Calculation of Betweenness centrality for large Sparse Graphs

National Institute of Technology,Karnataka

Team Members:

Hardik Rana
16CO138
hardikrana276@gmail.com

Harshal Shinde
16CO223
hsharshal9@gmail.com

Date: 24/11/2018

Table Of Contents

1.PROBLEM STATEMENT

Graph analysis is a fundamental tool for domains as diverse as social networks, computational biology, and machine learning. Real-world applications of graph algorithms involve tremendously large networks that cannot be inspected manually.

A popular metric used to analyze these networks is Betweenness Centrality (BC), which has applications in community detection, finding the best locations for stores within cities, power grid contingency analysis, and the study of the human brain. However, these analyses come with a high computational cost that prevents the examination of large graphs of interest. Also it is a popular analytic that determines vertex influence in a graph.

Recently, the use of Graphics Processing Units (GPUs) has been promising for efficient processing of unstructured data sets. Prior GPU implementations of BC suffer from large local data structures and inefficient graph traversals that limit scalability and performance. Unfortunately, the fastest known algorithm for calculating BC scores has $O(mn)$ complexity for unweighted graphs with n vertices and m edges, making the analysis of large graphs challenging. Hence there is a need for robust, high-performance graph analytics that can be applied to a variety of network structures and sizes.

Here our goal is to find the accurate betweenness centrality in large sparse graphs. Sparse graphs are the ones in which the number of edges is close to the minimal number of edges. So mathematically a sparse graph is a graph G = (V, E) in which |E| = O(|V|).

2.ABOUT BETWEENNESS CENTRALITY

Betweenness Centrality determines the importance of vertices in a network by measuring the ratio of shortest path passing through a particular vertex to the total number of shortest paths between all pairs of vertices. Intuitively, this ratio determines how well a vertex connects pairs of vertices in the network.

In graph theory, betweenness centrality is a measure of centrality in a graph based on shortest paths. For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices such that either the number of edges that the path passes through (for unweighted graphs) or the sum of the weights of the edges (for weighted graphs) is minimized. The betweenness centrality for each vertex is the number of these shortest paths that pass through the vertex.

Betweenness centrality finds wide application in network theory: it represents the degree of which nodes stand between each other. For example, in a telecommunications network, a node with higher betweenness centrality would have more control over the network, because more information will pass through that node. Betweenness centrality was devised as a general measure of centrality: it applies to a wide range of problems in network theory, including problems related to social networks, biology, transport and scientific cooperation.

Formally, the Betweenness Centrality of a vertex v is defined as:

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

where σ_{st} is the number of shortest paths between vertices s and t and $\sigma_{st}(v)$ is the number of those shortest paths that pass through v.

Note that the betweenness centrality of a node scales with the number of pairs of nodes as implied by the summation indices. Therefore, the calculation may be rescaled by dividing through by the number of pairs of nodes not including v, so that $g \in [0, 1]$. The division is done by (N-1)(N-2) for directed graphs and (N-1)(N-2)/2 for undirected graphs, where N is the number of nodes in the giant component. Note that this scales for the highest possible value, where one node is crossed by every single shortest path. This is often not the case, and a normalization can be performed without a loss of precision

$$\text{normal}(g(v)) = \frac{g(v) - \min(g)}{\max(g) - \min(g)}$$

which results in:

max(normal) = 1
min (normal) = 0

Note that this will always be a scaling from a smaller range into a larger range,so no precision is lost.

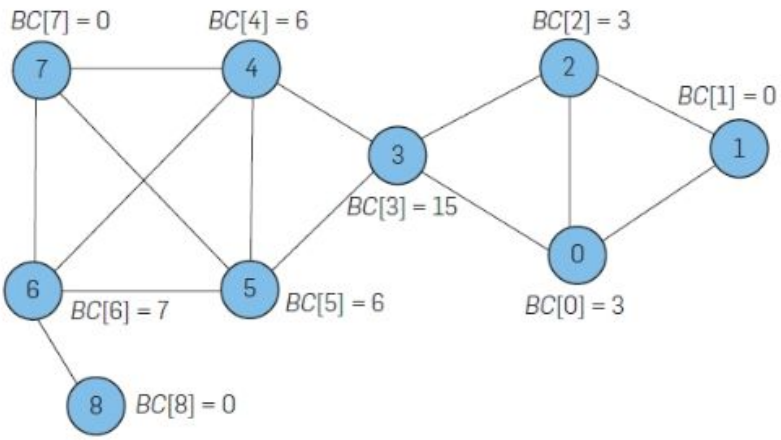


Figure 1. Sample Graph

For example in the given graph in figure1 vertex 3 is the only vertex that lies on paths from its left (vertices 4 through 8) to its right (vertices 0 through 2). Hence vertex 3 lies on all the shortest paths between these pairs of vertices and has a high BC score. In contrast, vertex 8 does not belong on a path between any pair of the remaining vertices and thus it has a BC score of 0.

Representation of Sparse graph to reduce memory uses:

Normally in graphs with small number of vertices or edges,we can represent them using adjacency matrix, but in sparse graphs we will be having very few or less edges,so if we represent them in the form of adjacency matrix then we will end up by wasting so much memory,which is not good. In the adjacency matrix element $A_{i,j}$ of the matrix is equal to 1 if an edge exists from i to j and is equal to 0 otherwise (for unweighted graphs). However, this method of storage requires $O(n^2)$ memory. So when we do parallel computation of betweenness centrality in

large sparse graphs,we will represent them in CSR [compressed sparse row format] format.

This CSR representation consists of two arrays: *column indices* and *row offsets*. The column indices array is a concatenation of each vertex's adjacency list into an array of m elements. The row offsets array is an $n + 1$ element array that points at where each vertex's adjacency list begins and ends within the column indices array.

For example, the adjacency list of vertex u starts at C[R[u]] and ends at C[R[u+1]-1] (inclusively). For example, for the given graph of figure1, we will have following CSR representation.

R = [0, 3, 5, 8, 12, 16, 20, 24, 27, 28]
C = [1,2,3 | 0,2 | 0,1,3 | 0,2,4,5 | 3,5,6,7 | 3,4,6,7 | 4,5,7,8 | 4,5,6 | 6]

3. OUR APPROACHES

Following section describe our all approaches [serial and parallel (cuda and mpi)] to find betweenness centrality in the given sparse graphs.

3.1 SERIAL IMPLEMENTATION

In a weighted network the links connecting the nodes are no longer treated as binary interactions, but are weighted in proportion to their capacity, influence, frequency, etc., which adds another dimension of heterogeneity within the network beyond the topological effects. A node's strength in a weighted network is given by the sum of the weights of its adjacent edges.

$$s_i = \sum_{j=1}^{N} a_{ij} w_{ij}$$

With a_{ij} and w_{ij} being adjacency and weight matrices between nodes i and j , respectively. Analogous to the power law distribution of degree found in scale free networks, the strength of a given node follows a power law distribution as well.

$$s(k) \approx k^{\beta}$$

A study of the average value of s(b) the strength for vertices with betweenness b shows that the functional behavior can be approximated by a scaling form.

$$s(b) \approx b^{\alpha}$$

To find betweenness centrality for the graph in serial implementation,we have taken the help of the python library called **networkx .** To know more about networkx library go to: http://networkx.readthedocs.io/en/networkx-1.10/index.html

Following is the sample code to find the between centrality for the graph shown in figure1.

```
import timeit                                   #Step1
import networkx as nx                           #Step2
Start = timeit.default_timer()                  #Step3
G = nx.Graph()                                  #Step4
G.add_nodes_from([0,1,2,3,4,5,6,7,8])           #Step5
#Step6
elist = [(0,1),(0,2),(0,3),(1,2),(2,3),(3,4),(3,5),(4,5),(4,6),(4,7),(5,6), (5,7),(6,7),(6,8)]
G.add_edges_from(elist)                         #Step7
#G=nx.erdos_renyi_graph(50,0.05)                #Step8
b=nx.betweenness_centrality(G)                  #Step9
print(b)                                        #Step10
Stop = timeit.default_timer()                   #Step11
print('Total execution time: ', Stop-Start);    #Step12
```

networkx library can be installed by the following command: pip install networkx

Detailed explanation of the steps of the code.

1. In **Step1** we are importing the python timeit library which is used to calculate the total execution time of our serial code.

2. In **Step2** we are importing the python networkx library which is used to calculate the betweenness centrality of the given graph in figure1.

3. In **Step3** we are storing the initial (starting) time of the program in Start variable,which is used at the end to calculate total execution time.

4. In **Step4** we are creating one empty graph with no nodes and edges. `Graph()` is an inbuilt function defined in networkx library which create an empty graph with no nodes and edges. In networkx, nodes of the graph can be any hashable object e.g., a text string, an image, an XML object, another Graph, a customized node object, etc.

5. In **Step5** we are adding nodes from the list [0,1,2,3,4,5,6,7,8] into the empty graph created in step4. For this step we are taking help of networkx function `add_nodes_from`

   `add_nodes_from(nodes_for_adding, **attr)` : This will Add multiple nodes.

Parameters:

- **nodes_for_adding** (iterable container) – A container of nodes (list, dict, set, etc.) OR A container of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.

- **attr** (keyword arguments, optional (default= no attributes)) – Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified via keyword arguments.

6. In **Step6,** we are just creating a list of pairs,which contain all pairs of nodes which are having edges between then in graph of figure1.

7. In **Step7**, we are adding the list of pairs of nodes which are having edge between them [created in step6] to the graph created in step4 and step5.
   For this step we are taking help of networkx function: add_edges_from

add_edges_from**(ebunch_to_add, **attr) :** Add all the edges in ebunch_to_add.

Parameters:

- **ebunch_to_add** (container of edges) – Each edge given in the container will be added to the graph. The edges must be given as 2-tuples (u, v) or 3-tuples (u, v, d) where d is a dictionary containing edge data.

- **attr** (keyword arguments, optional) – Edge data (or labels or objects) can be assigned using keyword arguments.

8. In **Step8**, we are just creating random graph of 50 nodes for just testing Purpose [It is not useful for our purpose,so it is commented].

9. In **Step9,** we are calculating the betweenness centrality of the graph created in the above steps by just invoking the networkx library's inbuilt function: betweenness_centrality

betweenness_centrality**(G, k=None, normalized=True, weight=None, endpoints=False, seed=None)**

Parameters:

- **G** (graph) – A NetworkX graph.
- **k** (int, optional (default=None)) – If k is not None use k node samples to estimate betweenness. The value of k <= n where n is the number of nodes in the graph. Higher values give better approximation.
- **normalized** (bool, optional) – If True the betweenness values are normalized by **2/((n-1)(n-2))** for graphs, and **1/((n-1)(n-2))** for directed graphs where **n** is the number of nodes in G.
- **weight** (None or string, optional (default=None)) – If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight.

- **k** (int, optional (default=None)) – If k is not None use k node samples to estimate betweenness. The value of k <= n where n is the number of nodes in the graph. Higher values give better approximation.
- **endpoints** (bool, optional) – If True include the endpoints in the shortest path counts.
- **seed** (integer, random_state, or None (default)) – Indicator of random number generation state.

10. In **Step10,** we are just printing the betweenness centrality for each node of The grpah in figure1.

11. In **Step11** and **Step12** , we are just calculating stop time (finishing time) and total execution time from that stop time and start time.

So this is all about the serial implementation.
The result of the serial implementation is added in the results section. To do this calculation we have taken help from this reference:
https://www.geeksforgeeks.org/betweenness-centrality-centrality-measure/

3.2 PARALLEL IMPLEMENTATION

This section contains the parallel implementation for calculating betweenness centrality in both cuda and mpi.

3.2.1 CUDA IMPLEMENTATION

Step 1: We fixed the node for which we are calculating the betweenness centrality based on threadIdx.x, it is referred to as the centrality node here

onwards. The total threads alloted is equal to the the total number of nodes in the graph.

Step 2: The source node was fixed and the shortest path to every other node in the graph was calculated using breadth first search. This gave us $sigma_{st,}$ where 's' is the fixed source and 't' is the every other node in the graph except for the centrality node.

Step 3: The idea was that if the node for which we are calculating the centrality is a neighbour of the node which is currently visited (we used the bfs traversal to travel the graph) then if we join this node to any neighbour of the centrality node then we will get the shortest path of traversal from the fixed source to the neighbour node of the centrality node, we can also say that the path will pass through the centrality node, so this will also give us $sigma_{st}(V)$ where V is the centrality node.

Step 4: Then we divide the $sigma_{st}(V)$ and $sigma_{st}$.

Step 5: Then take the summation of the quotients gives us the betweenness centrality for the centrality node.

Step 6: We run the loop to calculate the centrality for all the nodes, and we choose all the nodes in the graph as source nodes and calculate the shortest path from them to all other nodes, however as this will generate the same path two times one from 's' to 't' and from 't' to 's', so we divide the total number of shortest paths by 2.

Below here is the cuda code for the sample graph,shown in figure1.

- Cuda Code

```
#include<cuda.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
using namespace std;

#define num 9
#define num1 27
#define pt1 10
#define sig 500

__device__ void BFS(vector<int> adj[], int src, int dist[], int paths[], int n,int curr_ind)
{
    bool visited[n];
    for (int i = 0; i < n; i++)
        visited[i] = false;
```

```
dist[src] = 0;
paths[src] = 1;
int sigma[sig] = {0};
queue <int> q;
q.push(src);
visited[src] = true;
while (!q.empty())
{
    int curr = q.front();
    q.pop();
    // For all neighbors of current vertex do:
    for (auto x : adj[curr])
    {
        if(x == curr_ind)
       {
           for(auto x1: adj[curr_ind])
           {
               sigma[x1]++;
               // if the current vertex is not yet  visited, then push it to the queue.
               if (visited[x1] == false)
               {
```

```
            q.push(x1);
            visited[x1] = true;
        }
        // check if there is a better path.
        if (dist[x1] > dist[curr] + 1)
        {
            dist[x1] = dist[curr] + 1;
            paths[x1] = paths[curr];
        }
        // additional shortest paths found
        else if (dist[x1] == dist[curr] + 1)
            paths[x1] += paths[curr];
    }
}
else
{
// if the current vertex is not yet  visited, then push it to the queue.
if (visited[x] == false)
{
    q.push(x);
    visited[x] = true;
```

```
            }
            // check if there is a better path.
            if (dist[x] > dist[curr] + 1)
            {
                dist[x] = dist[curr] + 1;
                paths[x] = paths[curr];
            }

            // additional shortest paths found
            else if (dist[x] == dist[curr] + 1)
                paths[x] += paths[curr];
            }
        }
    }
}
// function to find number of different shortest paths form given vertex s.
// n is number of vertices.

__global__void findShortestPaths(vector<int> adj[], int n, int dist[], int paths[])
{
    for (int i = 0; i < n; i++)
```

```
        dist[i] = INT_MAX;
    for (int i = 0; i < n; i++)
        paths[i] = 0;

    int curr_ind = threadIdx.x;
    for(int i=0;i<n && i!=curr_ind;i++)
    {
    sum = 0;
    BFS(adj, i, dist, paths, n,curr_ind);
    cout << "Numbers of shortest Paths are: ";
    for (int i = 0; i < n; i++)
    {
        cout << paths[i] << " ";
        sum = sum + (sigma[i] / (0.5 * paths[i]));
    }
        cout << " Betweenness centrality of node" << curr_ind <<" ";
        cout << sum <<" ";
}
// A utility function to add an edge in a  directed graph.
void addEdge(vector<int> adj[], int u, int v)
{  adj[u].push_back(v); }
```

```
int main()
{

        int n = 7;
         int *adj = (int*)malloc(sizeof(int)*n);
         int *dist = (int*)malloc(sizeof(int)*n);
         int *paths = (int*)malloc(sizeof(int)*n);

        addEdge(adj, 0, 1);
        addEdge(adj, 0, 2);
        addEdge(adj, 1, 2);
        addEdge(adj, 1, 3);
        addEdge(adj, 2, 3);
        addEdge(adj, 3, 4);
        addEdge(adj, 3, 5);
        addEdge(adj, 4, 6);
        addEdge(adj, 5, 6);

       cudaMalloc((void **)&d_adj, sizeof(int)*n);
       cudaMalloc((void **)&d_dist, sizeof(int)*n);
```

```
        cudaMalloc((void **)&d_path, sizeof(int)*n);


        cudaMemcpy(d_adj, adj, n * sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_dist, dist, n * sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_path, path, n * sizeof(int), cudaMemcpyHostToDevice);

         findShortestPaths<<<1,7>>>(d_adj, n, d_dist, d_path);

         cudaMemcpy(adj, d_adj, n * sizeof(int), cudaMemcpyDeviceToHost);
         cudaMemcpy(dist, d_dist, n * sizeof(int), cudaMemcpyDeviceToHost);
         cudaMemcpy(path, d_path, n * sizeof(int), cudaMemcpyDeviceToHost);

         cudaFree(d_adj);
         cudaFree(d_dist);
         cudaFree(d_path);
}
```

The details about the result of the parallel implementation is added in the results section. To do this calculation we have taken help from this reference: https://www.geeksforgeeks.org/number-shortest-paths-unweighted-directed-graph/

3.2.2 MPI IMPLEMENTATION

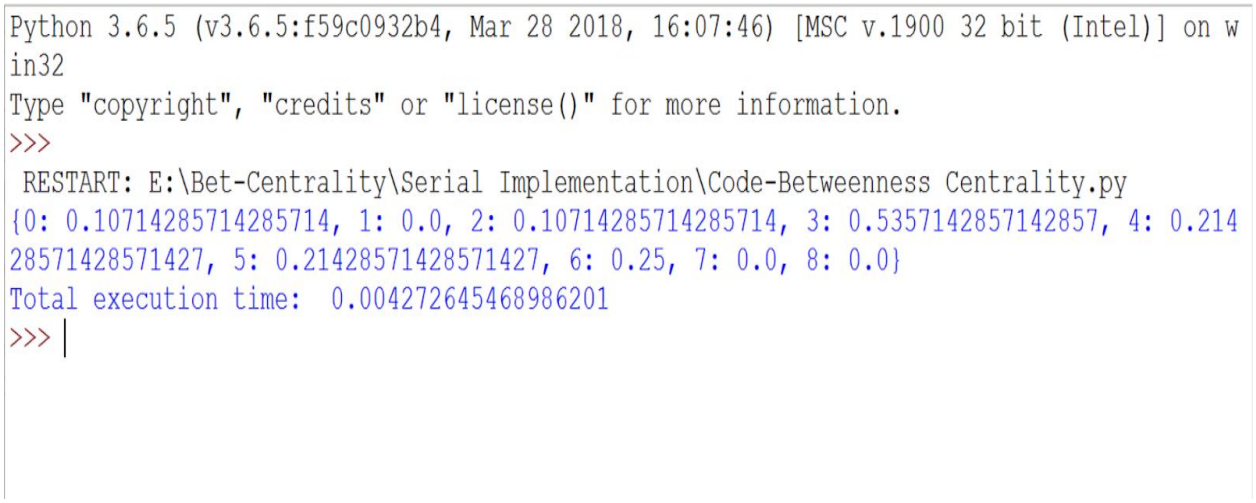
We have not implemented the mpi implementation for finding betweenness centrality by ourselves.

We have just taken that implementation from the following link for reference purpose only [to compare the performance of that code with our cuda code] : https://github.com/ssgandham/Betweenness-Centrality-using-MPI

4. RESULTS

This section contains the results of the serial and parallel implementation for finding betweenness centrality in graphs.

The following image is the result of the serial code described in the section 3.1
[Note: we haven't tried it for large graphs]



For parallel implementation, we are getting error at some line of the code, we have tried a lot to solve that error,but somehow not able to resolve that error. So we could not get the result for parallel (cuda) implementation for finding betweenness centrality.

5. REFERENCES

1. https://devblogs.nvidia.com/accelerating-graph-betweenness-centrality-cuda/
2. https://cacm.acm.org/magazines/2018/8/229768-accelerating-gpu-betweenness-centrality/fulltext
3. https://en.wikipedia.org/wiki/Betweenness_centrality
4. https://github.com/settyblue/BetweennessCentrality
5. https://www.geeksforgeeks.org/number-shortest-paths-unweighted-directed-graph/
6. https://www.sci.unich.it/~francesc/teaching/network/betweeness.html
7. https://www.coursera.org/lecture/python-social-network-analysis/betweenness-centrality-5rwMl
8. https://www.researchgate.net/publication/289502777_On_Betweenness_Centrality_Measures_of_Large_Scale_Networks
9. https://www.researchgate.net/publication/261959412_Finding_k-highest_betweenness_centrality_vertices_in_graphs
10. http://algo2.iti.kit.edu/schultes/hwy/betweennessFull.pdf

6. CONCLUSION

We observed a significant decrease in execution time of the betweenness centrality on a parallel implementation when compared to the serial implementation for large graphs. This is because in the serial implementation the betweenness centrality of each node was calculated one by one while in the parallel implementation the calculation of the betweenness centrality for each node is being done parallely. However for small graphs the serial implementation is faster as the parallel implementation gets occupied in the overheads associated with the allocation and deallocation of the device memory.