Generalizing k-Betweenness Centrality Using Short Paths and a Parallel Multithreaded Implementation

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

Abstract

We present a new parallel algorithm that extends and generalizes the traditional graph analysis metric of betweenness centrality to include additional non-shortest paths according to an input parameter k. Betweenness centrality is a useful kernel for analyzing the importance of vertices in a graph and has found uses in social networks, biological networks, and power grids among others. k-Betweenness centrality captures the additional information provided by paths whose length is within k units of the shortest path length. These additional paths provide robustness that is not captured in traditional betweenness centrality computations, and they may become important shortest paths if key edges are removed (by link failure or other means). We implement our parallel algorithm using lock-free methods on a massively multithreaded Cray XMT. We apply this implementation to a real-world data set of pages on the World Wide Web and show the importance of the additional data incorporated by our algorithm.

1. Introduction

The development of algorithms for the analysis of large graph data sets has driven much research in high performance computing today. Real-world networks from application domains, such as computational biology, economics, sociology, and computer networks, are sparse in nature and often exhibit "small world" properties. The algorithms often exhibit low amounts of temporal and spatial locality, while revealing little computation to hide the memory access times. Thus, their performance is often limited by the speed of main memory. Additionally, it can be difficult to obtain a balanced partition across processors in a modern parallel system when working with massive real-world graphs.

A useful analysis kernel for large graphs has been the computation of betweenness centrality. As defined by Freeman in [6], betweenness centrality is a measure of the number of shortest paths in a graph passing through a given vertex. For a 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:

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

Betweenness centrality can be used to identify critical vertices in a network. High centrality scores indicate that a vertex lies on a considerable fraction of shortest paths connecting pairs of vertices. This metric has been applied extensively to the study of various networks including biological networks [8], sexual networks and the transmission of the AIDS virus [10], identifying key actors in terrorist networks [5], organizational behavior, and transportation networks [7].

In our earlier work, we developed the first parallel algorithm for betweenness centrality [1], [12]. In the remainder of this paper, we will motivate and present an extension of Freeman's betweenness centrality and our previous algorithm. We generalize the definition to include paths in the graph whose length is within a specified value k of the length of the shortest path. We extend our recent parallel, lock-free algorithm for computing betweenness centrality to compute generalized k-betweenness centrality scores. Next, we will give details of an implementation of our new algoirthm on the massively multithreaded Cray XMT and describe the performance effects of this extension in terms of execution time and memory usage. Last, we will compare the results of this algorithm on synthetic and real-world data sets.

2. Extending Betweenness Centrality

The traditional definition of betweenness centrality given in [6] enumerates all shortests paths in a graph and defines betweenness centrality in terms of the percentage of shortest paths passing through a vertex v. This metric has proved valuable for a number of graph analysis applications, but fails capture the robustness of a graph. A vertex that lies on a number of paths whose length is just one greater than the shortest path receives no additional value compared to a vertex with an equally large number of shortest paths, but few paths of length one greater.

We will define k-betweenness centrality in the following manner. For an arbitrary graph $G(V, E)$, let $d(s, t)$ denote the length of the shortest path between vertices s and t. We define σ_{st_k} to be the number of paths between s and t whose length is less than or equal to $d(s, t) + k$. Likewise, $\sigma_{st}(v)$ is the count of the subset of paths that pass through vertex v. Therefore, k-betweenness centrality is given by:

$$BC_k(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st_k}(v)}{\sigma_{st}} \tag{2}$$

This definition of k-betweenness centrality subsumes Freeman's definition of betweenness centrality for $k = 0$. From this definition, it is clear that there are multiple ways to count paths whoses length is greater than $d(s, t)$, but less than or equal to $d(s, t) + k$. Above, each of the paths is counted equal to a shortest path. It might also be reasonable to count each of according to a linearly decreasing or exponentially decreasing function. This is a topic for future work.

3. A Parallel Algorithm for k-Betweenness Centrality

Much research has been done to develop fast algorithms for computing betweenness centrality. Brandes offered the first algorithm for computing betweenness in O(mn) time for an unweighted graph [3]. Madduri and Bader developed a parallel version of Brandes' algorithm exploiting both coarse- and fine-grained parallelism in low-diameter graphs in [1] and improved performance of this algorithm using lock-free methods in [12]. In this paper, we will extend the work in [12] to incorporate our new definition of k-betweenness centrality.

We define $d(s, t)$ to be the length of the shortest path between s and t. Paths must be acyclic and directed outwards from the source vertex. Let $\sigma_{st_k}(v)$ be the number of paths between s and t with length equal to $d(s, t)+k$ and passing through v. Let σ_{st_k} be the number of paths with length less than or equal to $d(s, t)+k$ between s and t. And let $\sigma_{st_{=k}}$ be the number of paths with length exactly equal to $d(s, t)+k$. Then, $\sigma_{st_k}(v)$ is given by:

$$\sigma_{st_k}(v) = \sum_{i=0}^{k} \sigma_{sv_{=i}} \cdot \sigma_{vt_{=k-j}} \tag{3}$$

Clearly, for $k = 0$, we have reproduced the original value of $\sigma_{st}(v)$ from Brandes where $d(s, t) \geq d(s, v) + d(v, t)$ (the Bellman criterion).

The k-betweenness centrality of vertex v is obtained by summing the pair-dependencies for that vertex:

$$BC_k(v) = \sum_{i=0}^{k} \sum_{s \neq v \neq t \in V} \delta_{st_k}(v) \tag{4}$$

$\delta_{st_k}(v)$ is given by a ratio of the number of paths whose length is equal to $d(s, t) + k$ passing through vertex v over the total count of the paths of length less than or equal to $d(s, t) + k$ between s and t.

$$\delta_{st_k}(v) = \frac{\sigma_{st_k}(v)}{\sigma_{st_k}} \tag{5}$$

In his work, Brandes cleverly derives a recursive relation for the *dependency* of s on any other vertex v in the graph. Likewise, we have derived the general expression for any path length k greater than the shortest path. We define $\Delta D(w, v)$ as $d(s, v) - d(s, w) + 1$, where s is the source vertex and w is a neighbor of v. ΔD is bounded by k for neighbors lying on a $+k-$path in which we are interested. We define $\delta_{s_k}(v)$, the dependence of s on v through paths of length $d(s, t) + k$, to be:

$$\delta_{s_k}(v) = \sum_{t \in V, t \neq s} \delta_{st_k}(v) \tag{6}$$

$$BC_k(v) = \sum_{i=0}^{k} \sum_{s \in V} \delta_{s_k}(v) \tag{7}$$

It follows that $BC_k(v)$ can be directly calculated from a sum of these dependence values. A formula for this relation is given in Figure 1. Note that for negative i, σ_{sv_i} is defined to be zero. The reasoning for this equation is as follows: In [3], Brandes gives a definition for $\delta_{st}(v, \{v, w\})$ being the ratio of shortest paths from s to t that travel through both the vertex v and the edge $\{v, w\}$. Similarly we can define $\delta_{st_k}(v, \{v, w\})$ as being the ratio of paths of length $d(s, t) + k$ from s to t that travel through v and $\{v, w\}$. If we do this, it is easy to see that we can express δ_{st_k} as a sum of $\delta_{st_k}(v, \{v, w\})$ across all vertices w such that w is a neighbor of v and $\Delta D(w, v) \leq k$.

Now we see that if $t = w$ then $\delta_{st_k}(v, \{v, w\})$ is simply $\frac{\sigma_{sv_{k-\Delta D}}}{\sigma_{sw}}$ where σ_{sw} is the number of paths from s to w which are of length $d(s, w) + k$ or less. The interesting part is when $t \neq w$. In the shortest paths case, $\delta_{st}(v, \{v, w\})$ can be derived as the ratio of shortest paths to w that go through v times the fraction of shortest paths from s to t that go through w, via the Bellman criterion. However we do not have the luxury of doing this as $\delta_{st_k}(v, \{v, w\})$ depends not only on $\sigma_{st_k}(w)$ but on $\sigma_{st_i}(w)\ \forall\ i \leq k$. To construct this dependency we take our general formula for $\sigma_{st_k}(w)$ and extend it to take into account the constraint that we need to pass through a certain edge $\{v, w\}$. To do this we can consider $\sigma_{sw_k}(v, \{v, w\})$, which is exactly $\sigma_{sv_{k-\Delta D(w,v)}}(v)$, and $\sigma_{wt_{=k}}$, which is the number of shortest paths from w to t with length exactly $d(w, t) + k$, and which is unknown in our algorithm. It can be seen that for $w \neq t$:

$$\delta_{st_k}(v, \{v, w\}) = \frac{1}{\sigma_{st_k}} \sum_{i=0}^{k} \sigma_{sv_{k-\Delta D(w,v)}}(v) \sigma_{wt_{=k}} \tag{8}$$

But here the problem is that we do not compute the value $\sigma_{wt_{=k}}$ at all. Therefore, we must infer it from other data. We want to get some expression in terms of $\sigma_{st_k}(w)$, since this will reduce in summation to a recurrence on $\delta_{s_k}(w)$. From equation (3) we can derive a recurence relation for $\sigma_{wt_{=k}}$:

$$\sigma_{wt_{=k}} = \frac{\sigma_{st_k}(w) - \sum_{i=1}^{k} \sigma_{sw_{=i}} \cdot \sigma_{wt_{=k-1}}}{\sigma_{sw_0}} \tag{9}$$

Then we are prepared to calculate a formula for $\delta_{s_k}(v)$. Combining (9) with (8), and summing over all ts, if we notice that $\delta_{s_k}(v) = \Sigma_t \frac{\sigma_{st_k}(w)}{\sigma_{st_k}}$, we obtain equations (10) and (11).

For the case when $k = 0$, this formula quickly reduces to shortest path dependency of s on v from Brandes' betweenness centrality algorithm:

$$\delta_s(v) = \sum_w \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_s(w)) \tag{12}$$

The recursive relation involves a combinatorial number of terms and quickly becomes intractable for large values of k. For our purposes, small values of k (less than 5) are interesting, meaning that we are interested in how a few edge changes could heavily influence a vertex's importance. For these, it may be possible to directly derive the expression for $\delta_{s_k}(v)$.

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

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

1: **for all** $v \in V$ **in parallel do**
2: $kBC[v] \leftarrow 0$
3: **for all** $s \in V$ **do**
I. Initialization
4: **for all** $t \in V$ **in parallel do** $d[t] \leftarrow -1$
5: **for** $0 \leq i \leq k$ **in parallel do**
6: $Succ[i][t] \leftarrow$ empty multiset, $\sigma[i][t] \leftarrow 0$,
7: $\sigma[0][s] \leftarrow 1$, $d[s] \leftarrow 0$
8: $phase \leftarrow 0$, $S[phase] \leftarrow$ empty stack
9: push $s \rightarrow S[phase]$

Now that a recurrence relation for the delta values has been established, we can summarize the algorithm outlined in Algorithm 1, 2, and 3. In the first stage we calculate $\sigma_{sv_{=k}}$ $\forall\ v, k$ from a particular source vertex s. Then we use these σ values to recursively calculate the $\delta_{s_k}(v)\ \forall\ v, k$. The first step is achieved through using breadth-first search to do a graph traversal. In the second stage, we traverse the graph in backward order from which it was explored during the

Algorithm 2 Part II - Graph traversal for shortest path discovery and counting

1: $count \leftarrow 1$
2: **while** $count > 0$ **do**
3: $count \leftarrow 0$
4: **for all** $v \in S[phase]$ **in parallel do**
5: **for** each neighbor w of v **in parallel do**
6: **if** $d[w] < 0$ **then**
7: push $w \rightarrow S[phase + 1]$
8: $count \leftarrow count + 1$
9: $d[w] \leftarrow d[v] + 1$
10: $\Delta D = d[v] - d[w] + 1$
11: **if** $\Delta D \leq min(k, 1)$ **then**
12: $\sigma[\Delta D][w] \leftarrow \sigma[\Delta D][w] + \sigma[0][v]$
13: **if** $\Delta D \leq k$ **then**
14: append $w \rightarrow Succ[\Delta D][v]$
15: $phase \leftarrow phase + 1$
16: **for** $1 \leq i \leq k$ **do**
17: **for** $0 \leq p < phase$ **do**
18: **for all** $v \in S[p]$ **in parallel do**
19: **for all** $w \in Succ[0][v]$ **in parallel do**
20: $\sigma[i][w] \leftarrow \sigma[i][w] + \sigma[i][v]$
21: **if** $i < k$ **then**
22: **for** $0 < j \leq i + 1$ **in parallel do**
23: **for all** $w \in Succ[j][v]$ **in parallel do**
24: $\sigma[i + 1][w] = \sigma[i + 1][w] + \sigma[i + 1 - j][v]$

Algorithm 3 Part III - Dependency accumulation by back-propagation

1: $phase \leftarrow phase - 1$
2: $\delta[i][t] \leftarrow 0\ \forall\ t \in V, 0 \leq i \leq k$
3: **for** $0 \leq k' \leq k$ **do**
4: $p \leftarrow phase$
5: **while** $p > 0$ **do**
6: **for all** $v \in S[p]$ **in parallel do**
7: **for** $0 \leq d \leq k'$ **in parallel do**
8: **for all** $w \in Succ[d][v]$ **in parallel do**
9: **for** $0 \leq i \leq k' - d$ **do**
10: $sum \leftarrow 0$
11: $e \leftarrow k' - d - i$
12: **for** $0 \leq j \leq e$ **do**
13: $sum \leftarrow sum + W(e - j, e, w, \sigma) * \sigma[j][v]$
14: $\delta[k'][v] \leftarrow \delta[k'][v] + sum * \frac{\delta[i][w]}{\sigma[0][w]^{e+1}}$
15: $\delta[k'][v] \leftarrow \delta[k'][v] + \frac{\sigma[k'-d][v]}{\Sigma_i \sigma[i][w]}$
16: $kBC[v] \leftarrow kBC[v] + \delta[k'][v]$
17: $p \leftarrow p - 1$

$$\delta_{s_k}(v) = \sum_{w \in Succ[v]} \left(\frac{\sigma_{sv_{=(k-\Delta D)}}}{\sigma_{sw}} + \sum_{i=0}^{k} \sum_{j=0}^{k-i} \left[W(k-i-j, k-i) \cdot \sigma_{sv_{=(j-\Delta D)}} \right] \frac{\delta_{s_i}(w)}{\sigma_{sw_0}^{k-i+1}} \right) \tag{10}$$

$$W(n,d) = \begin{cases} \sigma_{sw_0}^{d}, & n = 0 \\ -\sum_{i=1}^{n} \sigma_{sw_{=i}} \cdot W(n-i, d-1), & n > 0 \end{cases} \tag{11}$$

Figure 1. Recursive relation for the *dependency* of a vertex s on a vertex v in the graph.

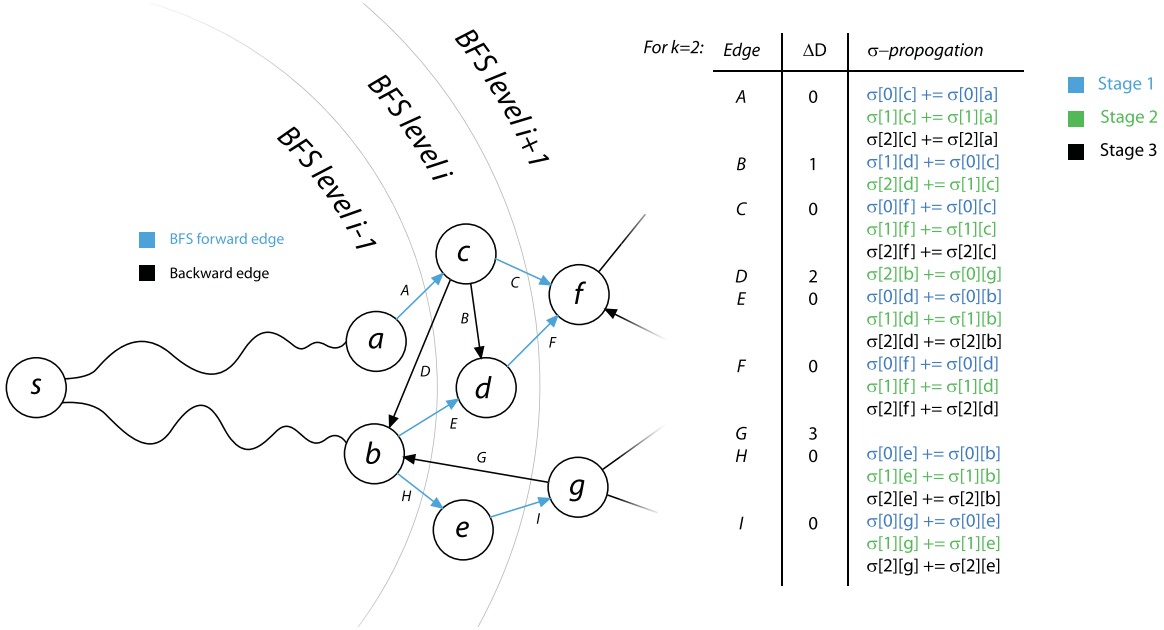

Figure 2. Illustration of σ propogation in the $k = 2$ case. Shown is a segment of the breadth-first search. The table represents the ΔD value as well as the propogation occurring as the result of each edge. The color of the σ propogation represents in which stage that addition will occur.

Algorithm 4 Function $W(k, d, w, \sigma)$: a recursive method producing a polynomial expansion with constant subscript sum.

Input: k, d, w, two-dimensional array σ

Output: A multivariate polynomial in $\sigma[x_i][w]$ where each term has sum of exponents d and also in each term $\Sigma_i x_i = k$, evaluated with values from σ.

1: **if** $k = 0$ **then return** $\sigma[0][w]^d$
2: **else**
3: $sum \leftarrow 0$
4: **for** $0 < i \leq k$ **in parallel do**
5: $sum \leftarrow sum - \sigma[i][w] * W(k-i, d-1, w, \sigma)$
return sum

search stage. We repeat this for each source vertex s and sum the δ values for each vertex v to obtain the BC_k value.

We must modify the graph traversal phase from that of our previous work in [12] in order to correctly propagate values of $\sigma_{st_{=k}}$. When $k = 0$, it suffices to do a single breadth first search and propagate the value of σ from level to the next. For $k > 0$, we must do $k + 1$ breadth first searches. Notice that when we are updating σ values, for neighbors on the next level in the breadth-first search, we have that $\sigma_{sw_{=i}} = \sigma_{sv_{=i}}$ $\forall$ i, based on our generalized Bellman criterion. But we must also increment the σ values for neighbors in the current or previous levels of the search, which can be a problem since these neighbors then need to also pass on these values to their neighbors.

We solve this by only passing on σ values of a certain rank to forward vertices and of another rank to vertices that are behind the breadth first search frontier. Specifically, the forward propagation always trails the backward propagation by one level. In the first step, we calculate and forward-propagate σ_0 and back-propagate σ_1. In the second step, we forward-propagate σ_1 and back-propagate σ_2. If $k = 2$, in the final step, we forward-propagate σ_2. See Figure 2 for a demonstration of this process. We "back-propagate" σ_k to vertices at the same level or a previous level when we identify them as a neighbor of a vertex in the current level and whose ΔD is appropriate for the phase of σ_k we

are propagating. We "forward-propagate" σ_k to neighboring vertices on the shortest path that we discover during the breadth first search. During the first breadth first search we store $k+1$ successor (or child) arrays. When we find a neighbor during breadth first seach whose $\Delta D \leq k$, we append that neighbor's index to the ΔD^{th} successor array.

Therefore, we need not re-run the breadth first search after the first time: as long as we synchronize on the phases, we may directly scan the successor arrays to perform our propagation, avoiding contention on a common vertex queue, and also allowing us to ignore far away neighbors and jump to exactly the neighbors we are interested in. In each case we exploit parallelism in the traversal by exploring the neighbors of the current level of vertices concurrently. For a small-world graph, where graph diameter is small, the number of levels in the breadth-first search is correspondingly small, and parallelism is thus high. In the first traversal, all the vertices must add newly discovered vertices to a atomically accessed queue, which is the main bottleneck in the search. Since that work is done in the first phase, we can store some data so that subsequent searches do not rely on this queue and thus do not have a sequential bottleneck to their parallelism.

For the $\delta-$accumulation step, we start by performing shortest-path accumulation as before, in Brandes's original algorithm. However, having these δ_0 values allows us to repeat the backward traversal to recursively calculate δ_1, and so on and so forth. Notice that in Figure 1, the δ_k value of the current vertex sometimes relies on the δ_k values of vertices after, before, and on the same level of the breadth first search. This may seem to preclude recursion, however as it works out, for the same level of δ, the $-\Delta D$ term on the subscript of σ_{sv} cancels out any dependencies on vertices on the same level or lower, meaning we can perform the recursive step as before, so long as we only calculate one level of δ per recursive traversal.

Complexity analysis: In Brandes' original algorithm the betweenness centrality of any vertex can be calculated in $\mathrm{O}(mn)$ time and $\mathrm{O}(m+n)$ space. It is easily seem that the memory requirements of our new algorithm is exactly $k+1$ times the original, as we make $k+1$ copies of the arrays for σ, δ, and the $successor$ and $successor\_count$ arrays, one for each level between 0 and k. The sequential *time* analysis is a bit more difficult. The breadth-first search phase is roughly $k+1$ times $\mathrm{O}(mn)$, as we are essentially traversing the graph $k+1$ times (and in the worst case we traverse each edge during each iteration). However in the accumulation phase we are actually weighed down by the calculation of the W function which grows combinatorically: more specifically, the number of terms in $W(k,d)$ grows as the number of integer partitions of k, which is $\mathrm{O}\left(e^{\sqrt{k}}\right)$. So the amount of work in this algorithm is on the order of $\mathrm{O}\left(e^{\sqrt{k}}mn\right)$. For large k we can see that this becomes quite difficult to compute, however at small k it is quite manageable. For example, if we are only interested in $k<5$ then the multiplier is bounded by $e^2 << mn$.

4. Computing k-Betweenness on the Cray XMT

Current hardware utilize a hierarchy of caches to hide the latency to main memory. This approach works well when the memory access pattern is predictable or when application codes demonstrate significant levels of temporal or spatial locality. Graph analysis kernels usually exhibit fairly low levels of spatial or temporal locality, and execution on these platforms is limited by the speed of the memory subsystem. Hardware multithreading has been shown to be effective for producing efficient implementations of parallel graph algorithms when a signficant amount of parallelism can be revealed [11].

The Cray XMT [9] uses massive numbers of hardware threads to *tolerate* latency to main memory. The XMT uses a 500 MHz 64-bit Threadstorm processor that supports 128 hardware streams of execution. Context switching between threads is lightweight and requires a single clock cycle. Each processor can support up to 16 GB of main memory that is hashed and globally addressed. The memory has a 128 KB, 4-way set associative data buffer that caches local data only. The system is built around Cray's XT infrastructure and can scale to 8,024 processors.

The Cray XT infrastructure provides the I/O facilities and the interconnection network for the XMT. The system utilizes the Seastar-2 interconnection network that connects nodes in a 3D-torus topology. As a result, per-processor bisection bandwidth decreases as the number of processors is increased. Service nodes provide access to a Lustre file system for the storage and retrieval of large data sets.

A novel feature of the Cray XMT is its support for lightweight word-level synchronization mechanisms. Each 64-bit word of memory has a full/empty bit associated with it. C-language primitives are provided to the programmer for managing this bit and mutual exclusion locks are common. The architecture also provides an atomic `int_fetch_-add` instruction that allows for integer read-modify-write operations. This is commonly used for zero-overhead shared data structures like queues and stacks. See [12] for a more detailed explanation of the synchronization and lock-free methods used in this algorithm.

Implementation and optimization: In terms of implementation on the XMT, the underlying structure is similar to that used in our previous work [12]. As before, utilization of the child arrays allows us to update our $\delta-$values without locking. The further optimizations in our code for $k-$betweenness can be seen as utilizing the fact that we will mostly be using small k values. As one can see in Algorithm 3, there are many nested loops, however most of them are

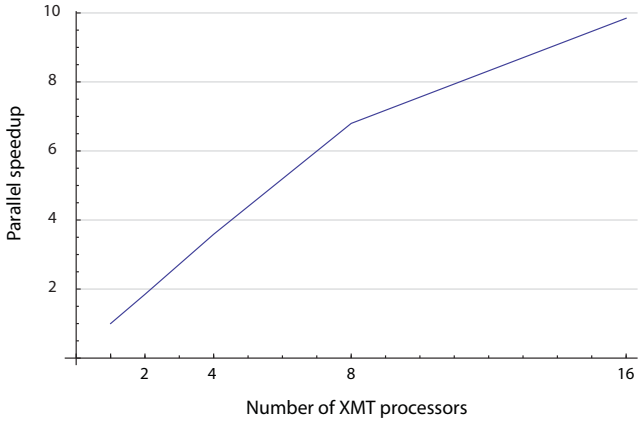

Figure 3. Parallel scaling on the Cray XMT, for an R-MAT generated graph of scale 23 (2^{23} vertices). Scaling is linear up to 8 processors and speedup is roughly 10 at all 16 processors. $k = 1, K4approx = 8$ for balanced runtime and complexity (single node time 56 minutes).

very simple for small k and unfurl quickly. By manually coding these loops for smaller values of k', we significantly reduce the execution time since the time to set up and iterate over the small number of loop iterations quickly outstrips the actual useful work inside of them. For a scale 20 R-MAT (Recursive MATrix graph generator [4]) graph (having 2^{20} vertices and 2^{23} edges), the time to compute 1-betweenness drops by a factor of two with this optimization. We use this R-MAT generator to realize inputs that are similar to small-world graphs in degree distribution.

We begin unrolling at the loop over d values. Notice that for $d = k'$ the loops collapse into a very nice form, in that it becomes the following simplified algorithm:

Algorithm 5 Loop at line 8 from Algorithm 3, reduced form. Note that $W(0, 0) = 1$.

1: **for all** $w \in Succ[k'][v]$**in parallel do**
2: $\quad \delta[k'][v] \leftarrow \delta[k'][v] + \sigma[0][v] * \left(\frac{\delta[0][w]}{\sigma[0][w]} + \frac{1}{\Sigma_i \sigma[i][w]} \right)$

Just from this simple equation we have reduced three loops into one. Now since we have a formulation for $d = k'$ in the d loop, we may pull it out of the loop and run the rest of the loop from 0 to $k' - 1$. Similarly we may wish to pull the formulation when $d = k' - 1$, which we have done, or beyond, depending on the number of iterations we wish to work out by hand. However, as i in $d = k' - i$ grows, the complexity of the resulting reduced formula grows at an exponential rate, meaning that for larger i this reduction is intractable.

In addition, just as we wish to take advantage of small k values in this outer loop, we may also utilize this property in writing our recursive W function, which creates this relatively messy σ polynomial. For small values of k' this will already be built into the reduced loop iterations, however since larger inputs to the W function recurse downward to more manageable numbers, it will be convenient to quickly return the correct answer at lower input sizes, saving a couple recursion steps.

Apart from manually optimizing for lower loop iterations, other considerations were taken for this architecture. For example, the `malloc` system call is rather expensive in the case of these lightweight threads: if we have a `malloc` in parallel its cost compared to the number of operations in that thread's lifetime can actually be quite significant. Initially temporary arrays were kept to store the number of children accumulated for a particular vertex during graph traversal, however since temporary arrays require dynamic allocation, we modified the code to skip these temporary arrays and access the source arrays directly, at the cost of addressing the larger array repeatedly. Reorganizing memory accesses to avoid dynamic allocation within the loop reduced runtime by more than 75%. Furthermore, since the system has a plentiful amount of memory which can in essence be accessed with minimal latency, we are encouraged to utilize extra memory in lieu of performing extra calculations (as long as we have sufficient network bandwidth): thus, the expression $\Sigma_i \sigma[i][w]$ in Algorithm 3 is precomputed and stored in an array for all values of w.

In Figure 3 we show the parallel scaling of our optimized code on the Cray XMT from 1 to 16 processors. We have reduced the execution time from nearly an hour down to a few minutes for this problem. On 16 processors a graph of scale 23 takes a little over 320 seconds to run for $k = 1$ *approximate* betweenness. This approximation is based on selecting a random sample of source vertices s, in this case, when $K4approx = 8$, the number of starting vertices is $2^8 = 256$. The plot shows good scaling up to our machine size.

5. Evaluating k-Betweenness

In order to explore the effect of various values of k on the calculation of k-betweenness centrality, we apply our Cray XMT implementation to the nd-www graph data set [2]. This graph represents the hyperlink connections of web pages on the Internet. It is a directed graph with 325,729 vertices and 1,497,135 edges. Its structure demonstrates a power-law distribution in the number of neighbors. The graph displays characteristics typical of scale-free graphs found in social networks, biological networks, and computer networks.

To examine the graph data, we ran *approximate* k-betweenness centrality for k from 0 (traditional betweenness centrality) to 4. In an approximate calculation, a subset of vertices are chosen at random as starting points for the breadth first search. An exact computation would use all vertices in the graph. For our experiments, we set

Percentile	$k=1$	$k=2$
90th	981	2358
95th	366	644
99th	59	100

Figure 4. The number of vertices ranked in selected percentiles for $k = 1$ and $k = 2$ whose betweenness centrality score was 0 for $k = 0$ (traditional BC). There were 51,870 vertices whose traditional BC score was 0, but whose BC_k score for $k = 1$ was greater than 0. The ND-www graph contains 325,729 vertices and 1,497,135 edges.

$K4approx = 8$ as before. The betweenness scores are compared for each value of k. An analysis directly follows. Also, after computing betweenness centrality for $k = 0$, we remove one or more of the highest ranking vertices and re-examine the results.

Looking at the highest ranking vertices going from $k = 0$ to $k = 4$, the subset of vertices and the relative rankings change little. This would seem to indicate that the paths k longer than the shortest path lie along the same vertices as the shortest path in this graph. Moreover, as predicted, the traditional betweenness centrality metric fails to capture all of the information in the graph. When examining the BC_k score for $k > 0$ of vertices whose score for $k = 0$ was 0 (no shortest paths pass through these vertices), it becomes clear that a number of very important vertices in the graph are not counted in traditional betweenness centrality. For $k = 1$, 981 vertices are ranked in the top 10 percent, but received a score of 0 for $k = 0$. In the 99th percentile are 59 vertices. Likewise, 100 vertices received a traditional BC score of 0, but ranked in the top 1 percent for $k = 2$. In total, there were 51,870 vertices whose betweenness centrality score for $k = 0$ was 0, but had a k-betweenness centrality score of greater than 0 for $k = 1$.

These vertices that get missed by traditional betweenness centrality play an important role in the network. They do not lie along any shortest paths, but they lie along paths that are just one unit longer than the shortest path. If an edge is removed that breaks one or more shortest paths, these vertices could likely become very central to the graph. The traditional definition of betweenness centrality fails to capture this subtle importance, but k-betweenness centrality makes it possible to identify these vertices.

When a vertex of high betweenness is removed from the graph, it causes a number of changes in betweenness scores for all values of k that we are studying. Many vertices gain a small number of shortest paths and their ranking is fairly unchanged. In general, those vertices ranked very highest on the list remain at the top. This would seem to indicate that there is a network of short paths between vertices of extremely high betweenness. Interestingly, however, other vertices jump wildly within the rankings. Often, several of these are neighbors of the removed vertex. This underscores the previous conclusion that a vertex of relatively little importance in the graph can become extremely important if the right vertex or combination of vertices are removed. Future work will study the effect of removing edges of high betweenness, rather than vertices, on the rankings of k-betweenness centrality in these real-world networks.

6. Conclusions and Future Work

Betweenness centrality has proven itself in the past to be a useful metric for graph analysis. We have extended traditional betweenness centrality to the general case taking into account additional paths longer than the shortest paths. We have begun to analyze the effects of this metric on real world graphs. We believe that this new tool will have important consequences in the study of contingency analysis and planning, failover routing in computer networks, and extended relationships in social networks. In future work, we look to apply this new tool to a wider range of graphs stemming from real world data in an effort to understand the impact that various types of paths have on the structure, robustness, and resiliency of the graph.

Acknowledgments

This work was supported in part by the PNNL CASS-MT Center and NSF Grant CNS-0614915. We would like to thank PNNL for providing access to the Cray XMT. We are grateful to Kamesh Madduri, Daniel Chavarría, 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 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.

[2] A.-L. Barabási, "Network databases," 2007, http://www.nd.edu/~networks/resources.htm.

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

[4] 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.

[5] 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.

[6] L. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.

[7] 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.

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

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

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

[11] 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.

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