Designing Multithreaded Algorithms for Breadth-First Search and *st*-connectivity on the Cray MTA-2

David A. Bader Kamesh Madduri
College of Computing
Georgia Institute of Technology

February 26, 2006

Abstract

Graph abstractions are extensively used to understand and solve challenging computational problems in various scientific and engineering domains. They have particularly gained prominence in recent years for applications involving large-scale networks. In this paper, we present fast parallel implementations of three fundamental graph theory problems, Breadth-First Search, *st*-connectivity and shortest paths for unweighted graphs, on multithreaded architectures such as the Cray MTA-2. The architectural features of the MTA-2 aid the design of simple, scalable and high-performance graph algorithms. We test our implementations on large scale-free and sparse random graph instances, and report impressive results, both for algorithm execution time and parallel performance. For instance, Breadth-First Search on a scale-free graph of 200 million vertices and 1 billion edges takes less than 5 seconds on a 40-processor MTA-2 system with an absolute speedup of close to 30. This is a significant result in parallel computing, as prior implementations of parallel graph algorithms report very limited or no speedup on irregular and sparse graphs, when compared to the best sequential implementation.

1 Introduction

Graph theory concepts are widely applied in many traditional and emerging scientific disciplines such as VLSI Design, Combinatorial Optimization, Databases, and Computational Biology. Some examples include phylogeny reconstruction [36, 35], protein-protein interaction networks [42], placement and layout in VLSI chips [30], data mining [23, 25], and

clustering in semantic webs. Graph abstractions are also finding increasing relevance in the relatively new domain of large-scale and social network analysis [13, 29]. Empirical studies show that many social and economic interactions tend to organize themselves in complex network structures. These networks may contain billions of vertices with degrees ranging from small constants to thousands [7, 19]. The Internet and other communication networks, transportation and power distribution networks also share this property. The two key characteristics studied in these networks are *centrality* (which nodes in the graph are best connected to others, or have the most influence) and *connectivity* (how nodes are connected to one another). Popular metrics for analyzing these networks, like betweenness centrality [20, 9], are computed using fundamental graph algorithms like Breadth-First Search (BFS) and shortest paths.

In recognition of the importance of graph abstractions for solving large-scale problems on High Performance Computing (HPC) systems, several communities have proposed graph theoretic computational challenges. For instance, the recently announced 9th DIMACS Implementation Challenge [17] is targeted at finding shortest paths in graphs. The DARPA High Productivity Computer Systems (HPCS) [16] program has developed a synthetic graph theory benchmark called SSCA#2 [26, 27] which is composed of four kernels operating on a large-scale, directed multi-graph. (We describe our implementation of SSCA#2 on symmetric multiprocessors in [6])

Graph theoretic problems are typically memory intensive, and the memory accesses are fine-grained and highly irregular. This leads to poor performance on cache-based systems. On distributed memory clusters, few parallel graph algorithms outperform their best sequen-

tial implementations due to long memory latencies and high synchronization costs. Parallel shared memory systems are a more supportive platform. They offer higher memory bandwidth and lower latency than clusters, as the global shared memory avoids the overhead of message passing. However, parallelism is dependent on the cache performance of the algorithm and scalability is limited in most cases. While it may be possible to improve the cache performance to a certain degree for some classes of graphs, there are no known general techniques for cache optimization because the memory access pattern is largely dependent on the structure of the graph.

1.1 Preliminaries

The Cray MTA-2 is a high-end shared memory system offering two unique features that aid considerably in the design of irregular algorithms: fine-grained parallelism and zero-overhead synchronization. The MTA-2 has no data cache; rather than using a memory hierarchy to hide latency, the MTA-2 processors use hardware multithreading to tolerate the latency. The low-overhead synchronization support complements multithreading and makes performance primarily a function of parallelism. Since graph algorithms often have an abundance of parallelism, these architectural features lead to superior performance and scalability.

The computational model for the MTA-2 is thread-centric, not processor-centric. A thread is a logical entity comprised of a sequence of instructions that are nominally issued in order, respecting jumps and skips. It has a finite state, which is defined at any given point by the values in its registers and its program counter. At any point in time, an executing program will include one or more threads. No thread is bound to any particular processor.

System memory size and the inherent degree of parallelism within the program are the only limits on the number of threads used by a program.

Synchronization among threads within an executing program is easy and efficient because of special hardware support. Each 64-bit word of memory also has an associated *full/empty bit* which can be used to synchronize load and store operations. A synchronous load or store operation retries until it succeeds or traps. The thread that issued the load or store remains blocked until the operation completes, but the processor that issued the operation continues to issue instructions from non-blocked streams.

BFS [14] is one of the basic paradigms for the design of efficient graph algorithms. Given a graph $G = (V, E)$ (m edges and n vertices) and a distinguished source vertex s, BFS systematically explores the edges of G to *discover* every vertex that is reachable from s. It computes the *distance* (smallest number of edges) from s to each reachable vertex. It also produces a *breadth-first tree* with root s that contains all the reachable vertices. All vertices at a distance k (or *level* k) are first visited, before discovering any vertices at distance $k+1$. The *BFS frontier* is defined as the set of vertices in the current level. Breadth-First Search works on both undirected and directed graphs. A queue-based sequential algorithm runs in optimal $O(m+n)$ time.

st-connectivity is a related problem, also applicable to both directed and undirected graphs. Given two vertices s and t, the problem is to decide whether or not they are connected, and determine the shortest path between them, if one exists. It is a basic building block for more complex graph algorithms, has linear time complexity, and is complete for the class SL of problems solvable by symmetric, non-deterministic, log-space computations

[31].

Here, we present fast parallel algorithms for Breadth-First Search and *st*-connectivity, for directed and undirected graphs, on the MTA-2. We extend these algorithms to compute single-source shortest paths, assuming unit-weight edges. The implementations are tested on four different classes of graphs – random graphs generated based on the Erdős-Rényi model, scale-free graphs, synthetic sparse random graphs that are hard cases for parallelization, and SSCA#2 benchmark graphs. We also outline a parallel implementation of BFS for handling high-diameter graphs.

1.2 Related Work

Distributed BFS [2, 37, 43] and *st*-connectivity [8, 21] are both well-studied problems, with related work on graph partitioning and load balancing schemes [3, 40] to facilitate efficient implementations. Other problems and algorithms of interest include shortest paths variants [18, 12, 39, 38, 33, 15] and external memory algorithms and data structures [1, 10, 32] for BFS. However, there are very few parallel implementations that achieve significant parallel speedup on sparse, irregular graphs when compared against the best sequential implementations. In [5], we demonstrated superior performance for list ranking and connected components on the MTA-2 when compared with symmetric multiprocessor implementations and attained considerable absolute speedups over the best sequential implementations. This work serves as the primary motivation for our current experimentation on the MTA-2.

Input: $G(V, E)$, source vertex s

Output: Array $d[1..n]$ with $d[v]$ holding the length of the shortest path from s to $v \in V$, assuming unit-weight edges

```
1  for all v ∈ V in parallel do
2      d[v] ← −1;
3  d[s] ← 0;
4  Q ← φ;
5  Enqueue s ← Q;
6  while Q ≠ φ do
7      for all u ∈ Q in parallel do
8          Delete u ← Q;
9          for each v adjacent to u in parallel do
10             if d[v] = −1 then
11                 d[v] ← d[u] + 1;
12                 Enqueue v ← Q;
```

Algorithm 1: Level-synchronized Parallel BFS

2 A Multithreaded Approach to Breadth-First Search

Unlike prior parallel approaches to BFS, on the MTA-2 we do not consider load balancing or the use of distributed queues for parallelizing BFS. We employ a simple level-synchronized parallel algorithm (Alg. 1) that exploits concurrency at two key steps in BFS:

1. All vertices at a given *level* in the graph can be processed simultaneously, instead of just picking the vertex at the head of the queue (step 7 in Alg. 1)

2. The adjacencies of each vertex can be inspected in parallel (step 9 in Alg. 1).

We maintain an array d to indicate the level (or distance) of each visited vertex and process the global queue Q accordingly. Alg. 1 is however a very high-level representation, and hides the fact that thread-safe parallel insertions to the queue and atomic updates of the distance array d are needed to ensure correctness. Alg. 2 details the MTA-2 code required to achieve this (for the critical steps 7 to 12), which is simple and very concise. The loops

```
/* While the Queue is not empty */
#pragma mta assert parallel
#pragma mta loop future
for (i = startIndex; i < endIndex; i++)
    u = Q[i];
    /* Inspect all vertices adjacent to u */
    #pragma mta assert parallel
    for (j = 0; j < degree[u]; j++)
        v = neighbor[u][j];
        /* Check if v has been visited yet? */
        dist = readfe(&d[v]);
        if (dist == -1)
            writeef(&d[v], d[u] + 1);
        else
            writeef(&d[v], dist);
        /* Enqueue v */
        Q[int_fetch_add(&count, 1)] = v;
```

Algorithm 2: MTA-2 parallel C code for steps 7-12 in Alg. 1

will not be automatically parallelized as there are dependencies involved. The compiler can be forced to parallelize them using the *assert parallel* directive on both the loops. We then note that we have to handle and exploit the nested parallelism in this case. We can explicitly indicate that the iterations of the outer loop can be handled concurrently, and the compiler will dynamically schedule threads for the inner loop. We do this using the compiler directive *loop future* (see Alg. 2) to indicate that the iterations of the outer loop can be concurrently processed.

We use the low-overhead synchronization calls `int_fetch_add`, `readfe()`, and `writeef()` to atomically update the value of d, and insert elements to the queue in parallel. `int_fetch_add` offers synchronized updates to data representing shared counters without using locks. The `readfe` operation atomically reads data from a memory location only after that locations full/empty bit is set full, and sets it back to empty. If the bit is not full to start with, the

thread executing the read operation suspends in hardware and is later retried. Similarly, a `writeef` writes to a memory location when the full/empty bit is empty and then sets it to full. A `readfe` should be matched with a `writeef`, or else the program might deadlock.

Once correctness is assured, we optimize the code further. Note that we used the *loop future* directive on the outer loop in Alg. 2 to concurrently schedule the loop iterations. Using this directive incurs an overhead of about 200 instructions. So we do not use it when the number of vertices to be visited at a given level is less than 50 (an experimentally determined figure for this particular loop). Clearly, the time taken to spawn threads must be considerably less than the time spent in the outer loop.

High-degree vertices pose a major problem. Consider the case when the majority of vertices at a particular level are of low-degree (less than 100), but a few vertices are of very high-degree (order of thousands). If Alg. 1 is applied, most of the threads will be done processing the low-degree vertices quickly, but only a few threads will be assigned to inspect the adjacencies of the high-degree nodes. The system will be heavily under-utilized then, until the loop finishes. To prevent this, we first need to identify high-degree nodes at each level and work on them sequentially, but inspect their adjacencies in parallel. This ensures that work is balanced among the processors. We can choose the low-degree cutoff value appropriately so that parallelization of adjacency visits would be sufficient to saturate the system. We take this approach for BFS on Scale-free graphs. In general, given an arbitrary graph instance, we can determine which algorithm to apply based on a quick evaluation of the degree distribution. This can be done either during graph generation stage (when reading from an uncharacterized data set and internally representing it as a graph) or in a

pre-processing phase before running the actual BFS algorithm.

We observe that the above parallelization schemes will not work for high-diameter graphs (for instance, consider a chain of vertices with bounded degree). For arbitrary sparse graphs, Ullman and Yannakakis offer high-probability PRAM algorithms for transitive closure and BFS [41] that take $\tilde{O}(n^{\epsilon})$ time with $\tilde{O}(mn^{1-2\epsilon})$ processors, provided $m \geq n^{2-3\epsilon}$. The key idea here is as follows. Instead of starting the search from the source vertex s, we expand the frontier up to a distance d in parallel from a set of randomly chosen *distinguished* vertices (that includes the source vertex s also) in the graph. We then construct a new graph whose vertices are the distinguished vertices, and we have edges between these vertices if they were pair-wise reachable in the previous step. Now a set of *superdistinguished* vertices are selected among them and the graph is explored to a depth t^2. After this step, the resulting graph would be dense and we can determine the shortest path of the source vertex s to each of the vertices. Using this information, we can determine the shortest paths from s to all vertices.

3 *st*-connectivity and Shortest Paths

We can easily extend the Breadth-First Search algorithm for solving the st-connectivity problem too. A naïve implementation would be to start a Breadth-First Search from s, and stop when t is visited. However, we note that we could run BFS concurrently both from s and to t, and if we keep track of the vertices visited and the expanded frontiers on both sides, we can correctly determine the shortest path between s and t. The key steps are outlined in Alg. 3 (termed STCONN-FB), which has both high-level details as well as MTA-specific synchronization constructs. Both s and t are added to the queue initially, and newly

Input: $G(V, E)$, vertex pair (s, t)

Output: The smallest number of edges *dist* between s and t, if they are connected

```
1  for all v ∈ V in parallel do
2      color[v] ← WHITE;
3      d[v] ← 0;
4  color[s] ← RED;  color[t] ← GREEN;  Q ← φ;  done ← FALSE;  dist ← ∞;
5  Enqueue s ← Q;  Enqueue t ← Q;
6  while Q ≠ φ and done = FALSE  do
7      for all u ∈ Q in parallel do
8          Delete u ← Q;
9          for each v adjacent to u in parallel do
10             color ← readfe(&color[v]);
11             if color = WHITE then
12                 d[v] ← d[u] + 1;
13                 Enqueue v ← Q;
14                 writeef(&color[v], color[u]);
15             else
16                 if color ≠ color[u] then
17                     done ← TRUE;
18                     tmp ← readfe(&dist);
19                     if tmp > d[u] + d[v] + 1 then
20                         writeef(&dist, d[u] + d[v] + 1);
21                     else
22                         writeef(&dist, tmp);
23                 writeef(&color[v], color);
```

Algorithm 3: *st*-connectivity (STCONN-FB): concurrent BFSes from s and t

discovered vertices are either colored RED (for vertices reachable from s) or GREEN (for vertices that can reach t). When a *back edge* is found in the graph, the algorithm terminates and the shortest path is evaluated. As in the previous case, we encounter nested parallelism here and apply the same optimizations. The pseudo-code is elegant and concise, but must be carefully written to avoid the introduction of race conditions and potential deadlocks (see [4] for an illustration).

We also implement an improved algorithm for *st*-connectivity (STCONN-MF, denoting *minimum frontier*) applicable to graphs with a large percentage of high degree nodes, detailed

Input: $G(V, E)$, vertex pair (s, t)

Output: The smallest number of edges $dist$ between s and t, if they are connected

```
1  for all v ∈ V in parallel do
2      color[v] ← WHITE;
3      d[v] ← 0;
4  color[s] ← GRAY;  color[t] ← GRAY;  Qs ← φ;  Qt ← φ;
5  done ← FALSE;  dist ← −1;
6  Enqueue s ← Qs;  Enqueue t ← Qt;  extentS ← 1;  extentT ← 1;
7  while (Qs ≠ φ or Qt ≠ φ) and done = FALSE  do
8      Set Q appropriately;
9      for all u ∈ Q in parallel do
10         Delete u ← Q;
11         for each v adjacent to u in parallel do
12             color ← readfe(&color[v]);
13             if color = WHITE then
14                 d[v] ← d[u] + 1;
15                 Enqueue v ← Q;
16                 writeef(&color[v], color[u]);
17             else
18                 if color ≠ color[v] then
19                     dist ← d[u] + d[v] + 1;
20                     done ← TRUE;
21                 writeef(&color[v], color);
22     extentS ← |Qs|;  extentT ← |Qt|;
```

Algorithm 4: st-connectivity (STCONN-MF): alternate BFSes from s and t

in Alg. 4. In this case, we maintain two different queues Q_s and Q_t and expand the smaller frontier (Q in Alg. 4 is either Q_s or Q_t, depending on the values of $extentS$ and $extentT$) on each iteration. This algorithm would be faster for some graph instances (see [4] for an illustration). Both Alg. 3 and 4 are discussed in more detail in an extended version of this paper [4].

4 Experimental Results

This section summarizes the experimental results of our BFS and *st*-connectivity implementations on the Cray MTA-2. We report results on a 40-processor MTA-2, with each processor having a clock speed of 220 MHz and 4GB of RAM. From the programmer's viewpoint, the MTA-2 is however a global shared memory machine with 160GB memory.

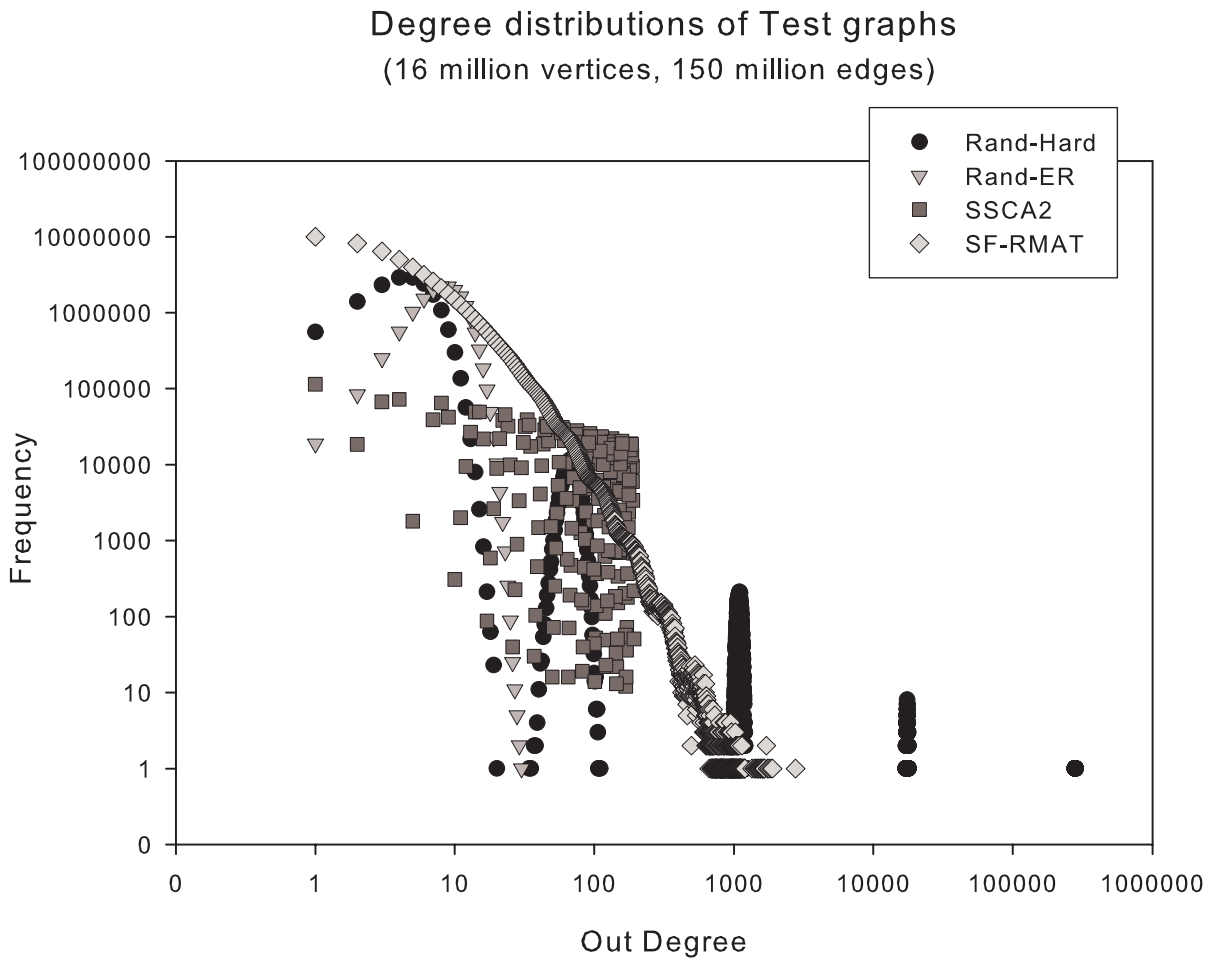

Figure 1: Degree distributions of the four test graph classes

We test our algorithms on four different classes of graphs (see Fig. 1):

- Random graphs generated based on the Erdős-Rényi $G(n, p)$ model (Rand-ER): A

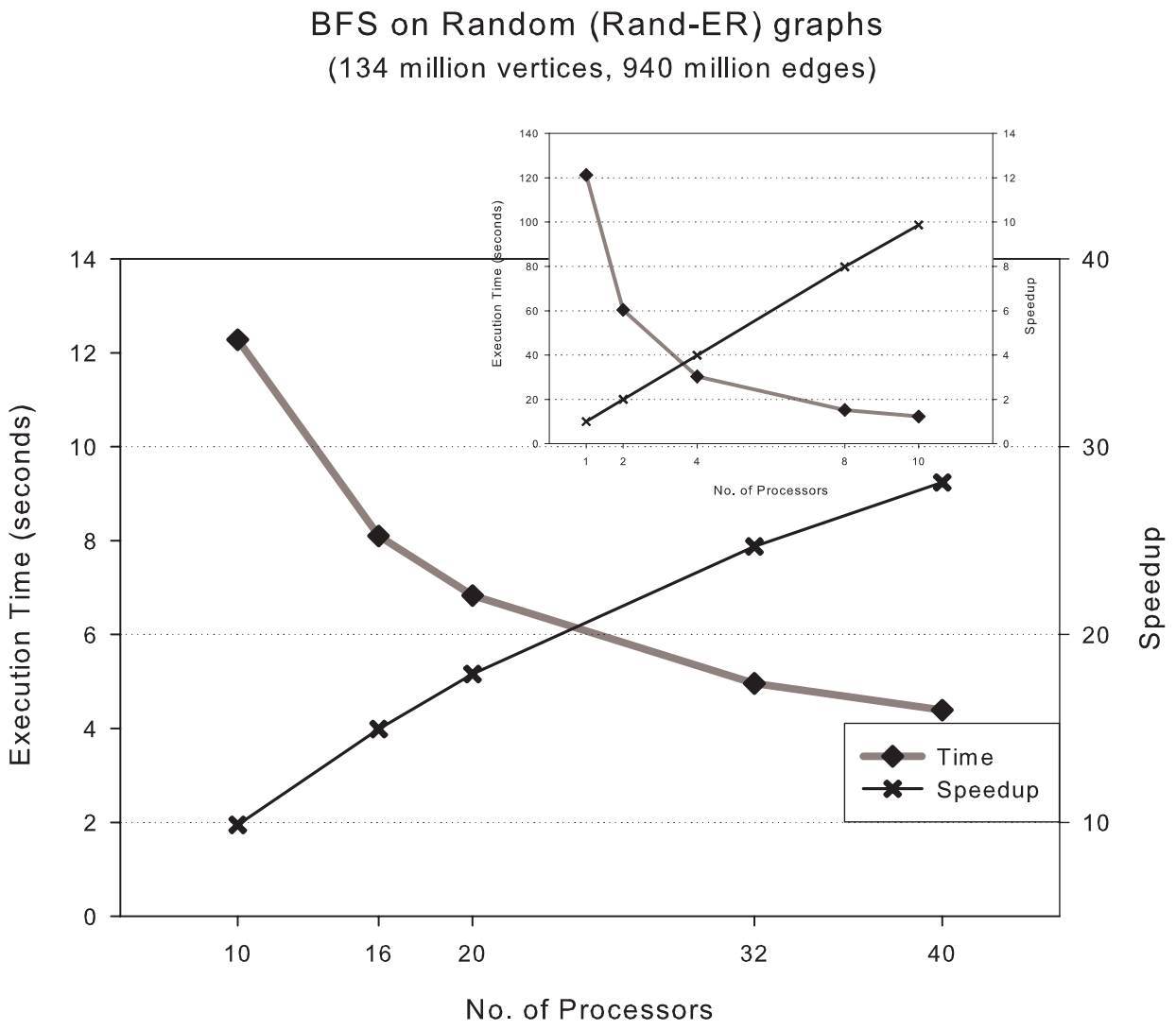

Figure 2: Breadth First Search Performance Results: Execution Time and Speedup for Random graphs: 1-10 processors (inset), and 10-40 processors

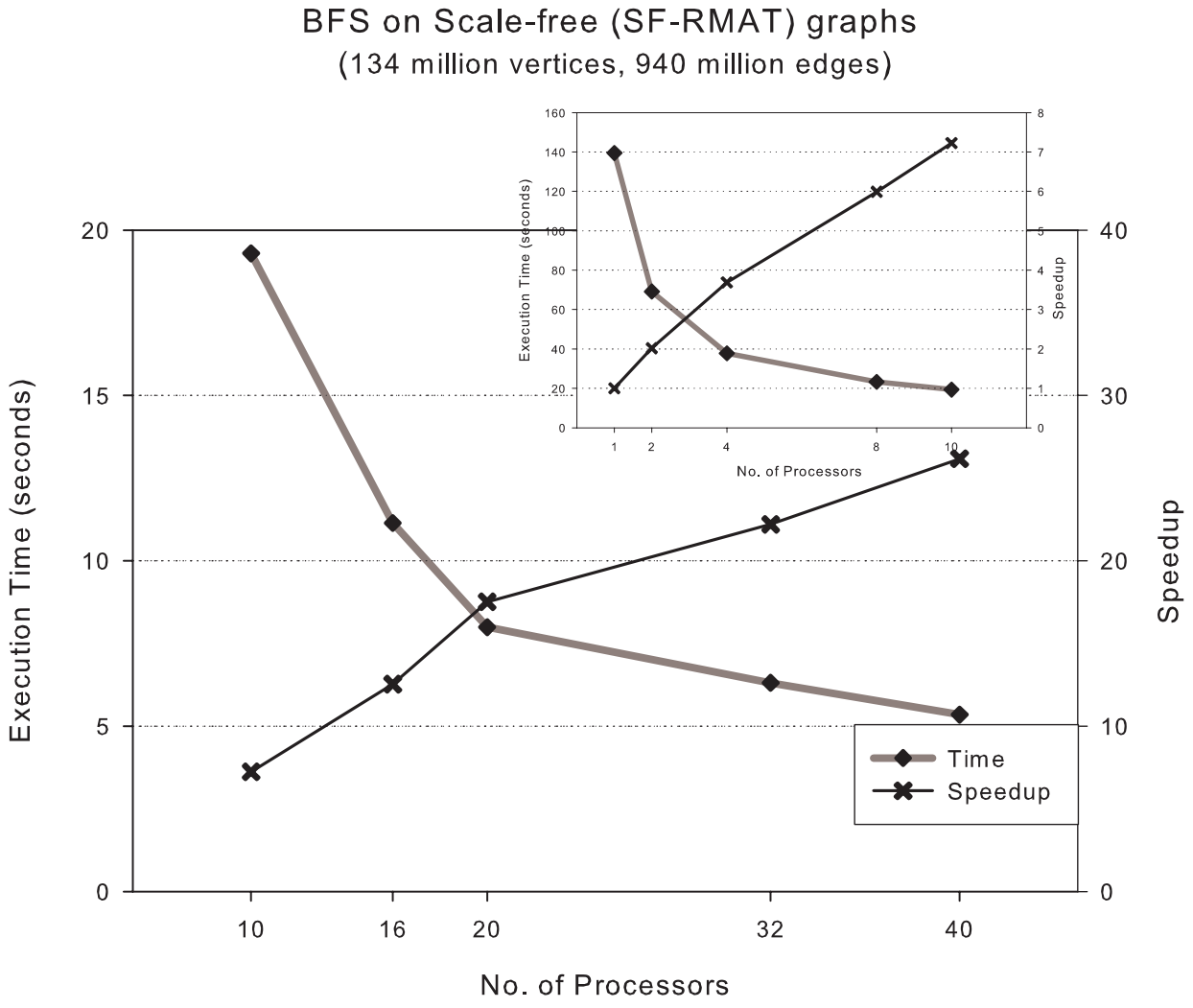

Figure 3: Breadth First Search Performance Results: Execution Time and Speedup for Scale-free (SF-RMAT) graphs: 1-10 processors (inset), and 10-40 processors

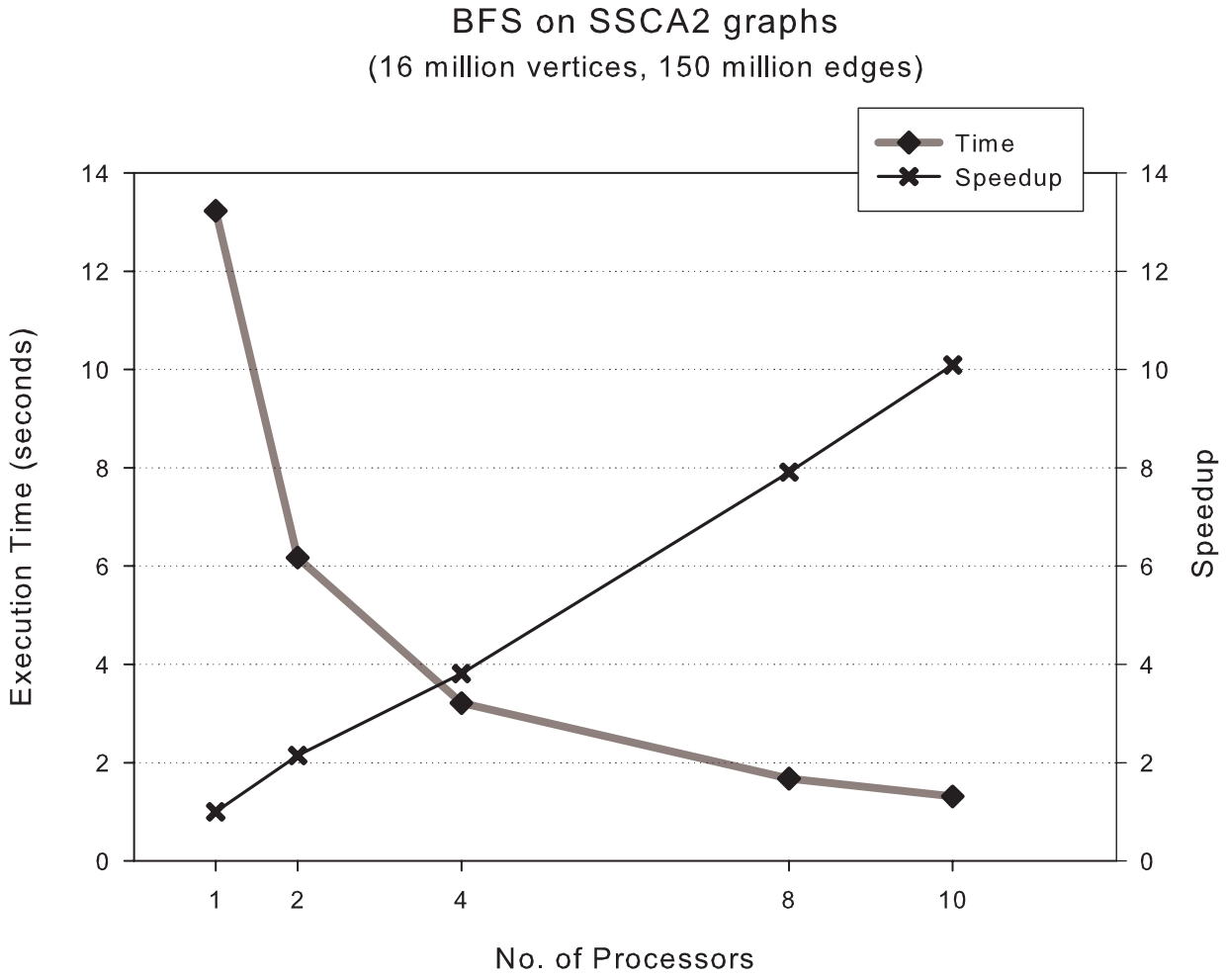

Figure 4: Breadth First Search Performance Results: Execution Time and Speedup for SSCA2 graphs

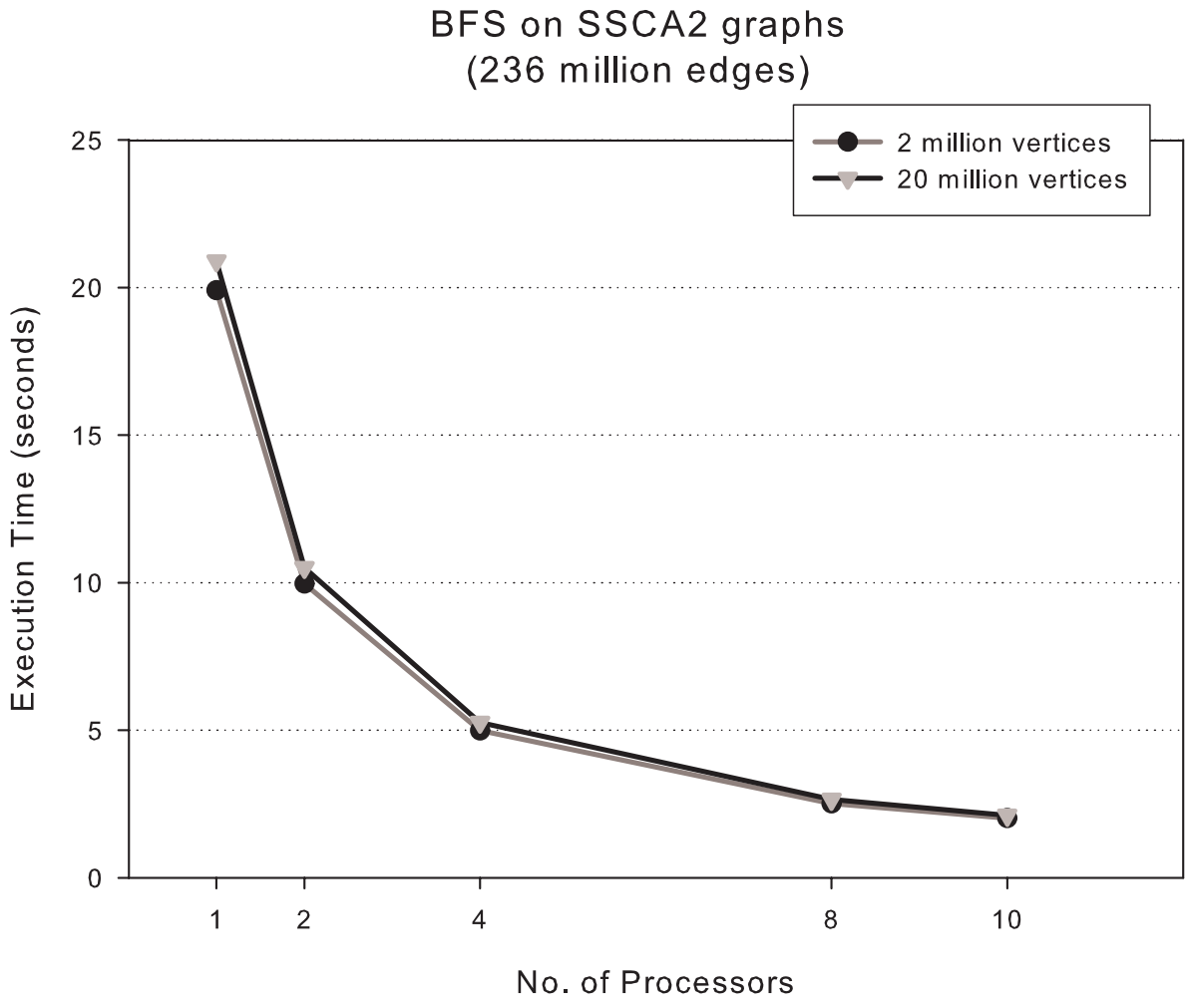

Figure 5: Breadth First Search Performance Results: Execution Time variation as a function of average degree for SSCA2 graphs

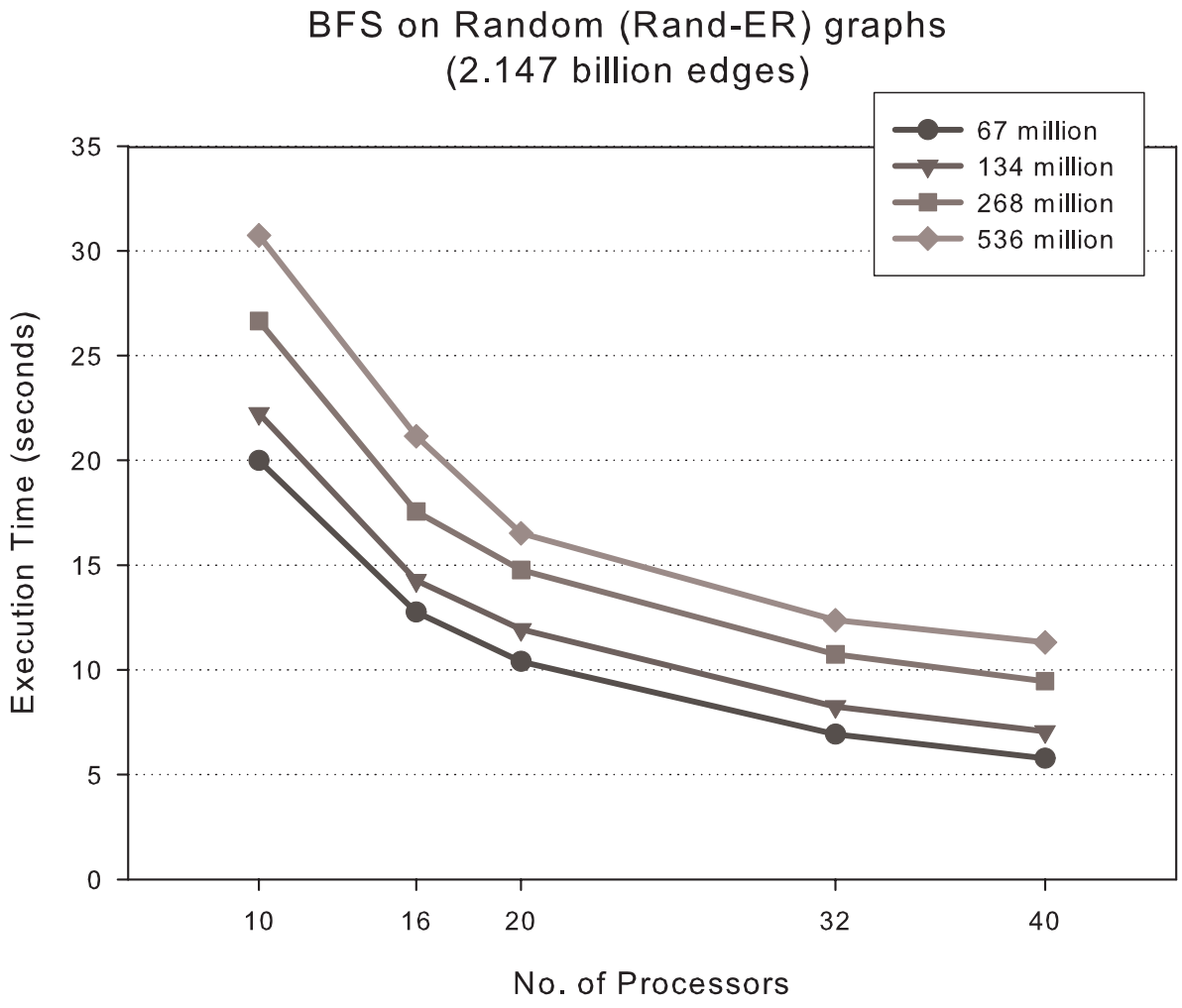

Figure 6: Breadth First Search Performance Results: Execution time variation as a function of average degree for Rand-ER graphs

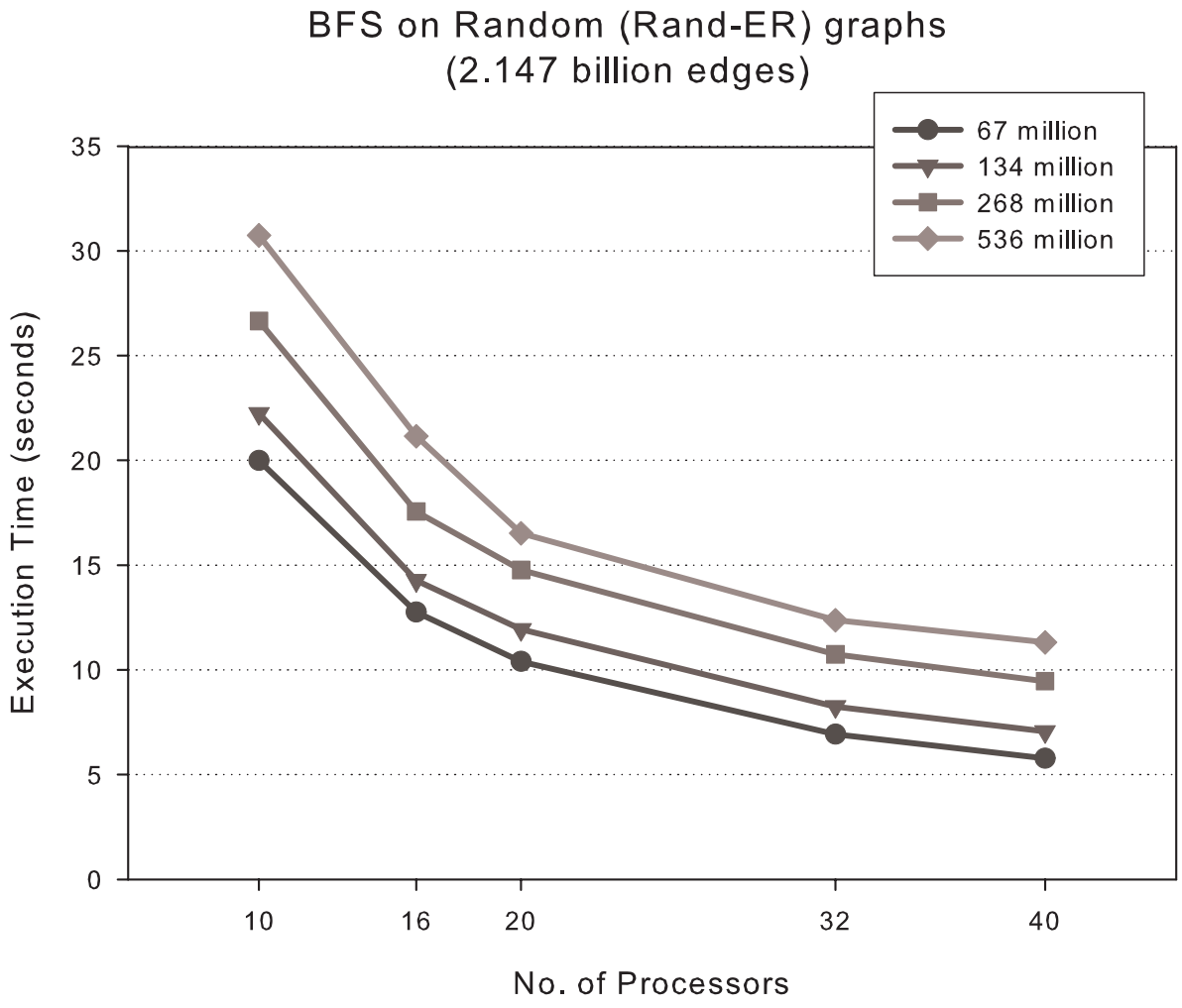

Figure 7: Breadth First Search Performance Results: Execution time variation as a function of average degree for Rand-Hard graphs

random graph of m edges is generated with $p = \frac{m}{n^2}$ and has very little structure and locality.

- Scale-free graphs (SF-RMAT), used to model real-world large-scale networks: These graphs are generated using the R-MAT graph model [11]. They have a significant number of vertices of very high degree, although the majority of vertices are low-degree ones. The degree distribution plot on a log-log scale is a straight line with a heavy tail, as seen in Fig. 1.

- Synthetic sparse random graphs that are hard cases for parallelization (Rand-Hard): As in scale-free graphs, a considerable percentage of vertices are high-degree ones, but the degree distribution is different.

- DARPA SSCA#2 benchmark (SSCA2) graphs: A typical SSCA#2 graph consists of a large number of highly interconnected clusters of vertices. The clusters are sparsely connected, and these inter-cluster edges are randomly generated. The cluster sizes are uniformly distributed and the maximum cluster size is a user-defined parameter. For the graph used in the performance studies in Fig. 1, a maximum cluster size of 10 was specified.

We generate directed graphs in all four cases. Our algorithms work for both directed and undirected graphs, as each vertex stores all its neighbors, and the edges in both directions. In this section, we report results for the undirected case. By making minor changes to our code, we can analyze directed graphs also.

Fig. 2 plots the execution time and speedup attained by the Breadth-First Search algo-

rithm on a random graph of 134 million vertices and 940 million edges (average degree 7). The plot in the inset shows the scaling when the number of processors is varied from 1 to 10, and the main plot for 10 to 40 processors. We define the *Speedup* on p processors of the MTA-2 as the ratio of the execution time on p processors to that on one processor. Since the computation on the MTA is thread-centric, system utilization is also an important metric to study. We observed utilization of close to 97% for single processor runs. We also note that the system utilization was consistently high (around 80% for 40 processor runs, see Table 1 in [4] for more details) across all runs. We achieve a speedup of nearly 10 on 10 processors for random graphs, 17 on 20 processors, and 28 on 40 processors. This is a significant result, as we are not aware of any s random graphs have no locality and such instances would offer very limited on no speedup on cache-based SMPs and other shared memory systems. The decrease in efficiency as the number of processors increases to 40 can be attributed to two factors: hot spots in the BFS queue, and a performance penalty due to the use of the future directive for handling nested parallelism.

Fig. 3 gives the BFS execution time for a Scale-free graph of 134 million vertices and 940 million edges, as the number of processors is varied from 1 to 40. The speedups are slightly lower than the previous case, due to the variation in the degree distribution. We have a pre-processing step for high-degree nodes as discussed in the previous sections; this leads to an additional overhead in execution time (when compared to random graphs), as well as insufficient work to saturate the system in some cases. Figs. 4 and 5 summarize the BFS performance for SSCA#2 graphs. The execution time and speedup (4) are comparable to random graphs. We also varied the user-defined cluster size parameter to see how BFS

performs for dense graphs. Fig. 5 shows that the dense SSCA#2 graphs are also handled well by our BFS algorithm.

Fig. 7 and 6 show the performance of BFS as the edge density is varied for Rand-ER and Rand-Hard graphs. We consider a graph of 2.147 billion edges and vary the number of vertices from 16 million to 536 million. In case of Rand-ER graphs, the execution times are comparable as expected, since the dominating term in the computational complexity is the number of edges, 2.147 billion in this case. However, in case of the Rand-Hard graphs, we note an anomaly: the execution time for the graph with 16 million vertices is comparatively more than the other graphs. This is because this graph has a significant number of vertices of very large degree. Even though it scales with the number of processors, since we avoid the use of nested parallelism in this case, the execution times are higher.

Fig. 8 summarizes the performance of st-connectivity. Note that both the st-connectivity algorithms are based on BFS, and if BFS is implemented efficiently, we would expect st-connectivity also to perform well. Fig. 8 (top) shows the performance of STCONN-MF on random graphs as the number of processors is varied from 1 to 10. Note that the execution times are highly dependent on (s, t) pair we choose. In this particular case, just 45,000 vertices were visited in a graph of 134 million vertices. The st-connectivity algorithm shows near-linear scaling with the number of processors. The actual execution time is bounded by the BFS time, and is dependent on the shortest path length and the degree distribution of the vertices in the graph. In Fig. 8 (bottom), we compare the performance of the two algorithms, concurrent Breadth-First Searches from s and t (STCONN-FB), and expanding the smaller frontier in each iteration (STCONN-MF). Both of them scale linearly with the number of

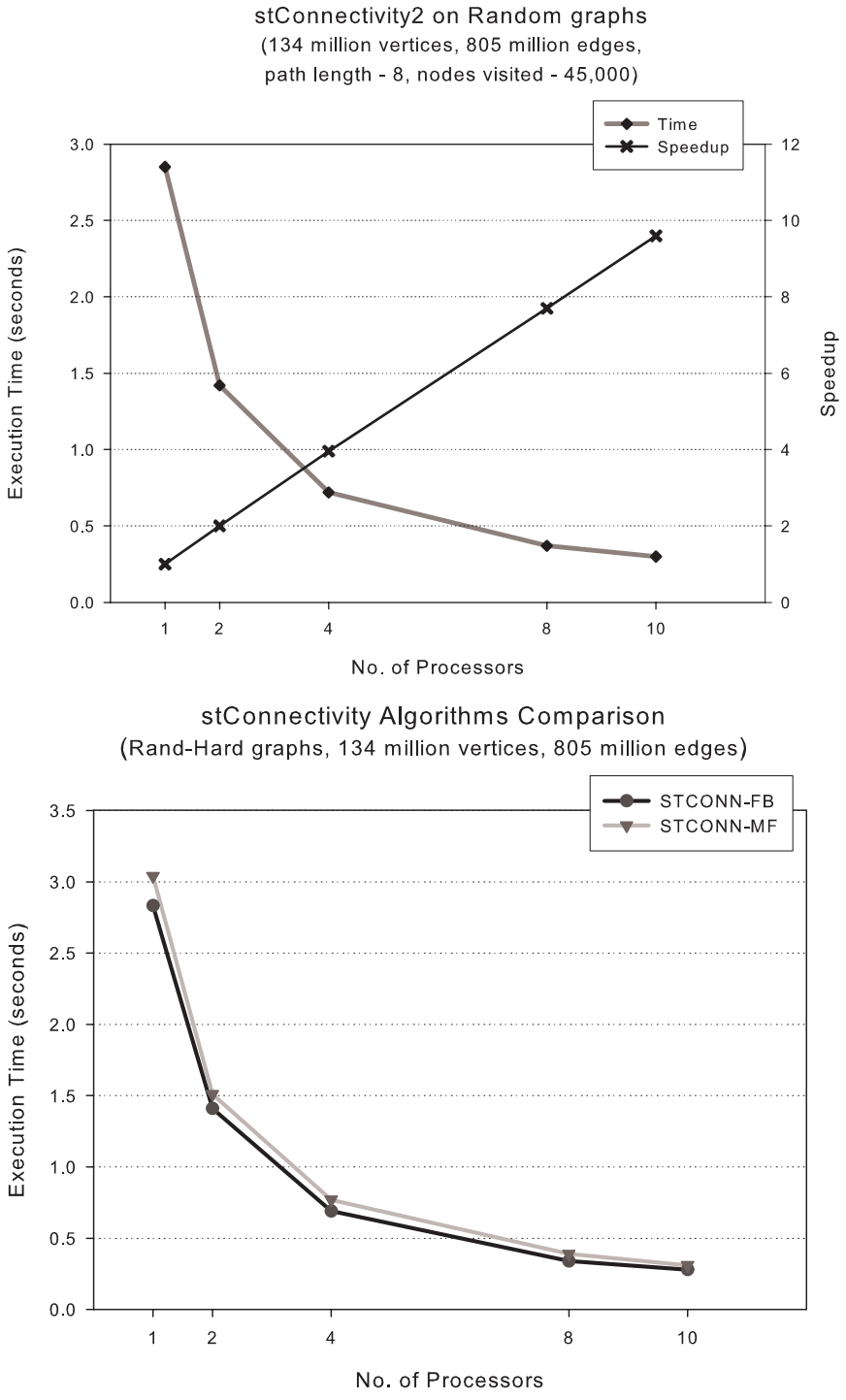

Figure 8: *st*-connectivity Results: Execution time and Speedup for Rand-ER graphs (top) and comparison of the two *st*-connectivity algorithms (bottom)

processors for a problem size of 134 million vertices and 805 million edges. STCONN-FB performs slightly better for this graph instance. They were found to perform comparably in other experiments with random and SSCA#2 graphs.

5 Conclusions

We present fast multithreaded algorithms for fundamental graph theory problems. Our implementations show strong scaling for irregular and sparse graphs chosen from four different graph classes, and also achieve high system utilization. The absolute execution time values are significant; Problems involving large graphs of billions of vertices and edges can be solved in seconds to minutes. With its latency tolerant processors, high bandwidth network, global shared memory and fine-grained synchronization, the MTA-2 is the first parallel machine to perform extraordinarily well on sparse graph problems. It may now be possible to tackle several key PRAM algorithms [24, 28, 22, 34] that have eluded practical implementations so far. Another attractive feature of the MTA-2 is the ease of programming. It is possible to write concise and elegant code, focusing on exploiting the concurrency in the problem, rather than optimizing for cache locality (or minimizing communication in distributed memory systems).

Acknowledgements

This work was supported in part by NSF Grants CAREER ACI-00-93039, CCF-0611589, NSF DBI-0420513, ITR ACI-00-81404, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654; and DARPA Contract NBCH30390004. We would like to thank

Richard Russell for sponsoring our MTA-2 accounts. We are grateful to John Feo for providing the SSCA#2 graph generator source. We acknowledge the significant algorithmic inputs and MTA-2 programming help from Jonathan Berry and Bruce Hendrickson. Finally, we would like to thank Simon Kahan, Petr Konecny, John Feo and other members of the Cray Eldorado team for their valuable advice and several suggestions on optimizing code for the MTA-2.

References

[1] J. M. Abello and J. S. Vitter, editors. *External memory algorithms*. Amer. Math. Soc., Boston, MA, USA, 1999.

[2] B. Awerbuch and R. G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Trans. Inf. Theor.*, 33(3):315–322, 1987.

[3] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Plotkin. Network decomposition and locality in distributed computation. In *IEEE Symp. on Found. of Comp. Sci.*, pages 364–369, 1989.

[4] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and *st*-connectivity on the Cray MTA-2. Tech. report, Georgia Institute of Technology, January 2006.

[5] D.A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proc. 34th Int'l Conf. on Parallel Processing (ICPP)*, Oslo, Norway, June 2005.

[6] D.A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proc. 12th Int'l Conf. on High Performance Computing*, Goa, India, December 2005. Springer-Verlag.

[7] A. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509, 1999.

[8] G. Barnes and W. L. Ruzzo. Deterministic algorithms for undirected s-t connectivity using polynomial time and sublinear space. In *Proc. 23rd Annual ACM Symp. on Theory of Computing*, pages 43–53, NY, USA, 1991. ACM Press.

[9] U. Brandes. A faster algorithm for betweenness centrality. *J. Mathematical Sociology*, 25(2):163–177, 2001.

[10] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. 11th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 859–860, Philadelphia, PA, USA, 2000.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. 4th SIAM Intl. Conf. on Data Mining*, Florida, USA, April 2004.

[12] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.

[13] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47(3):45–47, 2004.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Inc., Cambridge, MA, 1990.

[15] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *MFCS '98: Proc. of the 23rd Int'l Symp. on Mathematical Foundations of Computer Science*, pages 722–731, London, UK, 1998. Springer-Verlag.

[16] Defense Advanced Research Projects Agency (DARPA). High productivity computing systems program. `http://www.darpa.mil/ipto/programs/hpcs/`.

[17] C. Demetrescu, A. Goldberg, and D. Johnson. 9th DIMACS implementation challenge – shortest paths. `http://www.dis.uniroma1.it/~challenge9/`.

[18] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.

[19] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *SIGCOMM*, pages 251–262, 1999.

[20] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[21] H. Gazit and G. L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Inf. Process. Lett.*, 28(2):61–65, 1988.

[22] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proc. 6th Ann. Symp. Parallel Algorithms and Architectures (SPAA-94)*, pages 16–25, Cape May, NJ, June 1994.

[23] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. 4th Europ. Conf. on Principles of Data Mining and Knowledge Discovery*, pages 13–23, London, UK, 2000.

[24] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, New York, 1992.

[25] G. Karypis, E. Han, and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[26] J. Kepner, D. P. Koester, and *et al.* HPCS Scalable Synthetic Compact Application (SSCA) Benchmarks, 2004. `http://www.highproductivity.org/SSCABmks.htm`.

[27] J. Kepner, D. P. Koester, and *et al. HPCS SSCA#2 Graph Analysis Benchmark Specifications v1.0*, April 2005.

[28] P.N. Klein and J.H. Reif. An efficient parallel algorithm for planarity. *J. Comp. and System. Sci.*, 37(2):190–246, 1988.

[29] V.E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.

[30] T. Lengauer. *Combinatorial algorithms for integrated circuit layout.* John Wiley & Sons, Inc., New York, NY, USA, 1990.

[31] H. R. Lewis and C. H. Papadimitriou. Symmetric space-bounded computation (extended abstract). In *Proc. 7th Colloquium on Automata, Languages and Programming*, pages 374–384, London, UK, 1980.

[32] U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proc. 12th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 87–88, Philadelphia, PA, USA, 2001.

[33] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[34] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proc. 26th Ann. IEEE Symp. Foundations of Computer Science*, pages 478–489, Portland, OR, October 1985.

[35] B. M.E. Moret, D.A. Bader, T. Warnow, S.K. Wyman, and M. Yan. GRAPPA: a high-performance computational tool for phylogeny reconstruction from gene-order data. In *Proc. Botany*, Albuquerque, NM, August 2001.

[36] B.M.E. Moret, D.A. Bader, and T. Warnow. High-performance algorithm engineering for computational phylogenetics. In *Proc. Int'l Conf. on Computational Science*, volume 2073–2074 of *Lecture Notes in Computer Science*, San Francisco, CA, 2001. Springer-Verlag.

[37] P. M. Pardalos, M. G. Resende, and K. G. Ramakrishnan, editors. *Parallel Processing of Discrete Optimization Problems: DIMACS Workshop April 28-29, 1994.* American Mathematical Society, Boston, MA, USA, 1995.

[38] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995.

[39] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[40] J. L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Comput.*, 21(9):1505–1532, 1995.

[41] J. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. In *Proc. 2nd Ann. Symp. Parallel Algorithms and Architectures (SPAA-90)*, pages 200–209, New York, NY, USA, 1990. ACM Press.

[42] A. Vazquez, A. Flammini, A. Maritan, and A. Vespignani. Global protein function prediction in protein-protein interaction networks. *Nature Biotechnology*, 21:697, 2003.

[43] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and Ü. V. Çatalyürek. A scalable distributed parallel breadth-first search algorithm on Bluegene/L. In *Proc. Supercomputing (SC 2005)*, Seattle, WA, November 2005.