2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Investigating Graph Algorithms in the BSP Model on the Cray XMT

David Ediger David A. Bader
Georgia Institute of Technology
Atlanta, GA, USA

***Abstract*—Implementing parallel graph algorithms in large, shared memory machines, such as the Cray XMT, can be challenging for programmers. Synchronization, deadlock, hot-spotting, and others can be barriers to obtaining linear scalability. Alternative programming models, such as the bulk synchronous parallel programming model used in Google's Pregel, have been proposed for large graph analytics. This model eases programmer complexity by eliminating deadlock and simplifying data sharing. We investigate the algorithmic effects of the BSP model for graph algorithms and compare performance and scalability with hand-tuned, open source software using GraphCT. We analyze the innermost iterations of these algorithms to understand the differences in scalability between BSP and shared memory algorithms. We show that scalable performance can be obtained with graph algorithms in the BSP model on the Cray XMT. These algorithms perform within a factor of 10 of hand-tuned C code.**

I. Introduction

Internet-scale graphs are motivating research into algorithms and systems for large-scale, data-intensive analytics. A parallel system with hundreds of thousands of threads can be a challenge to program complex analytics. Sophisticated synchronization protocols are used to avoid deadlock. Overuse of a single memory location for coordination leads to hot-spotting and sequentialization. In the search for easier parallel frameworks, the programming model can influence the execution and scalability of an algorithm in a subtle manner.

Cray XMT programmers leverage the large global shared memory and loop-level parallelism to obtain good performance and unprecedented scalability on graph algorithms. To the best of our knowledge, alternative styles of programming have not yet been investigated on the Cray XMT for these problems. In this paper, we will examine three common graph algorithms in the traditional shared memory programming model and how they can be expressed using bulk synchronous parallel programming (BSP). We will show that BSP graph algorithms can be efficiently implemented on the Cray XMT with scalable performance.

II. Background

Real world networks challenge modern computing in several ways. These graphs are often "small-world" [1] networks with small diameters and skewed degree distributions. All reachable vertices are found in a small number of hops. A highly skewed degree distribution, where most vertices have a few neighbors and several vertices have many neighbors, is challenging to parallelize. A parallel runtime system must be able to handle dynamic, fine-grained parallelism among hundreds of thousands of lightweight threads. A breadth-first search from a single vertex quickly touches the entire graph, with the size of the frontier (and parallelism) varying greatly from level to level.

The Cray XMT [2] is a supercomputing platform designed to accelerate massive graph analysis codes. The architecture tolerates high memory latencies using massive hardware multithreading. Fine-grained synchronization constructs are supported through full-empty bits as well as atomic fetch-and-add instructions. A large, fully shared memory enables the analysis of graphs on the order of one billion vertices using a well-understood programming model.

Each Threadstorm processor within a Cray XMT contains 128 *hardware streams*. Streams may block temporarily while waiting for a long-latency instruction, such as a memory request, to return. The processor will execute one instruction per cycle from hardware streams that have instructions ready to execute. The Cray XMT does not require locality to obtain good performance. Instead, latency to memory is tolerated entirely by hardware multithreading, making this machine a good candidate for memory-intensive codes like those found in graph analysis.

The Cray XMT located at Pacific Northwest National Lab contains 128 Threadstorm processors running at 500 MHz. These 128 processors support over 12 thousand hardware thread contexts. The globally addressable shared memory totals 1 TiB. Memory addresses are hashed globally to break up locality and reduce hot-spotting.

A global shared memory provides an easy programming model for the graph application developer. Due to the "small-world" nature of the graphs of interest, finding a balanced partition of vertices or edges across multiple distinct memories can be difficult. The Cray XMT has proved useful in a number of irregular applications including string matching [3], document clustering [4], triangle counting [5], hash tables [6], static graph analysis [7], [8], [9], [10], [11] and streaming graphs [12], [13].

GraphCT [14] is a framework for developing parallel and scalable static graph algorithms on multithreaded platforms. It is designed to enable a workflow of graph analysis algorithms

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.107

to be developed through a series of function calls. Graph kernels utilize a single, efficient graph data representation that is stored in main memory and served read-only to analysis applications. Graph data-file input and output, utility functions such as subgraph extraction, and many popular algorithms are provided. These algorithms include clustering coefficients, connected components, betweenness centrality, k-core, and others, from which workflows can easily be developed. GraphCT supports weighted and unweighted graphs with directed or undirected edges. It is freely available as open source, and can be built on the Cray XMT with XMT-C or on a commodity workstation using OpenMP. Our experiments build on top of GraphCT's existing capabilities.

Pregel is a distributed graph processing system with a C++ API developed by Google [15]. To avoid issues of deadlock and data races, Pregel uses a bulk synchronous parallel programming style. A graph computation is broken up into a sequence of iterations. In each iteration, a vertex is able to 1) receive messages from the previous iteration, 2) do local computation or modify the graph, and 3) send messages to vertices that will be received in the next iteration. Similar to MapReduce in many ways, chains of iterations are used to solve a graph query in a fault-tolerant manner across hundreds or thousands of distributed workstations. Unlike MapReduce, however, vertices in Pregel can maintain state between iterations, reducing the communication cost. Apache Giraph is an open source project implementing a Pregel-like environment for graph algorithms on top of Apache Hadoop [16].

To support this model of computation in a distributed cluster environment, Pregel assigns vertices to machines along with all of their incident edges. A common operation is for a vertex to send a message to its neighbors, which are readily available, but the model allows for sending messages to any vertex that is known by the sender through other means. By default, the assignment of vertex to machine is based on a random hash function yielding a uniform distribution of the vertices. Real-world graphs, however, have the scale-free property. In this case, the distribution of edges will be uneven with one or several machines acquiring high-degree vertices, and therefore a disproportionate share of the messaging activity.

We will consider the bulk synchronous parallel (BSP) style of programming, rather than the BSP computation model or BSPlib [17]. Bulk synchronous parallel programming is popular in the scientific community and is the basis for many large-scale parallel applications that run on today's supercomputers. An application is composed of a number of *supersteps* that are run iteratively. Each superstep consists of three phases. In the first phase, compute nodes process incoming messages from the previous superstep. In the second phase, nodes compute local values. In the third phase, nodes send messages to other nodes that will be received in the next superstep. The restriction that messages must cross superstep boundaries ensures that the implementation will be deadlock-free.

Applying bulk synchronous parallel programming to graph analytics is straightforward. Each vertex becomes a first-class citizen and an independent actor. The vertex implicitly knows its neighbors (they do not have to be read from disk each iteration). Each vertex is permitted to maintain state between supersteps. A vertex can send a message to one or all of its neighbors or to any other vertex that it can identify (such as through a message that it has received). If a vertex has no messages to send or nothing to compute, it can vote to stop the computation, and will not be re-activated until a future superstep in which it has messages to receive.

In the following sections, we will compare and contrast popular graph algorithms in the shared memory and BSP models. We will present performance results of connected components, breadth-first search, and triangle counting algorithms on a 128-processor Cray XMT. Shared memory algorithms from GraphCT will be examined as the baseline. We implemented BSP variants of these graph algorithms with GraphCT in order to obtain a comparison with fewer variables. Our results will show that graph algorithms are relatively easy to express in the BSP model and can be parallelized easily on the Cray XMT. Performance is within a factor of 10 of the hand-tuned C code.

III. Connected Components

Connected components is a reachability algorithm that labels all vertices in a connected component with the same label. An algorithm for connected components in the BSP model is shown in Algorithm 1. Each active vertex will execute this algorithm for each superstep. For this algorithm, the vertex state will store the component label of the component to which each vertex belongs. In the first superstep, each vertex will set its state (label) to be itself, or each vertex will begin by belonging to its own component, as in the Shiloach-Vishkin approach [18]. Each vertex then sends its component label to all neighbors.

In each subsequent superstep, all active vertices (those vertices with at least 1 waiting message) will receive their incoming messages and check each one to see if it contains a component label that is smaller than the current state. If it finds such a new component label, it will update the current state and send the new component label to all of its neighbors, to be received in the next superstep. When all vertices have found no changes and have voted to stop the computation, the algorithm terminates with the correct vertex-component mapping.

The shared memory algorithm in GraphCT, based on Shiloach-Vishkin, considers all edges in all iterations. When a new component label is found, the label is updated and available to be read by other threads. In this way, new component labels can propagate the graph within an iteration. Label propagation in shared memory decreases the number of iterations required, as seen in Figure 1.

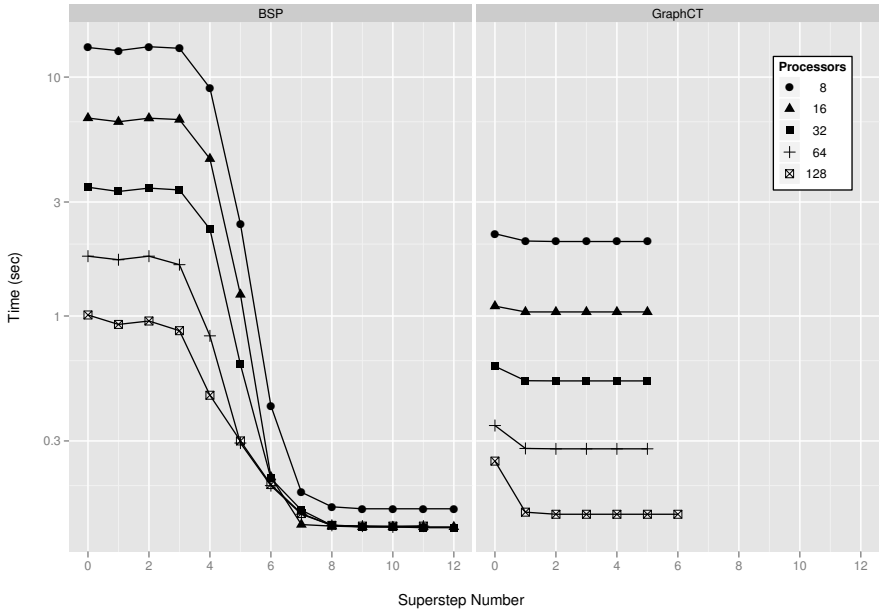

Fig. 1. Connected components execution time by iteration for an undirected, scale-free graph with 16 million vertices and 268 million edges. On the 128-processor Cray XMT, BSP execution time is 5.40 seconds and GraphCT execution time is 1.31 seconds.

Algorithm 1 A parallel algorithm for connected components in the BSP model

Input: Superstep s, Vertex v, Current label L, Set of Messages M_v
Output: Outgoing message set M'_v

```
1:  Vote ← 0
2:  for all m ∈ M_v do
3:      if m < L then
4:          L ← m
5:          Vote ← 1
6:  if s = 0 then
7:      L ← min(M_v)
8:      for all n ∈ Neighbors(v) do
9:          Send L to n
10: else
11:     if Vote = 1 then
12:         for all n ∈ Neighbors(v) do
13:             Send L to n
```

Figure 1 plots the execution time on the Cray XMT for each iteration of connected components. The input graph is an undirected, scale-free RMAT [19] graph with 16 million vertices and 268 million edges. On the left, the BSP algorithm completes in 13 iterations. In the first four iterations, almost all vertices are active, sending and receiving label updates. As the labels begin to converge, the number of active vertices, and execution time, drops significantly. In the last six iterations, only a small fraction of the graph is active.

On the right, the shared memory algorithm in GraphCT completes in 6 iterations. The amount of work per iteration is constant, and the execution time reflects this. Label propagation early in the algorithm reduces the number of iterations compared to the BSP algorithm, which uses four iterations to resolve the majority of the graph. In the shared memory algorithm, most labels are fixed by the end of the first iteration.

Each line in Figure 1 plots performance for a different Cray XMT processor count. Time is on a log scale and the number of processors doubles with each line. Even vertical spacing between points indicates linear scaling for a given iteration. All iterations of the shared memory algorithm in GraphCT demonstrate linear scaling. In the BSP algorithm, the first iterations that involve the entire vertex set also have linear scaling. As the number of active vertices becomes small, the parallelism that can be exposed also becomes small and scalability reduces significantly.

The total time to compute connected components on a 128-processor Cray XMT using GraphCT is 1.31 seconds. The time to compute connected components using the BSP algorithm on the Cray XMT is 5.40 seconds.

In a recent presentation, Sebastian Schelter discussed the application of Apache Giraph on real-world networks [20]. Giraph is an open-source alternative to Pregel that executes vertex-centric graph algorithms in a BSP style on distributed systems. On a graph from Wikipedia containing 6 million vertices and 200 million edges, in-memory processing using Giraph was 10 times faster than Hadoop MapReduce. The test cluster contains 6 compute nodes, each having two 8-

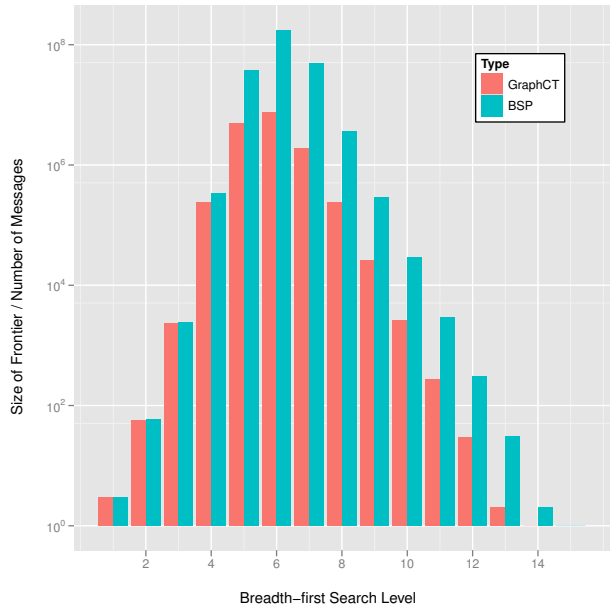

Fig. 2. Size of the breadth-first search frontier (red) or number of messages generated (green) by the BSP superstep.

Algorithm 2 A parallel algorithm for breadth-first search in the BSP model

Input: Superstep s, Vertex v, Current distance D, Set of Messages M_v
Output: Outgoing message set M'_v

```
1:  Vote ← 0
2:  for all m ∈ M_v do
3:      if m + 1 < D then
4:          D ← m + 1
5:          Vote ← 1
6:  if s = 0 then
7:      if D = 0 then
8:          Vote ← 1
9:          for all n ∈ Neighbors(v) do
10:             Send D to n
11: else
12:     if Vote = 1 then
13:         for all n ∈ Neighbors(v) do
14:             Send L to n
```

core AMD Opteron processors and 32 GiB main memory for an aggregate memory of 192 GiB. Connected components on this system runs in approximately 4 seconds. Schelter also notes that the connected components algorithm on Wikipedia requires 12 supersteps to converge with steps 6 to 12 running several orders of magnitude faster than 1 through 5.

IV. Breadth-first Search

Breadth-first search, the classical graph traversal algorithm, is used in the Graph500 benchmark [21]. In the BSP algorithm, the vertex state stores the current distance from the source vertex. In the first superstep, the source vertex sets it distance to zero, and then sends its distance to all of its neighbors. All other vertices begin with a distance of infinity. Algorithm 2 gives a description in the BSP model. Each active vertex will execute this algorithm for each superstep.

In each subsequent superstep, all vertices that are potentially on the frontier will receive messages from the previous superstep and will become active. If the current distance is infinity, the vertex will process the incoming messages and set the distance appropriately. The vertex then sends its new distance to all of its neighbors, and votes to stop the computation.

The computation continues to iterate as long as vertices have a distance that is infinity. Once all vertices have computed their distance, the computation can terminate.

The BSP algorithm closely mimics the parallel level-synchronous, shared memory breadth-first search algorithm [22] with the exception that messages are sent to vertices that *may be* on the frontier, while the shared memory algorithm enqueues only those vertices that are definitively unmarked and on the frontier. The level-synchronous shared memory breadth-first search algorithm is the comparison in GraphCT in Figures 2 and 3.

In contrast, the number of messages generated by each BSP superstep is plotted in Figure 2. In the BSP algorithm, a message is generated for every neighbor of a vertex on the frontier, or alternatively every edge incident on the frontier. Initially, almost every neighbor of the frontier is on the next frontier, and the number of messages is approximately equal to the size of the frontier. As the majority of the graph is found, messages are generated to vertices that have already been touched. The number of messages from superstep four to the end is an order of magnitude larger than the real frontier. However, the number of messages does decline exponentially.

Scalability is directly related to parallelism, which is itself related to the size of the frontier. In Figure 2, the frontier begins small, grows quickly, reaches an apex in iteration 6, and then contracts. Figure 3 plots the scalability of iterations 3 through 8. The early and late iterations show flat scaling and are omitted. On the right, the GraphCT breadth-first search has flat scalability in levels 3 and 4. Levels 5 and 8 have good scalability to 64 processors, but reduce at 128, implying contention among threads. Levels 6 and 7 have linear scalability, which matches the apex of the frontier curve.

Likewise, the BSP scalability on the left of the figure shows flat scaling in levels 3 and 8. Levels 4 and 8 have good scalability at 32 processors, but tails off. Levels 5, 6 and 7 have almost linear scalability. Because the number of messages generated is an order of magnitude larger than the

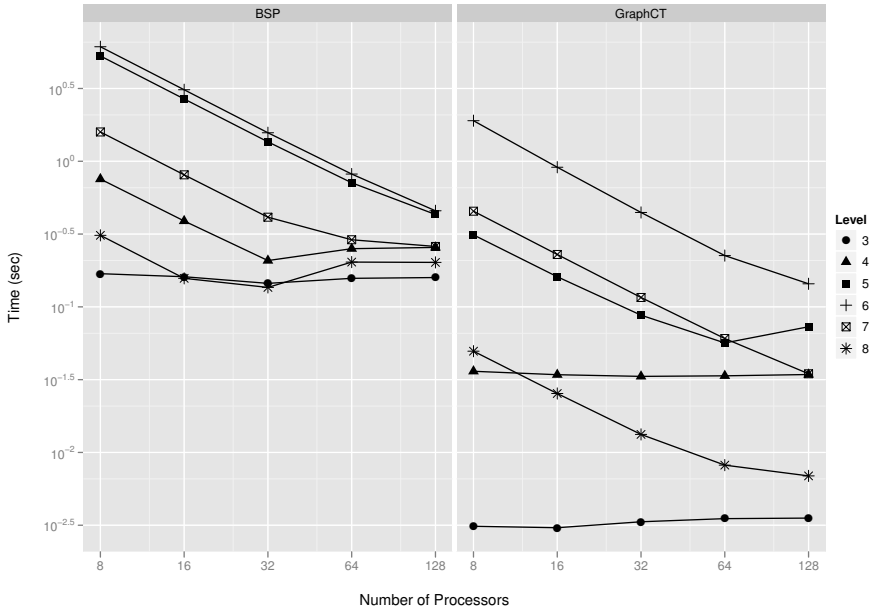

Fig. 3. Scalability of breadth-first search levels 3 to 8 on an undirected, scale-free graph with 16 million vertices and 268 million edges. Total execution time on the 128-processor Cray XMT is 3.12 seconds for BSP and 310 milliseconds for GraphCT.

size of the frontier, the contention on the message queue is higher than in the GraphCT breadth-first search and scalability is reduced as a result. Overall, the innermost BSP levels have similar execution times and scalability as the shared memory algorithm. However, the overhead of the early and late iterations is two orders of magnitude larger.

The total time to compute a single breadth-first search on a 128-processor Cray XMT using GraphCT is 310 milliseconds. The time to compute a breadth-first search from the same vertex using the BSP algorithm on the Cray XMT is 3.12 seconds. The input graph is an RMAT graph with 16 million vertices and 268 million edges.

Kajdanowicz *et al.* computes Single Source Shortest Path on a graph derived from Twitter with 43.7 million vertices and 688 million edges. On a cluster with 60 machines, Giraph completes in the algorithm in an average of approximately 30 seconds [23]. Scalability is flat from 30 to 85 machines.

Microsoft Research implements a BSP-style, vertex-centric graph analysis framework called the Trinity Graph Engine [24]. Trinity is also a distributed, in-memory graph engine. In this technical report, researchers describe performance varying number of edges and number of machines in the cluster. On an RMAT graph with 512 million vertices and approximately 6.6 billion edges, breadth-first search completes in approximately 400 seconds on 14 machines.

V. Triangle Counting

The computationally challenging aspect of computing the clustering coefficients of a graph is counting the number of triangles. In a shared memory model, this is the intersection of each vertex's neighbor list with the neighbor list of each of its neighbors, for all vertices in the graph.

The BSP algorithm takes a different approach. First, a total ordering on the vertices is established such that $V = v_1, v_2, v_3, \ldots v_N$. We define a triangle as a triple of vertices $\langle v_i, v_j, v_k \rangle$ such that $i < j < k$. We will count each triangle exactly once. The algorithm details are given in Algorithm 3. Each active vertex will execute this algorithm for each superstep.

In the first superstep, all vertices send a message to all neighbors whose vertex ID is greater than theirs. In the second superstep, for each message received, the message is retransmitted to all neighbors whose vertex ID is greater than the vertex that received the message. In the final step, if a vertex receives a message that originated at one of its neighbors, a triangle has been found. A message can be sent to itself or to another vertex to indicate that a triangle has been found.

Although this algorithm is easy to express in the model, the number of messages generated is much larger than the number of edges in the graph. This has practical implications for implementing such an algorithm on a real machine architecture.

We calculate the clustering coefficients of an undirected, scale-free graph with 16 million vertices and 268 million edges. This algorithm is not iterative. In the shared memory

Algorithm 3 A parallel algorithm for triangle counting in the BSP model

Input: Superstep s, Vertex v, Set of Messages M_v
Output: Outgoing message set M'_v

```
1: if s = 0 then
2:     for all n ∈ Neighbors(v) do
3:         if v < n then
4:             Send v to n
5: if s = 1 then
6:     for all m ∈ M_v do
7:         for all n ∈ Neighbors(v) do
8:             if m < v < n then
9:                 Send m to n
10: if s = 2 then
11:    for all m ∈ M_v do
12:        if m ∈ Neighbors(v) then
13:            Send m to m
```

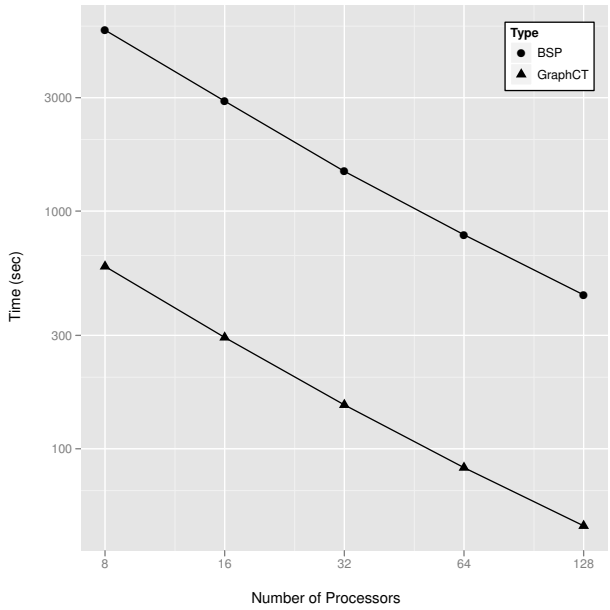

Fig. 4. Scalability of triangle counting algorithms on an undirected, scale-free graph with 16 million vertices and 268 million edges. Execution time on the 128-processor Cray XMT is 444 seconds for BSP and 47.4 seconds for GraphCT.

GraphCT implementation of triangle counting, the algorithm is expressed as a triply-nested loop. The outer loop iterates over all vertices. The middle loop iterates over all neighbors of a vertex. The inner-most loop iterates over all neighbors of the neighbors of a vertex.

The BSP algorithm replaces the triply-nested loop with three supersteps. The first two supersteps enumerate all possible triangles (restricted by a total ordering). The third and final superstep completes the neighborhood intersection and enumerates only the actual triangles that are found in the graph.

Both algorithms perform the same number of reads to the graph. The BSP algorithm must emit all the possible triangles as messages in the second superstep. For the graph under consideration, this results in almost 5.5 billion messages generated. In the last superstep, we find that these 5.5 billion possible triangles yield only 30.9 million actual triangles. It is worth noting that the RMAT graph under consideration contains far fewer triangles than a real-world graph. The number of intermediate messages will grow quickly with a higher triangle density.

The shared memory implementation, on the other hand, only produces a write when a triangle is detected. The total number of writes is 30.9 million, compared with 5.6 billion for the BSP implementation. The BSP clustering coefficient implementation produces 181 times as many writes as the shared memory implementation.

Figure 4 plots execution time and scalability of the GraphCT and BSP triangle counting algorithms on a 128-processor Cray XMT. The BSP implementation scales linearly and completes in 444 seconds on 128 processors. The shared memory implementation completes in 47.4 seconds on 128 processors.

TABLE I
EXECUTION TIMES ON A 128-PROCESSOR CRAY XMT FOR AN UNDIRECTED, SCALE-FREE GRAPH WITH 16 MILLION VERTICES AND 268 MILLION EDGES.

	Time (sec.)		
Algorithm	BSP	GraphCT	Ratio
Connected Components	5.40	1.31	4.1:1
Breadth-first Search	3.12	0.31	10.0:1
Triangle Counting	444	47.4	9.4:1

VI. DISCUSSION

By implementing BSP in a C-language environment on the same shared memory platform on which we conduct our GraphCT experiments, we can observe the algorithmic differences imposed by the bulk synchronous parallel programming model. Table I compares the total execution times for each algorithm on the Cray XMT. The Cray XMT enables scalable, parallel implementations of graph algorithms in both programming models. In some cases, such as connected components, the scalability and execution profiles are quite different for the same algorithm. In others, such as breadth-first search and triangle counting, the main execution differences are in the overheads, both memory and time.

Breadth-first search is the BSP algorithm that bears the most resemblance to its shared memory counterpart. Both operate in an iterative, synchronous fashion. The only real difference lies in how the frontier is expressed. The shared memory algorithm

only places vertices on the frontier if they are undiscovered, and only places one copy of each vertex. The BSP algorithm does not have this knowledge, so it must send messages to every vertex that could possibly be on the frontier. Those that are not will discard the messages. As a result, the two algorithms perform very similarly. In fact, many of the fastest performing Graph500 [21] entries on large, distributed clusters perform the breadth-first search in a bulk synchronous fashion with varying 1-D and 2-D decompositions [25].

In the connected components algorithms, we observe different behavior. Since messages in the BSP model cannot arrive until the next superstep, vertices processing in the current superstep are processing on stale data. Because data cannot move forward in the computation, the number of iterations required until convergence is at least a factor of two larger than in the shared memory model. In the shared memory algorithm, once a vertex discovers its label has changed, that new information is available to all of its neighbors immediately and can be further consumed. While the shared memory algorithm requires edges and vertices to be read and processed that will not change, the significantly lower number of iterations results in a significantly shorter execution time.

In the clustering coefficients algorithms, we observe very similar behavior in reading the graph. Each vertex is considered independently and a doubly-nested loop of the neighbor set is required (although the exact mechanisms of performing the neighbor intersection can be varied–see [12]). The most significant difference between the algorithms is the nature of the possible triangles. In the shared memory algorithm, the possible triangles are implicit in the loop body. In the BSP algorithm, the possible triangles must be explicitly enumerated as messages. The result is an overwhelming number of writes that must take place. Despite 181 times greater number of writes, the Cray XMT only experiences a 9.4x slow down in execution time when performing the BSP algorithm.

VII. Conclusion and Future Work

The global shared address space on the Cray XMT is advantageous for prototyping alternative programming environments for large graph analysis. Expressing popular static graph algorithms is straightforward in the bulk synchronous parallel model, similar to Google's Pregel framework. The Cray XMT compiler is able to automatically parallelize BSP iterations, vertex messaging, and neighbor list traversal. Without native support for message features such as enqueueing and dequeueing, serialization around a single atomic fetch-and-add is possible, inhibiting scalability.

We peered within the innermost iterations of graph algorithms and observed differences in parallelism from iteration to iteration. We demonstrated linear scalability to 128 processors on connected components, breadth-first search, and triangle counting using the bulk synchronous parallel model. Performance was within a factor of 10 of hand-tuned C code. Connected components was limited by the number of iterations requiring every vertex to be active. Breadth-first search was only limited by processing on vertices that had already been processed. Triangle counting was limited by message overhead.

Despite these factors, the scalability demonstrated in these algorithms indicates a promising area of study for parallel graph algorithms on large, shared memory platforms.

Acknowledgments

This work was supported in part by the Pacific Northwest National Lab (PNNL) Center for Adaptive Supercomputing Software for MultiThreaded Architectures (CASS-MT). We thank PNNL and Cray for providing access to Cray XMT systems.

References

[1] D. Watts and S. Strogatz, "Collective dynamics of small world networks," *Nature*, vol. 393, pp. 440–442, 1998.

[2] P. Konecny, "Introducing the Cray XMT," in *Proc. Cray User Group meeting (CUG 2007)*. Seattle, WA: CUG Proceedings, May 2007.

[3] O. Villa, D. Chavarria-Miranda, and K. Maschhoff, "Input-independent, scalable and fast string matching on the Cray XMT," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–12.

[4] J. Mogill and D. Haglin, "Toward parallel document clustering," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 1700–1709.

[5] G. Chin, A. Marquez, S. Choudhury, and K. Maschhoff, "Implementing and evaluating multithreaded triad census algorithms on the Cray XMT," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–9.

[6] E. Goodman, D. Haglin, C. Scherrer, D. Chavarria-Miranda, J. Mogill, and J. Feo, "Hashing strategies for the Cray XMT," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–8.

[7] J. Shuangshuang, H. Zhenyu, C. Yousu, D. Chavarria-Miranda, J. Feo, and W. Pak Chung, "A novel application of parallel betweenness centrality to power grid contingency analysis," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1–7.

[8] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. Farber, and W. N. Reynolds, "Massive social network analysis: Mining twitter for social good," *Parallel Processing, International Conference on*, pp. 583–593, 2010.

[9] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *9th International Conference on Parallel Processing and Applied Mathematics (PPAM11)*. Springer, September 2011.

[10] K. Jiang, D. Ediger, and D. A. Bader, "Generalizing k-Betweenness centrality using short paths and a parallel multithreaded implementation," in *The 38th International Conference on Parallel Processing (ICPP 2009)*, Vienna, Austria, September 2009.

[11] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarría-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP'09)*, Rome, Italy, May 2009.

[12] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *Workshop on Multithreaded Architectures and Applications (MTAAP)*, Atlanta, Georgia, April 2010.

[13] D. Ediger, E. J. Riedy, D. A. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks," in *5th Workshop on Multithreaded Architectures and Applications (MTAAP)*, May 2011.

[14] D. Ediger, K. Jiang, J. Riedy, and D. Bader, "Graphct: Multithreaded algorithms for massive graph analysis," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2012.

[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[16] A. Ching and C. Kunz, "Apache giraph," 2012. [Online]. Available: http://incubator.apache.org/giraph/

[17] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, August 1990.

[18] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *J. Algs.*, vol. 3, no. 1, pp. 57–67, 1982.

[19] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM)*. Orlando, FL: SIAM, April 2004.

[20] S. Schelter, "Large scale graph processing with apache giraph," May 2012, invited talk, GameDuell Berlin.

[21] "Graph 500," 2012. [Online]. Available: http://www.graph500.org

[22] D. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*. Columbus, OH: IEEE Computer Society, August 2006.

[23] T. Kajdanowicz, W. Indyk, P. Kazienko, and J. Kukul, "Comparison of the efficiency of mapreduce and bulk synchronous parallel approaches to large network processing," in *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*, December 2012, pp. 218–225.

[24] B. Shao, H. Wang, and Y. Li, "The trinity graph engine," Microsoft Research, Tech. Rep. 161291, 2012.

[25] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 65:1–65:12.