All code was compiled with g++ version 4.1.2 with -O3.

The only difference from the in-memory algorithm implementation is that the prioritized visitor queues have an additional secondary sorting parameter, the vertex identifier. This increases access locality to the storage devices by semi-sorting access if possible. Because we use a compressed sparse row graph format, the adjacency set for vertex i will be close in terms of data locality to the adjacency set of vertex $i+1$. When the visitor queues are able to visit vertices with close identifiers, aggregate page-level locality can be exploited. Using BFS as an example, not only will level 1's vertices be processed before level 2's, the vertices in level 1 will be visited in a semi-sorted order to increase locality.

V. Experimental Studies

A. Graph Types and Sizes

We performed experiments using both synthetic and real graph inputs of various sizes. For the three algorithms tested in this work, the input graphs were organized as follows:

- BFS - Directed synthetic graphs, unweighted;
- SSSP - Directed synthetic graphs, two forms of random weights;
- CC - Undirected graphs, synthetic and real.

1) Synthetic Graphs: For synthetic graphs, we used scale-free graphs generated by the RMAT [20] graph generator. The RMAT graph generator uses a 'recursive matrix' model to create graphs that model 'real-world' graphs. We generated directed graphs with unique edges ranging from 2^{25} – 2^{30} vertices and an average out-degree of 16. The vertex identifiers are permuted after graph generation. Undirected versions of these graphs for use with Connected Components were created by adding reverse edges. Properties of the RMAT graphs are shown in Table I. We generated 2 types of RMAT graphs with different RMAT parameters:

- RMAT-A : $a = 0.45, b = 0.15, c = 0.15, d = 0.25$: This creates a scale-free graph with moderate out-degree skewness.
- RMAT-B : $a = 0.57, b = 0.19, c = 0.19, d = 0.05$: This creates a scale-free graph with heavy out-degree skewness.

For weighted SSSP experiments, we added edge weights to the RMAT graphs in the following manner:

- UW - uniform weights range from $[0, num\_vertices)$
- LUW - log-uniform weights range from $[0, 2^i)$, where i is chosen uniformly from $[0, lg(num\_vertices))$

2) Real Graphs: For real graphs, we experimented with five different web traces that were treated as undirected. Properties of the five web graphs are shown in Table I

B. Hardware Resources

Our implementation and experiments were performed using Linux computers at Lawrence Livermore National Laboratory:

- AMD256GB – Single compute node; 16-core AMD Opteron(tm) 8356 with 256 GB of main memory. This machine is used for Async, MTGL, SNAP, DIMACS-SSSP, and BGL discussed in V-C1.
- AMDCluster – Linux cluster; 16-core AMD Opteron(tm) 8356 with 32 GB of main memory. This cluster is used only for PBGL discussed in V-C1
- AMD16GB – Single compute node; 16-core AMD Opteron(tm) 8356 with 16 GB of main memory. In addition, this machine can be configured with 3 different types of NAND Flash storage.

Using the AMD16GB machine, we experimented with 3 types of NAND Flash based storage:

- AMD16GB-FusionIO – 4x 80GB FusionIO SLC, PCI-E cards in a software RAID 0 configuration. Our experiments show that this configuration is capable of close to 200,000 random reads per second. This is the fastest device that we have tested, and much of its speed is due to its PCI-E interface.
- AMD16GB-Intel – 4x 80GB Intel X25-M MLC, SATA SSDs in a software RAID 0 configuration. Our experiments show that this configuration is capable of close to 60,000 random reads per second.
- AMD16GB-Cosair – 4x 128GB Corsair P128 MLC, SATA SSDs in a software RAID 0 configuration. Our

				Size on Flash Device	
graph	# nodes	# edges	# components	directed	undirected
RMAT-A	2^{25}	2^{29}	34,008	small	
	2^{26}	2^{30}	72,647	small	
	2^{27}	2^{31}	154,179	9 GB	17 GB
	2^{28}	2^{32}	327,072	18 GB	34 GB
	2^{29}	2^{33}	689,979	36 GB	68 GB
	2^{30}	2^{34}	1,448,438	72 GB	136 GB
RMAT-B	2^{25}	2^{29}	13,739,228	small	
	2^{26}	2^{30}	28,448,613	small	
	2^{27}	2^{31}	58,757,785	9 GB	17 GB
	2^{28}	2^{32}	121,037,055	18 GB	34 GB
	2^{29}	2^{33}	249,937,778	36 GB	68 GB
	2^{30}	2^{34}	510,267,039	72 GB	136 GB
ClueWeb09 [18]	1,667,267,985	7,939,647,897	3,149,668	n/a	
it-2004 [19]	41,291,595	1,150,725,436	979	n/a	
sk-2005 [19]	50,636,155	1,949,412,601	126	n/a	14 GB
uk-union [19]	133,633,041	5,507,679,822	2,097,197	n/a	36 GB
webbase-2001 [19]	118,142,156	1,019,903,190	2,721,051	n/a	

TABLE I
PROPERTIES OF GRAPH DATASETS USED IN EXPERIMENTS.

experiments show that this configuration is capable of close to 30,000 random reads per second.

Our semi-external approach requires a high level of IOPS to achieve good performance. For this reason, we have not studied our approach on traditional rotating media. Figure 1 shows the multithreaded random I/O performance of the three NAND Flash configurations that we test in this work.

C. In-Memory Experiments

1) In-Memory Graph Libraries: For experimental comparison, we show the performance of the following graph libraries:

- Async – our asynchronous approach.
- MTGL [6], [21] – a shared memory parallel graph library primarily designed for Cray's massively multithreaded machines. For commodity SMP systems, MTGL has implementations of BFS and CC that use the QThreads library [22] for threading support. It is important to note that MTGL's performance on SMP systems does not reflect on its performance on the Cray XMT. Our experiments use MTGL's Subversion Trunk revision 2827.
- SNAP [7] – a parallel graph library for shared memory which utilizes OpenMP for parallelism. Our experiments use SNAP version 0.3.
- PBGL [5] – a distributed memory parallel graph library. *Direct comparisons between distributed memory and shared memory implementations are not possible, however these experiments help to compare alternative techniques for large graph processing.* Our experiments use PBGL from version 1.43 of the Boost library. PBGL experiments are performed on AMDCluster up to 2048-cores; we report the core-count that performs optimally.
- BGL [9] – a serial graph library. BGL is used as an efficient serial baseline to compute speedup. Our experiments use BGL from version 1.43 of the Boost library.
- DIMACS-SSSP – an implementation of Goldberg's multilevel bucket shortest path algorithm [23]. It was used as the reference solver from the 9th DIMACS Implementation Challenge on shortest paths.

2) Breadth First Search (BFS): Figure 4 shows our asynchronous Breadth First Search (BFS) compared with MTGL, SNAP, PBGL all normalized by BGL. Our approach, MTGL, SNAP and BGL were tested on AMD256GB. PBGL is shown with the optimal number of cores between 64-2048 and 128GB-4TB of memory tested on AMDCluster.

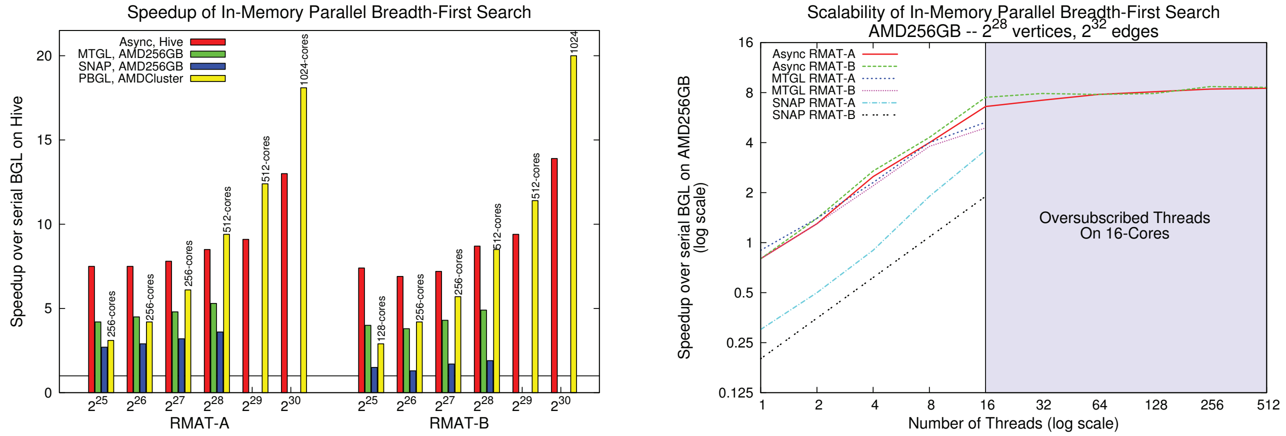
At full parallelism, our asynchronous BFS is roughly 1.6-1.8x faster than MTGL's and 2.3-5.3x times faster than SNAP's BFS for our test cases. SNAP's BFS struggles with the highly skewed degree distribution of the RMAT-B datasets, leading to poor scaling and speedup. MTGL's and SNAP's graph data structures are implemented using 64-bit integers and are unable to fit the 2^{29} and 2^{30} vertex graphs in 256GB of memory; our implementation can be configured to use 32 or 64-bit integers.

Using significant resources, 128-1024 cores with 256GB-2TB of memory, PBGL can compute BFS on the 2^{28}, 2^{29}, and 2^{30} vertex graphs the fastest. However, PBGL's parallel speedup is small when compared to the number of cores used.

The scalability of Async, MTGL, and SNAP on AMD256GB is shown in Figure 4(b) for the 2^{28} vertex graphs. The modest benefit of thread oversubscription for Async can be seen here, as in all test cases 512 threads outperform 16 threads using our approach; this indicates that further scaling is possible beyond 16-cores. MTGL and SNAP do not benefit from thread oversubscription.

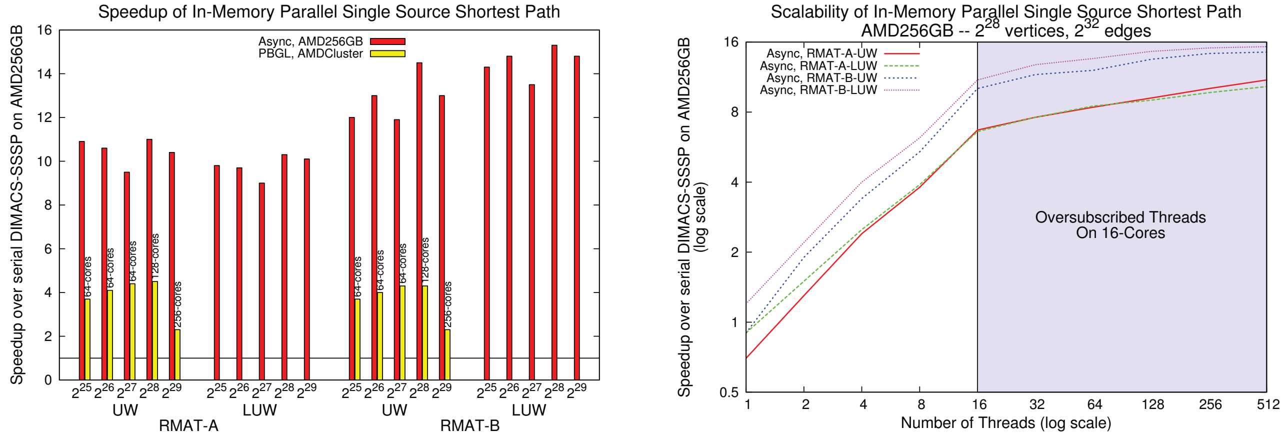
Overall, our approach outperforms MTGL and SNAP in all of our test cases, and was competitive with PBGL which uses 4-64x the number of cores and up to 8x the memory.

3) Single Source Shortest Path (SSSP): Figure 5 shows Async and PBGL normalized with the DIMACS-SSSP tested



(a) (b)

Fig. 4. Performance comparison of In-Memory Breadth First Search (BFS). Speedup (a) shows parallel libraries normalized by BGL. Scalability (b) shows shared-memory libraries only normalized by BGL. For PBGL, we report the optimal core-count up to 2048.



(a) (b)

Fig. 5. Performance comparison of In-Memory Singe Source Shortest Path (SSSP). Speedup (a) shows parallel libraries normalized by BGL. Scalability (b) shows shared-memory libraries only normalized by BGL. For PBGL, we report the optimal core-count up to 2048.

on AMD256GB. PBGL's Δ-stepping SSSP is shown with the optimal number of cores between 64-2048 and 128GB-4TB of memory AMDCluster. These experiments use the same directed graphs as the BFS experiments in Section V-C2 and add uniform (UW) and log-uniform (LUW) edge weights.

PBGL's Δ-stepping SSSP is unable to scale past 256-cores. The optimal core-count was 64-256 cores for the UW graphs. An appropriate *lookahead* value could not be found for the LUW graphs. For both the UW and LUW graphs, the default heuristic for choosing a *lookahead* value was poor.

Again, we see that Async benefits from thread oversubscription as all tests perform best using 512 threads on 16 cores, which indicates that further scaling is possible beyond 16-cores. Async achieves a speedup of up to 15.3 on 16-cores when normalized to DIMACS-SSSP.

4) Connected Components: Figure 6 shows Async, MTGL, PBGL normalized by BGL on AMD256GB. PBGL is shown with the optimal number of cores between 64-2048 and 128GB-4TB of memory on AMDCluster.

Our asynchronous CC is 2.8-4.5x faster than MTGL in all synthetic cases tested. For the real web-graphs, our asynchronous CC is 4-13 times faster than MTGL.

At 16 threads, our approach is consistently better than MTGL in our experiments. Oversubscribing to 512 threads further improves performance in all cases, which again indicates that further scaling is possible beyond 16-cores.

Using significant resources, PBGL is able to outperform both Async and MTGL on 3 of the largest RMAT-A graphs. However, PBGL performs extremely poorly on the RMAT-B and WebGraphs; in many cases it is unable to complete due to memory limitations.

D. Semi-External Memory Experiments

This section describes our experiments traversing Semi-External Memory (SEM) graphs stored on solid state FLASH devices on AMD16GB normalized to BGL running In-Memory on AMD256GB. It is important to note that the In-Memory BGL performance numbers are used as a baseline to evaluate the Semi-External performance. Also, the processors

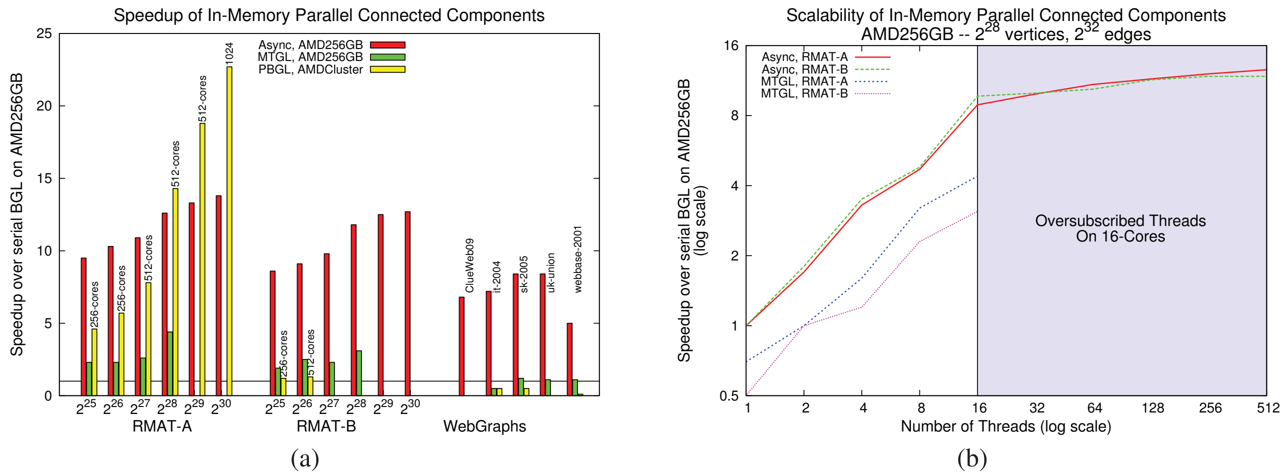


Fig. 6. Performance comparison of In-Memory Connected Components (CC). Speedup (a) shows parallel libraries normalized by BGL. Scalability (b) shows shared-memory libraries only normalized by BGL. For PBGL, we report the optimal core-count up to 2048.

and motherboard on AMD16GB and AMD256GB are identical (only memory size differs), so a direct comparison between IM and SEM can be made.

The ability to process large graphs in semi-external memory at comparable or better to in-memory performance is important. The cost of solid state devices like NAND Flash SSDs can be significantly less than DRAM costs, and offers persistent data storage. Our SEM experiments show that for moderate and fast SSDs, our asynchronous approach is consistently faster than a serial In-Memory alternative like BGL, with even the slowest SSD tested performing comparable to BGL.

1) Breadth First Search: Figure 7(a) shows our Semi-External asynchronous BFS compared with the In-Memory BGL BFS. Using the FusionIO or Intel SSDs, we typically outperform the serial BGL which requires a larger amount of memory to store the graph in-memory. The FusionIO drive offers the highest random I/O access speed and typically outperforms other SSDs we have tested. Even the Corsair, the slowest of the SSDs we tested, shows performance comparable to BGL's in-memory performance.

2) Connected Components: Figure 7(b) shows our Semi-External asynchronous CC compared with the In-Memory BGL CC. As with BFS, our semi-external approach to connected components can outperform BGL's in-memory using the FusionIO and Intel SSDs. Again, the FusionIO drive typically offers the highest semi-external performance.

VI. Related Work

A. HPC and Graphs

Processing of large graphs is receiving increasing attention by the HPC community as datasets quickly grow past the capacity of commodity workstations. Significant challenges arise for traditional HPC solutions because of the nature of these datasets. These challenges can be summarized into unstructured memory access and poor data locality [24], [25].

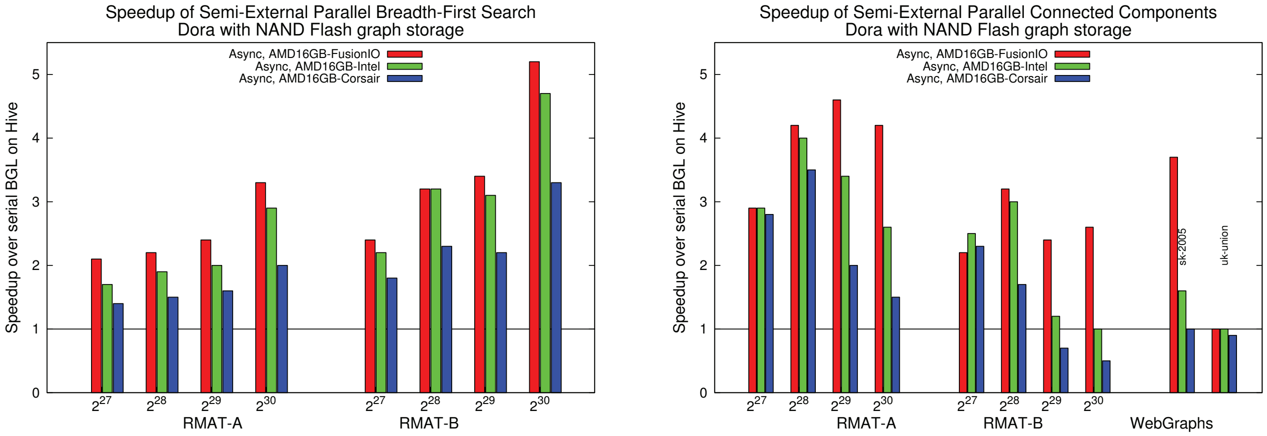
A popular approach to graph processing in HPC has been to use Distributed Memory computer clusters. Such clusters distribute the graph data amongst its processors and memory and process the graph by exchanging messages during computation phases. This approach works well when the graph exhibits nice load balancing properties (regular or uniformly random) [26] but suffers from significant load imbalance when processing power-law graphs [5]. Our approach addresses the load imbalance challenge by using an asynchronous approach.

Massive Multithreaded machines address the challenges of unstructured memory accesses and poor data locality by using little or no memory hierarchy. The Cray XMT has been successful at processing large graph datasets; these specialized supercomputers rely on massive multithreading to mask memory latency without using complex memory caches. The development of the Multithreaded Graph Library (MTGL) for this specialized computing platform as been shown to address many of the issues related to memory latency [21]. Our approach addresses the memory latency issues using commodity hardware and storage devices (NAND Flash) that are relatively slow compared with main memory.

Asynchronous algorithms for computing shortest paths in parallel have been previously studied [27], [28]. Our work builds on these techniques to create an asynchronous approach that can overcome load imbalance and data latencies.

B. External Memory Graph Algorithms

Many real world graphs are too large to fit into main memory of modern computers, necessitating the use of external storage devices such as disk. Due to the significant difference in access times between main memory and disk, many efficient in-memory algorithms become impractical when using external storage. To analyze the I/O complexity of algorithms using external storage, the Parallel Disk Model (PDM) [29] has been developed. PDM's main parameters are N (problem size), M (size of internal memory), B (block transfer size), D (number of independent disks), and P (number of CPUs). When designing I/O efficient algorithms, the key principles are *locality of reference* and *parallel disk access*. For an in-depth survey of EM Algorithms, see [30].



(a) (b)

Fig. 7. Performance comparison of Semi-External Memory on three FLASH memory configurations. Breadth-First Search (a) and Connected Components (b) show Async performance in semi-external memory normalized by In-Memory BGL's serial version on AMD256GB.

Graph traversal (e.g., Breadth First Search) is an algorithm that is efficient when computing in-memory, but becomes impractical in external memory. In-memory BFS incurs $\Omega(n+m)$ 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 [8].

For general undirected graphs, Munagala and Ranade [31] improve the worst-case I/O of BFS to $O(n + sort(m))$ by exploiting the fact that a node in BFS level i can only have edges to nodes in level $i-1$ or $i+1$, removing the need to check all previous BFS levels. The $O(n)$ term in Munagala and Ranade's algorithm is due to non-contiguous access to the adjacency lists, requiring separate access. Mehlhorn and Meyer [2] improved the adjacency list access by pre-processing the graph into subgraphs of low diameter and storing their adjacency lists contiguously, leading to sub-linear I/O complexity.

For general directed graphs, improvements over in-memory BFS and DFS have not been made, their I/O complexity is $O((n + m/B)lg(n/B) + sort(m))$ [4]. This is considered impractical for general sparse directed graphs. For an in-depth survey of EM graph traversal algorithms, see [4].

VII. Conclusions

In this work, we present a novel asynchronous graph traversal approach and demonstrate it on Breadth First Search (BFS), Single Source Shortest Paths (SSSP), and Connected Components (CC) computations for large graphs in shared memory. Our approach allows the computation to proceed in an asynchronous manner, reducing the number of synchronizations. As clock speeds flatten and massive threading becomes mainstream, asynchronous approaches will become necessary to overcome the increasing cost of synchronization.

We present an experimental study applying our technique to both In-Memory (IM) and Semi-External Memory (SEM) graphs utilizing multi-core processors and solid-state memory devices. We provide a quantitative study comparing our approach to existing implementations such as the Boost Graph Library (BGL) [9], the Parallel Boost Graph library (PBGL) [5], the Multithreaded Graph Library (MTGL) [6], and the Small-world Network Analysis and Partitioning library (SNAP) [7]. Our experimental study evaluates both synthetic and real-world datasets, and shows that our asynchronous approach is able to overcome data latencies and provide significant speedup over alternative approaches. Our In-Memory experiments show that our asynchronous BFS is 1.6-1.8x faster than MTGL's and 2.3-5.3x faster than SNAP's BFS, and our asynchronous CC is 2.8-13 times faster than MTGL's CC. Our Semi-External Memory experiments show that for moderate and fast SSDs, our asynchronous approach is consistently faster than a serial In-Memory alternative like BGL, with even the slowest SSD tested performing comparable to BGL.

Overall, we have shown that our asynchronous approach overcomes the cost of synchronization and data latencies in shared memory. Our technique works with both In-Memory and Semi-External Memory graphs, and shows potential for scaling as the number of cores on a processor increases.

VIII. 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-427572). Funding partially provided by LDRD 07-ERD-063. Portions of experiments were performed at the Livermore Computing facility resources. This research supported in part by NSF awards CRI-0551685, CCF-0833199, CCF-0830753, IIS-096053, IIS-0317266, by NSF/DNDO award 2008-DN-077-ARI018-02, by the DOE NNSA under the Predictive Science Academic Alliances Program by grant DE-FC52-08NA28616, by THECB NHARP grant 000512-0097-2009, by Chevron, IBM, Intel, HP, Oracle/Sun and by King Abdullah University of Science and Technology (KAUST) Award KUS-C1-016-04 Pearce is supported in part by a Lawrence Scholar Fellowship and a Dept. of Education Graduate Fellowship (GAANN).

REFERENCES

[1] T. Kolda, D. Brown, J. Corones, T. Critchlow, T. Eliassi-Rad, L. Getoor, B. Hendrickson, V. Kumar, D. Lambert, C. Matarazzo, K. McCurley, M. Merrill, N. Samatova, D. Speck, R. Srikant, J. Thomas, M. Wertheimer, and P. C. Wong, "Data sciences technology for homeland security information management and knowledge discovery: Report of the DHS workshop on data sciences," Jointly released by Sandia National Laboratories and Lawrence Livermore National Laboratory, Tech. Rep. UCRL-TR-208926, September 2004.

[2] 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*. London, UK: Springer-Verlag, 2002, pp. 723–735.

[3] J. Abello, A. L. Buchsbaum, and J. R. Westbrook, "A functional approach to external graph algorithms," *Algorithmica*, vol. 32, no. 3, pp. 437–458, 2002.

[4] D. Ajwani and U. Meyer, "Design and engineering of external memory traversal algorithms for general graphs," pp. 1–33, 2009.

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

[6] B. W. Barrett, J. W. Berry, R. C. Murphy, and K. B. Wheeler, "Implementing a portable multi-threaded graph library: The MTGL on Qthreads," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–8, 2009.

[7] D. A. Bader and K. Madduri, "SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *IPDPS*, 2008, pp. 1–12.

[8] 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 algorithm*. New York, NY, USA: ACM, 2006, pp. 601–610.

[9] 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.

[10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms, 2nd edition*. MIT Press and McGraw-Hill, 2001.

[11] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[12] U. Meyer and P. Sanders, "[delta]-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114 – 152, 2003, 1998 European Symposium on Algorithms.

[13] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982.

[14] J. JáJá, *An introduction to parallel algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1992.

[15] D. Ajwani, A. Beckmann, R. Jacob, U. Meyer, and G. Moruz, "On computational models for flash memory devices," in *Experimental Algorithms*, 2009, pp. 16–27.

[16] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo, "Characterizing the performance of flash memory storage devices and its impact on algorithm design," in *Proceedings of the 7th International Workshop on Experimental Algorithms (WEA)*, 2008, pp. 208–219.

[17] "BOOST Threads," www.boost.org/doc/libs/release/libs/thread/.

[18] "ClueWeb09 dataset," http://boston.lti.cs.cmu.edu/Data/clueweb09/.

[19] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[20] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Fourth SIAM International Conference on Data Mining*, April 2004.

[21] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, March 2007, pp. 1–14.

[22] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *IPDPS*, 2008, pp. 1–8.

[23] A. V. Goldberg, "A simple shortest path algorithm with linear average time," in *ESA '01: Proceedings of the 9th Annual European Symposium on Algorithms*. London, UK: Springer-Verlag, 2001, pp. 230–241.

[24] 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.

[25] 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.

[26] 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 *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 25.

[27] D. P. Bertsekas, F. Guerriero, and R. Musmanno, "Parallel asynchronous label-correcting methods for shortest paths," *J. Optim. Theory Appl.*, vol. 88, no. 2, pp. 297–320, 1996.

[28] F. Guerriero and R. Musmanno, "Parallel asynchronous algorithms for the k shortest paths problem," *Journal of Optimization Theory and Applications*, vol. 104, no. 1, pp. 91–108, 2000.

[29] J. S. Vitter and E. A. Shriver, "Algorithms for parallel memory I: Two-level memories," *Algorithmica*, vol. 12, no. 2-3, pp. 110–147, 1994.

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

[31] 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*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999, pp. 687–694.