CS 406 - Progress Report

Parallelizing Closeness Measures on Graphs

24071 - Barış Batuhan Topal *Furkan Ergün - 24083*
20588 - Çağhan Köksal *Hakan Ogan Alpar - 23462*

A Detailed Analysis of Our Model

In this project, we aimed to create a parallel running model that calculates all the degree 1 & 2, closeness and betweenness centralities at once (or a subset of these measurements based on the condition passed into this function. And for this step, we have successfully managed to create this function and apply CPU parallelism by using OpenMP.

As specified in the proposal report, graphs we have worked on are assumed as undirected and unweighted graphs. These graphs are read and stored by using *row sparse matrix* technique as we indicated in the previous report. For the degree 2, closeness and betweenness centralities, the functions are provided below ($\sigma_{s,\,t}$ is the number of shortest paths from s to t):

$$deg2(v) = \{u \varepsilon V : dist(v, u) \leq 2\}$$

$$CC(v) = \sum_{u \varepsilon V} \frac{n^2}{dist(u, v)} \qquad BC(v) = \sum_{s \neq v \neq t \varepsilon V} \frac{\sigma_{s,t}(v)}{\sigma_{s,\,t}\, x\, n^2}$$

The more the number of nodes in graphs increase, the smaller the closeness centrality and the bigger the betweenness centrality scores become. Therefore, for scaling purposes the actual results are multiplied with square of the number of nodes in the graph in closeness centrality calculations and divided with the same value for betweenness centrality measurements.

In the main model, a coarse grained approach [3] is being used for parallelizing purposes. Regarding this approach, all the computations starting with each source vertex are seen as separate tasks and calculated parallelly. While this model ensures a communication overhead as little as possible, it may also bring some disadvantages. First of all, if the workload between threads is not balanced, then the speedup may be lower than expected and lastly, if each of the tasks requires a big amount of memory, then physical memory may be exceeded with the increasing number of threads. However, in our model running centrality algorithms and BFS for each single node provides a balanced workload between threads and for the sizes of graphs we have worked on so far, memory did not create any problem. Moreover, coarse grained design created a good baseline for including GPU to the model afterwards, since it enables that the individual processes in a single thread may run in GPUs in parallel. The flow of our model can be seen below in Algorithm 1:

```
Algorithm 1: Source Parallel Centrality Measurement Algorithm

Input:
$g \leftarrow$ a graph to analyse
$n \leftarrow$ number of nodes in $g$
$wanted \leftarrow$ a boolean array with size 4, set true for wanted measurements [deg1, deg2, cc, bc]
Output: $result \leftarrow$ array in 2D with shape 4 x $n$
Algorithm:
$results[i][j] \leftarrow 0$ for $i \in [0, 3]$ and $j \in [0, n)$
for $s \in g.nodes$ in parallel do
    $dist[t] \leftarrow -1$ for $t \in [0, n)$
    $dist\_counter[color] \leftarrow 0$ for each possible $color \in [0, n)$
    $\sigma[t] \leftarrow 0$ for $t \in [0, n)$
    $Q[t] \leftarrow 0$ for $t \in [0, n)$, holds nodes read in BFS in non-decreasing distances
    if $wanted[0]$ is true then $result[0][s] \leftarrow g.row\_ptr[s+1] - g.row\_ptr[s]$;
    bfs(g, n, s, dist, dist_counter, Q, $\sigma$, $wanted$)
    if $wanted[1]$ is true then $result[1][s] \leftarrow dist\_counter[2]$;
    if $wanted[2]$ is true then $result[2][s] \leftarrow step\_closeness(n, dist\_counter)$;
    if $wanted[3]$ is true then $step\_betweenness(result, g, Q, dist, \sigma)$ ;
    return result
end
```

Because of the row sparse matrix technique, degree 1 centrality is easily calculated at first for each source node in O(1) time. However, for the other centrality measurements a BFS result is needed. After running BFS, each centrality measure is calculated separately.

For the BFS algorithm, a top-down approach is preferred for this step. However, a hybrid approach will be implemented for the following steps, since it provides a faster calculation time [4]. To fasten the calculation times for degree 2 and closeness centralities, the number of vertices for each single distance is also saved during BFS to a parameter called *dist_counter*. In the betweenness centrality part of the whole implementation, the algorithm of Brandes [1] is preferred since it is the state-of-art algorithm for exact calculation [2]. Therefore, in BFS function 2 more additional parameters *Q and* σ are also added. While Q holds the visited order of nodes with non-decreasing distance values, σ holds the number of shortest paths from source vertex to another selected vertex. BFS implementation s as follows:

```
Algorithm 2: BFS

Algorithm:
$Q \leftarrow$ add the node $s$
$dist[s] \leftarrow 0$
$dist\_counter[0] \leftarrow dist\_counter[0] + 1$
$\sigma[d] \leftarrow 1/n^2$
$front \leftarrow 0$
for $front <$ number of elements in $Q$ do
    $v \leftarrow$ Q[front]
    front $\leftarrow$ front + 1
    if dist[v] == 2 and not wanted[2] and not wanted[3] then break;
    for all neighbors w of v in g do
        if dist[w] $< 0$ then
            Q $\leftarrow$ add w
            dist[w] = dist[v] + 1
            dist_counter[dist[w] $\leftarrow$ dist_counter[dist[w]] + 1
        else
        end
        if wanted[3] and dist[w] == dist[v] + 1 then $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
    end
end
```

Algorithm 3: Step Closeness

```
Output: closeness centrality value of node
Algorithm:
$sum \leftarrow 0$
for $i \in [1, dist\_counter.size)$ do
    if $dist\_counter[i]$ is 0 then break ;
    sum ← sum + (i x dist_counter[i])
end
return $n^2/sum$
```

Algorithm 4: Step Betweenness

```
Algorithm:
$\delta[t] \leftarrow 0$ for $t \in [0, n)$
for $w$ from Q.end to Q.start do
    for neighbors $v$ of $w$ in $g$ do
        if $dist[w] == dist[v] + 1$ then $\delta[v] \leftarrow \delta[v] + (\sigma[v] / \sigma[w]) \times (1 + \delta[w])$;
    end
    result[3][w] ← result[3][w] + $\delta[w]$
end
```

Closeness centrality is also implemented as stated on the left side. Since the number of vertices for each single distance is calculated beforehand, the complexity for calculating closeness values is decreased from O(n) to O(max_distance).

For betweenness, the approach of Brandes is followed. For each vertex in Q, an addition to the result is made. Since the calculation runs in parallel, this update is made atomically.

In the overall structure, to decrease memory complexity, creation of the predecessor array and 2 different parameters as queue and stack in the Brandes's algoritm are not followed. Only one array Q is created for both queueing and stacking processes and instead of the predecessor array, distance check of the neighbors of each node in Q is preferred in the betweenness step (Algorithm 4). To decrease the total execution time, changing sizes of the vectors is avoided. In the C++ implementation, all vector parameters are initialized with a predetermined size and their sizes are not updated during the whole process.

The execution times are tested in 3 different types of graphs taken from Suite-Sparse Matrix Collection [5]: small (50 graphs) , medium (3 graphs) and large (2 graphs) sized. Also most of the graphs are chosen from different problem domains. For small graphs, the computations are repeated 50 times and for medium sized ones 10 times. Since the number of small graphs is so many, the average result of all of them will be given. The results are as follows (time in seconds):

Graphs	Num Nodes	Num Edges	Threads 1	Threads 2	Threads 4	Threads 8	Threads 16
small avg.	252.10	1895.44	0.0088	0.0058	0.0042	0.0029	***0.0023***
medium 1	11125	67400	9.28654	4.99327	2.77868	1.60677	***0.900295***
medium 2	10937	75488	25.9425	13.9101	7.69223	4.03841	***2.12877***
medium 3	10429	57014	6.9701	3.79833	2.15311	1.3581	***0.798775***
large 1	102158	406858	1785.31	962.854	514.273	264.684	***139.762***
large 2	156317	1059331	4578.67	2342.54	1198.08	631.664	***332.555***

Main Overheads

There were several overheads that caused problems during programming. Starting with first touch policy, the graph is read to the program during the main thread so the memory distribution is not even and when different CPUs try to access this memory there's a communication overhead. Mostly these are used for traversal and are read only so this happens until the graph is in the cache of different CPUs, but when the graph is large different caches of the CPU's are not able to hold them and then the CPU's need to pay this uneven memory usage cost cost more often. Second overhead came with scheduling, in our program we are computing centralities for each node and we tried all three different scheduling algorithms. Dynamic was the winner and the reason was the following: work required to compute the centralities for a given node may depend on the number of neighbours this node has, if static was used, there would be an assumption that all nodes have a similar number of neighbours. Guided also gives large chunk sizes at the beginning and depending on graph structure this could also lead to problems.

Our next overhead came from trying to parallelize breadth first search. We tried many techniques, first was a brute force technique using critical regions on our push to queue and pop from queue operations. This was not very scalable because main operation is the graph traversal in our program so the main work came from the push and pop operations, making them critical was close to turning the whole bfs search sequential. Our second method was to use a top down bfs algorithm where each thread would get their own temporary queue at each level iteration, and after the level was complete they would merge the queues again. This was a little better than running the critical bfs algorithm but there were still big problems. Copying and merging the queues were costly, and also the cost of having a shared array to check for visited nodes were costly. There were some improvements in this approach, but it was not scalable. Here we decided to change our approach from just trying to parallelize BFS to trying to parallelize compute centralities function for each node. This resulted in a linear speed up with the number of threads.

Complexity of Algorithm

As mentioned before, the project works on unweighted graphs. Then let:

- P_s= {i in V | (i, v) in E && dist(s, v) = dist(s, i) + 1} where G=(V, E) such that V is the vertices of the graph and E is the edges of the graph.

As the Lemma's from 'A Faster Algorithm for Betweenness Centrality' by Brandes [1] explains, the Lemma's are:

- Number of Shortest Paths between s and v is equal to total number of Shortest Paths between s and u such that u $\in P_s(v)$
- If there is one and only one Shortest Path from s to any vertex v $\in$ V, then $\delta(s|t) = \mathbf{\Sigma}_{k:t \text{ in } Ps(k)} (1 + \delta(s|k))$

Then the way Brandes Algorithm works is:

- Performs a BFS traversal for the graph. The BFS traversal computes the number of Shortest Paths that goes through a given node v ∈ V.
- Then by using this δ(s,t|k) can be used to calculate δ(s|k), which decreases the amount of computation needed.

Therefore Space Complexity for the betweenness centrality algorithm to work is O(E+V) and time complexity is O(E*V) for a Graph G=(V, E) where V is the number of vertices and E is the number of edges.

For the degree centrality, the approach is in it's most naive yet useful form. Algorithm just calculates the degree of every vertex by iterating through every vertex with the time complexity of O(V). Furthermore, for the closeness centrality, the BFS part of the algorithm is modified as explained above. Furthermore, the BFS part improves calculating closeness values to O(max_distance) from O(V) as stated in the detailed analysis part of the paper. Therefore, complexity of the closeness is O(V*max_distance).

Speed-up and Scalability

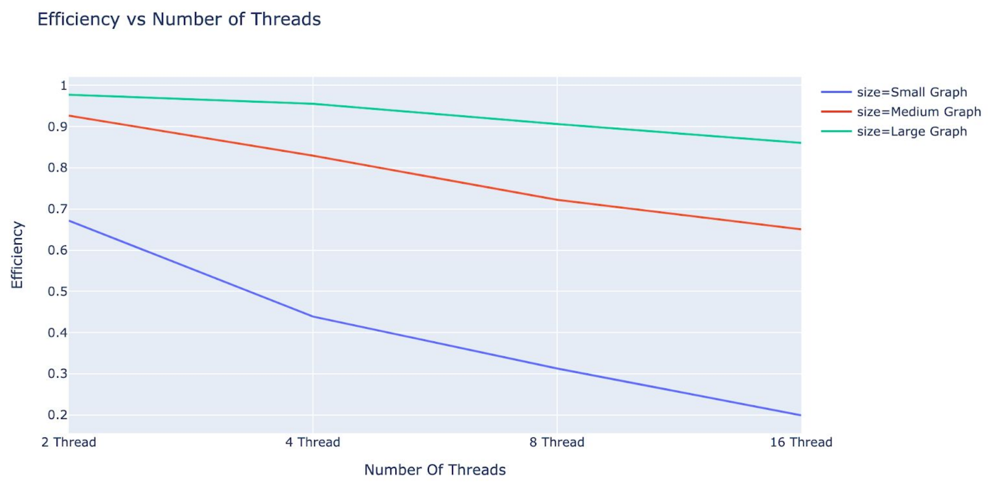

Figure 1 : Efficiency vs Number of Threads

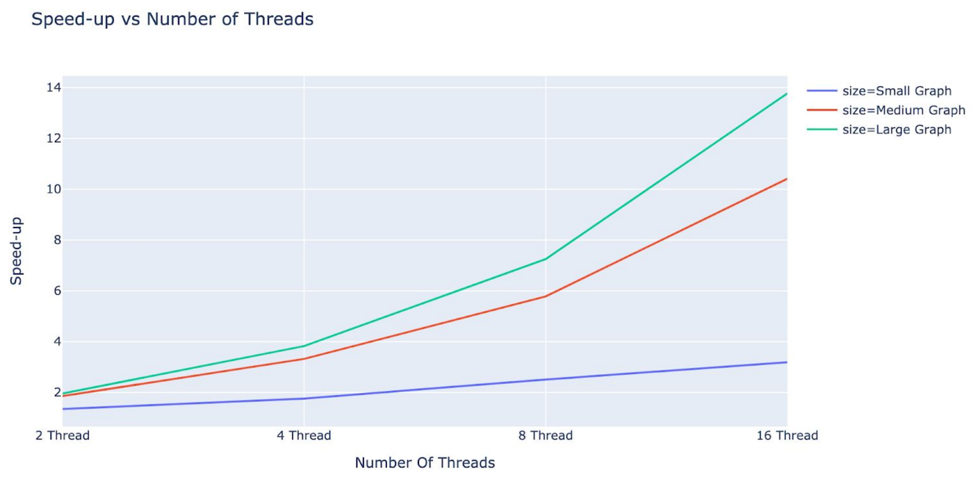

Figure 2: Speed-up vs Number of Threads

Our algorithm has almost linear speed up on medium and large graphs whereas, on small graphs, speed up stays sub-linear. While calculating betweenness centrality, at each iteration threads update the result vector, which is shared among all threads. In order to avoid mutual exclusion, updates occur in an atomic manner and since the number of nodes in small graphs are fewer, updates occur more frequently and threads wait for each other which prohibits expected speed-up on small graphs. Likewise, because of the mutual exclusion, efficiency also decreases, if the size of the graph also decreases.

References

[1] Brandes U. (2001) ‘A Faster Algorithm for Betweenness Centrality’. Available at: https://www.eecs.wsu.edu/~assefaw/CptS580-06/papers/brandes01centrality.pdf

[2] Baglioni M., Geraci F., Pellegrini M., Lastres E. ‘Fast Exact and Approximate Computation of Betweenness Centrality in Social Networks’. Available at: http://wwwold.iit.cnr.it/staff/marco.pellegrini/papiri/betweenness-full.pdf

[3] Tan G., Tu D., Sun N. (2009) ‘A Parallel Algorithm for Computing Betweenness Centrality’. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.668.8430&rep=rep1&type=pdf

[4] Beamer S., Asanovíc K., Patterson D. (2012) ‘Direction-Optimizing Breadth-First Search’. Available at: http://www.scottbeamer.net/pubs/beamer-sc2012.pdf

[5] Suite-Sparse ‘A Suite of Sparse Matrix Software’ [online]. Available at: http://faculty.cse.tamu.edu/davis/suitesparse.html