

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 computational domains. Unfortunately, many algorithms and implementations do not scale with the demand for increasing graph sizes. As a result, researchers have attempted to meet the growing data demands using parallel and external memory techniques. Our work, targeted to chip multi-processors, takes a highly parallel asynchronous approach to hide the high data latency due to both poor locality and delays in the underlying graph data storage.

We present a novel asynchronous approach to compute Breadth First Search (BFS), Single Source Shortest Path (SSSP), and Connected Components (CC) for large graphs in shared memory. 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. Our experiments using both synthetic and real-world datasets show that our asynchronous approach is able to overcome data latencies and provide significant speedup over alternative approaches.

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.

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.

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 store and search

efficiently.

- 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 such graphs. Currently, External Memory and Parallel 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 research for the RAM computational model, many realistic datasets are too large to fit in the memory of a single computer. To address this problem, researchers have explored using External Memory (EM) and Distributed Memory (DM); these

approaches are discussed in Section VI. The 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. External Memory approaches suffer from poor data locality and unstructured memory accesses such that techniques like prefetching, blocking, and pipelining provide little improvement in the general case. Some experiments have shown that BFS designed for the RAM computation model runs orders of magnitude slower when forced to use external memory [2].

D. Our Contribution

We present a novel asynchronous approach to graph traversal and demonstrate the approach by computing Breadth First Search (BFS), Single Source Shortest Path (SSSP), and Connected Components (CC) 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. We provide a quantitative study comparing our approach to existing implementations such as the Boost Graph Library (BGL) [3], the Parallel Boost Graph library (PBGL) [4], the Multithreaded Graph Library (MTGL) [5], and the Small-world Network Analysis and Partitioning library (SNAP) [6]. 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 10-18% faster than MTGL's and 1.5-3 times faster than SNAP's BFS, and our asynchronous CC is 2-13 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.

II. PRELIMINARIES

In this work, we are concerned with three basic graph traversals that are fundamental to many other areas of graph analysis: Breadth First Search (BFS), Single Source Shortest Path (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.

B. Single Source Shortest Paths (SSSP)

A SSSP algorithm computes the shortest path through a weighted graph to each vertex from a single source 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 [7] and Dijkstra's [8] SSSP. Bellman-Ford is a *label-correcting* algorithm to compute SSSP by making $|V| - 1$ loops over all vertices, *relaxing* the path length of each vertex. Similarly, Dijkstra's SSSP algorithm iteratively relaxes vertices but proceeds in a greedy manner, *relaxing* only the shortest-path vertex at each iteration.

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.

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 [9], [10]. 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 [9], [10] 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 in order 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) is seen as an increasing number of threads issue read requests. The Flash configuration details are 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, Single Source Shortest Path, and Connected Components.

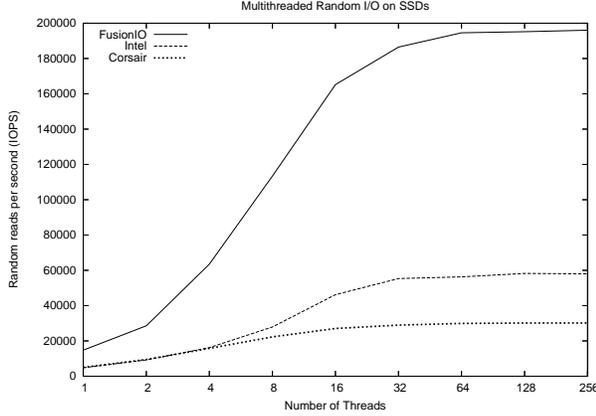


Fig. 1. Multithreaded random read I/O performance for three NAND Flash configurations. Configuration details discussed in Section IV-C.

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 multiple priority queues with a hash function controlling the selection of a 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 is started in Algorithm 1 at the

source. 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 the visitor 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

- 1: **INPUT:** $g \leftarrow$ input graph
 - 2: **INPUT:** $dist_array \leftarrow$ an array holding the path length to each vertex, initialized to ∞
 - 3: **INPUT:** $parent_array \leftarrow$ an array holding the path parent to each vertex, initialized to ∞
 - 4: **INPUT:** $start \leftarrow$ starting vertex for SSSP
 - 5: $pri_q_visit \leftarrow$ a multithreaded visitor queue, prioritized by SSSPVertexVisitor’s cur_dist
 - 6: $pri_q_visit.push(SSSPVertexVisitor(g, pri_q_visit, dist_array, parent_array, start, 0, start))$
 - 7: $pri_q_visit.wait() //$ wait for queued work to finish
-

Algorithm 2 Single Source Shortest Path – SSSPVertexVisitor

- 1: **INPUT:** $g \leftarrow$ input graph
 - 2: **INPUT:** $pri_q_visit \leftarrow$ a multithreaded visitor queue
 - 3: **INPUT:** $dist_array \leftarrow$ an array holding the path length to each vertex
 - 4: **INPUT:** $parent_array \leftarrow$ an array holding the path parent to each vertex
 - 5: **INPUT:** $v \leftarrow$ vertex to visit
 - 6: **INPUT:** $cur_dist \leftarrow$ current shortest path length to vertex
 - 7: **INPUT:** $cur_parent \leftarrow$ current shortest path parent
 - 8: **if** $cur_dist < dist_array[v]$ **then**
 - 9: $dist_array[v] = cur_dist //$ relax vertex information
 - 10: $parent_array[v] = cur_parent$
 - 11: $adj_list \leftarrow g.adj_list(v)$
 - 12: **for all** $v_j \in adj_list$ **do**
 - 13: $edge_weight = g.edge_weight(v,v_j)$
 - 14: $pri_q_visit.push(SSSPVertexVisitor(g, pri_q_visit, dist_array, parent_array, v_j, edge_weight, v))$
 - 15: **end for**
 - 16: **end if**
-

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, 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. In the worst case, where the traversal is serialized, the algorithm is bounded by Dijkstra’s SSSP performance of $O(|E| \log |V|)$ for sparse graphs. In the best case, with at least p parallelism in the graph traversal, the algorithm is bounded by $O((|E|/p) \log(|V|/p))$. 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 an average case performance bounded by the best parallel case upper bound.

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 vertex 1 with length 2, and a visitor to vertex 2 with length 5.
 - (b) Vertex 1 is visited with length 2, updates its path length to 2, and queues a visitor to vertex 2 with length 6, and a visitor to vertex 3 with length 9.
Vertex 2 is visited with length 5, updates its path length to 5, and queues a visitor to vertex 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 vertex 0 with length 10 and visitor to vertex 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 vertex 4 with length 8, and a visitor to vertex 0 with length 7.
Vertex 4 is visited with length 11, updates its path length to 11, and queues a visitor to vertex 0 with length 14.
 - (e) Vertex 0 is visited with length 14, 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 vertex 0 with length 11.
 - (f) Vertex 0 is visited with length 7, and does not update its path length; no new visitors queued. For the subsequent time step, vertex 0 is visited with length 11, and does not update its path length; no new visitors queued.
- End*: The computation terminates when all message

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

- 1: **INPUT**: $g \leftarrow$ input graph
 - 2: **INPUT**: $ccid_array \leftarrow$ an array holding the component id for each vertex, initialized to ∞
 - 3: $pri_q_visit \leftarrow$ a multithreaded visitor queue, prioritized by UCCVertexVisitor's cur_ccid
 - 4: **for all** $v \in g.vertex_list()$ **parallel do**
 - 5: $pri_q_visit.push(UCCVertexVisitor(g, pri_q_visit, ccid_array, v, v))$
 - 6: **end for**
 - 7: $pri_q_visit.wait()$ // wait for queued work to finish
-

Algorithm 4 Undirected Connected Components – UCCVertexVisitor

- 1: **INPUT**: $g \leftarrow$ input graph
 - 2: **INPUT**: $pri_q_visit \leftarrow$ a multithreaded visitor queue
 - 3: **INPUT**: $ccid_array \leftarrow$ an array holding the component id for each vertex
 - 4: **INPUT**: $v \leftarrow$ vertex to visit
 - 5: **INPUT**: $cur_ccid \leftarrow$ current component id for vertex
 - 6: **if** $cur_ccid < ccid_array[v]$ **then**
 - 7: $ccid_array[v] = cur_ccid$ // relax vertex information
 - 8: $adj_list \leftarrow g.adj_list(v)$
 - 9: **for all** $vj \in adj_list$ **do**
 - 10: $pri_q_visit.push(UCCVertexVisitor(g, pri_q_visit, ccid_array, vj, cur_ccid))$
 - 11: **end for**
 - 12: **end if**
-

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

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

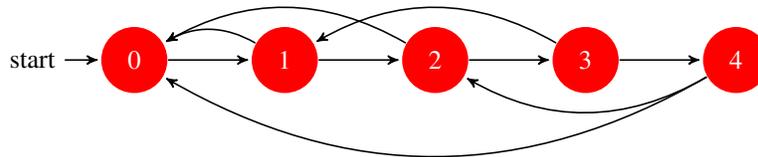


Fig. 2. An example directed graph with poor parallelism for BFS and SSSP

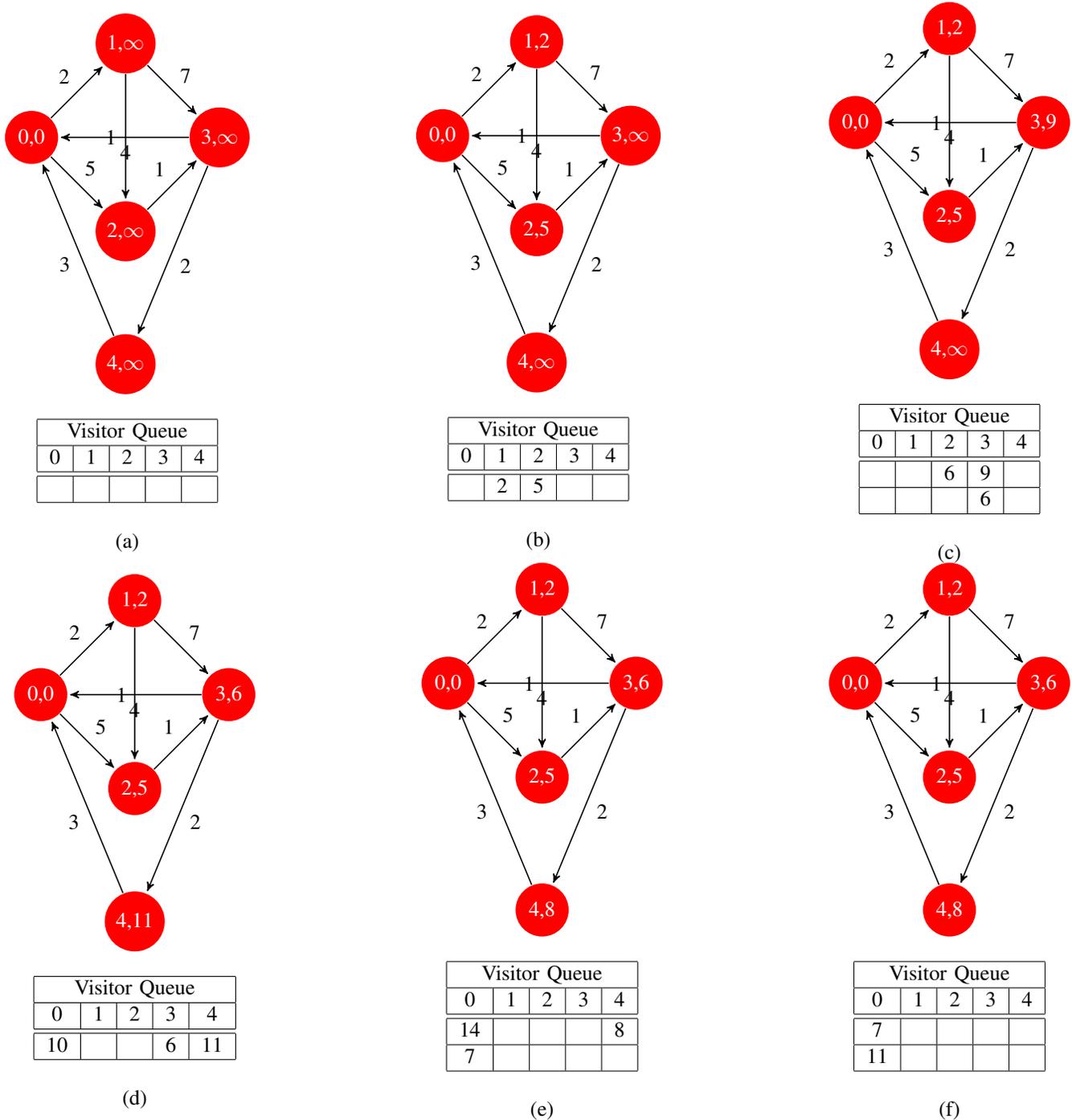


Fig. 3. An example of an asynchronous Single Source Shortest Path (SSSP) traversal of a simple weighted directed graph. Section III-B2 discusses the details of this example.

Two similar implementations were created for In-Memory and Semi-External Memory graphs.

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 and corresponding 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 [3]. For POSIX thread support, we used the Boost Thread library [11]. All code was compiled with g++ version 4.1.2 with -O3 and run on a quad core, quad processor (16 cores) AMD Opteron(tm) 8356 with 256 GB of main memory.

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. We define a semi-external graph as having enough memory to store algorithmic information about the vertices but not edges. The entire graph structure is stored on the persistent storage device, and the visitor queues and the output of the algorithm are stored in main memory.

For POSIX thread support, we used the Boost Thread library [11]. All code was compiled with g++ version 4.1.2 with -O3 and run on a dual core, quad processor (8 cores) AMD Opteron(tm) 2378 with 16 GB of main memory.

An additional difference from the in-memory 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. 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.

We experimented with 3 types of FLASH based storage:

- 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.
- Intel – 4x 80GB Intel 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.
- Cosair – 4x 128GB Corsair P128 MLC, SATA SSDs in a software RAID 0 configuration. Our 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 random 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.

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 synthetic graphs, and undirected real graphs.

1) *Synthetic Graphs*: For synthetic graphs, we used scale-free graphs generated by the RMAT [12] graph generator. The RMAT graph generator uses a ‘recursive matrix’ model to create graphs that models ‘real-world’ graphs. We generated directed graphs with unique edges ranging from $2^{25} - 2^{30}$ vertices and an average out-degree of 16. Undirected versions of these graphs for use with Connected Components were created by adding reverse edges. 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.

- ClueWeb09 [13]: To fit the vertex identifiers in a 32 bit integer space, we removed all vertices that had zero incoming and outgoing edges. The trimmed graph has 1,667,267,985 vertices and 7,939,647,897 edges.
- it-2004 [14]: 41,291,595 verts, 1,150,725,436 edges.
- sk-2005 [14]: 50,636,155 verts, 1,949,412,601 edges.
- uk-union [14]: 133,633,041 verts, 5,507,679,822 edges.
- webbase-2001 [14]: 118,142,156 verts, 1,019,903,190 edges.

B. In-Memory Experiments

This section describes our experiments traversing In-Memory graphs. For experimental comparison, we show the performance of the Multithreaded Graph Library (MTGL) [5], [15], the Parallel Boost Graph Library (PBGL) [4], the Small-world Network Analysis and Partitioning (SNAP) library [6], and the serial Boost Graph Library (BGL) [3] when available.

- MTGL [5], [15] is 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 [16] 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 2597.

- PBGL [4] is 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 1.41 of the Boost library.
- BGL [3] is a serial graph library. In our experiments, BGL is used as an efficient serial baseline to compute speedup. Our experiments use BGL from version 1.41 of the Boost library.
- SNAP [6] is a parallel graph library for shared memory which utilizes OpenMP for parallelism. Our experiments use SNAP version 0.3.

1) *Breadth First Search (BFS)*: Table I shows our asynchronous Breadth First Search (BFS) compared with MTGL, SNAP, PBGL and BGL. Our approach, MTGL, SNAP and BGL were tested on a 16-core AMD machine with 256 GB of memory. PBGL is shown with the optimal number of cores between 64-1024 and 128GB-2TB of memory on an AMD cluster with 16-cores and 32 GB per compute node.

At full parallelism, our asynchronous BFS is roughly 10-18% faster than MTGL’s and 1.5-3 times faster than SNAP’s BFS in all 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 structure are implemented using 64-bit integers and those libraries 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, 256-1024 cores with 512GB-2TB of memory, PBGL can compute BFS on the 2^{28} and 2^{29} vertex graphs the fastest. Like MTGL and SNAP, PBGL also runs out of memory with the 2^{29} and 2^{30} vertex graphs.

Overall, our approach outperforms MTGL and SNAP in all of our test cases, and performs well compared with PBGL which uses 32-64 times the number of cores and 4-8 times the memory. The benefit of thread oversubscription 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.

2) *Single Source Shortest Path (SSSP)*: Table II shows our asynchronous Single Source Shortest Path (SSSP) compared with BGL on a 16-core AMD machine with 256GB of main memory. These experiments use the same directed graphs as the BFS experiments in Section V-B1 and add two types of edges weights: uniform weights (UW) and log-uniform weights (LUW).

Again, we see the benefit of thread oversubscription as all tests perform best using 512 threads on 16 cores, which indicates that further scaling is possible beyond 16-cores. We see scalability of at least 10 on 16 cores with both edge weight types.

3) *Connected Components*: Table III shows our asynchronous Connected Components (CC) compared with MTGL, PBGL and BGL. Our approach, MTGL and BGL were tested on a 16-core AMD machine with 256GB of memory. PBGL is

graph type	# verts	# edges	% vis	# levs	BGL BFS		MTGL BFS, 16-core AMD			SNAP-graph BFS, 16-core AMD			Asynchronous BFS, 16-core AMD				PBGL BFS, AMD cluster					
					time (s)	time (s)	time (s)	16 thrds	scaling	speedup BGL	time (s)	16 thrds	scaling	speedup BGL	time (s)	16 thrds	512 thrds	scaling	speedup BGL	# cores	time (s)	speedup BGL
RMAT A	2^{25}	2^{29}	99.0%	12	52.2	57.3	12.4	4.6	4.2	45.0.3	19.1	2.4	2.7	79.8	14.6	10.7	7.5	4.9	256	24.4	2.1	
	2^{26}	2^{30}	98.9%	13	117.4	142.6	26.3	5.4	4.5	95.4	40.5	2.4	2.9	188.0	31.4	23.6	8.0	5.0	256	39.5	3.0	
	2^{27}	2^{31}	98.8%	12	292.0	363.8	60.8	6.0	4.8	231.2	90.8	2.5	3.2	455.9	71.2	55.1	8.3	5.3	512	64.0	4.6	
	2^{28}	2^{32}	98.8%	13	727.1	855.2	137.3	6.2	5.3	546.0	202.1	2.7	3.6	1073.7	155.1	122.0	8.8	6.0	512	86.8	8.4	
	2^{29}	2^{33}	98.7%	13	1630.5	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	2594.8	317.3	251.4	10.3	6.5	Out of Memory	Out of Memory	Out of Memory	Out of Memory
	2^{30}	2^{34}	98.7%	15	3742.5	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	6400.5	636.1	521.0	12.3	7.2	Out of Memory	Out of Memory	Out of Memory	Out of Memory
RMAT B	2^{25}	2^{29}	48.9%	8	34.4	37.1	8.6	4.3	4.0	27.3	23.8	1.1	1.4	51.1	9.1	7.8	6.5	4.4	256	14.7	2.3	
	2^{26}	2^{30}	47.6%	8	77.3	96.9	20.7	4.7	3.7	57.5	58.1	1.0	1.3	122.2	20.4	17.5	7.0	4.4	256	24.9	3.1	
	2^{27}	2^{31}	46.2%	8	197.2	263.2	45.8	5.7	4.3	134.0	117.4	1.1	1.7	299.8	48.4	41.8	7.2	4.7	256	43.2	4.6	
	2^{28}	2^{32}	45.0%	8	510.1	633.5	105.8	6.0	4.8	313.4	275.4	1.1	1.9	709.7	109.6	94.4	7.5	5.4	512	70.6	7.2	
	2^{29}	2^{33}	43.6%	9	1434.9	Out of Memory	Out of Memory	Out of Memory	Out of Memory	1823.4	230.1	Out of Memory	Out of Memory	1823.4	230.1	195.3	9.3	7.3	1024	169.2	8.5	
	2^{30}	2^{34}	42.7%	9	3342.5	Out of Memory	Out of Memory	Out of Memory	Out of Memory	4078.5	480.8	Out of Memory	Out of Memory	4078.5	480.8	406.8	10.0	8.2	Out of Memory	Out of Memory	Out of Memory	Out of Memory

TABLE I
PERFORMANCE COMPARISON OF IN-MEMORY BREADTH FIRST SEARCH (BFS). BEST PERFORMANCE FOR EACH TEST INSTANCE IS SHOWN IN BOLD. ‘SCALING’ SHOWS BEST PARALLEL PERFORMANCE OVER SINGLE THRD. ‘SPEEDUP BGL’ SHOWS BEST PARALLEL PERFORMANCE OVER BGL’S SERIAL VERSION.

Graph type	Weight Type	# nodes	# edges	BGL	Asynchronous SSSP, 16-core AMD				
				time (s)	time (s) 1 thread	time (s) 16 threads	time (s) 512 threads	scaling	speedup BGL
RMAT-A	UW	2^{25}	2^{29}	319.2	265.1	32.2	18.1	14.6	17.6
		2^{26}	2^{30}	764.0	600.6	69.7	41.2	14.6	18.5
		2^{27}	2^{31}	1827.4	1340.8	151.5	93.8	14.3	19.5
		2^{28}	2^{32}	5098.4	3201.1	336.1	227.2	14.1	22.4
		2^{29}	2^{33}	13467.6	7365.2	797.0	503.8	14.6	26.7
		2^{30}	2^{34}	25286.8	OOM	4598.7	821.8	n/a	30.8
	LUW	2^{25}	2^{29}	233.1	262.8	31.7	18.7	14.1	12.5
		2^{26}	2^{30}	538.5	597.7	71.9	42.9	13.9	12.6
		2^{27}	2^{31}	1242.6	1354.5	155.9	101.4	13.4	12.3
		2^{28}	2^{32}	3367.9	3153.8	345.2	229.2	13.8	14.7
		2^{29}	2^{33}	8188.9	7850.3	838.3	515.4	15.2	15.9
		2^{30}	2^{34}	15774.8	OOM	1297.7	863.0	n/a	18.3
RMAT-B	UW	2^{25}	2^{29}	185.5	129.3	16.0	12.9	10.0	14.4
		2^{26}	2^{30}	428.7	291.0	35.9	27.5	10.6	15.6
		2^{27}	2^{31}	1003.2	653.8	77.9	63.3	10.3	15.9
		2^{28}	2^{32}	2801.4	1597.0	185.0	141.2	11.3	19.8
		2^{29}	2^{33}	6142.1	3937.0	399.4	346.0	11.4	17.8
		2^{30}	2^{34}	11703.2	14734.3	1806.9	862.8	17.1	13.6
	LUW	2^{25}	2^{29}	150.7	118.1	15.5	10.3	11.4	14.6
		2^{26}	2^{30}	341.3	269.8	32.4	24.7	10.9	13.8
		2^{27}	2^{31}	785.5	632.3	77.2	54.4	11.6	14.4
		2^{28}	2^{32}	2168.7	1477.8	175.6	123.8	11.9	17.5
		2^{29}	2^{33}	5587.8	3685.3	334.6	270.8	13.6	20.6
		2^{30}	2^{34}	10642.7	6837.2	630.4	430.8	15.9	24.7

TABLE II

PERFORMANCE COMPARISON OF IN-MEMORY SINGLE SOURCE SHORTEST PATH (SSSP). BEST PERFORMANCE FOR EACH TEST INSTANCE IS SHOWN IN BOLD. ‘SCALING’ SHOWS BEST PARALLEL PERFORMANCE OVER SINGLE THREAD. ‘SPEEDUP BGL’ SHOWS BEST PARALLEL PERFORMANCE OVER BGL’S SERIAL VERSION.

shown with the optimal number of cores between 64-1024 and 128GB-2TB of memory on an AMD cluster with 16-cores and 32 GB per compute node. Our asynchronous CC is roughly 2 times faster than MTGL’s CC in all synthetic cases tested. For the real web-graphs, our asynchronous CC is 4-13 times faster than MTGL’s CC.

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.

PBGL very slightly outperforms our approach in two cases while using 32-64 times the number of cores and 4-8 times the memory.

C. Semi-External Memory Experiments

This section describes our experiments traversing Semi-External Memory (SEM) graphs stored on solid state FLASH devices on an 8-core AMD machine with 16GB of main memory. We compare performance to the serial Boost Graph Library (BGL) running In-Memory on a 16-core AMD machine with 256GB of main memory. The In-Memory BGL performance numbers are used as a baseline to evaluate the Semi-External performance.

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*: Table IV 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 a comparable performance to BGL’s in-memory performance.

2) *Connected Components*: Table V 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 [17], [18].

A popular approach to graph processing in HPC has been with the use of Distributed Memory computer clusters. Such

Semi-External Asynchronous BFS, 8-core AMD									
graph	# nodes	# edges directed	Size on EM device	FusionIO		Intel		Corsair	
				time (s) 256 threads	speedup IM BGL	time (s) 256 threads	speedup IM BGL	time (s) 256 threads	speedup IM BGL
RMAT-A	2^{27}	2^{31}	9 GB	134.4	2.2	306.1	1.0	400.5	0.7
	2^{28}	2^{32}	18 GB	352.0	2.1	578.8	1.3	876.8	0.8
	2^{29}	2^{33}	36 GB	947.1	1.7	1368.8	1.2	1852.7	0.9
	2^{30}	2^{34}	72 GB	2138.2	1.8	2805.2	1.3	3966.8	0.9
RMAT-B	2^{27}	2^{31}	9 GB	78.1	2.5	82.5	2.4	132.5	1.5
	2^{28}	2^{32}	18 GB	176.8	2.9	189.2	2.7	245.9	2.1
	2^{29}	2^{33}	36 GB	482.3	3.0	477.0	3.0	852.4	1.7
	2^{30}	2^{34}	72 GB	1199.7	2.8	1284.0	2.6	1949.8	1.7

TABLE IV

PERFORMANCE COMPARISON OF SEMI-EXTERNAL MEMORY BREADTH FIRST SEARCH (BFS) ON THREE FLASH MEMORY CONFIGURATIONS. BEST PERFORMANCE FOR EACH TEST INSTANCE IS SHOWN IN BOLD. ‘SPEEDUP BGL’ SHOWS BEST PARALLEL PERFORMANCE OVER IN-MEMORY BGL’S SERIAL VERSION. SEMI-EXTERNAL EXPERIMENTS PERFORMED ON AN 8-CORE AMD SYSTEM WITH 16 GB MEMORY.

Semi-External Asynchronous Connected Components, 8-core AMD									
graph	# nodes	# edges undirected	Size on EM device	FusionIO		Intel		Corsair	
				time (s) 256 threads	speedup IM BGL	time (s) 256 threads	speedup IM BGL	time (s) 256 threads	speedup IM BGL
RMAT-A	2^{27}	2^{31}	17 GB	174.6	2.9	245.1	2.0	225.1	2.2
	2^{28}	2^{32}	34 GB	371.7	3.9	408.1	3.6	501.4	2.9
	2^{29}	2^{33}	68 GB	947.2	3.8	1236.8	2.9	1470.9	2.5
	2^{30}	2^{34}	136 GB	2776.4	3.0	4029.3	2.1	9118.9	0.9
RMAT-B	2^{27}	2^{31}	17 GB	300.0	1.3	277.9	1.4	191.8	2.0
	2^{28}	2^{32}	34 GB	659.5	1.8	554.4	2.1	537.3	2.2
	2^{29}	2^{33}	68 GB	1673.0	1.7	2017.9	1.4	6318.7	0.4
	2^{30}	2^{34}	136 GB	4354.7	1.5	5972.6	1.1	10209.8	0.6
sk-2005	50M	3.6B	14 GB	13.2	3.7	30.9	1.6	48.3	1.0
uk-union	133M	9.4B	36 GB	126.3	1.0	119.7	1.0	134.5	0.9

TABLE V

PERFORMANCE COMPARISON OF SEMI-EXTERNAL MEMORY CONNECTED COMPONENTS (CC) ON THREE FLASH MEMORY CONFIGURATIONS. BEST PERFORMANCE FOR EACH TEST INSTANCE IS SHOWN IN BOLD. ‘SPEEDUP BGL’ SHOWS BEST PARALLEL PERFORMANCE OVER IN-MEMORY BGL’S SERIAL VERSION. SEMI-EXTERNAL EXPERIMENTS PERFORMED ON AN 8-CORE AMD SYSTEM WITH 16 GB MEMORY.

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) [19] but suffers from significant load imbalance when processing power-law graphs [4]. 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 [15]. Our approach addresses the memory latency issues using commodity hardware and storage devices (NAND Flash) that are relatively slow compared with main memory.

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) [20] 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 [21].

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 [2].

For general undirected graphs, Munagala and Ranade [22] improve the worst-case I/O of BFS to $O(n + \text{sort}(m))$ by exploiting the fact that for 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 a non-contiguous access to the adjacency lists, hence every adjacency list must be accessed separately. An improvement on the adjacency list

graph type	# verts	# edges	# CCs	BGL CC			MTGL CC, 16-core AMD			Asynchronous CC, 16-core AMD			PBGL CC, AMD cluster			
				time (s)	time (s) 1 thrd	speedup BGL	time (s) 16 thrds	scaling	speedup BGL	time (s) 16 thrds	time (s) 512 thrds	scaling	speedup BGL	# cores	time (s)	speedup BGL
RMAT-A	²²⁵	²²⁹	34,008	93.3	142.9	40.9	3.7	2.3	150.8	22.3	16.8	9.0	5.6	256	28.9	3.2
	²²⁶	²³⁰	72,647	206.6	549.5	90.3	3.7	2.3	332.3	45.6	34.2	9.7	6.0	256	48.3	4.3
	²²⁷	²³¹	154,179	498.0	1194.6	190.2	4.3	2.6	825.1	98.9	75.2	11.0	6.6	512	74.7	6.7
	²²⁸	²³²	327,072	1458.2	2209.0	331.5	6.1	4.4	2034.5	233.7	174.0	11.7	8.4	1024	177.4	8.2
	²²⁹	²³³	689,979	3618.1	Out of Memory	Out of Memory	Out of Memory	Out of Memory	4479.8	497.0	391.3	11.4	9.2	1024	378.7	9.6
	²³⁰	²³⁴	1,448,438	8281.0	Out of Memory	Out of Memory	Out of Memory	9775.4	879.1	832.3	11.7	9.9	9.9	Out of Memory	Out of Memory	Out of Memory
RMAT-B	²²⁵	²²⁹	13,739,228	68.7	121.2	35.6	3.6	1.9	127.5	18.9	15.6	8.2	4.4	256	73.9	0.9
	²²⁶	²³⁰	28,448,613	154.7	474.9	62.9	4.4	2.5	274.3	39.4	32.7	8.4	4.7	256	Error	Error
	²²⁷	²³¹	58,757,785	388.1	1043.2	170.9	3.9	2.3	660.7	86.6	68.6	9.6	5.7	256	Error	Error
	²²⁸	²³²	121,037,055	1158.6	1947.8	379.8	4.2	3.1	1582.6	199.7	167.3	9.5	6.9	256	Error	Error
	²²⁹	²³³	249,937,778	2827.4	Out of Memory	Out of Memory	Out of Memory	3714.0	437.2	363.6	10.2	7.8	7.8	256	Error	Error
	²³⁰	²³⁴	510,267,039	6433.0	Out of Memory	Out of Memory	Out of Memory	7535.9	905.1	793.3	9.5	8.1	8.1	256	Out of Memory	Out of Memory
ClueWeb09	1.7B	7.9B	3,149,668	1872.5	Out of Memory	Out of Memory	Out of Memory	4661.4	502.6	276.3	16.9	6.8	256	Out of Memory	Out of Memory	
it-2004	41M	1.2B	979	26.8	201.2	48.8	4.1	0.5	75.7	7.9	3.7	20.3	7.2	256	59.3	0.5
sk-2005	50M	1.9B	126	49.3	168.8	41.8	4.0	1.2	110.5	12.3	5.8	18.9	8.4	256	104.1	0.5
uk-union	133M	5.5B	2,097,197	120.6	905.2	114.3	7.9	1.1	303.8	31.6	14.4	21.1	8.4	256	Out of Memory	Out of Memory
webbase-2001	118M	1B	2,721,051	35.1	386.4	33.0	11.7	1.1	201.6	17.1	7.0	29.0	5.0	64	405.1	0.1

TABLE III

PERFORMANCE COMPARISON OF IN-MEMORY CONNECTED COMPONENTS (CC). BEST PERFORMANCE FOR EACH TEST INSTANCE IS SHOWN IN BOLD. ‘SCALING’ SHOWS BEST PARALLEL PERFORMANCE OVER SINGLE THRD. ‘SPEEDUP BGL’ SHOWS BEST PARALLEL PERFORMANCE OVER BGL’S SERIAL VERSION.

access was made by Mehlhorn and Meyer [23] which preprocesses the graph into subgraphs of low diameter and stores their adjacency list 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 $O((n + m/B)lg(n/B) + sort(m))$ [24]. This is considered impractical for general sparse directed graphs. For an in-depth survey of EM graph traversal algorithms, see [24].

VII. CONCLUSIONS

In this work, we present a novel asynchronous approach to graph traversal and demonstrate the approach by computing Breadth First Search (BFS), Single Source Shortest Path (SSSP), and Connected Components (CC) 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. We provide a quantitative study comparing our approach to existing implementations such as the Boost Graph Library (BGL) [3], the Parallel Boost Graph library (PBGL) [4], the Multithreaded Graph Library (MTGL) [5], and the Small-world Network Analysis and Partitioning library (SNAP) [6]. 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 10-18% faster than MTGL’s and 1.5-3 times faster than SNAP’s BFS, and our asynchronous CC is 2-13 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 scaling potential as the number of cores on a processor continues to increase.

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] 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. [Online]. Available: <http://doi.acm.org/10.1145/1109557.1109623>

- [3] *The boost graph library: user guide and reference manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [4] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," in *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.
- [5] 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.
- [6] 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.
- [7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms, 2nd edition*. MIT Press and McGraw-Hill, 2001.
- [8] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.
- [9] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02011-7_4
- [10] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo, "Characterizing the performance of flash memory storage devices and its impact on algorithm design," in *Experimental Algorithms*, 2008, pp. 208–219. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68552-4_16
- [11] "Boost threads," www.boost.org/doc/libs/release/libs/thread/.
- [12] 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.
- [13] "Clueweb09 dataset," <http://boston.lti.cs.cmu.edu/Data/clueweb09/>.
- [14] 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.
- [15] 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. [Online]. Available: <http://dx.doi.org/10.1109/IPDPS.2007.370685>
- [16] 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.
- [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. [Online]. Available: <http://dx.doi.org/10.1109/MCSE.2008.56>
- [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. [Online]. Available: <http://dx.doi.org/10.1142/S0129626407002843>
- [19] 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.
- [20] 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.
- [21] J. S. Vitter, "Algorithms and data structures for external memory," *Found. Trends Theor. Comput. Sci.*, vol. 2, no. 4, pp. 305–474, 2008. [Online]. Available: <http://dx.doi.org/10.1561/0400000014>
- [22] 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.
- [23] "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.
- [24] D. Ajwani and U. Meyer, "Design and engineering of external memory traversal algorithms for general graphs," in *Algorithmics of Large and Complex Networks*, 2009, pp. 1–33. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02094-0_1