On Ranking Nodes using kNN Graphs, Shortest-paths and GPUs

Ahmed Shamsul Arefin** Regina Berretta Pablo Moscato*

Centre for Bioinformatics, Biomarker Discovery and Information-Based Medicine, School of Electrical Engineering and Computer Science, Faculty of Engineering and Built Environment, The University of Newcastle, Callaghan, NSW 2308, Australia
Email: Ahmed.Arefin, Regina.Berretta, Pablo.Moscato@newcastle.edu.au
*Currently at the ICT Services, University of Southern Queensland, Qld * Contact author

Abstract

In this paper, we present graphics processing unit (GPU) based implementations of three popular shortest-path centrality metrics—closeness, eccentricity and betweenness. The basic method is designed to compute the centrality on gene-expression networks, where the network is pre-constructed in the form of kNN graphs from DNA microarray data sets. The relationship among the genes in the kNN graph is determined by the similarity of their expression levels. The proposed method has been applied to a well known breast cancer microarray study and we highlighted the correlation of the highly ranked genes to the time to relapse of the disease. The method is readily applicable to other datasets, where the data points can be recognised in a multidimensional space. It can be applied to other networks (e.g., social networks, the Internet, etc.) with minimal modifications.

Keywords: Shortest paths, breadth first search, centrality, kNN, CUDA, microarrays, gene-expression.

1 Introduction

Centrality analysis measures the relative importance of the elements in a given network based on their connectivity within the network structure. In other words, centrality measures help to rank the network elements according to their importance within the network structure. Formally, the centrality of a network is defined as follows (Junker et al. 2006), let $G(V, E)$ be a directed or undirected graph (network), then the centrality on $G$ is defined as a function $C : V \rightarrow \mathbb{R}$ that assigns a real number to each vertex. For a pair of two vertices, $u$ and $v$, if $C(u) > C(v)$, one can say that $u$ is more central than $v$. Although many of the popular centrality metrics are actually originated from the classical analysis of social networks, now they have successfully been investigated on many other practical networks, e.g., the Internet (Page et al. 1998, Gkorou et al. 2011), public transport networks (Kazerani & Winter 2009), power grid network (Jin et al. 2010), biological networks (Potomyap et al. 2005, Bader & Madduri 2008), etc. A brief review of the existing centrality metrics and their applications can be found in (Junker & Schreiber 2011, Newman 2010). The main problem with many of the metrics is that their sequential implementations can often become very time consuming. For instance, the betweenness centrality computation of all nodes in a graph requires $O(n^3)$ time with Floyd-Warshall algorithm, so for network with 1M nodes, a sequential method may take decades of computation on a general purpose computer. Even though there exists some faster approximate methods (Jacob et al. 2005, Eppstein & Wang 2001), their high error rates on larger networks can severely limit their applicability (Jia et al. 2008).

One feasible way to compute the centrality of such large-scale networks would be to parallelize the computation and interestingly, a number of parallelization approaches for such purpose have already been developed. Some of them are quite fast and scalable, but unfortunately, require highly sophisticated and expensive computer systems with parallel processing capabilities. For instance, Bader and Madduri (Bader & Madduri 2006) implemented several parallel shortest path based metrics using shared memory multi-processors on CRAY MTA-2. Later, Madduri et al. (Madduri et al. 2009) presented a refinement of the same work by proposing a lock-free variant on CRAY-XMT system (Mizell & Maschhoff 2009). Jin et al. (Jin et al. 2010) utilized the same system for computing the betweenness of power grid contingency measurements utilizing the same set of algorithms. Edmonds et al. (Edmonds et al. 2010) presented a set of distributed memory algorithms for computing centralities using cluster computers with at least 100’s of compute nodes. Alternatively, there exist a few GPU implementations, which can be considered as relatively inexpensive approaches. However, a common problem with these implementations is their relatively lower scalability, when compared with the CPU based parallel counter-parts.

In this work, we present fast methods for computing three shortest-path based centrality metrics, closeness, eccentricity and betweenness. We apply the methods on gene-expression networks constructed from DNA microarray data sets. We use a GPU-based fast and scalable method (GPU-FS-kNN) for constructing the network. The proposed centrality computation methods are adapted from the sequential Breadth First Search (BFS) and Brandes’s (Brandes 2001) shortest-path computation method for a given graph.

2 Literature Review

The shortest path based centrality metrics are usually implemented by using the basic path finding algorithms (e.g., single source shortest paths (SSSP), all-pair shortest paths (APSP), etc.). Along with the super-computer based implementations (as discussed above), there exist a few GPU implementations of the basic graph traversal algorithms (e.g.,
as breadth first search (BFS), Single Source Shortest Path (SSSP) (Harish & Narayanan 2007), which can be used as building blocks for computing a shortest path based centralities. In addition, Sriram et al. (Sriram et al. 2009) proposed the first GPU-based implementation of the shortest path betweenness centrality that “parallelize the BFS wavefront from different source nodes at different thread blocks”. However, this so-called node parallel approach performs sub-optimally when some nodes have more neighbors than the others. For instances, if a block has less workload and finishes early, it must stay idle until all others are finished, which can lead to serious load-imbalance. Jia (Jia 2010) solved this problem by proposing an edge-parallel and achieved a comparatively better parallelism by “exploiting neighbors of each wavefront nodes in parallel”. However, similar to (Sriram et al. 2009), they not only perform the BFS from different source nodes on different thread blocks but also duplicates several data structures to each of these blocks. This data replication severely limits the number of thread blocks that can be launched at a time and hence, their implementation can run with only atmost 10 – 30 CUDA blocks (the number of standard SMs in the current series of GPUs). To overcome, the best optimized GPU implementation of the shortest path based centralities are proposed by Shi and Zhang (Shi & Zhang 2011). Even though their implementations are the adaptations of those in (Jia 2010), they eliminated the need for data duplication (to each block) by maintaining a two dimensional flag matrix which keeps track of the traversed paths. However, their approach is still unscalable to very large-scale networks, as it requires duplicated edge lists (on device) to maintain the neighborhood structures. There are a few other approaches developed more recently (e.g., see (McLaughlin & Bader 2014)) that are relatively more scalable but at the expense of hundreds of GPUs and more expensive hardware setup.

3 Proposed Centrality Computation Method

The proposed centrality computation method is designed to identify central elements in DNA microarray gene-expression data sets. It works in two steps, first it constructs a k-nearest neighbor (kNN) graph from a given multi-dimensional (gene-expression) data set. Next, it computes the centrality metrics on that graph.

3.1 Construction of the kNN Graphs from Gene-expression Data Sets

The k nearest neighbor (kNN) graph is an important graph structure where each node is connected to its k nearest or, closest nodes and the closeness is defined by a distance metric. The computation of a distance matrix is a fundamental task in constructing a kNN graph, if it is not provided as the input. Although there exist several methods that do not require the complete (or half, as symmetric) distance matrix, they suffer from a different problem called curse of dimensionality (e.g., see kd-Trees). From gene-expression data sets, the kNN graphs can be constructed in many ways, e.g., by using exhaustive search techniques, such as brute-force kNN search. The basic construction is quite simple, for a set of data elements (in a given metric space) the kNN graph can be produced by creating an edge from each element to its k nearest elements.

However, the computation of the distances of the nearest neighbours for large-scale instances becomes very slow on general purpose computers. Fortunately, the nearest neighbours of each vertex can be computed and searched independently and hence, the brute force approach is highly parallelizable. It may be noted that the most common problem with the existing GPU-based brute force kNN algorithms are two-fold, firstly, they can only work if all the distances between query and reference points, i.e., the distance matrix, can fit into GPU’s in-memory (e.g., see (Garcia et al. 2008)); secondly, they assume that the value of k is relatively small in comparison with the instance size (Liang et al. 2009). In contrast, we utilized a scaled and parallelized variant of the simple brute force kNN algorithm that is implemented using a chunking-based approach called GPU-FS-kNN. It can efficiently utilize GPUs and can handle instances with more than one million objects and fairly larger values of k (e.g., tested with k up to 64) on a single GPU. On multiple GPUs, if data partitioning is applied, then the method is capable of handling much larger instances and higher dimension sizes. Details of our GPU-based kNN graph construction method can be found in (Arefin et al. 2012b) and its other applications in (Arefin et al. 2012a,c,d, 2013).

3.2 Representation of Graphs

Whilst the graph representation methods on the CPU have been studied extensively, many of these methods are not suitable for GPUs. The GPUs have limited on-board memory and their memory access patterns are completely different from CPU. Traditional methods, such as adjacency matrix, stores a graph \(G(V,E)\) using \(O(|V|^2)\) space. Using this approach, a graph with millions of vertices will require terabytes of storage memory. Therefore, adjacency matrix-based representations are infeasible on GPUs. In contrast, adjacency lists are more relaxed and can store graphs using much less memory \(O(|V| + |E|)\). There exist several approaches for storing graphs on GPUs, for instance, adjacency list based (Harish & Narayanan 2007, Leist et al. 2009) and the traditional adjacency matrix (Katz & Kider 2008), but by accelerating it using the device shared memory.

In this work, we store our kNN graph (noted by \(Gk\)) in a single dimensional array of edge structure, where the basic structure has at least 3 members \(\{\text{source}, \text{target} \text{and weight}\}\). There may be some additional members of the structure depending upon the problem in question. All the graphs maintain the following two properties, to facilitate our algorithm developments. First, edges are sorted by the source vertex indices, then by weights. Second, each distinct source vertex has exactly k neighbors and hence, source vertex id changes exactly after k edges.

We store our the graphs in the device global memory, which is slower but much larger than any of the other device memory types. Therefore, a higher scalability in terms of input is expected. Moreover, for very large graphs, the proposed structure allows to load graphs from the host to device part by part (i.e., by chunks). Furthermore, for extremely large graphs, an external memory approach (Vitter 2001) may be used to store the chunks of the graph into the external hard-drives and then load back to host and subsequently to the device memory.
3.3 Single Source Shortest Paths (SSSP) for the \(k\)NN graphs

Given a graph \(G(V, E)\) and a source vertex \(s \in V\), the SSSP problem finds the shortest paths (or geodesic distance) from \(s\) to every other vertex \(v \in V\). There exist several algorithms (e.g., Dijkstra’s, Bellman-Ford, etc.) to compute the SSSP on directed graphs. We consider that the input \((Gk)\) as unweighted, therefore we simply extend the BFS algorithm.

Table 1: Variables and arrays used in the shortest path based centralities.

<table>
<thead>
<tr>
<th>Names</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d)</td>
<td>The shortest path distance</td>
</tr>
<tr>
<td>(dist)</td>
<td>An array to hold the distance from the source (s) to each node (v \in V)</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>An array to hold the number of shortest paths from the source (s) to each node (v \in V)</td>
</tr>
<tr>
<td>(\delta)</td>
<td>An array to hold the dependency of the source (s) on each node (v \in V)</td>
</tr>
<tr>
<td>(P)</td>
<td>A bit matrix to track the traversed paths</td>
</tr>
<tr>
<td>(flag, found)</td>
<td>Boolean flags</td>
</tr>
</tbody>
</table>

The data-parallel implementation of the SSSP problem presented in Algorithm 1 (variables are explained in Table 1, see also (Arefin 2013)) is an adaptation of the parallel BFS algorithm presented in (Shi & Zhang 2011). However, there is a difference between their graph representation and ours. To facilitate the graph traversal in both directions (i.e., forward and backward, which is required by the betweenness centrality) they keep each edge twice using two longer arrays that maintain the one-to-one neighborhood correspondence with each other. In contrast, we achieve the same goal, but without performing any data replication. We incorporate a simple modification in the parallel BFS Algorithm (line 4 – line 10, Algorithm 2), so that we can use the \(k\)NN graph straight into the kernel. Each thread checks both ends of each \(k\)NN edge and investigates if any of them is in the current wavefront (BFS level) (see Figure 1). If the source node is in the current level, then the respective thread updates the distance of the target. Otherwise values of source and target are swapped and the corresponding distances are updated. It reduces the workload balance and hence, the execution times may slightly increase, but we consider this as a trade-off between the space and run-time complexity of the algorithm.

Figure 1: An illustration of the parallel BFS and SSSP algorithm on a network of four vertices, where node 0 is considered as the source vertex (i.e., \(s \leftarrow 0\) and \(d \leftarrow 0\)). (a) First launch of the BFS kernel, thread 0 and 1 explore the node 1 and 2, respectively (in parallel). The respective locations in the dist array are initialized by the distance, \(d \leftarrow 1\). (b) Second launch of the kernel, now both threads explore node 3 in parallel and initialize the respective location in dist by the node distance \(d \leftarrow 2\).

---

Algorithm 1: Single Source Shortest Path Algorithm

Input: \(Gk, s, k\);  
Output: \(dist\) array is initialized;  
1. Initialize \(dist[v] \leftarrow -1, \forall v \in V\);  
2. \(dist[s] \leftarrow 0\);  
3. \(flag \leftarrow true\);  
4. while \(flag = true\) do  
5. \(host \rightarrow device (flag, d)\);  
6. BFS \((Gk, dist, d, flag, n, k)\);  
7. device \(\rightarrow host (flag, d)\);  
8. \(d \leftarrow d + 1\);  
9. \(flag \leftarrow false\);  
10. device \(\rightarrow host (dist)\);  
11. return \(dist\);

Algorithm 2: BFS (Parallel)

Input: \(Gk, dist, d, flag, n, k\);  
Output: \(dist\) array is initialized;  
1. \(tid \leftarrow thread id\);  
2. if \(tid < n \times k\) then  
3. \(source \leftarrow Gk[tid].source;\)  
4. \(target \leftarrow Gk[tid].target;\)  
5. \(/* Path discovery */\)  
6. if \(dist[source] = d\) then  
7. \(found \leftarrow true;\)  
8. else if \(dist[target] = d\) then  
9. \(swap(target, source) // device function;\)  
10. \(found \leftarrow false;\)  
11. \(/* Path Traversal */\)  
12. if \(found = true\) then  
13. if \(dist[target] = -1\) then  
14. \(dist[target] \leftarrow d + 1;\)  
15. \(flag \leftarrow true;\)
Algorithm 3: kNN Closeness Centrality

Input : \( Gk, k \)
Output: Closeness centrality of each vertex stored in \( cc \) array;
1 Initialize \( dist[i] \leftarrow -1, \forall i \in V \);
2 \( flag \leftarrow \text{true}; d \leftarrow 0; \)
3 foreach vertex \( s \) in \( V \) do
4 \( dist[s] \leftarrow 0; \)
5 while \( flag = \text{true} \) do
6 host \( \rightarrow \) device \( (flag, d); \)
7 BFS \( (Gk, dist, d, flag, n, k) \);
8 device \( \rightarrow \) host \( (flag, d); \)
9 \( d \leftarrow d + 1; \)
10 \( flag \leftarrow \text{false}; \)
11 device \( \rightarrow \) host \( (dist); \)
12 Accumulate_Closeness \( (dist, n, s, cc) \);
13 return \( cc; \)

Algorithm 4: Accumulate_Closeness

Input : \( dist, n, s, cc \)
Output: Closeness centrality of \( s \) is accumulated in \( cc[s] \);
1 \( sum \leftarrow 0; \)
2 \( c \leftarrow n; \)
3 for \( i \leftarrow 0 \) to \( n \) do
4 if \( dist[i] = -1 \) then \( c \leftarrow c - 1; \)
5 \( \hat{\text{else}} \) \( \sum \leftarrow \sum + dist[i]; \)
6 \( \hat{\text{else}} \) if \( c = 0 \) then \( cc[s] \leftarrow 0; \)
7 \( \hat{\text{else}} \) \( cc[s] \leftarrow ((c - 1)^2/(n - 1))/sum; \)
8 return \( cc; \)

3.4 Closeness Centrality

The closeness centrality measures the closeness between a pair of nodes in terms of their shortest path distance, in other words, for each node it computes the reciprocal of the sum of all pairwise distances within the network (Sabidussi 1966).

Therefore, for a given network and a source node \( s \), closeness centrality is computed as follows,

\[
cc(s) = \frac{1}{\sum_{t \in V} dist(s, t)}
\]  

(1)

The closeness centrality can only be applied to connected networks, yet it has a wide range of applications (Ma & Zeng, 2003; Yang & Zuluhaudar, 2011). We compute the centrality by simply extending the proposed GPU-based SSSP algorithm (Algorithm 1). The modified algorithm is demonstrated in Algorithm 3 (kNN Closeness Centrality). The idea is to run the BFS kernel from each node in each separate iteration (line 3 – line 11, Algorithm 3) and subsequently, accumulate the closeness. For a given source node, the accumulation kernel (Algorithm 4) sums all the distances stored in \( dist \) array (computed by the BFS kernel) and finds closeness.

3.5 Eccentricity Centrality

The eccentricity centrality (Harary et al. 1965) computes for every vertex, the reciprocal of the maximum (shortest path) distance to all other nodes. For a given network and a source node \( s \in V \), it is defined as,

\[
cc(s) = \frac{1}{\max \{dist(s, t) : t \in V \}}
\]  

(2)

This can be computed by updating the accumulation kernel using Algorithm 5 in Algorithm 3.

3.6 Betweenness Centrality

The (shortest path) betweenness centrality (Freeman 1977) counts the number of communications a vertex can monitor, or in other words, the rate of shortest paths experienced by an interior vertex. A node becomes more central not only for being on many shortest paths, but also on most of the shortest paths. It is based on a notion of pair-dependency \( \delta_{uv}(s) \), which measures the fractions of shortest paths between \( u \) and \( v \) passing through the node \( s \),

\[
\delta_{uv}(s) = \frac{\sigma_{uv}(s)}{\sigma_{uv}}
\]  

(3)

The betweenness centrality of \( s \) is computed by summing up all the pairwise dependencies,

\[
bc(s) = \sum_{u \neq s \neq v \in V} \delta_{uv}(s)
\]  

(4)

As the metric involves computation of shortest paths among all pairs of vertices, a straightforward implementation may require \( O(n^3) \) computations (e.g., using Floyd-Warshall algorithm). However, for a graph without loops or multiple edges, Brandes (Brandes 2001) proposed a dynamic programming approach that can reduce the search space and significantly lower the computational complexity. For a graph with \( n \) nodes and \( |E| \) edges, the metric can be computed in \( O(n|E|) \) (unweighted) \( O(n|E| + n^2 \log n) \) (weighted) times. The author introduced a notion of dependency (in contrast to the pairwise dependency in (Freeman 1977)) of a vertex \( u \in V \) on a single vertex \( s \in V \), as follows,

\[
\delta_{uv}(s) = \sum_{v \in V} \delta_{uw}(s)
\]  

(5)

Which always obeys a recursive relation. For instance, the dependency of \( u \in V \) on any \( s \in V \) obeys following relation,

\[
\delta_{uv}(s) = \sum_{w: \text{dist}(u,w) = \text{dist}(u,s)+1} \frac{\sigma_{uw}}{\sigma_{uw}}(1 + \delta_{uw}(w))
\]  

(6)
Therefore, following Brandes (Brandes 2001) the betweenness centrality of a node \( s \) can be computed by summing up all the dependencies as follows,

\[
bc(s) = \sum_{u \neq s \in V} \delta_{su}(s)
\]  

(7)

This approach is the fastest known technique for computing the betweenness centrality. However, its sequential variant is still too costly for the large graphs. Therefore a number of CPU-based parallel variants are proposed so far and many of them require highly expensive hardware setup (discussed earlier). In contrast, we propose two data-parallel variants of the Brandes’s approach on GPUs. Same as before, we consider that the input is an unweighted \( k \)NN graph and the edges are sorted by the source vertices.

3.6.1 Parallel Brandes’s Betweenness Centrality

There are two major components of the Brandes’s algorithm (Brandes 2001). A SSSP component that keeps track of the number of shortest paths (\( \sigma \)) arriving at each node and the predecessors (\( \text{Pred} \)), where the predecessors of a node is a group of adjacent nodes that are on the shortest paths to the node. An accumulation component that computes the dependency (\( \delta \)) of each predecessor node on the source node (\( s \)) and sums the betweenness centrality from the dependencies (Equation 7).

Table 2: Variables in the Brandes’s betweenness centrality (See (Brandes 2001))

<table>
<thead>
<tr>
<th>Names</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q )</td>
<td>A queue (initially empty)</td>
</tr>
<tr>
<td>( S )</td>
<td>A stack (initially empty)</td>
</tr>
<tr>
<td>( \text{Pred}[v] )</td>
<td>A list of predecessors on shortest paths from source</td>
</tr>
<tr>
<td>( p,q )</td>
<td>Two arrays of size ( k )</td>
</tr>
</tbody>
</table>

In our naïve parallelization, we only parallelize the BFS portion of the SSSP component (Algorithm 6). Our aim is to explore only the neighbors of each source node in parallel, but process (i.e., identify the respective shortest paths) in a group of adjacent nodes (predecessors) the sources sequentially. Duplicate edges are not considered during this exploration process and hence, they are removed apriori. The adapted method is presented as Algorithm 7, which we term as Parallel Brandes’s Betweenness Centrality 1. We call it naïve parallel, as it parallelizes the centrality computation partially and in an optimized way. Detailed working procedure of the method can be found in (Brandes 2001). Since, it utilizes the data structures proposed in the original algorithm (see Tables 1 and 2), it is applicable to both directed and undirected graphs.

3.6.2 Parallel Brandes’s Betweenness Centrality 2.

To improve the performance of the naïve parallel implementation, we adapt the edge parallel approach as proposed in (Shi & Zhang 2011). Same as in the closeness and eccentricity centrality, our SSSP computation takes \( k \)NN graphs straight as the input and does not require any lengthy pre-processing of neighbors (as required by the methods (Jia 2010, Shi & Zhang 2011)). However, we perform a slight modification in the BFS kernel (Algorithm 2) of the SSSP computation, as now we need to store the number of shortest paths (\( \sigma \)) arriving at each node and moreover, we need to keep track of the traversed paths. We add the following four lines at the end of the BFS kernel.

```c
S.push(v);
host -> device (p,q);
BFS (Gk, dist, \sigma, p,q, v, k) (Algorithm 6);
device -> host (p,q);
S.push(q(i)), \forall i \in k;
p[i] append -> Pred(v) \forall i \in k;
```

Algorithm 6: Naïve Parallelization of BFS in Brandes (Brandes 2001)

```c
Input : Gk, dist, \sigma, p,q,v,k;
Output: \sigma, p and q arrays are initialized;

1. tid ← thread id;
2. start ← v × k; limit ← start + k;
3. if tid ≥ start and tid < limit then
   4. i ← tid − start;
   5. w ← Gk[tid].source;
   6. if dist[w] < 0 then
      7. dist[w] = dist[v] + 1;
      8. q[i] ← w; // keep the visited nodes
   9. if dist[w] = dist[v] + 1 then
      10. atomicAdd(\sigma[w],\sigma[v]);
      11. p[i] ← w; // keep the predecessors
```

Algorithm 7: Parallel Brandes’s (Brandes 2001) Betweenness Centrality 1

```c
Input : Gk and k;
Output: Betweenness centrality of each vertex stored in bc array;

1. Create Q, S, Pred, \sigma, \delta;
2. foreach s ∈ V do
   3. Initialize dist[t] ← ∞, \( \sigma[t] \leftarrow 0 \) \forall t \in V;
   4. dist[s] ← 0; \( \sigma[s] \leftarrow 1; \)
   5. Q.push(s);
   6. while Q ≠ ∅ do
      7. p[i], q[i] ← 0, \forall i \in k;
      8. v ← Q.pop();
      9. S.push(v);
      10. host -> device (p,q);
      11. BFS (Gk, dist, \sigma, p,q, v, k) (Algorithm 6);
      12. device -> host (p,q);
      13. Q.push(q(i)), \forall i \in k;
      14. p[i] append -> Pred(v) \forall i \in k;
      15. device -> host (\sigma);
      16. Initialize \( \delta[v] \leftarrow 0, \forall v \in V; \)
      17. while S ≠ ∅ do
         18. w ← S.pop();
         19. for v ∈ Pred[w] do
            20. \( \delta[v] ← \delta[v] + \frac{\sigma[v]}{\sigma[w]} \times (1 + \delta[w]); \)
            21. if w ≠ s then bc[w] ← bc[w] + \delta[w];
      22. ;
23. return bc;
```
dependencies (\(\delta\)) and computing the betweenness centrality \((bc)\). This procedure can be seen as a **BFS in reverse direction**. For a given node, when the SSSP computation is finished along with the distances \((\text{dist})\) and shortest paths \((\sigma)\) computations, we perform the backtracing by traversing the nodes from the farthest distance to the nearest distance and at the same time, we compute dependencies \((\delta)\) from \(\sigma\). An illustration of this process is demonstrated in Figure 2, where we show the BFS (starting from node 0) in forward and reverse directions. During the forward phase (Figure 2(a)), it explores node 1, 2 and 3 in parallel at the first invocation, then node 4, 6 and 5, 7 in the subsequent invocations and the corresponding values of \(\sigma\) are also computed at the same time. Then during the reverse phase (Figure 2(b)), the exploration starts from the nodes in the farthest distance, i.e., node 5 and 7 and subsequently exploration of all the nodes in reverse order are performed along with the computation of corresponding dependencies \((\delta)\) (using Equation 6). The kernel to perform the BFS in reverse order and respective dependency computation is presented in Algorithm 8.

During the backtrace, we also need to accumulate the betweenness of the node in question. We utilize the accumulation kernel of (Shi & Zhang 2011) to perform this accumulation (Algorithm 9). The complete edge parallel variant of the Brandes’s method is presented in Algorithm 10. The algorithm is optimized to have a better performance than the naïve parallel implementation, as the number of shortest paths and respective dependencies are computed in parallel.

![Figure 2: Demonstration of BFS in forward and reverse direction and the computation of the betweenness centrality.](image)

Algorithm 8: BFS in Reverse Order

Input: \(Gk, \text{Pred}, \sigma, \delta, n,k;\)
Output: \(\delta\) is initialized;
1 \(tid \leftarrow\) thread id;
2 if \(tid < n \times k\) then
3 \(source \leftarrow Gk[tid],source;\)
4 \(target \leftarrow Gk[tid],target;\)
5 if \(\text{dist}[u] = d - 1\) then
6 found \(\leftarrow\) true;
7 else if \(\text{dist}[w] = d - 1\) then
8 swap(target, source) // device function;
9 found \(\leftarrow\) true;
10 else found \(\leftarrow\) false;
11 if found \(\leftarrow\) true then
12 \(bit \leftarrow target \times n + source;\)
13 if \(\text{Pred}(bit)\) then
14 atomicAdd\((\delta[i], \sigma[i]/\sigma[j] \times (1 + \delta[j]));\)

Algorithm 9: Accumulate Betweenness

Input: \(bc, \delta;\)
Output: Betweenness centrality of each vertex is stored in \(bc\) array;
1 \(tid \leftarrow\) thread id;
2 if \(tid < n\) then
3 if \(tid \neq s\) and \(\text{dist[tid]} = d - 1\) then
4 \(bc[tid] \leftarrow bc[tid] + \delta[tid];\)

Algorithm 10: Parallel Brandes’s (Brandes 2001) Betweenness Centrality

Input: \(Gk\) and \(k;\)
Output: Betweenness centrality of each vertex stored in \(bc\) array;
1 Initialize \(bc[v] \leftarrow 0, \forall v \in V;\)
2 foreach \(s \in V\) do
3 \(\text{Single Source Shortest Path} (Gk, s, k)\) (Algorithm 1, see BFS);
4 Initialize \(\delta[v] \leftarrow 0, \forall v \in V;\)
5 /* Backtrace and Accumulate*/;
6 while \(d > 1\) do
7 host \(\to\) device \((d);\)
8 \(\text{BFS Reverse Order} (Gk, \text{dist}, \sigma, \delta, P, d)\) (Algorithm 8);
9 \(\text{Accumulate Betweenness}\)
10 \((s, d, dist, \delta, bc)\) (Algorithm 9);
11 device \(\to\) host \((d);\)
12 device \(\to\) host \((bc);\)
13 return \(bc;\)

However, the one limitation of this approach is that it was originally designed for sparser (e.g., scale-free) graphs (Shi & Zhang 2011, Jia 2010) and it can only work with undirected graphs (due to the mechanism of forward and reverse traversal). Thus, it assumes a directed \(k\)NN graph as an undirected graph and computes the centrality respectively. In contrast, the naïve parallel implementation can handle the directed and relatively denser graphs, as it follows the original data structures as proposed in (Brandes 2001).
4 Results

4.1 Test Environment

We implemented all the proposed algorithms on the following hardware setup. A total of four NVIDIA Tesla C2050 GPU cards were installed on a X8DTG-Q Supermicro server containing 2×Intel Xeon E5620 2.4GHz processors, 32GB DDR3 RAM and 800GB of Local Hard Disk. The programs were written in C++ and CUDA (toolkit 4.0) and compiled using the g++ v4.4.4 and nvcc compilers on a Linux x86_64 OS (kernel version 2.6.9). The computational times were measured using CUDA timer utility (NVIDIA 2007).

4.2 Comparison Tools

We compared the performance of the GPU implementations against respective CPU and GPU variants (a standard implementation of Brandes’s algorithm is provided in BOOST Graph Library (Siek et al. 2000), however we used the simpler implementation given in GPU-FAN (GPU-based Fast Analysis of Networks) (Shi & Zhang 2011)). Its GPU variant and the CPU closeness and eccentricity centralities were also taken from the same source: http://bioinfo.vanderbilt.edu/gpu-fan/)

4.3 Data Collection and Preprocessing

To assess the methods performance, We utilized a renowned breast cancer gene-expression study contributed by (van de Vijver et al. 2002) (see also, (Van T Veer et al. 2002)). The original published data set has a total of 24,479 probe sets (with 1,281 control probes) in 295 breast cancer patients. For each patient the published data set has five attributes: log(ratio), log(ratio) error, p-value for log(ratio) significance, log(intensity), and a flag for each spot (= 0 for control or bad spot, = 1 for valid measurement). In this experiment we only utilized the log(ratio) attribute for each valid measurement that resulted in a total of 24,158 probe sets (for all the 295 patients/samples). To test the scalability of the proposed method, we extended our search space by creating an artificial data set: BC_Expanded, containing a total of 384,125 (details in (Arefin 2013)). The expansion was performed using ‘difference’ operator between each pair of genes. Then we created several sub data sets (n = 50 000, 75 000 and 100 000) from this extended data.

4.4 Performance Evaluation

4.4.1 Performance Evaluation

Next, using the same set of graphs, we computed the speed-up gains achieved by the two GPU variants of the Brandes’s betweenness centrality, i.e., naïve parallel (adapted from (Brandes 2001)), edge parallel (adapted from (Shi & Zhang 2011)) and compared them against the speed-up gains achieved by the original GPU variant proposed in (Shi & Zhang 2011). The results are presented in Figure 4.

We observed that the Betweenness Centrality 1 (naïve parallel) and the Betweenness-GPUFAN by Shi and Zhang (Shi & Zhang 2011) achieved the lowest and highest speed-ups, respectively. Even though the speed-ups achieved by the proposed Betweenness Centrality 2 variant was as slightly lower than the original approach (due to the workload imbalance introduced among the threads), on the largest instance only the proposed approach could scale properly in our device (due to the elimination of edge data duplication and hence, data redundancy as explained in Section 3.2). The usage of the kNN graphs instead of the replicated adjacency edge list (as proposed in (Shi & Zhang 2011)) aided us to gain the extended scalability. We found the proposed edge-parallel approach as a preferable method for the large-scale kNN graphs. However, it can only be applied to the undirected graphs. In contrast, the naïve parallel approach can be applied to directed graphs, but it performed only twice faster than the original Brandes’s algorithm (Brandes 2001).

Next, we used the original data metrics (van de Vijver et al. 2002) (n = 24 158) to construct five different kNN graphs (k = 10, 15, 20, 25 and 50) again utilized the three centrality metrics (Betweenness Centrality 2). The results are shown in Figure 5. The centralities did not receive much affects by the change in k (for a fixed value of n =25,158).

We experimented with the expanded data sets by replicating the probe lists (as proposed in (Shi & Zhang 2011)) aided us to gain the extended scalability. We found the proposed edge-parallel approach as a preferable method for the large-scale kNN graphs. However, it can only be applied to the undirected graphs. In contrast, the naïve parallel approach can be applied to directed graphs, but it performed only twice faster than the original Brandes’s algorithm (Brandes 2001).

4.4.2 Performance Evaluation

First, we applied the CPU implementations on these data sets and computed the performance gains of each of the respective GPU implementation (Close-ness, Eccentricity and Betweenness Centrality 2) (Figure 3). It can be noted that the CPU implementations were not optimized to handle the kNN graphs (which may contain a number of duplicate edges). Therefore, we created respective graph data structures for the removal of self-loops from the input graphs. The times required for the preprocessing have been added to the respective experimental evaluations. The speed-ups obtained at this stage is depicted in Figure 3.

Next, we used the original data metrics (van de Vijver et al. 2002) (n = 24 158) to construct five different kNN graphs (k = 10, 15, 20, 25 and 50) again utilized the three centrality metrics (Betweenness Centrality 2). The results are shown in Figure 5. The centralities did not receive much affects by the change in k (for a fixed value of n =25,158).
changes in the value of $k$. The reason behind this behavior can be understood from Section 3.6. For each source vertex, the parallel BFS spawn threads using the number of its nearest neighbors and hence, an increase in the value of $k$, in fact further improves the parallel hardware utilization.

### 4.5 Significance of the Highly Ranked Genes

We have plotted the intersection of top 1,000 central probes in Figure 6 that we identified using the three centrality measures (ECC - eccentricity centrality, CC - closeness centrality and BCC - betweenness centrality). The input gene co-expression network was constructed from the original data matrix ($n = 25,158$) from the $k$NN graph with fixed value of $k = \ln(n)$. This result brings some interesting insight about the data, as it may facilitate further research in the direction of finding genes that experience the most shortest paths over them (see Tables 3 and 4).

Figure 6: Intersection of the top 1,000 probes in the breast cancer data set (van de Vijver et al. 2002) identified by the three shortest path centralities, where the input gene co-expression network was constructed as a $k$NN graph for a fixed value of $k = \ln(n)$).

Table 3: Top 10 most central probes at the intersection of probes selected by two of the three centrality measures.

<table>
<thead>
<tr>
<th>CC $\cup$ BC</th>
<th>ECC $\cup$ CC</th>
<th>BC $\cup$ ECC</th>
</tr>
</thead>
<tbody>
<tr>
<td>D42055[NEDD4]</td>
<td>AA045642</td>
<td>AA044906</td>
</tr>
<tr>
<td>AW139627</td>
<td>AA101173(HESR6)</td>
<td>AA045749(CPXM2)</td>
</tr>
<tr>
<td>AI825936[BIGN5T]</td>
<td>AA131323(TGTA5)</td>
<td>A053806</td>
</tr>
<tr>
<td>AB028998[TENC1]</td>
<td>AA176629</td>
<td>AA057596</td>
</tr>
<tr>
<td>NM_002051[GATA3]</td>
<td>AA189151</td>
<td>AA059342</td>
</tr>
<tr>
<td>NM_004396[FOXA1]</td>
<td>AA190858</td>
<td>AA205599(TUD7A)</td>
</tr>
<tr>
<td>AK000604[COL4A3BP]</td>
<td>AA398575</td>
<td>AA201704</td>
</tr>
<tr>
<td>AA9035783</td>
<td>AA407470</td>
<td>AA2011418</td>
</tr>
<tr>
<td>AI135555[ADAMTS5]</td>
<td>AA403192(TAF7L)</td>
<td>AA284301</td>
</tr>
<tr>
<td>AA831836</td>
<td>AA434109(FBXL6)</td>
<td>AA406164</td>
</tr>
</tbody>
</table>

For instance, the top BC scoring gene TWISTNB - TWIST neighbor is also in correlation $\rho = 0.245141$ with patients’ times to relapse (see Figure 7). We computed the correlation using expression levels of the selected genes against patients’ times to relapse given in years (see (van de Vijver et al. 2002)), using a robust correlation function, $\rho = abs(Pearson(x, y) + Spearman(x, y))$. Further, descriptions of the identified genes are out of the scope of this work. However, apart from the hypothetical proteins, some of the genes e.g., FOXA1, GATA3 have previously been reported for their significance in causing tumors and cancers.

Table 4: Top 10 most central probes at the intersection of probes selected by the three centrality measures.

<table>
<thead>
<tr>
<th>Description</th>
<th>AA061928</th>
<th>AA013439</th>
<th>AA034111</th>
<th>AA035279</th>
<th>AA086248</th>
<th>AA121481(TWISTNB)</th>
<th>AA1290725</th>
<th>AA150107(COBL1)</th>
<th>AA192224(SMTNL1)</th>
<th>AA233912</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWIST neighbor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COBL-like 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: A scatter plot showing correlation of the top 1000 most central genes (betweenness centrality) against ‘time to relapse’ of cancer in the 295 patient (van de Vijver et al. 2002). The input gene co-expression network was constructed as a $k$NN graph for a fixed value of $k = \ln(n)$).

### 5 Conclusion

We proposed GPU implementations of three popular centrality metrics. Although, they are designed to work with the $k$NN graphs, simple modifications can make them applicable to other graphs. We choose the $k$NN graphs as the input, as our main goal is to identify the central elements from multi-variate data sets (e.g., microarrays, time series data etc.). Generally, it is not possible to make any distinction or ranking among the elements in a multi-variate data set if they are given in the form of a distance matrix (complete graph). We have shown that the $k$NN graphs can be successfully utilized in such case. Notably, they contain the most important proximity relations. We showed that the scalability of the existing GPU variant of a shortest path based centrality metrics (betweenness) can further be improved using $k$NN graphs. The proposed methods should be able to scale up to million nodes on the current series of GPUs (e.g., NVIDIA Tesla and the most recent Kepler architectures). However, an optimization of the bit matrix is required for a further scalability.

### 6 References

#### References


Jia, Y. (2010), Large graph simplification, clustering and visualization, PhD thesis, Champaign, IL, USA. AAB4309988.


