Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 399 – 408

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Faster Betweenness Centrality Based on Data Structure Experimentation

Oded Green, David A. Bader

College of Computing
Georgia Institute of Technology
Atlanta,Georgia, USA

Abstract

Betweenness centrality is a graph analytic that states the importance of a vertex based on the number of shortest paths that it is on. As such, betweenness centrality is a building block for graph analysis tools and is used by many applications, including finding bottlenecks in communication networks and community detection. Computing betweenness centrality is computationally demanding, $O(V^2 + V \cdot E)$ (for the best known algorithm), which motivates the use of parallelism. Parallelism is especially needed for large graphs with millions of vertices and billions of edges. While the the memory requirements for computing betweenness are not as demanding, $O(V + E)$ (for the best known sequential algorithm), these bound increase for different parallel algorithms. We show that is possible to reduce the memory requirements for computing betweenness centrality from $O(V + E)$ to $O(V)$ at the expense of doing additional traversals. We show that not only does this not hurt performance it actually improves performance for coarse grain parallelism. Further, we show that using the new approach allows parallel scaling that previously was not possible. One example is that the new approach is able to scale to 40 x86 cores for a graph with 32M vertices and 2B edges, whereas the previous approach is only able to scale upto 6 cores because of memory requirements. We also do analysis of fine-grain parallel betweenness centrality on both the x86 and the Cray XMT.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science

Keywords: Parallel algorithms, Graph algorithms, Betweenness Centrality, Optimizations, Experimental algorithms

1. Introduction

Betweenness centrality is computed for graphs $G = (V, E)$ where V represents the set of vertices and E represents the set of links between the vertices. The graph can be either directed or undirected and either weighted or unweighted. Betweenness centrality is applicable to many fields. A few applications that use betweenness centrality as a building block are 1) community detection in social networks[1], 2) brain network analysis [2], 3) transportation network analysis [3], and 4) deploying detection devices in communication networks [4].

A path between source vertex $s \in V$ and the destination vertex $t \in V$ is defined as the sequence of vertices $s, v_1, v_2, .., v_k, t$ such that $(v_i, v_{i+1}) \in E$ for the entire sequence. The length of a path is the sum of the weights of all the edges in the path. For an unweighted graph, the length of the path is the number of edges in the sequence. The shortest path between two vertices, also known as the geodesic, is the sequence of vertices that has the smallest summed weight. It is worth noting that there can be more than one shortest path connecting any pair of vertices.

Finding the shortest path from a single vertex (source) is known as the Single Source Shortest Path (SSSP) problem. Finding the shortest paths from all the vertices in the graph is known as the All Pairs Shortest Path (APSP) problem. The complexity of computing APSP using the Floyd-Warshall algorithm [5] [6] is $O(V^3)$. For a more detailed discussion on SSSP and APSP the reader is referred to [7].

*Corresponding author: Oded Green
E-mail address: ogreen@gatech.edu.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.203

Centrality is used for finding important vertices/edges in graphs. In social networks the vertices refer to people/actors and the edges refer to relationships, where the relationship is dependent on the type of social network. In a communication network, the vertices might be servers and the edges might be physical connections between the servers. For email networks, the vertices will be the senders/receivers and the edges refer to emails sent between the sender and receiver.

Freemen[8] defined betweenness centrality as the number of shortest paths between s and t going through a vertex v ,denoted $\sigma_{st}(v)$, divided by all the shortest paths between s and t (including those that do not go through v),denoted σ_{st}. Then betweenness centrality is computed as following [8]:

$$C_B(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}. \quad (1)$$

2. Related Work

2.1. Dependency-Accumulation Based Betweenness Centrality

In [9], Brandes presents a fast algorithm for computing betweenness centrality based on a dependency accumulation technique which accesses the vertices in the reverse order of the BFS traversal. The dependency accumulation technique present by Brandes is faster than previous approaches that required summing up all of the pair-wise dependencies due to its recursive nature.

The algorithm for computing betweenness centrality presented in [9] is made up of four stages. The first two stages, Stage 0 and Stage 1, are data structure initialization stages, where Stage 0 is a global initialization stage and Stage 1 is local initialization that is completed once for each vertex in the graph. In Stage 0 the output of the algorithm is initialized, which is the betweenness centrality score of each vertex, and set to zero. Stages 1, 2, and 3 are executed for each vertex in the graph. Each vertex is considered a root in the iteration over the vertices. In Stage 1, the data structures that will be used in Stages 2 and Stage 3 are initialized. This includes a stack, queue, and three additional arrays. The first array, σ, counts the number of shortest paths from each vertex to the root of the current shortest path tree, s. The second array, d, measures the distance of each vertex from the root. As the graph is unweighted, this is the minimum number of edges between the vertex and the root. We refer to the distance as the level in which the vertex is in the BFS tree. Initially the distance of all vertices from the root is set to ∞. The third array, P, is an array of linked lists. Each vertex v has a linked list $P[v]$, that contains all the vertices that precede v in the BFS traversal. These are the parent vertices of v. Note that the wrong selection of the data structure for the parent list can significantly reduce performance. This will be further discussed in Section 3

Stage 2 and Stage 3 are the key components of the betweenness centrality computation. Stage 2 is a BFS traversal from a given root that finds the shortest path to the remaining vertices. In this stage, each element is placed in a queue when it is found. It is later placed in the stack when it is dequeued from the queue. As part of the BFS traversal the distance from the root vertex, s, to each vertex is also computed. For each vertex, v, found in the BFS traversal there is a list of parental vertices that are all one hop closer to the root. Thus, all of v's shortest paths goes through his parents and these are accumulated in $\sigma[v]$.

By setting s to be a specific value (i.e. the root of the tree) it is possible to compute both numerator and denominator $\sigma_t(v)$ and σ_t using the BFS traversal for each root vertex s.

Stage 3 computes betweenness centrality using the dependency accumulation technique of Brandes [9]. The pair-dependency for a pair of vertices s, t is defined as following:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}. \quad (2)$$

Using Eq. (2) with Eq. (1) changes the computation of betweenness centrality based on the pair-dependency:

$$C_B(v) = \sum_{s \neq t \neq v} \delta_{st}. \quad (3)$$

In [9] the following recursive relationship, a.k.a the dependency accumulation, is shown and proved:

$$\delta_s(v) = \sum_{\{w | v \in P_s(w)\}} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)). \quad (4)$$

The immediate outcome of this is that it is no longer necessary to sum all the pair-dependencies as they follow a recursive relation. In addition to this, it is possible to compute each of the $\delta_s(w)$, by computing the shortest path from a given root, s, to the rest of the graph using a single source shortest path algorithm.

The time complexity of a BFS is $O(V + E)$. The time complexity of the dependency accumulation is also $O(V + E)$ as the maximal number of steps is bound by the number of parents, $O(E)$, and the vertices accessed,

$O(V)$. As this computation is done for each vertex, the time complexity is $O(V^2 + VE)$. For all purposes this is $O(VE)$.

The memory requirement for the stack, queue and the arrays σ and d are $O(V)$ as the sizes of these data structures are bound by the number of vertices in the graph V. The memory needed by the array of linked lists is bound by the number of edges in the graph $O(V + E)$ as the maximum number of parents a vertex has is bound by the number of edges it has. The sum of all the parents is bound by the total number of edges in the graph.

We give the following memory complexity analysis for the sequential algorithm. As each BFS traversal is computed independently, only a single copy of these data structures need to be maintained, which is $O(V + E)$. The memory required by the array C_B for computing vertex betweenness centrality is bound by the number of the vertices $O(V)$. The memory required for maintaining the parent array is bound by the number of the edges $O(E)$. Therefore, the total memory requirement of this algorithm is $O(V + E)$.

2.2. Betweenness Centrality Algorithms

In Madduri *et al.* [10] the first parallel implementation for computing betweenness centrality is presented. This algorithm uses a multi-level parallelism to achieve fine-grain parallelism. In Madduri *et al.* [11] the authors make changes to the data structure used by the algorithm which removes the need for locks (atomic instructions are still needed) in the dependency accumulation stage by maintaining a successor list for each vertex rather that a predecessor list. They show a 2.3X speedup using the successor list over the predecessor list on the Cray XMT. In their anaylsis, the showed linear scaling upto 8 Cray XMT processors. The scaling was no longer linear when the number of processors was increased from 8 to 16. We offer some additional insights on these scaling issues in Section 4.

In Tan *et al.* [12] an additional parallel algorithm for computing betweenness centrality is presented. Their algorithm was tested using an x86 platform. A taxonomy of the different types of parallel granularity for betweenness centrality is also given. Both [10, 12] have a $O(V + E)$ space complexity. Tan *et al.* [13] present several optimization strategies for computing betweenness centrality on the IBM Cylcops64.

In Bader *et al.* [14] the authors suggest reducing the complexity requirements of betweenness by computing an approximation. This is done by selecting a subset of vertices and computing betweenness centrality for these vertices alone. In their paper, the authors show that the approximation gives considerably good results for artificial networks. In [11, 15] execution times are presented for the computation of approximate betweenness centrality for graphs with an edge count of half a billion. In Buluç and Gilbert [16] the authors show a framework, Combinatorial BLAS, for computing betweenness centrality using algebraic computation. Using Combinatorial BLAS, they show that the computation of approximate betweenness centrality is scalable and can be distributed to multiple cores.

In Green *et al.* [17] the authors present an algorithm for computing betweenness for streaming graphs. Streaming graphs are graphs that change over time by edge insertion and deletion. Vertices can also be inserted-in or deleted-from the graph. The algorithm maintains BFS like trees between the updates. As such, the algorithm has a higher memory bound of $O(V \cdot (V + E))$. Using the methods discussed in this paper we will reduce those bounds to $O(V^2)$ for exact betweenness centrality and $O(K \cdot V)$ for approximate betweenness centrality.

2.3. Parallel Betweenness Centrality Taxonomy

Complexity analysis for parallel algorithms is dependent on additional parameters such as the granularity of the algorithm and the synchronization model. In [12], the authors suggest three different granularity of concurrency: 1) coarse-grained, 2) medium-grained, and 3) fine-grained. P denotes the number of processors used for the parallel computation.

For coarse-grained granularity, each of the processors gets a subset of the vertices of the graph and executes the algorithm suggested by Brandes. Each of these processors maintains the mentioned data structures, thus the total memory complexity is $O(P \cdot (V + E))$. The additional parameter in the complexity greatly limits the usage of this method.

The medium-grained granularity achieves parallelism by exploring the neighbors of all the vertices in a given frontier in parallel. This means that a parallel for-loop is executed on all the elements in a given level of a BFS tree for one specific root. As each vertex potentially has a different number of neighbors, this approach can lead to uneven load balancing.

To overcome this problem, it is possible to add an additional level of parallelism in which the neighbors of a given vertex are traversed concurrently. This is the fine-grained approach.

Consequently, the medium-grained and fine-grained granularity have a lower memory complexity of $O(V + E)$ as only single a copy of the data structures is shared by the processors. This lower memory-bound comes at the expense of using atomic instructions and locks. Obviously, atomic instructions and locks impact the performance of the algorithms.

2.4. Real World Graph Properties

In Albert *et al.* [18] the authors present the small-world phenomena which states the distance between two vertices in the graph is a small number of hops away. In Broder *et al.* [19] the authors show that World Wide Web (WWW) has one huge connected component that contains 90% of the vertices in the graph. In Barabasi *et al.* [20] also show that the edge distribution follows a power law. The work of Leskovic *et al.* [21] confirms that many real world networks have these properties. In Leskovec *et al.* [21] the effective diameter is defined as the 90th percentile distance of all the vertices. They also show show that the effective diameter for social networks is considerably smaller than the actual diameter (the maximal distance between any two vertices).

3. Data Structure Experimentation

3.1. Relevant Terminology

In Brandes's algorithm [9], each vertex maintains a list of all its neighbors that are in the level above it. We will refer to these list as either the parent list or the predecessor list, interchangeably, when referring to Brandes's algorithm. In Madduri *et al.* [11], the vertices maintain a list of all the the neighbors that are in the level below it. We will refer to these list as either the children list or the successor list, interchangeably, when referring to this algorithm. Obviously, these lists can only be a subset of the adjacency for a given vertex. The final implementation of the algorithm is dependent on the selection of either the predecessor or successor approach. We refer the reader to both [9] and [11] for further reading. We note that for a sequential implementation, there is no preference to either of these approaches.

As these algorithms share many implementation properties, they do not require separate explanations on many common issues. As such we will refer to these lists as ancestry lists whenever the explanation is relevant for both the approaches. In Section 4 we will specify the exact approaches that we used for each test.

3.2. Ancestor-List implementation

It is well known that the implementation of a data structure can significantly influence the performance of an algorithm. In some cases, including the case of betweenness centrality, additional parameters about the data structure are known such as the maximal size it will grow and access pattern to the data structure. For Brandes's algorithm we know both of these. The length of each ancestor list is bounded by the vertex's incoming edges. The access pattern is also known - in the BFS stage, elements are pushed to the end of the list and in the dependency accumulation stage traverse the list from the beginning to the end.

Given this additional information, the ancestor lists can be implemented with an array. This is done by pre-allocating an array of $|E|$ element where each vertex, v, is allocated space in the array based on its adjacency. In the Results section we show that despite the increase in the memory requirements, the performance of the array based list is significantly better than the linked-list, as can be expected. Both these implementations have a memory bound of $O(V + E)$.

Obviously the lists can also be implemented using dynamic memory allocations such as a linked-list implementation. The linked-list approaches suffers from several performance pit falls. The first, insertion of an element into the list requires adding a new link to the end of the list. If the node is dynamically allocated, this requires a system call which is usually costly. If the link is added to the list by getting a 'new' link from a pool of preallocated link, then the pool needs to be the size of $O(E)$. As such, it is preferable to use the array based approach because of spacial locality and the reduced number of memory allocations.

3.3. Neighbor-Traversal

In this section we present an alternative to maintaining the ancestry lists. This approach reduces the memory requirements from $O(V + E)$ to $O(V)$, increases parallel scalability, reduces synchronization requirement and improves performance for sparse graph. We will see that this new approach, which we refer to as neighbor-traversal, is more computationally demanding the the ancestral based approaches, yet it does not increase the time complexity of the betweenness centrality algorithm of $O(V^2 + V \cdot E)$. In the neighbor traversal approach we eliminate the ancestry list altogether. As such, lists for the vertices are not created during the BFS traversal. Consequently, in the dependency accumulation stage all the neighbors of a given vertex are traversed rather than just the ancestors.

Before presenting the advantages of this approach, we show that the complexity of neighbor traversal approach does not increase the time complexity of the algorithm. As such, each vertex that is found in the BFS traversal for the first time is placed in a stack and is popped out in the dependency accumulation in the reverse order that it was found. Each vertex is popped out exactly once. This is the same is when maintaining the ancestry lists. Even for the worst case where all the neighbors of all the vertices are traversed, an upper-bound of $O(V + E)$ is given. This is the same as the complexity of algorithm suggested by Brandes for a single root. The upper bound complexity of both ancestry lists and neighbor traversals are not influenced by the graph density .

As it is no longer required to maintain the ancestry lists/array, the memory bound of betweenness centrality goes down from $O(V + E)$ to $O(V)$. Note that the only data structure that requires $O(E)$ memory is the ancestor list. The remaining data structures only require $O(V)$ memory. As these memory requirements do not change, the memory requirements for computing betweenness centrality is reduced to $O(V)$.

The coarse-grain parallel algorithm requires that each core maintain a copy of the data structure required by Brandes' algorithm. Each core is responsible for computing the betweenness centrality values for an independent set of vertices. Thus, the storage complexity of the coarse grain granularity using the ancestry list is $O(P \cdot (V + E))$ given that each core needs $O(V + E)$ memory. Consequently, Tan *et al.* state that the coarse-grain parallelism is not scalable due to these memory requirements. Following the removal of the ancestry list, the memory requirement of the coarse-grain approach is reduced from $O(P \cdot (V + E))$ to $O(P \cdot V)$. As such, the coarse grain approach becomes practical and is more scalable than the ancestor approaches. This will become apparent in the Results section.

We now discuss the implications of our new approach on existing algorithms. The medium-grain [12] and fine-grain algorithms [10, 11] require atomic instructions to maintain the ancestry lists. In the first parallel implementation of betweenness centrality [10] the parents list is used. To update the betweenness centrality of a parent, in the dependency accumulation stage, it is necessary to acquire a lock on the parent. In [11] a lock-free algorithm for betweenness centrality is shown where the predecessor list is replaced with a successor list. This approach swaps the roles between the parents and children. For both these algorithms, it is possible to replace the ancestry lists with the neighbor-traversal approach. The benefit of using the neighbor-traversal over ancestry lists for these parallel granularities is atomic instructions are no longer needed for serializing the insertions of elements into the ancestry lists. We note that the neighbor-traversal approach can be implemented either using the predecessor approach or successor approach in which in the dependency accumulation stage the algorithm looks for parents in the level above or children in the level below, respectively. For undirected graphs, both methods will work fine, however, for directed graphs the parent approach can not be used in the dependency accumulation stage of the neighbor-traversal as a vertex will not have access to its incoming edge as is customary for many graph representations such as in the CSR (Compressed Sparse Row) representation. As such, we recommend using the successor approach when using the neighbor-traversal approach.

4. Results

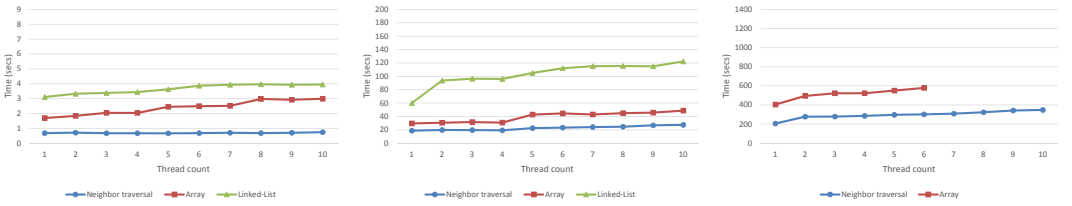
We present experimental results of the traversal-based algorithm in this section on both the x86 architecture and the Cray XMT2. The x86 system used for testing is a multi-processor multi-core Intel server. The Intel system has four processors each containing a 10-core Intel Xeon E7-8870 with at 2.4 GHz clock rate. This gives a total of 40 physicals cores. As the E7-8870 supports Hyper-Threads, the system has a total of 80 logical cores; however, for our testing the Hyper-Threads was disabled. Each processor has 30MB of L3 cache shared by all the cores on that processor. This system has a total of 256 GB of DDR3 RAM clocked at 1066 MHz. The Cray XMT2 is the second generation of the Cray XMT [22] system. The XMT architecture is a shared memory system with a massive thread count. The XMT2 that used for results is the system at the Swiss National Supercomputing Centre that has 64 processors and 2 TB of main memory. Each of the processors contains 128 hardware streams. A different stream can be executed at every clock cycle. The massive thread count enables applications with irregular memory accesses to overcome the latency. In addition to this, the XMT offers low-overhead synchronization through atomic fetch-and-add instructions and full-empty bit memory semantics. As such the XMT is appropriate for graph problems.

4.1. Random Graphs

Recursive Matrix (R-MAT) [23] is a graph generator used to create synthetic scale-free graphs that follow properties found in real-world networks. For simplicity, we present R-MAT using an adjacency matrix. Initially, the adjacency matrix is empty, and edges are added one at a time. For each newly inserted edge, the adjacency matrix is divided into equal-size quadrants where each has a different probability of being selected. One of the quadrants is selected using a random number generator. This quadrant is recursively subdivided into smaller equal-size quadrants from which the next random selection is made. This process is repeated until each quadrant contains only a single element in the adjacency matrix. The last round randomly selects a single element and creates the corresponding edge. The probabilities assigned to the quadrants are designated a, b, c, and d where $a + b + c + d = 1$ and $0 \leq a, b, c, d \leq 1$. The R-MAT generator can also create Erdős-Rènyi (ER) [24][25] random graphs when $a = b = c = d = 0.25$. ER graphs are random graphs with a uniform edge distribution. We use RMAT-I to denote an RMAT graph with 2^I vertices. For example an RMAT-24 graph has $2^{24} = 16M$ vertices.

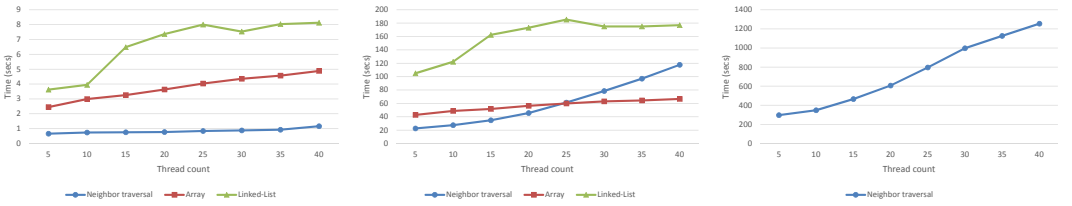
4.2. Coarse Grain Parallelism

In this section we will take a look at experimental results of the coarse grain parallel neighbor-traversal implement ion. The coarse grain approach is appropriate for the x86 architecture as it does not require synchronization



(a) $|V| = 2^{18}, |E| = 64 \cdot |V|$. (b) $|V| = 2^{22}, |E| = 64 \cdot |V|$. (c) $|V| = 2^{25}, |E| = 64 \cdot |V|$.

Fig. 1. Weak scale testing for thread counts of 1 to 10. Lower is better.



(a) $|V| = 2^{18}, |E| = 64 \cdot |V|$. (b) $|V| = 2^{22}, |E| = 64 \cdot |V|$. (c) $|V| = 2^{25}, |E| = 64 \cdot |V|$.

Fig. 2. Weak scale testing for thread counts of 15 to 40. Lower is better.

(locks and atomic instructions) which is usually costly. Furthermore, this architecture does not benefit from increasing the number of threads beyond the number of physicals cores, as such, we limited the thread count by the maximal number of cores available.

On this system we compared the different algorithmic approaches for computing betweenness centrality: neighbor-traversals, parent array, and parent list. As both scalability and performance were the metrics of interest, we ran a variety of tests on the system. We ran the algorithms on multiple scales (vertex count), average edge degree, and thread count. All tests were conducted on undirected graphs. For our performance and scalability tests we did weak scaling in which each core is supplied with a constant amount of work, regardless of the number of the cores. This means that each of the algorithms was responsible for computing betweenness for the same number of roots. For fairness, all the algorithms and respective cores were given the same roots.

For each graph size and average edge adjacency, we checked the scalability for a single processor (upto 10 cores) and for the entire system (all 40 cores). Tests on the single processor allow a speedup analysis that does not require dealing with memory subsystem issues such as cache coherency and cross processor communication. The memory contention becomes apparent for the higher thread counts. Note, that Hyper-Threads option was not used for testing performance, however, the neighbor-traversal approach could scale to 80 threads for many of the test cases because of the considerably lower memory requirements.

We tested the algorithms on multiple graph sizes from RMAT-18 to RMAT-27. For all the graphs between RMAT-18 and RMAT-25, four different average edge degrees of $8, 16, 32$, and 64 were tested. For RMAT-26, average edge degrees of $8, 16$, and 32 were tested. For RMAT-27, average edge degrees of 8 and 16 were tested. All these, were tested for the above mentioned thread counts. We note that for both the array based approach and the linked list approach, there were instances in which the algorithm was not able to complete due to the memory requirements of these methods. Also, there were instances in which the algorithms were stopped as they exceeded the time of the algorithms by a substantive amount of time. We elaborate on these cases.

We divide the tests into 4 sets: small graphs (RMAT-18 - RMAT-19), medium sized graphs (RMAT-20 - RMAT-22), large graphs (RMAT-23 - RMAT-25), and extra large graphs (RMAT-26 - RMAT-27). We analyze each of these independently and in the following order: small graphs, large graphs, extra large graphs, and medium graphs. The order of the explanations moves from the simplest case to the more complicated case.

For the small graphs, all the different algorithms completed for all the different graph sizes, edge counts and thread counts. In Fig. 1 (a) and Fig. 2 (a) the execution times for the three different implementations are depicted. The small graph is a RMAT-18 graph with an average edge degree of 64. Note that for all thread counts the neighbor-traversal approach is faster than the other implementations.

For the larger graphs, there were many cases that the array approach algorithm was not able to scale to the to 30-40 threads due to memory constraints, as such the neighbor-traversal out preformed the array approach for

these thread counts. Note that the amount of memory required for array approach was so large, that there was increased number of cache misses. The linked-list approach simply timed out even for the smaller thread counts. The neighbor-traversal scaled to all 40 cores for all the graph scales and average edge degreess. In Fig. 1 (c) and Fig. 2 (c) the execution times for the three different implementations are depicted. The large graph is a RMAT-25 graph with an average edge degree of 64. The array based approach was not able to scale beyond 6 cores, whereas, the neighbor-traversal approach scaled up to 40 cores. For the extra large graphs it is clear that the neighbor-traversal out preformed the other algorithms, simply as it was able to scale to all the thread counts. For the sake of brevity we, do not present an additional figures, yet, we note that such a figure would be similar with fewer points on the graph for the array approach.

For the medium size graphs a more qualitative explanation is presented given that for some of the test cases the neighbor-traversal approach was outperformed by the array approach. In Table 1 it is possible to see the cases that the array based approach out performed the neighbor-traversal approach. These cases are marked with a red circle. For all remaining graph sizes, average edge degrees and thread counts the neighbor-traversal approach outperformed the array based approach.

In Fig. 1 (b) and Fig. 2 (b) the execution times for the three different implementations are depicted. The medium graph is a RMAT-22 graph with an average edge degree of 64. It is well worth noting that the array approach has the better performance only for the case that two or more processors (20 cores) are used. This can be explained by the fact the neighbor-traversal approach has a greater number of memory access and requires more data to be transmitted between different sockets. The increase in the number of processor present several communication limitations which include limited bandwidth, cache coherency, and non-uniform memory access (NUMA).

As such, an increase in communication (memory requests) can reduce the overall performance of the system as it happens for many of the medium sized graph scenarios when moving from 2 sockets (maximum of 20 threads) and upwards. However, as the graph becomes denser and the vertices have more parents in the adjacent level, the number of memory requests to the parent array increases. So does the number of cache misses due to contention. Cache misses can be just as expensive as over saturating the memory bandwidth and thus the performance of the array approach goes done and the neighbor-traversal approach starts outperforming the array approach again.

4.3. Fine Grain Parallelism

In this subsection we present performance results of the fine grain parallel betweenness centrality using both the ancestral approach and the neighbor approach on both the Cray XMT and the x86. For both these architectures we have taken known software packages: GraphCT [26] for the Cray XMT and Georgia Tech's SNAP [27, 28] for the x86.

4.3.1. GraphCT on the Cray XMT

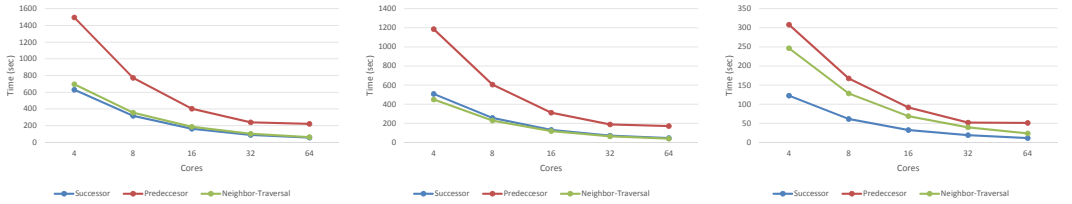
We compare the execution times of three variations of computing approximate betweenness centrality on the Cray XMT using fine grain parallelism: parent array approach [10], successor array approach [11], and successor neighbor-traversal approach. The first two are part of the GraphCT [26] software package.

Due to the large memory of the system and the reduced memory requirements of doing fine grain parallel betweenness centrality, $O(V+E)$ memory compared to $O(P\cdot(V+E))$, we focus in this section purely on performance of these algorithms.

For the XMT testing we used a RMAT-24 graphs with two different average edge degrees: 8 and 64. We refer to the first graph as the sparse graph and the second graph as the dense graph. For each graph 256 vertices

Table 1. The scenarios for the medium size graphs where the different approaches have better performance. The array based approach is denoted with ○ and the neighbor-traversal is denoted as ◇. The first column denotes the graph size and average edge adjacency. The remaining columns represent the thread count. Note that the neighbor-traversal approach dominates the table.

(Size, Avg|E|)	1	2	3	4	5	6	7	8	9	10	15	20	25	30	35	40
(20,8)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇
(20,16)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○
(20,32)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○	○
(20,64)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○	○
(21,8)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○
(21,16)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○	○	○
(21,32)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○	○	○
(21,64)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○	○	○
(22,8)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○	○
(22,16)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○	○	○	○
(22,32)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○	○	◇	◇
(22,64)	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	◇	○	○	◇	◇

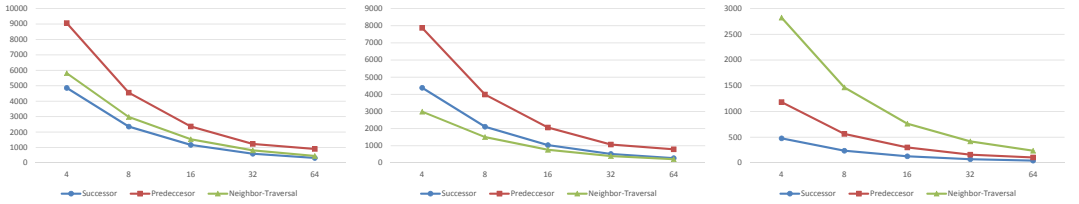


(a) Execution for both BFS and dependency accumulation stages.

(b) Execution for BFS stage.

(c) Execution for dependency accumulation stage.

Fig. 3. Execution time for approximate betweenness centrality on the Cray XMT2 for a considerably sparse graph. An RMAT-24 graph is used with an average edge degree of 8. 256 roots were used as defined in the HPCS SSCA [29] specifications.



(a) Execution for both BFS and dependency accumulation stages.

(b) Execution for BFS stage.

(c) Execution for dependency accumulation stage.

Fig. 4. Execution time for approximate betweenness centrality on the Cray XMT2 for a slightly denser sparse graph. An RMAT-24 graph is used with an average edge degree of 64. 256 roots were used as defined in the HPCS SSCA [29] specifications.

were selected as roots as is specified in the HPCS SSCA [29] specifications. We timed the the two main stages of betweenness centrality, the breadth first search stage and the dependency accumulation stage. We show the times for the entire computation, for the BFS stage, and for the dependency accumulation stage as a function of the number of XMT cores used. Fig. 3 and Fig. 4 present the timing the running times of the algorithms for the sparse and dense graph, respectively. The Successor approach in these figures refers to the [11] algorithm and the Predecessor approach refers to the [10] algorithm.

As can be seen, the new traversal algorithm out preforms the predecessor array and successor array based approaches in the BFS stage. The time spent in the parallel BFS dominates the execution time for both the sparse and dense graphs for all the implementations. On the other hand, the dependency accumulation stage in the predecessor array approach outperforms the dependency accumulation stage in the neighbor-traversal approach. The initial intuition behind the reduced performance can be can be explained by the fact the neighbor-traversal approach requires more memory operations. However, that is only a partial explanation.

The main reason for the reduced performance is due to load balancing issues that occur because of the different number of adjacencies that the vertices have. A single vertex with an extremely high number of adjacencies can cause the imbalance. This was also a problem for the successor based approach [11].

By generating an ER random graph we are able to confirm the workload imbalance. As ER graphs have a uniform distribution of edges over the vertices, the workload is considerably balanced for both the BFS traversal and the dependency accumulation. As expected both stages achieved better scalability (near linear scalability). For the sake of brevity, we do not present these graphs. It is also worth noting that detecting these workload imbalances for a small number of cores is difficult.

4.3.2. SNAP on the x86

We compare the execution times of the coarse parent array based approach and the neighbor-traversal approach on the x86. For these tests we used the implementations taken from the Georgia Tech's SNAP [27, 28] graph package. For the neighbor-traversal approach, we simply removed the none relevant code. For these tests we used a RMAT-21 21 graph with an average edge degree of 8. Once again a total of 256 roots were selected, as specified in the HPCS SSCA [29] specifications. Fig. 5 depicts the run-time of the fine grain algorithms. For these implementations, the neighbor-traversal code outperforms the array based approach.

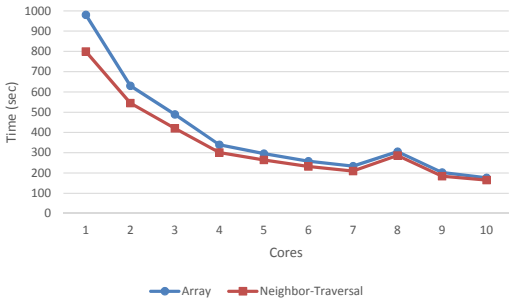


Fig. 5. Execution time of the fine grain parallelism using the SNAP package.

Table 2. Memory bounds for different betweenness centrality algorithms.

Algorithm	Ancestor-array based approach	Neighbor-traversal approach
Exact static [9]	$O(V+E)$	$O(V)$
Approximate static [14]	$O(V+E)$	$O(V)$
Exact streaming [17]	$O(V \cdot (V+E))$	$O(V^2)$
Approximate streaming	$O(K \cdot (V+E))$	$O(K \cdot V)$
Parallel fine grain [10, 11]	$O(V+E)$	$O(V)$
Parallel coarse grain	$O(P \cdot (V+E))$	$O(P \cdot V)$

4.4. Additional Uses

In this subsection we discuss the benefit of using neighbor-traversal in additional betweenness centrality algorithms. Table 2 contains a list of different betweenness centrality algorithms and their storage complexity using both the ancestor array approach and the new neighbor-traversal approach. We differentiate between computation of betweenness centrality for static graphs and streaming graphs. Streaming graphs are graphs that dynamically change over time with updates such as edge insertion or deletion. Further, we differentiate between exact and approximate betweenness centrality.

Note that the storage complexity for all the algorithms in Table 2 has been reduced using the neighbor-traversal approach. Also note, that the storage complexity of these algorithms is no longer dependent on the $O(E)$. This is especially crucial for the exact streaming algorithm [17] which previously had a storage complexity of $O(V \cdot (V + E))$ and is now $O(V^2)$. This improvement is also be helpful for computing approximate streaming betweenness centrality, where K refers to the number of roots that will be used in the computation, giving the approximate algorithm a storage complexity of $O(K \cdot V)$.

5. Conclusions

In this paper we present an additional approach for computing betweenness centrality. While the new approach does more operations it does not increase the theoretical complexity. Furthermore, the new approach increases the scalability due to a reduced storage complexity and allows for analyzing larger graphs. The performance of the new algorithm outperforms the previous ones. We showed the increased performance on two different architectures, Intel x86 and Cray XMT, using two different parallel granularities: coarse grain and fine grain. For the x86 architecture we showed both weak scaling and strong scaling results.

An additional conclusion of our work, is that computing edge betweenness centrality, as suggested Newman and Girvan [30], using the coarse grain approach simply does not scale due to the storage requirements as each thread requires an $O(E)$ memory and thus a total of $O(P \cdot (V + E))$. Note that vertex betweenness centrality has reduced storage complexity as was discussed in this work.

In conclusion we leave several open ended issues that we believed should be addressed in the hope of better understanding computation of betweenness centrality:

1. Is there a certain graph density from which it becomes beneficial to use the ancestral approach over the neighbor-traversal approach?
2. Is there a way to model the computation of betweenness centrality based on graph properties (without actually computing betweenness centrality) such that the number of memory accesses for a given algorithm can be approximated? This would allow for auto-tuning for betweenness centrality and perhaps for additional graph algorithms.

3. As we saw for the fine-grain parallelism scenario (on the Cray XMT) , the dependency accumulation in the successor approach suffers from an unequal workload balance. Is there a way to address this problem that would not require locks? Or if locks are used, can they be used such that the performance will not be affected too much?

Acknowledgments

Funding was provided by the U.S. Army Research Office (ARO) and Defense Advanced Research Projects Agency (DARPA) under Contract Number W911NF-11-C-0088. This work was supported in part by the Pacific Northwest National Lab (PNNL) Center for Adaptive Supercomputing Software for MultiThreaded Architectures (CASS-MT). We thank the Swiss National Supercomputing Centre for providing access to the Cray XMT system. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] M. Girvan, M. E. J. Newman, Community structure in social and biological networks, Proceedings of the National Academy of Sciences (12) (2002) 7821–7826.
[2] M. Rubinov, O. Sporns, Complex network measures of brain connectivity: Uses and interpretations, NeuroImage 52 (3) (2010) 1059 – 1069, Computational Models of the Brain.
[3] R. Guimera, S. Mossa, A. Turtschi, L. Amaral, The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles, Proceedings of the National Academy of Sciences 102 (22) (2005) 7794–7799.
[4] R. Bye, S. Schmidt, K. Luther, S. Albayrak, Application-level simulation for network security, in: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops, ICST, Brussels, Belgium, 2008, pp. 33:1–33:10.
[5] R. W. Floyd, Algorithm 97: Shortest path, Commun. ACM 5 (1962) 345–345.
[6] S. Warshall, A theorem on Boolean matrices, J. ACM 9 (1962) 11–12.
[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, New York, 2001.
[8] L. C. Freeman, A set of measures of centrality based on betweenness, Sociometry 40 (1) (1977) pp. 35–41.
[9] U. Brandes, A faster algorithm for betweenness centrality, Journal of Mathematical Sociology 25 (2) (2001) 163–177.
[10] D. Bader, K. Madduri, Parallel algorithms for evaluating centrality indices in real-world networks, in: International Conference on Parallel Processing (ICPP), 2006, pp. 539 –550.
[11] K. Madduri, D. Ediger, K. Jiang, D. Bader, D. Chavarria-Miranda, A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets, in: International Symposium on Parallel and Distributed Processing (IPDPS), IEEE, 2009.
[12] G. Tan, D. Tu, N. Sun, A parallel algorithm for computing betweenness centrality, in: International Conference on Parallel Processing (ICPP), 2009, pp. 340 –347.
[13] G. Tan, V. Sreedhar, G. Gao, Analysis and performance results of computing betweenness centrality on IBM Cyclops64, The Journal of Supercomputing 56 (2011) 1–24.
[14] D. Bader, S. Kintali, K. Madduri, M. Mihail, Approximating betweenness centrality, in: A. Bonato, F. Chung (Eds.), Algorithms and Models for the Web-Graph, Vol. 4863 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2007, pp. 124–137.
[15] D. Ediger, K. Jiang, J. Riedy, D. Bader, C. Corley, R. Farber, W. Reynolds, Massive social network analysis: Mining Twitter for social good, in: International Conference on Parallel Processing (ICPP), 2010, pp. 583 –593.
[16] A. Buluç, J. R. Gilbert, The Combinatorial BLAS: design, implementation, and applications, International Journal of High Performance Computing Applications 25 (4) (2011) 496–509.
[17] O. Green, R. McColl, D. A. Bader, A fast algorithm for incremental betweenness centrality, in: Proceedings of the 4th ASE/IEEE International Conference on Social Computing,, 2012.
[18] R. Albert, H. Jeong, A. Barabási, Internet: Diameter of the world-wide web, Nature 401 (6749) (1999) 130–131.
[19] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Wiener, Graph structure in the web, Computer Networks 33 (2000) 309 – 320.
[20] A.-L. Barabási, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512.
[21] J. Leskovec, J. Kleinberg, C. Faloutsos, Graph evolution: Densification and shrinking diameters, ACM Trans. Knowl. Discov. Data 1 (1).
[22] P. Konecny, Introducing the Cray XMT, Seattle, WA, USA, 2007.
[23] D. Chakrabarti, Y. Zhany, C. Faloutsos, R-MAT: A recursive model for graph mining, in: SIAM Proceedings Series, 2004, pp. 442–446.
[24] P. Erdös, A. Rényi, On random graphs I, Publicationes Mathematicae (1959) 290–297.
[25] P. Erdös, A. Rényi, The evolution of random graphs, Magyar Tud. Akad. Mat. (1960) 17–61.
[26] GraphCT: A Graph Characterization Toolkit.
URL `http://www.cc.gatech.edu/~bader/code/`
[27] SNAP: Small-world Network Analysis and Partitioning.
URL `http://snap-graph.sourceforge.net//`
[28] D. Bader, K. Madduri, SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks, in: International Symposium on Parallel and Distributed Processing (IPDPS), IEEE, 2008, pp. 1–12.
[29] HPCS Scalable Synthetic Compact Applications 2 for Graph Analysis, 2007.
[30] M. Newman, M. Girvan, Finding and evaluating community structure in networks, Physical review E 69 (2) (2004) 026113.