A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets

Kamesh Madduri* KMadduri@lbl.gov
David Ediger† dediger@gatech.edu
Karl Jiang† k.jiang@gatech.edu
David A. Bader† bader@cc.gatech.edu
Daniel Chavarría-Miranda‡ daniel.chavarria@pnl.gov

Abstract

We present a new lock-free parallel algorithm for computing betweenness centrality of massive small-world networks. With minor changes to the data structures, our algorithm also achieves better spatial cache locality compared to previous approaches. Betweenness centrality is a key algorithm kernel in HPCS SSCA#2, a benchmark extensively used to evaluate the performance of emerging high-performance computing architectures for graph-theoretic computations. We design optimized implementations of betweenness centrality and the SSCA#2 benchmark for two hardware multithreaded systems: a Cray XMT system with the Threadstorm processor, and a single-socket Sun multicore server with the UltraSPARC T2 processor. For a small-world network of 134 million vertices and 1.073 billion edges, the 16-processor XMT system and the 8-core Sun Fire T5120 server achieve TEPS scores (an algorithmic performance count for the SSCA#2 benchmark) of 160 million and 90 million respectively, which corresponds to more than a 2× performance improvement over the previous parallel implementations. To better characterize the performance of these multithreaded systems, we correlate the SSCA#2 performance results with data from the memory-intensive STREAM and RandomAccess benchmarks. Finally, we demonstrate the applicability of our implementation to analyze massive real-world datasets by computing approximate betweenness centrality for a large-scale IMDb movie-actor network.

1 Introduction

Graphs are a fundamental abstraction for representing data sets, and graph-theoretic algorithms and analysis routines are pervasive in several application domains today. Computations involving sparse real-world graphs such as socio-economic interactions, the world-wide

*Computational Research Division, Lawrence Berkeley National Laboratory.
†College of Computing, Georgia Institute of Technology.
‡High Performance Computing, Pacific Northwest National Laboratory.

web, and biological networks only manage to achieve a tiny fraction of the computational peak performance on the majority of current computing systems. The primary reason is that sparse graph analysis routines tend to be highly memory-intensive: they typically have a large memory footprint, exhibit low degrees of spatial and temporal locality in their memory access patterns (compared to other workloads), and there is very little computation to hide the latency to memory accesses. Thus, the execution time of a graph-theoretic computation strongly correlates with the memory subsystem performance, rather than the processor clock frequency or the floating-point processing capabilities of the system. The design of efficient parallel graph algorithms is quite challenging as well [23], due to the fact that massive graphs that occur in real-world applications are not amenable to a balanced partitioning among processors of a parallel system [20, 21]. Also, the locality characteristics of parallel graph algorithms tend to be poorer than their sequential counterparts [11].

The HPCS [13] graph theory benchmark was introduced as part of the Scalable Synthetic Compact Applications (SSCA) benchmark suite [5], and is representative of key computations in graph informatics applications such as social network analysis, epidemiological studies, and network analysis in systems biology. It is designed to be a compact mini-application that has multiple analysis techniques (multiple kernels) accessing a single data structure representing a weighted, directed graph. The second version of the benchmark specification was released in August 2006 [1], and consists of four kernels that operate on a synthetic graph instance. We use the Recursive MATrix (R-MAT) [8] random graph generation algorithm to generate input data that are representative of real-world networks with a small-world topology. The most interesting computational kernel of the SSCA#2 benchmark is the parallel evaluation of betweenness centrality, which is the focus of the paper.

Betweenness centrality is a popular graph analysis technique based on shortest-path enumeration for identifying key entities in large-scale interaction networks. For any arbitrary graph $G(V, E)$, let σ_{st} denote the number of shortest paths between vertices s and t, and $\sigma_{st}(v)$ the count of shortest paths that pass through a specified vertex v. The betweenness centrality of v is defined as follows:

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

Intuitively, betweenness measures the control a vertex has over communication in the network, and can be used to identify critical vertices in the network. High centrality indices indicate that a vertex can reach other vertices on relatively short paths, or that a vertex lies on a considerable fraction of shortest paths connecting pairs of other vertices. This metric has been extensively used for analyzing key features in networks with small-world properties. Some applications include lethality in biological networks [17], study of sexual networks and AIDS [22], identifying key actors in terrorist networks [10, 19], organizational behavior, supply chain management processes, and transportation networks [14].

In [3], we present the first parallel algorithms for the exact evaluation of betweenness and other centrality metrics. The betweenness implementation in version 2.2 of the SSCA#2 benchmark is based on the fine-grained parallel algorithm discussed in [3]. In this paper, we present a **new parallel algorithm for computing betweenness centrality**, which significantly reduces the synchronization overhead in comparison to the previous algorithm, and also exhibits better cache locality. The key idea in the design of this algorithm –

eliminating predecessor multisets associated with each vertex – can be applied to build efficient algorithms for other path-based centrality metrics. We discuss this new algorithm in more detail in Section 2.

Current hardware designs utilize multi-level caches or multithreading to hide the latency to main memory. Hardware multithreading in particular has been shown to be very effective in the design and implementation of efficient parallel graph algorithms [4, 25]. In this paper, we focus on **optimizing betweenness centrality implementations** on two state-of-the-art multithreaded systems: the **Cray XMT** [12, 18] with the massively multithreaded Threadstorm processor, and the multicore **Sun UltraSPARC T2** [28] server. In Section 3, we discuss the architectural features of these parallel systems, and present centrality implementation details and architecture-specific optimizations. Our key performance results on these systems are summarized below:

- On a 16-processor 128 GB XMT system, we compute approximate betweenness centrality for a small-world network of 268 million vertices and 2.147 billion edges in 50.1 minutes. This corresponds to a Traversed Edges Per Second (or TEPS, an algorithmic performance count for the SSCA#2 benchmark) score of 160 million.
- We achieve a TEPS performance of 90 million on an 8-core 32 GB UltraSPARC T2 server, for a small-world network of 134 million vertices and 1.073 billion edges.
- The parallel speedup of the XMT implementation is on an average 10.5 on 16 processors for large-scale networks. On the UltraSPARC T2, with 64 threads of execution, we achieve an average relative speedup of nearly 40.
- We utilize the Cray XMT betweenness implementation to compute the approximate centrality scores of all the vertices in a large-scale social network, constructed from a recent snapshot of the IMDb movie-actor database [16]. In comparison to a parallel run on a quad-core Intel workstation, we are able to perform this computation roughly 4.75 times faster on the 16-processor XMT system.

2 Computing Betweenness

To evaluate the betweenness centrality of a vertex v (defined in Equation 1), we need to determine the number of shortest paths between every pair of vertices s and t, and the number of shortest paths that pass through v. There is no known algorithm to compute the exact betweenness centrality score of a single vertex without solving an all-pairs shortest paths problem instance in the graph. In this paper, we will constrain our discussion of parallel betweenness centrality algorithms to directed and unweighted graphs. To process undirected graphs with our new parallel algorithm, the network can be easily modified by replacing each edge by two oppositely directed edges. While the approach to parallelization for unweighted graphs [3] works for weighted low-diameter graphs as well, the concurrency in each parallel phase is dependent on the weight distribution.

Let the number of vertices in the graph $G(V, E)$ be given by n, and the number of edges by m. Let $d(s, v)$ denote the length of the shortest path to v from a source vertex s.

Traditionally, betweenness centrality was computed in two steps: first, the number and length of shortest paths between all pairs of vertices were computed, and second, the pair

dependencies (the fractions $\frac{\sigma_{st}(v)}{\sigma_{st}}$) for each s-t pair were summed. The complexity of this approach was $O(n^3)$ time and $O(n^2)$ space. Exploiting the sparse nature of real-world networks, Brandes [7] presented a sequential algorithm to compute the betweenness centrality score for all vertices in an unweighted graph in $O(mn)$ time and $O(m+n)$ space. The main idea is to perform n breadth-first graph traversals, and augment each traversal to compute the number of shortest path passing through each vertex. The second key idea is that pairwise dependencies $\delta_{st}(v)(= \frac{\sigma_{st}(v)}{\sigma_{st}})$ can be aggregated without computing all of them explicitly. Define the *dependency* of a source vertex $s \in V$ on a vertex $v \in V$ as $\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$. The betweenness centrality of a vertex v can be then expressed as $BC(v) = \sum_{s \neq v \in V} \delta_s(v)$. Brandes shows that the dependency values $\delta_s(v)$ satisfy the following recursive relation:

$$\delta_s(v) = \sum_{w:d(s,w)=d(s,v)+1} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)) \tag{2}$$

Thus, the sequential algorithm computes betweenness in $O(mn)$ time by iterating over all the vertices $s \in V$, and computing the dependency values $\delta_s(v)$ in two stages. First, the distance and shortest path counts from s to each vertex are determined. Second the vertices are revisited starting with the farthest vertex from s first, and dependencies are accumulated according to Equation 2.

In prior work, we presented new parallel algorithms for computing betweenness on low-diameter graphs [3] with the same work complexity as Brandes' algorithm. There are two main approaches to parallelize exact betweenness computation for sparse graphs. The first approach is a coarse-grained parallelization, where each processor independently executes an augmented breadth-first search computation, while ensuring that the final centrality scores are updated atomically. However, the memory requirements scale as $O((m+n)p)$ for this approach, where p is the number of processors in the parallel system (or the number of thread contexts in a multithreaded system). Thus, this is infeasible for processing massive graphs.

Our second approach is a fine-grained parallelization of each augmented breadth-first search computation. Algorithm 1 gives a high-level schematic describing the two main steps in each iteration. Starting the source vertex s, we successively expand the frontier of visited vertices and augment breadth-first graph traversal (we also refer to this as *level-synchronous* graph traversal) to count the number of shortest paths passing through each vertex. We maintain a multiset P of *predecessors* associated with each vertex. A vertex v belongs to the predecessor multiset of w if $\langle v, w \rangle \in E$ and $d(s,w) = d(s,v) + 1$. Clearly, the size of a predecessor multiset for a vertex is bounded by its in-degree. The predecessor information is used in the dependency accumulation step (step III in Algorithm 1. We also indicate the steps in the algorithm that are amenable to parallel execution. However, note that accesses to the shared data structures (such as the predecessor multisets and the stack) and updates to the distance and path counts need to be protected with appropriate synchronization constructs, which we do not indicate in Algorithm 1.

In Algorithms 2 and 3, we give more detailed pseudo-code for implementing the traversal and dependency accumulation steps. We assume that our target architectures support two atomic operations – `compare_and_swap` and `fetch_and_add` – and list the pseudo-code for these parallel steps using these operations. `fetch_and_add` atomically increments a memory

Algorithm 1: A level-synchronous parallel algorithm for computing betweenness centrality of vertices in unweighted graphs.

Input: $G(V, E)$
Output: $BC[1..n]$, where $BC[v]$ gives the centrality score for vertex v

```
1  for all v ∈ V in parallel do
2      BC[v] ← 0;
3  for all s ∈ V do
       I. Initialization
4      P[t] ← empty multiset, σ[t] ← 0, and d[t] ← −1 ∀ t ∈ V;
5      σ[s] ← 1, d[s] ← 0;
6      phase ← 0, S[phase] ← empty stack;
7      push s → S[phase];
8      count ← 1;
       II. Graph traversal for shortest path discovery and counting
9      while count > 0 do
10         count ← 0;
11         for all v ∈ S[phase] in parallel do
12             for each neighbor w of v in parallel do
13                 if d[w] < 0 then
14                     push w → S[phase + 1];
15                     count ← count + 1;
16                     d[w] ← d[v] + 1;
17                 if d[w] = d[v] + 1 then
18                     σ[w] ← σ[w] + σ[v];
19                     append v → P[w];
20         phase ← phase + 1;
21     phase ← phase − 1;
       III. Dependency accumulation by back-propagation
22     δ[t] ← 0 ∀ t ∈ V;
23     while phase > 0 do
24         for all w ∈ S[phase] in parallel do
25             for all v ∈ P[w] do
26                 δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w]);
27         BC[w] ← BC[w] + δ[w];
28     phase ← phase − 1;
```

location by the specified integer value, and returns the value initially read from the memory location. `compare_and_swap` atomically compares the value of the memory location with the given value, and swaps it with the new value if they are equal. We assume that `compare_and_swap` also returns the value read from the memory location before the comparison. In case these two instructions are not supported on a parallel system, we can protect access to the shared variables with fine-grained mutual exclusion locks. Note that increments to the path count and the predecessor multiset are performance bottlenecks in the graph traversal step, but we show that they can be implemented with atomic operations. However, the dependence accumulation step requires locks, as δ and BC values are stored as floating-point numbers.

Algorithm 2: Pseudo-code for the augmented breadth-first graph traversal step in Algorithm 1.

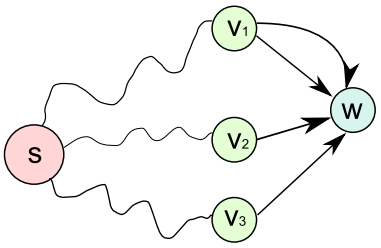
```
for all v ∈ S[phase] in parallel do
    for each neighbor w of v in parallel do
        dw ← compare_and_swap(&d[w], −1, phase + 1);
        if dw = −1 then
            p ← fetch_and_add(&count, 1);
            Insert w at position p of S[phase + 1];
            dw ← phase + 1;
        if dw = phase + 1 then
            p ← fetch_and_add(&Pcount[w], 1);
            Insert v at position p of P[w];
            fetch_and_add(&sigma[w], sigma[v]);
```

Algorithm 3: Pseudo-code for the dependency accumulation step in Algorithm 1.

```
for all w ∈ S[phase] in parallel do
    for all v ∈ P[w] do
        acquire lock on vertex v;
        δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w]);
        release lock on vertex v;
    BC[w] ← BC[w] + δ[w];
```

2.1 A lock-free parallel algorithm

As far as we know, all previous serial and parallel algorithms for computing betweenness centrality (and other shortest-path based centrality metrics) require the use of predecessor multisets. Updates to these multisets tend to limit concurrency in level-synchronous graph traversal, and can be a serious bottleneck in some cases. For instance, see Figure 1 for a possible scenario. Vertices v_1, v_2, v_3 are being processed in parallel and are all predecessors to vertex w, which is one hop farther away from the source vertex. Appends to the predecessor multiset of w will serialize in this case. Even in case of the serial betweenness centrality algorithm, the appends tend to be cache-unfriendly memory accesses, as we are touching



Predecessor updates in Algorithm 1	*New approach*
append $v_1 \rightarrow P[w]$	append $w \rightarrow Succ[v_1]$
append $v_1 \rightarrow P[w]$	append $w \rightarrow Succ[v_1]$
append $v_2 \rightarrow P[w]$	append $w \rightarrow Succ[v_2]$
append $v_3 \rightarrow P[w]$	append $w \rightarrow Succ[v_2]$

Figure 1: An illustration of the our new representation of the predecessor multisets.

w when we are processing v_i's. Similarly, we need fine-grained locking in the dependence accumulation step (Algorithm 3) due to this representation of the predecessor multisets.

Algorithm 4: Pseudo-code for the augmented breadth-first graph traversal step in our new parallel algorithm (replacing lines 11–19 in Algorithm 1).

```
for all v ∈ S[phase] in parallel do
    for each neighbor w of v in parallel do
        dw ← compare_and_swap(&d[w], −1, phase + 1);
        if dw = −1 then
            p ← fetch_and_add(&count, 1);
            Insert w at position p of S[phase + 1];
            dw ← phase + 1;
        if dw = phase + 1 then
            p ← fetch_and_add(&Succ_count[v], 1);
            Insert w at position p of Succ[v];
            fetch_and_add(&sigma[w], sigma[v]);
```

To improve the performance of our fine-grained parallel approach, we explored alternate representations for the predecessor multisets. We observe that it is possible to simplify both the serial and parallel algorithms by slightly restructuring the code, and by not storing the predecessors in their current form. Consider the illustration in Figure 1, where we store the adjacencies of a vertex that lie along shortest paths when processing the vertices themselves. Note that the correctness is not affected for an unweighted graph, as the presence of a directed edge $\langle u, v \rangle$ implies a shortest path along that edge, and that u belongs to the predecessor multiset of v. Since the shortest path counts are accumulated as the graph traversal proceeds, we still need to store them as before. However, with this simple change to the predecessor set representation, the dependence accumulation code is greatly simplified and we do not need locking at all for updates. Algorithms 4 and 5 list the new pseudo-code for the traversal and dependence accumulation steps respectively. We now maintain the *successor* multisets (denoted by the array `Succ` in the pseudo-code) instead, the size of which is conveniently bounded by the out-degree. Observe that the algorithm is more cache-friendly as well, as the updates are applied to the vertex that is currently being processed.

Algorithm 5: Pseudo-code for the dependency accumulation step in our new parallel algorithm (replacing lines 23–28 in Algorithm 1).

$phase \leftarrow phase - 1$;
for *all* $w \in S[phase]$ ***in parallel*** **do**
  $delta\_sum\_w \leftarrow 0$;
  $sigma\_w \leftarrow \sigma[w]$;
  for *all* $v \in Succ[w]$ **do**
    $delta\_sum\_w \leftarrow delta\_sum\_w + \frac{sigma\_w}{\sigma[v]}(1 + \delta[v])$;
  $\delta[w] \leftarrow delta\_sum\_w$;
  $BC[w] \leftarrow BC[w] + delta\_sum\_w$;

3 Centrality Implementations on Multithreaded Architectures

We next modify our shared memory implementations of the SSCA#2 graph analysis benchmark to reflect the improvements in the betweenness centrality algorithm. Kernel 4 of the SSCA#2 benchmark computes approximate betweenness centrality scores of all the vertices in a synthetic small-world network. We approximate betweenness values by traversing the graph from a randomly chosen subset of vertices in V (the number of vertices is specified by a benchmark parameter $K4Approx$), and then extrapolating the accumulated dependence scores. In practice, this approach generates a reasonably good approximation of the centrality scores for several real-world networks [2].

To compare the performance of this kernel across various implementations and architectures, we use a performance metric called *traversed edges per seconds*, or TEPS. Given the running time of the kernel to be t seconds, we define this normalized metric as follows:

$$BC\ TEPS = \frac{7n \cdot 2^{K4Approx}}{t} \tag{3}$$

$2^{K4Approx}$ is the number of vertices we perform graph traversals from, and $7n$ is the estimated number of edges visited in the SSCA#2 graph.

In this paper, we will focus on betweenness performance optimizations for two hardware multithreaded parallel systems, the Cray XMT and the Sun Fire T5120. We next describe the architectural characteristics of these systems, and present details of the shared-memory implementations for these systems.

3.1 The Cray XMT

The Cray XMT [12, 18, 9] is built on the idea of *tolerating latency* to memory by massive multithreading. The building block of the XMT is a 500 MHz 64-bit Threadstorm processor, which supports 128 hardware streams of execution mapped onto a single instruction pipeline. A processor can keep up to 1024 memory operations in flight, and so a memory reference latency of 1024 clock cycles can be tolerated without the use of cache memory. Context

switching between threads is extremely light-weight and takes just 1 clock cycle. Each processor can support up to 16 GB of commodity memory that is hashed and globally accessible in the system. A processor also has a 128 KB, 4-way set associate data buffer for caching local memory references.

The XMT system design differs significantly from its predecessor, the Cray MTA-2. XMT leverages the Cray XT infrastructure for its I/O, network, and operating system modules. It uses the Seastar-2 interconnection network; the network chips are connected in a 3D-torus network, which leads to a drop in per-processor bisection bandwidth as the system is scaled up. In contrast, the MTA-2 uses a custom modified Cayley graph interconnect, where the bisection bandwidth scales linearly with the number of processors. Also, the MTA-2 uses an older version of the Threadstorm processor clocked at 220 MHz, and there is no data buffer.

The XMT inherits several aspects of the MTA-2 software environment. Applications are developed in C and C++, and the compiler supports language extensions for lightweight thread creation and synchronization. In addition, the compiler can parallelize simple loops and linear recurrences. The programmer can specify dependencies and give additional hints to the compiler with the help of pragma statements. The programmer, however, has no control over data locality as the address space in uniformly hashed.

The Threadstorm processor provides excellent support for light-weight synchronization. Each memory word has an associated full-empty bit, which can be modified for fine-grained atomic reads and writes. Atomic increments (using the `int_fetch_add` operation) are light-weight and just cost one instruction cycle.

The Cray Apprentice2 environment are very helpful in analyzing the performance of applications on the XMT. Canal is a static analysis tool which shows compiler annotations for parallelized loops, and also generates a report indicating the number and type of instructions executed in the program. Hardware performance counters and profiling tools provide insight on the run-time behavior of the application.

While the XMT is very different from current shared-memory multicore and symmetric multiprocessor systems, the same design rules hold true for developing high-performance parallel implementations. Since a thread can have only one instruction in the pipeline at any given point of time, it is important to expose concurrency in the program and keep several memory references in flight on each processor. Due to the absence of caches, this is the only mechanism available for tolerating latency in memory-intensive applications. For parallel runs, it is critical to efficiently utilize bandwidth, as the bisection bandwidth per processor does not scale linearly with the processor count. We need to spawn enough threads to keep the processor pipeline busy, but we also need to keep in mind that additional threads may just increase memory traffic without substantial gains in instructions executed. Even though XMT synchronization is fairly light-weight, it is important to minimize it as much as possible, to avoid stalled threads and losses in instruction concurrency. Finally, balancing work among the processors is important.

Betweenness Implementation

It is relatively easy to adapt the pseudo-code listed in Algorithms 4 and 5 to implement approximate betweenness on the XMT. We use the `int_fetch_add` instructions for atomic increments. Since the Threadstorm processor does not have an atomic `compare_and_swap` instruction, we slightly modify the pseudo-code in Algorithm 4. We add an additional check to see if a vertex is visited for the first time, and only then add it to the stack. Its

corresponding distance value from the source vertex is then updated atomically. Alternately, we can simulate `compare_and_swap` with Threadstorm `readfe` and `writeef` instructions that modify the full/empty bit associated with each memory word. The primary difference between our new betweenness implementation and the old one (see Algorithm 1) is that we do not store the predecessors multisets associated with each vertices. This simplifies the accumulation step, removing the need for locking.

We need to parallelize two loops in every betweenness iteration, one in the graph traversal step and the other in the dependency accumulation step. On the XMT, this is achieved by just using a pragma to mark the loop parallel. Note that there are two levels of parallelism in the graph traversal step: all the vertices in the current frontier can be visited in parallel, and all the adjacencies of a vertex can be processed in parallel. Algorithm 4 gives the pseudo-code with both the loops parallelized. If we do not parallelize the inner loop, then we do not need to update `Succ_count[v]` atomically. Parallelizing the inner loop results in better work distribution among the threads, particularly in the case of small-world networks. This is because vertices in a small-world network tend to exhibit an unbalanced degree distribution, with a high percentage of low-degree vertices, and a few high-degree ($O(\sqrt{n})$) or higher) vertices. For SSCA#2 synthetic networks, however, since the vertex identifiers are randomly permuted in the graph generation stage, parallelizing just the outer loop results in a reasonably load-balanced work distribution.

Through inspection of our implementation and Canal annotations, we determine that the number of memory operations per iteration of centrality for SSCA#2 R-MAT graphs should be roughly $6.75m$, where m is the number of edges in the graph. This matches with the value of $6.5m$ obtained from performance counter data. Comparing with performance counter data helps us ensure that our implementation is frugal in the use of memory bandwidth, and that there are no extraneous memory references that we did not account for in manual inspection of the code.

The atomic increment instructions are potential performance bottlenecks to scalability on large XMT systems. Insertions of vertices to the stack representing the frontier of visited vertices can be alleviated by replicating stacks and merging them at the end of the iteration. Similarly, we can replicate the successor multisets for high-degree vertices to prevent any performance drop due to serialization of insertions into the multisets. These issues do not pose significant performance bottlenecks on the 16-processor XMT system we ran our experiments on.

3.2 The Sun UltraSPARC T2

The Sun UltraSPARC T2 is an eight-core processor, and the second-generation chip in Sun's Niagara architecture family. Each core is dual-issue and eight-way hardware multithreaded. Further, the hardware threads within a core are grouped into two sets of four threads each. There are two integer pipelines within a core, and each set of four threads share an integer pipeline. Each core also includes a floating point unit and a memory unit that are shared by all eight threads. Although the cores are dual-issue, each thread may only issue one instruction per cycle and so resource conflicts are possible. In comparison to the T2, the UltraSPARC T1 (the T2's predecessor) has 32 threads per processor, one integer pipeline per core, and one floating point unit shared by all eight cores.

The Sun Fire T5120 server we use in this study is a single-socket system with an UltraSPARC T2 processor that runs at 1167 MHz. The system has 32 GB DRAM memory, and the latency to main memory is in the order of 100's of clock cycles. Latency is hidden in part by the multilevel cache hierarchy, and tolerated to some extent with eight threads per core. Each core has an 8 KB L1 cache that is 4-way set associative. The eight cores share a 4 MB L2 cache which is 16-way set associative. The L1 and L2 cache line sizes are 16 and 64 bytes respectively. Note that the low per-thread L1 and L2 capacity and associativity values can result in significant cache capacity and conflict misses, in addition to memory bank conflicts. The presence of caches makes it considerably harder to understand and characterize performance of an application on the UltraSPARC T2.

Our parallel implementations are developed in C, with OpenMP for writing multithreaded code. On the Sun Fire T5120, we use the OpenMP implementation of the Sun Studio 12 compiler suite. As far as we can determine, the performance impact due to our choice of OpenMP as the threading library is minimal. We expect that POSIX threads, or any other light-weight threading package, would yield comparable performance.

Betweenness Implementation

Our UltraSPARC T2 multicore implementation has a few notable differences in comparison to the XMT implementation. Threads are spawned at the start of the program, and the number of threads is fixed throughout the execution of the program. The total number of threads in the system (64) is much smaller than the number of active threads on the XMT (128 per processor). Maximizing cache locality for each thread of execution is very important, as well as reducing the per-thread memory footprint. To characterize the spatial locality in our implementation, we maintain a count of the maximum number of non-contiguous memory accesses, a cost metric from the Helman-JaJa symmetric multiprocessor complexity model [15], for each step of the algorithm.

We use a compact array representation for the graph that requires just $m+n+1$ machine words. We assume that the vertices are labeled with integer identifiers between 0 and n-1. All the adjacencies of a vertex are contiguously stored in a block of memory, and the neighbors of vertex i are stored next to ones of vertex $i+1$. The size of the adjacencies array is m, and we require an array of pointers to this adjacency array, which is of size $n+1$ words. This representation is motivated by the fact that all the adjacencies of a vertex are visited in the graph traversal step after it is first discovered. Thus, we may incur two non-contiguous memory accesses when visiting adjacencies of an arbitrary vertex in the graph.

The minimum granularity of data transfer size from main memory to the L2 cache is the size of the cache line. Data is always fetched from main memory into the L2 cache as entire cache lines, or 64 bytes on the UltraSPARC T2. However, we may just require 4 or 8 bytes of data (for instance, reading the predecessor multiset size, updating the distance value, and incrementing the the shortest path count in Algorithm 2) when processing a vertex, and so the rest of the cache line tends to be wasted. To maximize reuse and minimize the number of non-contiguous memory accesses in our implementation, we use an array of structures representation for storing the auxiliary data structures associated with each vertex; thus, the number of shortest paths, the distance from the source vertex, the partial dependency value, and the shortest path count are stored in a contiguous block instead of separate arrays. The size of the structure is a multiple of 16 bytes and is aligned to the cache line to avoid any misses due to fragmentation. This representation cuts down the number of non-contiguous

memory references by a factor of four.

On the UltraSPARC T2, we use Solaris 10 C intrinsics for atomic increment and compare and swap operations. However, since atomic operations are expensive compared to the XMT, we try to minimize them as far as possible. As the number of threads are considerably smaller than the XMT, we maintain a separate stack of visited vertices for each thread in the graph traversal step, and the stacks are merged at the end of a phase. Thus, we avoid an atomic increment every time a vertex is inserted into the stack (see Algorithm 4). Also, we only parallelize the outer loop to avoid the successor array count increment.

Let m_s denote the average number of edges that lie along shortest paths in one iteration of betweenness computation. For SSCA#2 graphs, we observe that m_s is approximately $0.25m$. Since we know that the size of the successor multiset is bounded by the out-degree, we can pre-allocate memory for each vertex. However, it might be wasteful to do this if m_s is only a small fraction of m. For an arbitrary graph, we have no way of determining m_s beforehand, and the count varies depending on the source vertex. The value of m_s also determines the amount of computation in the dependency accumulation phase.

In summary, the maximum number of non-contiguous memory references per iteration in our new implementation is $3n + m$ in the graph traversal step, and $4n$ in the dependency accumulation step (neglecting the lower order terms). The m term is due to the fact that the auxiliary data structures used in the betweenness centrality kernel are constructed at the start of the kernel and not stored along with the graph representation. Hence we incur one non-contiguous memory access per edge for loading the auxiliary data (assuming that the vertex identifiers are randomly permuted), and this is the best we can do with the current algorithm.

4 Performance Analysis

We now present a detailed analysis of our new betweenness centrality algorithm performance, using the SSCA#2 graph analysis benchmark kernel 4 (approximate betweenness) as the reference implementation. We will refer to performance in terms of the normalized TEPS score, as defined in equation 3.

We report Threadstorm performance results on a 16-processor Cray XMT system with 128 GB memory. We built our code using the C compiler of the Cray XMT programming environment (version 5.2.1) and flags `-par -O3`. We also compare performance with a 40 processor Cray MTA-2 system with 140 GB memory. On the MTA-2, we use the C compiler of the Cray programming environment (version 6.0.3) with the same optimization flags. We do not need to make any changes to run the code on the MTA-2.

We build our multicore C implementation on the Sun Fire T5120 using the Sun Studio 12 C compiler and the flags `-fast -m64 -xopenmp -xtarget=ultraT2 -xchip=ultraT2`. The Sun server is a single-socket UltraSPARC T2 system with 32 GB memory.

We set K4Approx to 8 in the SSCA#2 benchmark, thus approximating centrality scores by traversing the graph from 256 randomly chosen vertices. A synthetic graph instance generated by the SSCA#2 benchmark is typically composed of one large strongly connected component with more than 95% vertices, and so we touch almost all the edges in the network in most of these 256 iterations. Using large K4Approx values gives better centrality score

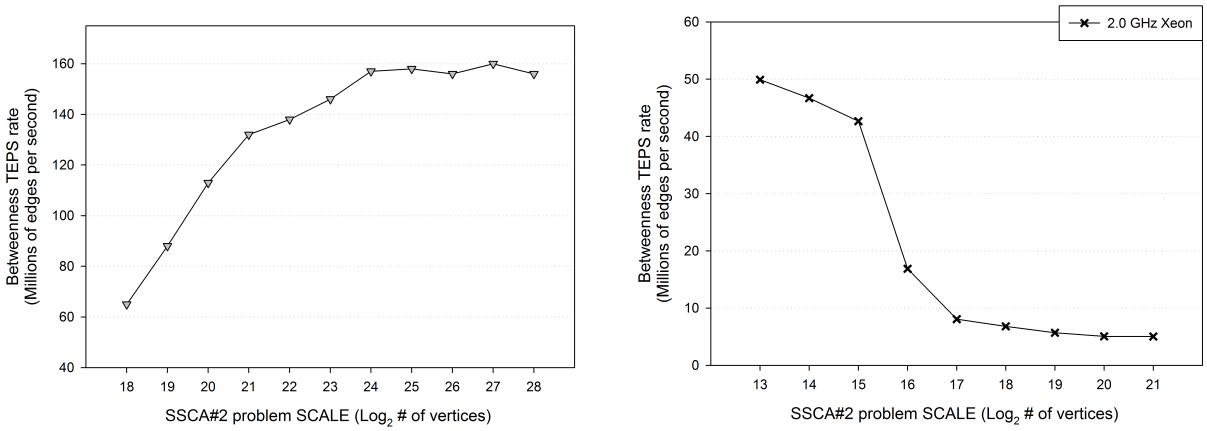
Scheduling mode	Average time (seconds)
Block	41.78
Block Dynamic	29.95
Interleaved	35.49
Dynamic	30.71

Table 1: Average execution time of the SSCA#2 centrality kernel (SCALE 21) on 16 processors of the XMT with various loop scheduling modes.

approximations. However, since we parallelize a single iteration of the centrality computation, the parallel scaling results are independent of the value of K4Approx we choose. The number of vertices and edges in the graph are set by a parameter SCALE: $n = 2^{SCALE}$ and $m = 8n$.

Stream Allocation on the XMT. Running our new implementation on 16 processors of the XMT, we achieved a TEPS score of 117 million for a graph of SCALE 21 (2.09 million vertices and 16.77 million edges). The loops in the centrality kernel were annotated using `#pragma mta assert no dependence` (indicating that the loop iterations can be scheduled independently) and the compiler automatically parallelized them. However, note that we parallelize only the outer loops in both the traversal and path accumulation steps. The static analysis tool Canal reported that 60 streams were being requested on each processor, for each loop. We experimented with several values between 60 and 120 using the `#pragma mta use k streams` statement and achieved peak performance with $k = 100$ streams on 16 processors. Thus, by hand-tuning the number of streams being requested, we were able to increase the TEPS score from 117 million to 131.7 million on 16 processors. Note however that the optimal number of streams is dependent on the graph topology, the problem size, as well as the number of processors. Using more streams than necessary increases the number of speculative loads, which would lead to a drop in performance as the number of processors in the system is scaled up (since the per-processor memory bandwidth decreases). In future work, we will try to automatically determine the optimal number of streams for each loop at run-time. In this paper, we will use a setting of 100 streams in all further experiments.

Parallel loop scheduling on the XMT. The Cray XMT compiler supports several different ways of scheduling iterations of a parallel loop, including block or static scheduling, dynamic, interleave, and block dynamic. The compiler picks an appropriate scheduling scheme, but the programmer can override it with a specific choice. We experiment with different scheduling modes for the two parallel loops in the betweenness implementation. For a graph of SCALE 21, we record the average execution time of five trials on 16 processors (see Table 1). We observe that the best performance is achieved with block dynamic scheduling, while static block scheduling performs poorly. This is most likely due to the power-law vertex degree distribution. Block dynamic scheduling, as the name suggests, combines aspects of block and dynamic scheduling. Threads are assigned blocks of iterations through a shared counter, and a thread gets its next block after completion of its current block by incrementing the counter. In pure dynamic scheduling, the block size can be just one iteration and the overhead is higher than static and block dynamic scheduling. We override the default dynamic scheduling with block dynamic scheduling in all further experiments. Again, this



(a) Cray XMT and Sun Fire T5120 **(b)** 2.0 GHz Intel Xeon

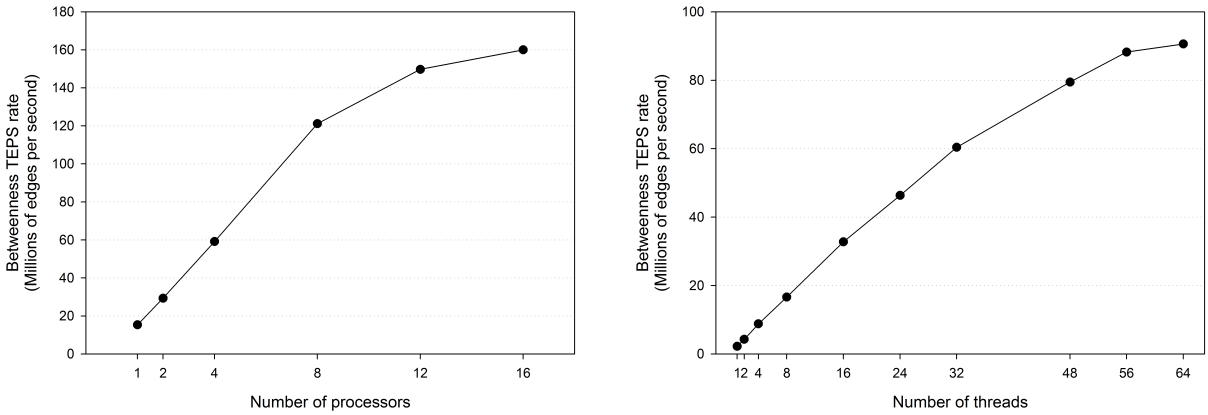
Figure 2: SSCA#2 betweenness kernel performance on the Sun Fire T5120, the Cray XMT system, and a 2.0 GHz Intel Xeon system for problem instances of various sizes.

setting is dependent on the graph topology, size, and number of processors. In future work, we will automate this process to pick the appropriate scheduling scheme.

Loop scheduling on the UltraSPARC T2. We try out the three possible OpenMP loop scheduling schemes on the UltraSPARC T2: static, dynamic, and guided. As in the case of the XMT, static performs poorly while the performance of guided and dynamic is similar. We use the guided mode for all the experiments in this paper.

Variation of performance with problem size. An important performance metric associated with the SSCA#2 benchmark is the largest problem instance that can be solved on a particular system. The performance on both the system varies as the problem size is varied. Figure 2(a) plots the performance on 16 processors of the XMT and 8 cores of the Sun Fire T5120 (64 threads) for SSCA#2 graphs of various sizes ($SCALE = 18$ to $SCALE = 28$). The largest problem instances we could run on the Sun server and the XMT were graphs of size SCALE 27 (memory footprint 24 GB) and SCALE 28 (memory footprint 90 GB) respectively. The factor-of-two difference in memory footprint is due to the fact that we use 32-bit unsigned integers for representing vertex identifiers on the Sun Fire T5120, and 64-bit words on the XMT. On the XMT, the performance picks up by a factor of 2.45 (64 to 160 million TEPS) as the problem size is increased from SCALE 18 to SCALE 24. This is due to the lack of sufficient concurrency for the problem instances to saturate all the threads on 16 processors. Similarly, on the Sun Fire T5120, the parallel overhead is significant for smaller problem sizes, but the performance plateaus SCALE 22 onwards. Comparing the performance of the XMT and the Sun Fire T5120 for SCALE 27, it is note-worthy that the single-socket Sun Fire system delivers performance equivalent to nearly eight XMT processors. At 160 million TEPS, the 16-processor XMT system is a factor of 1.77 faster than the Sun server for an identical problem instance of SCALE 27.

Parallel performance on the XMT. Since the XMT processor performance is relatively constant for problem instances greater than SCALE 24, we study strong scaling of the betweenness kernel for this problem instance. Figure 3(a) plots the TEPS score achieved



(a) Cray XMT **(b)** Sun Fire T5120

Figure 3: Parallel performance of SSCA#2 betweenness kernel on the Cray XMT and the Sun Fire T5120 for a graph of 16.77 million vertices and 134.21 million edges (SCALE 24).

by varying the number of processors from 1 to 16. On 16 processors, we achieve a parallel speedup of 10.41. The speedup is near-linear until 8 processors, but drops slightly for the case of 12- and 16-processor runs. Atomic insertions to a single stack of discovered vertices is likely one cause for the slowdown. This can be fixed by replicating the number of stacks to reduce contention, similar to the UltraSPARC T2 implementation. The reduced per-processor bisection bandwidth may be another reason for the performance drop. In future work, we will undertake a detailed analysis of performance bottlenecks, as well as study scaling on larger XMT systems.

Parallel performance on the UltraSPARC T2. We repeat the strong scaling experiment on the UltraSPARC T2. Figure 3(b) plots the performance of the betweenness kernel for SCALE 24, as the number of threads is varied from 1 to 64. Using the Solaris environment variable SUNW_MP_PROCBIND to bind threads to virtual processors, we experimented with several different configurations. In most cases, we achieved best results for a balanced distribution of threads: for instance, in case of a 16-thread run, we assigned 2 threads to each core, and one thread per thread group. Similarly, for a 32-thread run, assigning two threads to each thread group gave the best performance among all the possible scenarios. The operating system did a consistently good job scheduling threads, and the running time variation between the best combination and the OS-picked configuration was never more than 5%. We report performance for OS-scheduled runs in Figure 3(b). The relative speedup on 64 threads is 40.27 (ratio of the running time on 1 thread to the time on 64 threads). Further, the parallel code on 1 thread is just 4.4% slower than a serial implementation. Hence, the absolute speedup on the UltraSPARC T2 is 38.56. The final observed parallel speedup is remarkable, and the scaling is near-linear up to 56 threads. Note however that this is a single-socket system, and the performance may not scale as well on larger servers due to NUMA (non-uniform memory access) effects. The data layout in the current algorithm will need to be modified to improve performance on larger systems.

Performance comparison with predecessor systems. Table 2 compares the per-

Configuration	TEPS score
XMT, 1 processor	15.33
XMT, 16 processors	160.00
MTA-2, 1 processor	10.39
MTA-2, 16 processors	160.16
MTA-2, 40 processors	353.53
UltraSPARC T2	90.62
UltraSPARC T1	26.74

Table 2: Performance of the SSCA#2 betweenness centrality kernel for SCALE 24 on the Cray XMT, Cray MTA-2, Sun UltraSPARC T1, and the Sun UltraSPARC T2.

System/Algorithm	TEPS score	Speedup factor with new approach
Cray XMT, Alg. 1	69.14	2.31
UltraSPARC T2, Alg. 1	38.35	2.36
UltraSPARC T2, Alg. 4 (locks)	85.36	1.06

Table 3: A comparison of the performance of the new algorithm and the old approach on the UltraSPARC T2 and the Cray XMT.

formance of the XMT and UltraSPARC T2 with their respective predecessor architectures, the MTA-2 and the UltraSPARC T1 (please see [27] for a detailed discussion of architectural details and microbenchmark results on these architectures), for a problem instance of SCALE 24. We observe that the single-processor XMT implementation is 47% faster than the single-processor MTA-2. However, the performance is comparable on 16 processors, and the MTA-2 scales extremely well for even the 40-processor run, with a relative speedup of 34. We clearly notice the impact of XMT 3D-torus interconnect on the parallel performance scaling.

The UltraSPARC T2 is 3.4 times faster than its predecessor for this particular run. Note that the T2 has double the number of threads and a faster processor (1167 MHz compared to 1000 MHz). But the biggest difference is the number of floating point units. The T1 has just 1 FPU that is shared by all eight cores, which becomes a major performance bottleneck in the accumulation step of each iteration. We achieve better parallel speedup on the T2 due to the presence of one FPU per core.

Performance comparison with older algorithm. In Table 3, we report the performance of the previous parallel approach (using predecessor sets) on the XMT and the UltraSPARC T2, for the same SSCA#2 graph instance of SCALE 24. On the UltraSPARC T2, we also give performance of a code variant that uses OpenMP mutex locks instead of atomic operations. We observe that both our XMT and the T2 implementations are significantly faster than their corresponding older implementations. Using Solaris atomic instructions instead of OpenMP locks, we achieve a performance improvement of 6% for this problem instance.

Performance correlated with other benchmarks. To better characterize performance of the SSCA#2 graph analysis benchmark on these two systems, we give parallel

System/Configuration	STREAM avg. (MB/s)	RandomAccess (GUPS * 10^{-3})
XMT, 1 thread	48.47	0.55
XMT, 1 proc	1057.97	32.90
XMT, 16 proc	13375.32	338.60
UltraSPARC T2, serial	915.30	5.73
UltraSPARC T2, 8 threads	7030.75	44.69
UltraSPARC T2, 32 threads	7234.02	117.09
UltraSPARC T2, 64 threads	5562.53	119.97
UltraSPARC T2, best	8649.35	121.13

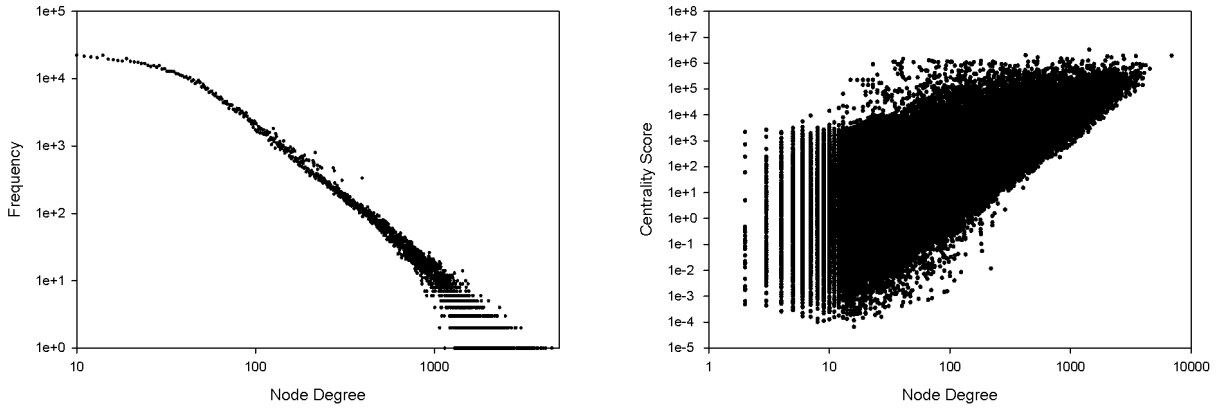
Table 4: STREAM and RandomAccess results on the XMT and the UltraSPARC T2.

performance results of two other memory-intensive benchmarks: STREAM [26] and RandomAccess [24]. STREAM is a synthetic benchmark that measures sustainable memory bandwidth in MB/s for a simple kernel with unit-stride memory references. In Table 4, we give the average value of the performance obtained for the four variants of STREAM. For XMT results, we modified the reference benchmark in C and OpenMP by replacing each `#pragma omp parallel for` with a corresponding `#pragma mta assert parallel`. On the XMT, the benchmark reports 1056.3 MB/s for a single processor run. Each read or write is seen by the benchmark as 8 bytes, but the XMT interconnection network transfers 64 bytes during each memory operation. Thus, the sustained bandwidth between the processor and the network is 8450 MB/s, which is approximately equal to the bandwidth of the Seastar2 link to the processor. With 16 processors, we see a 13× speed-up in total sustained memory bandwidth. On the UltraSPARC T2, we achieve a peak performance of 8649 MB/s for a 16-thread (binding two threads per core) run. The performance starts dropping after that point, likely due to contention for the single memory unit per core. The parallel speedup over a serial implementation is just 9.45. Note that our STREAM implementations do not have any architecture-specific optimizations, and we just run the code with the same optimization flags as the SSCA#2 benchmark.

RandomAccess measures the rate of random integer updates to memory. This computation is representative of a kernel with no spatial or temporal locality. On the XMT, we achieve a peak performance of $338.6 * 10^{-3}$ GUPS, with a relative speedup of 10.29. The UltraSPARC T2 peak performance of $121.13 * 10^{-3}$ is achieved for a 48-thread run, and the relative speedup is 21.13. Note that the performance saturates after 32 threads on the UltraSPARC T2. In future work, we will investigate reasons for the drop in parallel performance.

Comparing the peak performance of the three benchmarks, we observe that the 16-processor XMT is faster than the UltraSPARC T2 by a factor of 1.54, 1.77, 2.79 for STREAM, SSCA#2, and RandomAccess respectively. On the XMT, the 16-processor parallel speedups of RandomAccess and SSCA#2 are comparable, while STREAM achieves a slightly higher speedup of 13. However, on the Niagara2, the relative speedup for SSCA#2 is nearly 40, and is significantly higher than the speedup achieved by STREAM (9.45) and RandomAccess (21.13).

Performance on x86 cache-based multicore systems. Our C/OpenMP multithreaded implementation can also be executed on cache-based x86 architectures without any modifications. We first ran the serial version of the SSCA#2 benchmark on a single core of



(a) Node degree vs. frequency **(b)** Node degree vs. Centrality score

Figure 4: Centrality analysis of the IBDb movie actor data set (1.54 million vertices and 78 million edges). Vertices represent actors, and edges correspond to actors co-starring in movies.

a quad-core 2.0 GHz Intel Xeon *Harpertown* processor. The code was built with the Intel C compiler version 11.0. Figure 2(b) plots the TEPS score achieved as the problem size is varied from SCALE 13 (8192 vertices) to SCALE 21 (two million vertices). We observe that the performance drops by an order of magnitude when the graph does not fit in the L2 cache of the system.

We also ran the parallel version of the benchmark on a 2.4 GHz single-socket quad-core Intel Xeon workstation with 8 GB of main memory. The benchmark was compiled using the GNU C compiler version 4.2.1. For a graph of SCALE 21, the SSCA#2 betweenness centrality kernel performance on one and four threads is 9.1 million and 15.2 million respectively, which corresponds to a relative speedup of 1.67. We will investigate parallel performance bottlenecks on x86 architectures in future work.

Betweenness computation on the IMDb dataset. To demonstrate the applicability of our implementation to real-world data analysis, we perform a large-scale approximate betweenness calculation on a network constructed from the Internet Movie Database (IMDb) [16]. We obtained raw text data files that make up IMDb and used the actor, actress, and movie data to construct a graph with vertices representing actors (and actresses), and edges connecting actors who have co-starred in a movie. We removed television shows as well as uncredited roles.

The input dataset we developed produces an undirected graph with 1.54 million vertices (movie actors) and 78 million edges. Plotting the node degree versus frequency (Figure 4(a)), we observe an unbalanced degree distribution that can be approximated by a power-law graph, which is typical of real-world social networks [6]. On the XMT, the approximate betweenness calculation takes about 83.6 seconds (using 256 randomly selected sources). The same problem run on a 2.4 GHz quad-core Intel Xeon workstation requires 398 seconds to complete. Thus, we achieve a speedup of 4.75 using the 16-processor XMT. Studying the distribution of centrality scores (Figure 4(b)), it is interesting to note that the degree of the

actor with the highest centrality score is one order of magnitude less than highest degree in the network. It is also interesting that a number of actors appearing in movies with only 10 or fewer other actors have centrality scores 100 times less then the maximum, but a million times greater than the minimum. The actors of low-degree but high betweenness (or a high number of shortest paths passing through them) are particularly of interest in a social network, as we cannot identify them by a linear-time computation. Approximate centrality computation reveals these actors, and it is important to note that there are quite a few of these in the IMDb network.

5 Conclusions and Future Work

We present a new parallel approach for computing betweenness centrality, and conduct a detailed performance analysis of two optimized multithreaded implementations for the Cray XMT and the Sun UltraSPARC T2. We show that the new algorithm has a lower synchronization overhead and better cache locality compared to the previous approach, and this results in more than a 2× performance improvement for parallel runs on both the systems.

This paper raises several interesting questions that we hope to answer in future work. With our new algorithm, we have eliminated a few performance bottlenecks for any centrality implementation on the XMT. However, it is still unclear on how this approach will scale on larger XMT systems. The single-socket UltraSPARC T2 performance is promising, but non-uniform memory access (NUMA) effects on multi-socket Sun servers will likely lead to a significant performance slowdown. Continued performance scaling on larger systems may necessitate changes in the graph data structures we are using, as well as the data representations in the centrality algorithms. We are also working on adapting this fine-grained parallel betweenness centrality algorithm to develop optimized implementations for local-store memory based architectures such as the IBM Cell processor.

Acknowledgments

This work was supported in part by the PNNL CASS-MT Center, NSF Grant CNS-0614915, and the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We would like to thank PNNL for providing access to the Cray XMT, and Cray Inc. for access to the MTA-2. We are grateful to Jonathan Berry, Bruce Hendrickson, John Feo, Jeremy Kepner, and John Gilbert, for discussions on large-scale graph analysis and algorithm design for massively multithreaded systems.

References

[1] D.A. Bader, J.R. Gilbert, J. Kepner, and K. Madduri. HPC graph analysis benchmark, 2006. http://www.graphanalysis.org/benchmark.

[2] D.A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Proc. 5th Workshop on Algorithms and Models for the Web-Graph (WAW2007)*,

volume 4863 of *Lecture Notes in Computer Science*, pages 134–137, San Diego, CA, December 2007. Springer-Verlag.

[3] D.A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*, Columbus, OH, August 2006. IEEE Computer Society.

[4] D.A. Bader, K. Madduri, G. Cong, and J. Feo. Design of multithreaded algorithms for combinatorial problems. In S. Rajasekaran and J. Reif, editors, *Handbook of Parallel Computing: Models, Algorithms, and Applications*, chapter 31, pages 1–29. Chapman and Hall/CRC, 2007.

[5] D.A. Bader, K. Madduri, J.R. Gilbert, J. Kepner, T. Meuse, and A. Krishnamurthy. Scalable synthetic compact applications for benchmarking high productivity computing systems. *CTWatch Quarterly*, 2(4B):41–51, 2006.

[6] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

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

[8] 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, April 2004. SIAM.

[9] D.G. Chavarría-Miranda, A. Márquez, J. Nieplocha, K.J. Maschhoff, and C. Scherrer. Early experience with out-of-core applications on the Cray XMT. In *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP'08)*, Miami, FL, April 2008.

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

[11] G. Cong and S. Sbaraglia. A study on the locality behavior of minimum spanning tree algorithms. In *Proc. 13th Int'l Conf. on High Performance Computing (HiPC 2006)*, Bangalore, India, December 2006. Springer-Verlag.

[12] Cray, Inc. Cray XMT platform. http://www.cray.com/products/xmt, 2007.

[13] DARPA Information Processing Technology Office. High productivity computing systems project, 2004. http://www.highproductivity.org.

[14] R. Guimerà, S. Mossa, A. Turtschi, and L.A.N. Amaral. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proceedings of the National Academy of Sciences USA*, 102(22):7794–7799, 2005.

[15] D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265–278, 2001.

[16] IMDb.com, Inc. The internet movie database. http://www.imdb.com/interfaces, 2008.

[17] H. Jeong, S.P. Mason, A.-L. Barabási, and Z.N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411:41–42, 2001.

[18] P. Konecny. Introducing the Cray XMT. In *Proc. Cray User Group meeting (CUG 2007)*, Seattle, WA, May 2007. CUG Proceedings.

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

[20] K. Lang. Finding good nearly balanced cuts in power law graphs. Technical report, Yahoo! Research, 2004.

[21] K. Lang. Fixing two weaknesses of the spectral method. In *Proc. Advances in Neurals*

Information Proc. Systems 18 (NIPS 2005), Vancouver, Canada, December 2005.

[22] F. Liljeros, C.R. Edling, L.A.N. Amaral, H.E. Stanley, and Y. Åberg. The web of human sexual contacts. *Nature*, 411:907–908, 2001.

[23] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[24] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) benchmark suite. SC06 Conference Tutorial, Nov 2006. http://icl.cs.utk.edu/hpcc.

[25] K. Madduri, D.A. Bader, J.W. Berry, J.R. Crobak, and B.A. Hendrickson. Multithreaded algorithms for processing massive graphs. In D.A. Bader, editor, *Petascale Computing: Algorithms and Applications*, chapter 12, pages 237–262. Chapman and Hall/CRC, 2007.

[26] J.D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Tech. Comm. Comput. Arch. Newslett, 1995. http://www.cs.virginia.edu/stream.

[27] J. Nieplocha, A. Márquez, J. Feo, D.G. Chavarría-Miranda, G. Chin Jr., C. Scherrer, and N. Beagley. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *Proc. 4th Conf. on Computing Frontiers*, Ischia, Italy, May 2007.

[28] Sun Microsystems. UltraSPARC T2 processor. http://www.sun.com/processors/UltraSPARC-T2, 2007.