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

Kamesh Madduri
*Computational Research Division*
*Lawrence Berkeley National Laboratory*
*Berkeley, CA, USA*

David Ediger Karl Jiang David A. Bader
*College of Computing*
*Georgia Institute of Technology*
*Atlanta, GA, USA*

Daniel Chavarría-Miranda
*High Performance Computing*
*Pacific Northwest National Laboratory*
*Richland, WA, USA*

## Abstract

*We present a new lock-free parallel algorithm for computing betweenness centrality of massive complex networks that achieves better spatial locality compared with previous approaches. Betweenness centrality is a key kernel in analyzing the importance of vertices (or edges) in applications ranging from social networks, to power grids, to the influence of jazz musicians, and is also incorporated into the DARPA HPCS SSCA#2, a benchmark extensively used to evaluate the performance of emerging high-performance computing architectures for graph analytics. We design an optimized implementation of betweenness centrality for the massively multithreaded Cray XMT system with the Threadstorm processor. For a small-world network of 268 million vertices and 2.147 billion edges, the 16-processor XMT system achieves a TEPS rate (an algorithmic performance count for the number of edges traversed per second) of 160 million per second, which corresponds to more than a 2× performance improvement over the previous parallel implementation. We demonstrate the applicability of our implementation to analyze massive real-world datasets by computing approximate betweenness centrality for the large 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 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 tends to be highly memory-intensive: codes 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 [15], due to the fact that massive graphs that occur in real-world applications are often not amenable to a balanced partitioning among processors of a parallel system [13].

The DARPA HPCS [8] graph theory benchmark was introduced as part of the Scalable Synthetic Compact Applications (SSCA) benchmark suite [4], 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 (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) [6] random graph generation algorithm to generate input data sampled from a Kronecker product that are representative of real-world networks with a small-world topology. The most complex kernel of the SSCA#2 benchmark is the 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 $\sigma_{st}$ 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:

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

Intuitively, betweenness measures the control a vertex has over communication in the network, and can be used to identify critical vertices in the network (a similar measure of betweenness centrality exists for edges). 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 [11], study of sexual networks and AIDS [14], identifying key actors in terrorist networks [7], organizational behavior, supply chain management processes, and transportation networks [9].

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 [16]. In this paper, we focus on optimizing vertex betweenness centrality on the massively multithreaded Cray XMT system [12]. In Section 3, we discuss the architectural features of this system, and present centrality implementation details and architecture-specific optimizations. Our key performance results are summarized below:

- On a 16-processor XMT system with 128 GB of main memory, 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) rate of 160 million per second.
- The parallel speedup of the XMT implementation is on an average 10.5 on 16 processors for large-scale networks.
- 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 [10]. 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

We can evaluate the betweenness centrality of a vertex $v$ (defined in Eq. 1) by determining 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.

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 $s$. For unweighted and sparse real-world networks, Brandes [5] presents a sequential algorithm to compute the betweenness centrality score for all vertices in $\mathrm{O}(mn)$ time and $\mathrm{O}(m + n)$ space. The main idea is to perform $n$ breadth-first graph traversals, and augment each traversal to compute the number of shortest paths 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:

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

Thus, the sequential algorithm computes betweenness in $\mathrm{O}(mn)$ 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 Eq. 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. Algorithm 1 gives a high-level schematic describing the two main steps in each iteration of the fine-grained parallelization discussed in [3]. Starting from 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-

**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** $v \in V$ **in parallel do**
2: $BC[v] \leftarrow 0$
3: **for all** $s \in V$ **do**
*I. Initialization*
4: **for all** $t \in V$ **in parallel do**
5: $P[t] \leftarrow$ empty multiset, $\sigma[t] \leftarrow 0$, $d[t] \leftarrow -1$
6: $\sigma[s] \leftarrow 1$, $d[s] \leftarrow 0$
7: $phase \leftarrow 0$, $S[phase] \leftarrow$ empty stack
8: push $s \rightarrow S[phase]$
9: $count \leftarrow 1$
*II. Graph traversal for shortest path discovery and counting*
10: **while** $count > 0$ **do**
11: $count \leftarrow 0$
12: **for all** $v \in S[phase]$ **in parallel do**
13: **for** each neighbor $w$ of $v$ **in parallel do**
14: **if** $d[w] < 0$ **then**
15: push $w \rightarrow S[phase + 1]$
16: $count \leftarrow count + 1$
17: $d[w] \leftarrow d[v] + 1$
18: **if** $d[w] = d[v] + 1$ **then**
19: $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
20: append $v \rightarrow P[w]$
21: $phase \leftarrow phase + 1$
22: $phase \leftarrow phase - 1$
*III. Dependency accumulation by back-propagation*
23: $\delta[t] \leftarrow 0 \ \forall \ t \in V$
24: **while** $phase > 0$ **do**
25: **for all** $w \in S[phase]$ **in parallel do**
26: **for all** $v \in P[w]$ **do**
27: $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
28: $BC[w] \leftarrow BC[w] + \delta[w]$
29: $phase \leftarrow phase - 1$

code for implementing the traversal and dependency accumulation steps. We assume that our target architecture supports two atomic operations: `compare_and_swap` and `fetch_and_add`. `fetch_and_add` atomically increments a memory 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 $\delta$ and $BC$ values are stored as floating-point numbers.

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

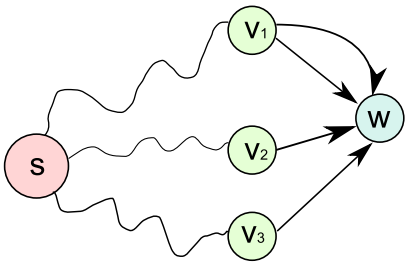
```
for all v ∈ S[phase] in parallel do
  for each neighbor w of v in parallel do
    dw ← compare_and_swap(&d[w], −1, phase + 1)
    if dw = −1 then
      p ← fetch_and_add(&count, 1)
      Insert w at position p of S[phase + 1]
      dw ← phase + 1
    if dw = phase + 1 then
      p ← fetch_and_add(&Pcount[w],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

To the best of our knowledge, 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 the case of the serial betweenness centrality algorithm, the appends tend to be cache-unfriendly memory accesses, as we are touching $w$ 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.




| *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.

**Algorithm 4** Pseudo-code for the augmented breadth-first graph traversal step in our new parallel algorithm (replacing lines 12–20 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 consider 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 24–29 in Algorithm 1).

```
phase ← phase − 1
for all w ∈ S[phase] in parallel do
  dsw ← 0
  sw ← σ[w]
  for all v ∈ Succ[w] do
    dsw ← dsw + (sw/σ[v])(1 + δ[v])
  δ[w] ← dsw
  BC[w] ← BC[w] + dsw
```

## 3. Centrality Implementations on the Cray XMT

We next demonstrate the empirical performance of our improved betweenness centrality algorithm by modifying the SSCA#2 graph analysis benchmark. Kernel 4 of the SSCA#2 benchmark computes exact or 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 rate 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:

$$BC\ TEPS = \frac{7n \cdot 2^{K4Approx}}{t} \text{ edges/second} \qquad (3)$$

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

We next present betweenness performance optimizations for the Cray XMT. We discuss implementation details on

cache-based multicore architectures such as the Sun Niagara blades in an extended version of this paper [17].

### 3.1. The Cray XMT

The Cray XMT [12] 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 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.

### 3.2. 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 modify slightly 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 emulate `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 the 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 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 only the outer loop results in a reasonably load-balanced work distribution.

Through inspection of our implementation, we determine that the number of memory operations per iteration of centrality for SSCA#2 R-MAT graphs is approximately $6.75m$, 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 on which we ran our experiments.

## 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 rate, as defined in Eq. 3.

We report Threadstorm performance results on a 16-processor 500 MHz Cray XMT system with 128 GB memory. We build our code using the C compiler of the

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

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

Cray XMT programming environment (version 5.2.1) and flags `-par -O3`. We also compare performance with a 40 processor 220 MHz Cray MTA-2 system with 140 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 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 achieve a TEPS rate of 117 million per second for a graph of SCALE 21 (2.09 million vertices and 16.77 million edges). The loops in the centrality kernel are annotated using `#pragma mta assert no dependence` (indicating that the loop iterations can be scheduled independently) and the compiler automatically parallelizes them. The static analysis tool Canal reports that 60 streams are requested on each processor, for each loop. We experimented with several values between 60 and 120, and achieved peak performance of 132 million (an increase of nearly 13%) with 100 streams on 16 processors.

**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 the worst. 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 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.

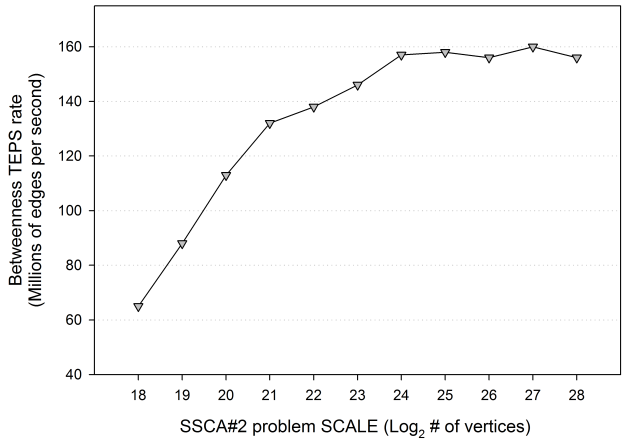



Figure 2. SSCA#2 betweenness kernel performance on the Cray XMT system.

**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. Figure 2 plots the performance on 16 processors of the XMT for SSCA#2 graphs of various sizes ($SCALE = 18$ to $SCALE = 28$). The largest problem instances we could run on the XMT was a graph of scale SCALE 28. 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.

**Parallel performance.** Since the XMT system 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 plots the TEPS rate achieved 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

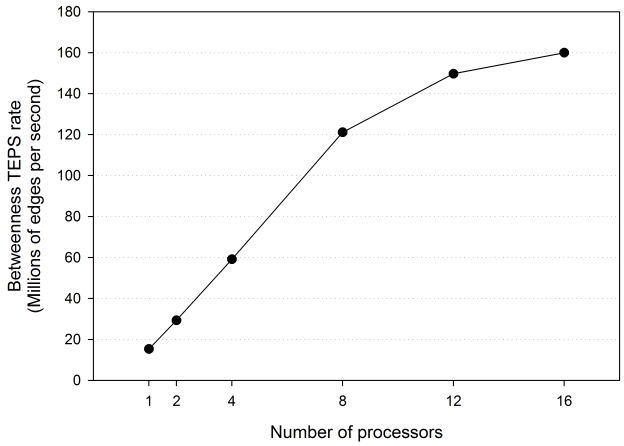



Figure 3. Parallel performance of SSCA#2 betweenness kernel on the Cray XMT for a graph of 16.77 million vertices and 134.21 million edges (SCALE 24).

Table 2. A comparison of the performance of the new algorithm and the old approach on the Cray XMT for a graph of SCALE 24.

| | |
|---|---|
| Original running time (seconds) | 434.84 |
| Original TEPS rate (Millions of edges per second) | 69.14 |
| Improved running time (this paper, seconds) | 187.90 |
| Speedup factor | 2.31 |

replicating the number of stacks to reduce contention. 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.

**Performance comparison with older algorithm.** In Table 2, we report the performance of the previous parallel approach (using predecessor sets) on the XMT for the same SSCA#2 graph instance of SCALE 24. We observe that the XMT implementation is significantly faster than the corresponding older implementation. We observe the same trend for larger problem instances as well.

**Performance comparison with the Cray MTA-2.** From the results in Table 3, 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.

**Betweenness computation on the IMDb dataset.** To demonstrate the applicability of our implementation to real-world data analysis, we perform a large-scale approximate

Table 3. Performance of the SSCA#2 betweenness centrality kernel for a graph of SCALE 24 on the Cray XMT and the Cray MTA-2.

| System/Configuration | TEPS rate (Millions of edges per second) |
|---|---|
| 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 |

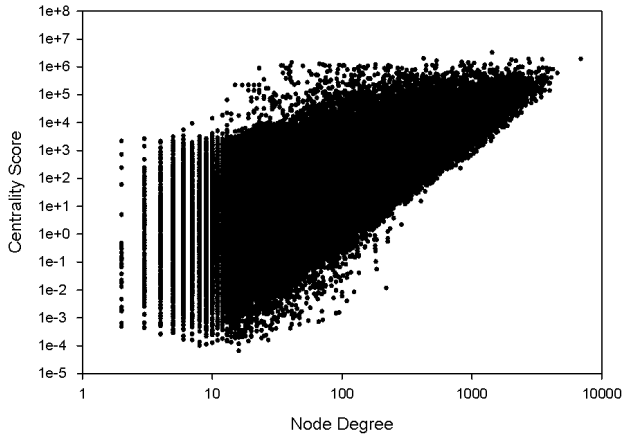



Figure 4. Centrality analysis (Node degree vs. approximate betweenness value) of the IBDb movie actor data set (1.54 million vertices and 78 million edges). Vertices represent actors, and edges correspond to actors co-starring in movies.

betweenness calculation on a network constructed from the Internet Movie Database (IMDb) [10]. 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. 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), 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 an optimized multithreaded implementation for the Cray XMT. 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 $2\times$ performance improvement for parallel runs.

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. 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. Bader, J. Gilbert, J. Kepner, and K. Madduri, "HPC graph analysis benchmark," 2006, http://www.graphanalysis.org/benchmark.

[2] D. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Proc. 5th Workshop on Algorithms and Models for the Web-Graph (WAW2007)*, ser. Lecture Notes in Computer Science, vol. 4863. San Diego, CA: Springer-Verlag, December 2007, pp. 134–137.

[3] D. 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: IEEE Computer Society, Aug. 2006.

[4] D. Bader, K. Madduri, J. Gilbert, J. Kepner, T. Meuse, and A. Krishnamurthy, "Scalable synthetic compact applications for benchmarking high productivity computing systems," *CT-Watch Quarterly*, vol. 2, no. 4B, pp. 41–51, 2006.

[5] U. Brandes, "A faster algorithm for betweenness centrality," *J. Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

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

[7] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-based technologies for intelligence analysis," *Communications of the ACM*, vol. 47, no. 3, pp. 45–47, 2004.

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

[9] R. Guimerà, S. Mossa, A. Turtschi, and L. Amaral, "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles," *Proceedings of the National Academy of Sciences USA*, vol. 102, no. 22, pp. 7794–7799, 2005.

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

[11] H. Jeong, S. Mason, A.-L. Barabási, and Z. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, pp. 41–42, 2001.

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

[13] K. Lang, "Finding good nearly balanced cuts in power law graphs," Yahoo! Research, Tech. Rep., 2004.

[14] F. Liljeros, C. Edling, L. Amaral, H. Stanley, and Y. Åberg, "The web of human sexual contacts," *Nature*, vol. 411, pp. 907–908, 2001.

[15] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[16] K. Madduri, D. Bader, J. Berry, J. Crobak, and B. Hendrickson, "Multithreaded algorithms for processing massive graphs," in *Petascale Computing: Algorithms and Applications*, D. Bader, Ed. Chapman and Hall/CRC, 2007, ch. 12, pp. 237–262.

[17] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarría-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," Lawrence Berkeley National Laboratory, Tech. Rep., 2009.