Parallelizing Centrality Measures

Barış Batuhan Topal
Çağhan Köksal
Furkan Ergün
Hakan Ogan Alpar

Contents

Problem Definition

Our Approach

Performance

References

Problem Definition

Graphs [1]:

- Google formalizes data mining and machine learning problems as graph problems.
- Graphs are also used in network systems (Star topology, Ring topology).
- Google uses this data structure in “Google Maps” by representing the roads that connect different places as edges.
- Facebook analyzes their social network by representing each person as a vertex and their relations as edges.
- Graph representations are also useful in solving many software engineering problems such as Travelling Salesman and Shortest Path Problems.

→ Therefore, in today’s world efficiency of graph algorithms is the key.

Centrality:

- Important concept in identifying the important nodes.
- Different types of centralities with different information gains.

Problem Definition

Degree One Centrality:

- For each single node, number of links held by this node is calculated.

Degree Two Centrality:

- For all vertices, the number of their neighbors and their neighbors' neighbors are found. In other words, for each single vertex, Breadth-First Search (BFS) algorithm is run and total count of nodes is computed, which have a distance closer than or equal to 2.

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

Closeness Centrality:

- A score is assigned to each node, based on its closeness to other nodes. The closeness of each node to other nodes is found by running BFS and adding all the distances found as a result.

$$CC(v) = \sum_{u \in V} \frac{n^2}{dist(u, v)}$$

Problem Definition

Betweenness Centrality:

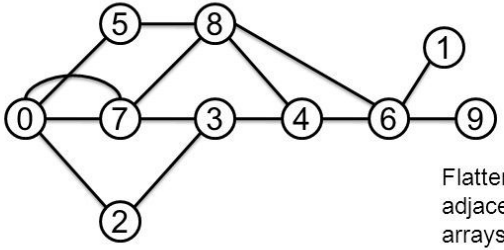
- Measures the number of shortest paths between 2 different nodes (s and t), in which our vertex (v) lies on. The more our node is between these 2 other nodes, the higher will be the value for our node. The shortest paths between 2 nodes are determined by BFS algorithms.

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{s,t}(v)}{\sigma_{s,t} \, x \, n^2}$$

Compressed Sparse Row Representation [3]

Graph representation

- Compressed Sparse Row-like



Vertex	Degree	Adjacencies			
0	4	2	5	7	7
1	1	6			
2	2	0	3		
3	3	2	4	7	
4	3	3	6	8	
5	2	0	8		
6	4	1	4	8	9
7	4	0	0	3	8
8	4	4	5	6	7
9	1	6			

Flatten adjacency arrays

Index into adjacency array

0	4	5	7	…	28

Size: n+1

Adjacencies

2	5	7	7	6	0	3	2	4	….	6	7	6

Size: 2*m

Minimal overhead allowing for fast traversals, lookups

Degree 1 Centrality

Space Efficiency

Spatial Locality

Our Approach

- For the implementations, self developed algorithms are developed based on the equations.
- In the betweenness centrality part of the whole implementation, the algorithm of Brandes [5] is preferred since it is the state-of-art algorithm for exact calculation [6].

Algorithm 1: Betweenness centrality in unweighted graphs

$C_B[v] \leftarrow 0,\ v \in V$;
for $s \in V$ **do**
  $S \leftarrow$ empty stack;
  $P[w] \leftarrow$ empty list, $w \in V$;
  $\sigma[t] \leftarrow 0,\ t \in V$; $\sigma[s] \leftarrow 1$;
  $d[t] \leftarrow -1,\ t \in V$; $d[s] \leftarrow 0$;
  $Q \leftarrow$ empty queue;
  enqueue $s \rightarrow Q$;
  while Q *not empty* **do**
    dequeue $v \leftarrow Q$;
    push $v \rightarrow S$;
    foreach *neighbor* w *of* v **do**
      // *w found for the first time?*
      if $d[w] < 0$ **then**
        enqueue $w \rightarrow Q$;
        $d[w] \leftarrow d[v] + 1$;
      end
      // *shortest path to w via v?*
      if $d[w] = d[v] + 1$ **then**
        $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
        append $v \rightarrow P[w]$;
      end
    end
  end
  $\delta[v] \leftarrow 0,\ v \in V$;
  // *S returns vertices in order of non-increasing distance from s*
  while S *not empty* **do**
    pop $w \leftarrow S$;
    for $v \in P[w]$ **do** $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$;
    if $w \neq s$ **then** $C_B[w] \leftarrow C_B[w] + \delta[w]$;
  end
end

Our Approach

Algorithm 1: Source Parallel Centrality Measurement Algorithm

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

Algorithm 2: BFS

```
Algorithm:
Q ← add the node s
dist[s] ← 0
dist_counter[0] ← dist_counter[0] + 1
σ[d] ← 1/n^2
front ← 0
for front < number of elements in Q do
    v ← Q[front]
    front ← 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 ← add w
            dist[w] = dist[v] + 1
            dist_counter[dist[w] ← dist_counter[dist[w]] + 1
        else

        end
        if wanted[3] and dist[w] == dist[v] + 1 then σ[w]← σ[w] + σ[v];
    end
end
```

Algorithm 3: Step Closeness

```
Output: closeness centrality value of node
Algorithm:
sum ← 0
for i ∈ [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:
δ[t] ← 0 for t ∈ [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 δ[v]← δ[v] + (σ[v] / σ[w]) x (1 + δ[w]);
    end
    result[3][w] ← result[3][w] + δ[w]
end
```

Our Approach

```
__global__
void bfs(int *rp, int *ci, int *dist, int *dist_counter, int *queue, int *q_size, double *sigma, int *n, int *improved, int level) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    __syncthreads();

    if(tid < *n){
        for(int edge = rp[tid]; edge < rp[tid + 1]; edge++) {
            int adj = ci[edge];
            if(dist[adj] == level && dist[tid] < 0) {
                dist[tid] = level + 1;
                atomicAdd(&dist_counter[dist[tid]], 1);
                *improved = 1;
                atomicAdd(q_size, 1);
                atomicCAS(&queue[*q_size], -1, tid);
            }
            if(dist[adj] == level && dist[tid] == level + 1) {
                sigma[tid] += sigma[adj];
            }
        }
    }
}
```

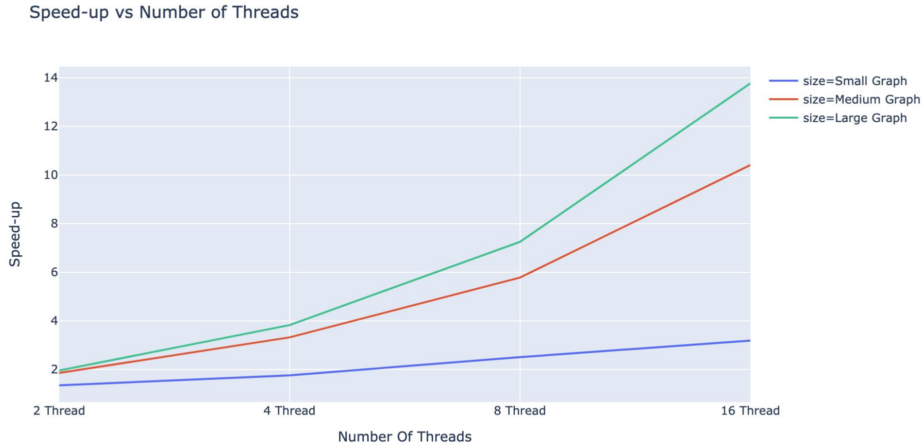
```
__global__
void closeness_step(int *dist_counter, int* n, int *sum) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid > 0 && tid <= *n && dist_counter[tid] > 0) {
        int partial = tid * dist_counter[tid];
        atomicAdd(sum, partial);
    }
}
```

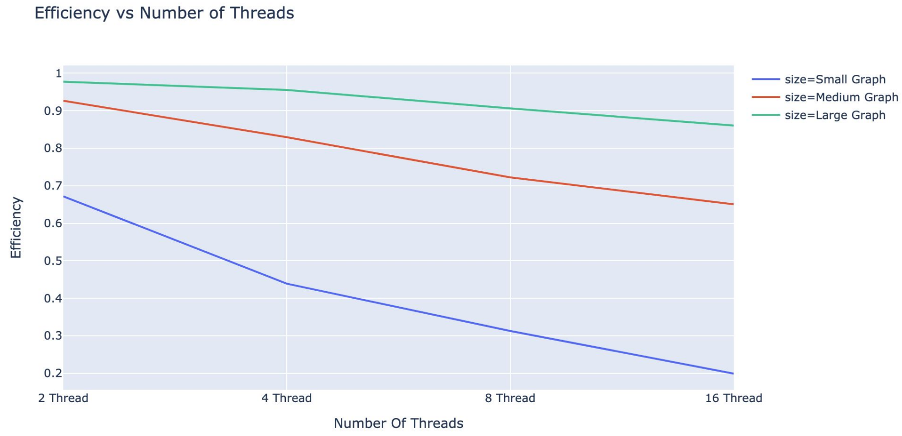
Runtime Results

Graphs are taken from Suite-Sparse Matrix Collection website [2].

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***

Graphs	Num Nodes	Sequential	GPU-Parallel
494_bus.mtx	494	0.0039	0.39
shuttle_eddy.mtx	10429	1.713	55.1124





Speedup & Efficiency

- In CPU:
  - Almost linear speedup
  - Efficiency close to 1
  - Better performance in large sized graphs
- In GPU:
  - Up to ~100 times worse runtime
  - Comparisons only available for centrality measurements except betweenness centrality

Comparing Our Speedup with Another Implementation

Graph	CPU Time (s)		GPU Time (s)		Speed up on Fermi	
	$nt = 1$	$nt = 16$	Tesla	Fermi	$nt = 1$	$nt = 16$
syn1.gr	24.19	5.47	5.65	2.64	9.16	2.07
syn2.gr	80.72	15.78	13.48	6.33	12.75	2.49
syn3.gr	184.11	32.68	23.31	11.31	16.28	2.89
syn4.gr	93.63	17.35	13.77	6.11	15.32	2.84
syn5.gr	232.79	33.87	19.98	11.83	19.68	2.86

[4]

References:

[1] Geeks For Geeks ‘Applications of Graph Data Structure’ [online]. Available at: https://www.geeksforgeeks.org/applications-of-graph-data-structure/

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

[3] Saad, Y. (2003). *Iterative methods for sparse linear systems* (Vol. 82). Siam.

[4] Pande, P.R., & Bader, D.A. (2011). Computing Betweenness Centrality for Small World Networks on a GPU.

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

[6] 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