Fast Parallel Algorithm for Betweenness Centrality

Aayush Adlakha
Jash Jayesh Jhatakia

November 2024

Introduction

Betweenness centrality is a critical measure in network analysis used to identify influential nodes in a graph based on the number of shortest paths passing through each node. Real- world applications of this measure include the IMDb dataset, where actors are nodes, and their connections are based on shared movie appearances and social media networks where identifying key influencers can help understand information flow.

Our task is to compute the Betweenness Centrality of all the nodes in an unweighted and undirected graph.

How to compute Betweenness Centrality?

Mathematically, for any node v, its Betweenness Centrality is defined as the summation of the fraction of shortest paths between any 2 nodes that contain the given node v over the total number of shortest paths between that pair.

$$\mathrm{BC}(v) = \sum_{\substack{s,t \in V \\ s \neq v \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Where:

- V: The set of all nodes in the graph.
- σ_{st}: The total number of shortest paths between nodes s and t.
- $\sigma_{st}(v)$: The number of shortest paths between nodes s and t that pass through the node v.
- The summation runs over all pairs of nodes s and t such that $s \neq v \neq t$, i.e., the pair does not involve the node v as either s or t.

The Betweenness Centrality quantifies how often node v acts as a bridge along the shortest paths between other nodes in the network.

The Naive Approach

We can use the fact that A vertex v ϵ V lies on a shortest path between vertices s, t ϵ V , if and only if $d_G(s,t) = d_G(s,v) + d_G(v,t)$

The quantity $\sigma_{st}(v)$ is defined as follows:

$$\sigma_{st}(v) = \begin{cases} 0 & \text{if } d_G(s,t) < d_G(s,v) + d_G(v,t) \\ \sigma_{sv} \cdot \sigma_{vt} & \text{otherwise} \end{cases}$$

So, the Naive Approach is to first compute the length and number of shortest paths between all pairs in $O(N \times M)$ time and accumulate the results by summing up all the pair-wise dependencies.

A pseudo code has been provided for the Naive Algorithm below.

Algorithm 1 Naive Betweenness Centrality

Input: Graph G with n vertices
Output: Betweenness Centrality vector BC
Initialize $BC \leftarrow \text{zeros}(n)$
Initialize $dist$ and $shortest\_paths$ as $n \times n$ matrices
{— Compute shortest paths using BFS for each source vertex —}
for each vertex src **do**
  Initialize queue Q
  Enqueue src into Q
  Set $dist[\text{src}][\text{src}] = 0$
  Set $shortest\_paths[\text{src}][\text{src}] = 1$
  while queue Q is not empty **do**
    Dequeue u from Q
    for each neighbor v of u in G **do**
      {— This node v is being seen for the first time —}
      if $dist[\text{src}][v] = \infty$ **then**
        $dist[\text{src}][v] = dist[\text{src}][u] + 1$
        Enqueue v into Q
      end if
      {— v is on the shortest path from src to v —}
      if $dist[\text{src}][v] = dist[\text{src}][u] + 1$ **then**
        $shortest\_paths[\text{src}][v] += shortest\_paths[\text{src}][u]$
      end if
    end for
  end while
end for
{— Calculate Betweenness Centrality for each vertex —}
for each vertex v **do**
  for each pair (s, t) where $s \neq v \neq t$ **do**
    if $dist[s][v] + dist[v][t] = dist[s][t]$ **then**
      $BC[v] \mathrel{+}= \frac{shortest\_paths[s][v] \times shortest\_paths[v][t]}{shortest\_paths[s][t]}$
    end if
  end for
end for
Return BC

Sequential Brandes' Algorithm

In this section, we present the Sequential Brandes' Algorithm for calculating Betweenness Centrality in an optimized manner compared to the Naive approach. The algorithm works by processing each vertex v and using a more efficient approach for accumulating dependencies. First, let us define some variables relevant to the Brandes Algorithm.

- $\delta_{st}(v)$: This represents the dependency of vertex v on a specific pair of vertices (s, t). It accumulates the fraction of shortest paths from s to t that pass through v.
$$\delta_{st}(v) = \frac{\sigma_{sv} \cdot \sigma_{vt}}{\sigma_{st}},$$
- $\delta_s(v)$: This is the dependency of vertex v on the source vertex s. It tracks the fraction of shortest paths from s that pass through v.
$$\delta_s(v) = \sum_{t \neq s} \delta_{st}(v)$$

Recall that σ_{st} is the number of shortest paths from the node s to t.

Brandes' derived a Mathematical relation between the δ_s of a node v in terms of it successors.

$$\delta_s(v) = \sum_{w : w \in S_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_s(w))$$

Where $S_S(v)$ is the set of successors of the node v. Node w is a successor of v, if there is an edge between v and w and $d_G(s, w) = d_G(s, v) + 1$

A pseudo code has been provided for the Brandes' Algorithm below.

Algorithm 2 Betweenness Centrality in Unweighted Graphs

Input: Graph $G = (V, E)$
Output: Betweenness Centrality vector BC
Initialize $BC[v] \leftarrow 0$, for all $v \in V$
for each source vertex $s \in V$ **do**
  Initialize $S \leftarrow \emptyset$ (stack for BFS order)
  Initialize $Succ[w] \leftarrow \emptyset$, for all $w \in V$ (successor lists)
  Initialize $\sigma[t] \leftarrow 0$, for all $t \in V$; $\sigma[s] \leftarrow 1$ (number of shortest paths)
  Initialize $d[t] \leftarrow -1$, for all $t \in V$; $d[s] \leftarrow 0$ (distance from s)
  Initialize $Q \leftarrow \emptyset$ (queue for BFS)
  Enqueue s into Q
  while queue Q is not empty **do**
    Dequeue v from Q
    Push v onto S (add to stack)
    for each neighbor w of v **do**
      if $d[w] < 0$ **then**
        {w found for the first time}
        Enqueue w into Q
        Set $d[w] = d[v] + 1$
      end if
      if $d[w] = d[v] + 1$ **then**
        {Shortest path to w via v}
        $\sigma[w]$ += $\sigma[v]$
        Append w to $Succ[v]$ (record w as a successor of w)
      end if
    end for
  end while
  Initialize $\delta[v] \leftarrow 0$, for all $v \in V$
  while stack S is not empty **do**
    Pop w from S
    for each successor $v \in Succ[w]$ **do**
      $\delta[w]$ += $\frac{\sigma[w]}{\sigma[v]} \times (1 + \delta[v])$
    end for
    if $w \neq s$ **then**
      $BC[w]$ += $\delta[w]$
    end if
  end while
end for
Return BC

Parallelization Strategies

Now that the problem and its sequential algorithm are clear, let us discuss the various parallelization strategies we have to make this algorithm more efficient.

- **Coarse-Grained**: We must construct the BFS trees assuming each vertex as the source. The computation starting from each source vertex can be considered as a task. We can parallelize this outer loop as it is completely independent. The only atomic value we would require in this strategy is $BC[w]$. This was initially a problem as atomics were not supported with double up to C++-17. However, they are supported in C++-20.

- **Medium-Grained**: BFS is a level-by-level process in which nodes are processed at a level k completely before level k+1. It explores all neighbours of each vertex in the current level, and the neighbouring vertices which are visited the first time are selected as the ones in the next level. Again, one exploration of a vertex can be considered as one task, thus, all tasks in one level could proceed totally in parallel if there was no shared neighbor between any two vertices. To account for the case of shared neighbours, we need to make certain accesses atomic, which would be discussed in the pseudo-code given below.

- **Fine-Grained**: The task of exploring the neighbours of a vertex itself can also be parallelized. In this approach every node explored by a vertex for BFS is considered as a separate task. In medium-grained, exploring all the neighbours of a node on level k was considered a task. Here, each neighbour exploration is considered a separate task. Our intuition suggests that this strategy considers very little work as a task and, hence, could be slower than medium-grained if the tasks are not chunked/grouped efficiently.

A high-level overview of Medium-grained parallelization is given below.

Algorithm 3 ProcessOne Function

1: **Input:** Graph G, stack S, atomic arrays S_size, d, σ, successor lists $Succ$, atomic arrays $Succ\_size$, phase phase, index ind
2: Let $v = S$[phase][ind]
3: **for** each neighbor $w \in G$.adj[v] **do**
4: $dw = -1$
5: **if** compare_and_swap(&d[w], -1, d[v] + 1) **then**
6: $p =$ fetch_and_add(S_size[phase + 1], 1)
7: Insert w at position p of S[phase + 1]
8: **end if**
9: **if** $d[w] = d[v] + 1$ **then**
10: fetch_and_add($\sigma[w]$, $\sigma[v]$)
11: $p =$ fetch_and_add($Succ\_size[v]$, 1)
12: Insert w at position p of Succ[v]
13: **end if**
14: **end for**

Algorithm 4 AccumulateOne Function

1: **Input:** vertex w, source s, atomic arrays σ, Succ_size, Succ, array δ, array BC
2: Let $dsw = 0.0$
3: Let $sw = \sigma[w]$
4: **for** each successor $v \in$ Succ[w] **do**
5: $dsw += \left(\frac{sw}{\sigma[v]}\right) \times (1.0 + \delta[v])$
6: **end for**
7: $\delta[w] = dsw$
8: **if** $w \neq s$ **then**
9: $BC[w] += dsw$
10: **end if**

Algorithm 5 Medium Grained Parallelization

1: **Input:** Graph G with n vertices
2: **Output:** Betweenness Centrality vector BC
3: Initialize BC as a zero vector of size n
4: Initialize atomic variables: σ, d, $Succ\_size$, S_size for each vertex
5: Initialize vectors: S (stack), $Succ$ (successors list), $delta$
6: **for** each source vertex s in V **do**
7: Initialize $\sigma[s] = 1$, $d[s] = 0$, $S[0] = [s]$, and $S\_size[0] = 1$
8: Initialize $\sigma[v] = 0$, $d[v] = -1$, $Succ\_size[v] = 0$, $delta[v] = 0$, $S\_size[v] = 0$ for all $v \neq s$
9: Set phase = 0
10: **while** S_size[phase] > 0 **do**
11: Increment S_size[phase + 1] = 0
12: Set current_phase_size = S_size[phase]
13: Create empty list of threads
14: start = 0, end = chunkSize
15: **for** each task in parallel **do**
16: create_thread(ProcessMany(start, end))
17: start = end, end += chunkSize
18: **end for**
19: Join all threads
20: Increment phase
21: **end while**
22: **while** phase > 0 **do**
23: Decrement phase
24: Set current_phase_size = S_size[phase]
25: Create empty list of threads
26: start = 0, end = chunkSize
27: **for** each task in parallel **do**
28: create_thread(AccumulateMany(start, end))
29: start = end, end += chunkSize
30: **end for**
31: Join all threads
32: **end while**
33: **end for**
34: **Return** BC

The pseudo-code has become a little messy and complicated. Some key

points explaining it are mentioned below:

- The Stack and Successor lists are shared vectors, to keep the insertions atomic, we only insert at a pre-allocated location determined by fetch_and_add call to the corresponding size vectors.
- The corresponding size vector to the Stack and Successor list has to be atomic.
- Phase is simply the level that is currently being processed in the BFS tree. The pseudo-code does not consistently showcase the fact that the Stack and Successor lits have been divided into phase-by-phase components to make the implementation simpler.
- The ProcessMany and AccumulateMany are helper functions that call ProcessOne and AccumulateOne in a desired range from start to end (the end is excluded).
- In the current implementation, threads are created at each level and destroyed. This means we are creating new K threads at each level, this has been done because these are level-by-level synchronization techniques. (We will later fix this and only create K threads)
- The chunkSize calculation has been omitted from the pseudo-code. It is simply the ceil of total_tasks divided by total_threads.

Why can't we create and destroy?

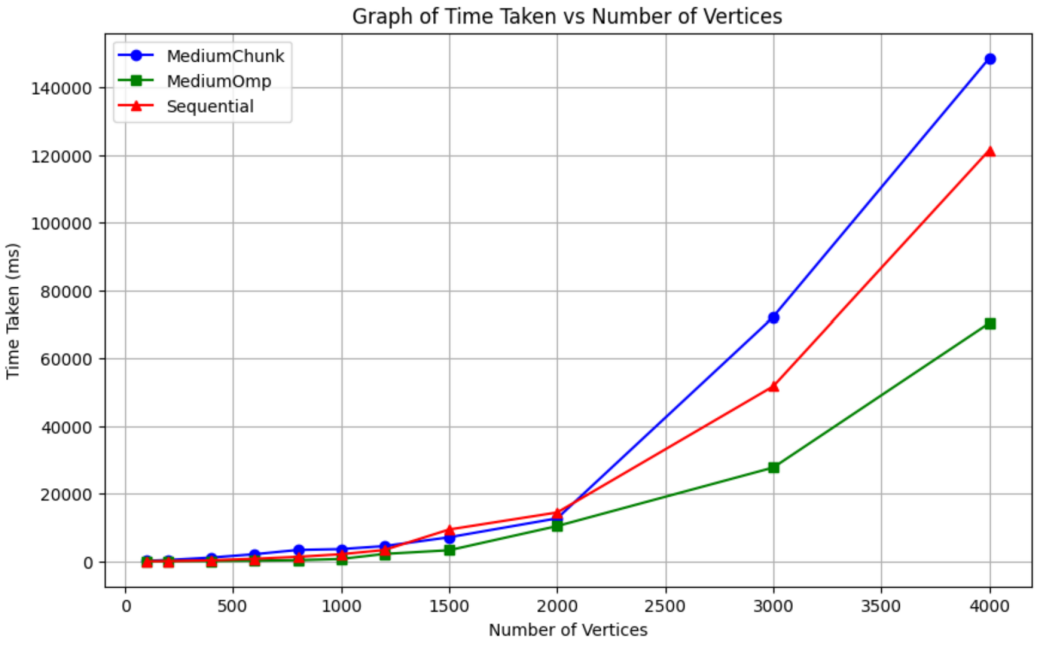


Figure 1: MediumChunk vs OpenMP vs Sequential

We see that our strategy is almost always worse than Sequential, but OpenMP is much faster than Sequential. This showcases that the overhead of creating and destroying threads on each level is very large and must be avoided.

Thread Pool

To improve the performance of our parallel algorithm, we need to ensure threads are not created and destroyed at each level. We decided to implement a thread pool for this such that:

- We maintain a task queue and all threads continuously loop to check for available tasks in the task queue.
- If no task is available for a thread, it waits on a conditional variable.

- When a new task is added, all the sleeping threads are signalled and woken up.
- The queue for the thread is not lock-free (as that was not available in the standard library).

A very rough sketch of a pseudo-code has been given for the thread pool implemented.

Algorithm 6 ThreadPool Pseudocode

```
1:  Class ThreadPool:
2:      Initialize:
3:          Create an empty task queue
4:          Initialize the number of tasks left to 0
5:          Initialize stop flag to false
6:          For each thread:
7:              Continuously wait for tasks
8:              If stop flag is true and no tasks are left, exit
9:              Otherwise, fetch the next task and execute it
10:             Decrease the number of tasks left and notify when tasks are complete
11:     Method enqueue(task):
12:         Lock the task queue
13:         Add task to the queue
14:         Unlock the task queue
15:         Increment tasks left count
16:         Notify one worker to pick up the task
17:     Method wait_for_tasks():
18:         Lock the tasks left counter
19:         Wait until tasks left is zero
```

Results for the Thread Pool

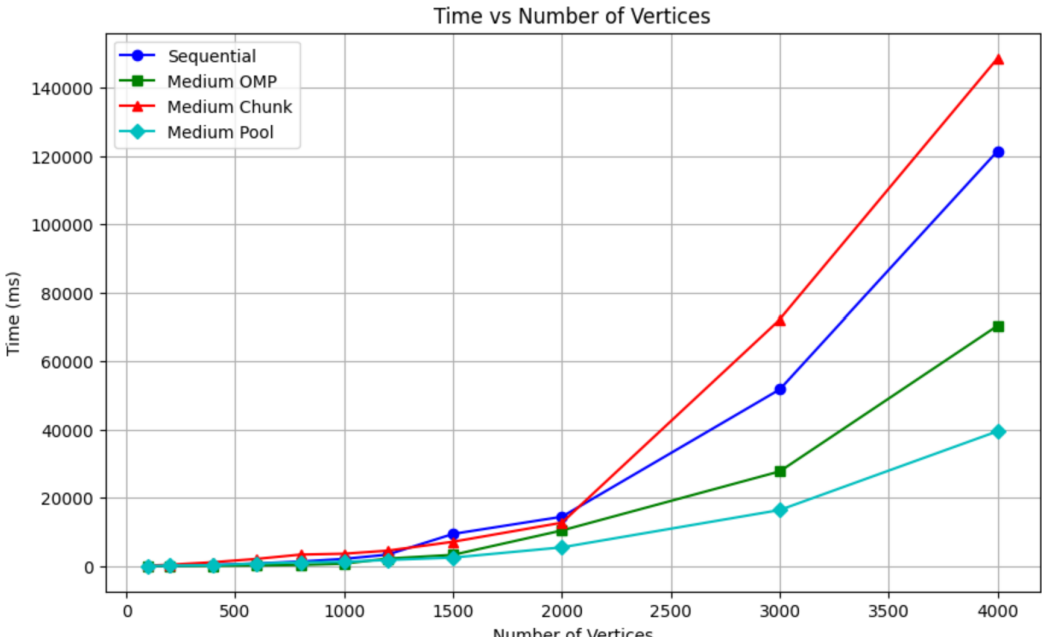
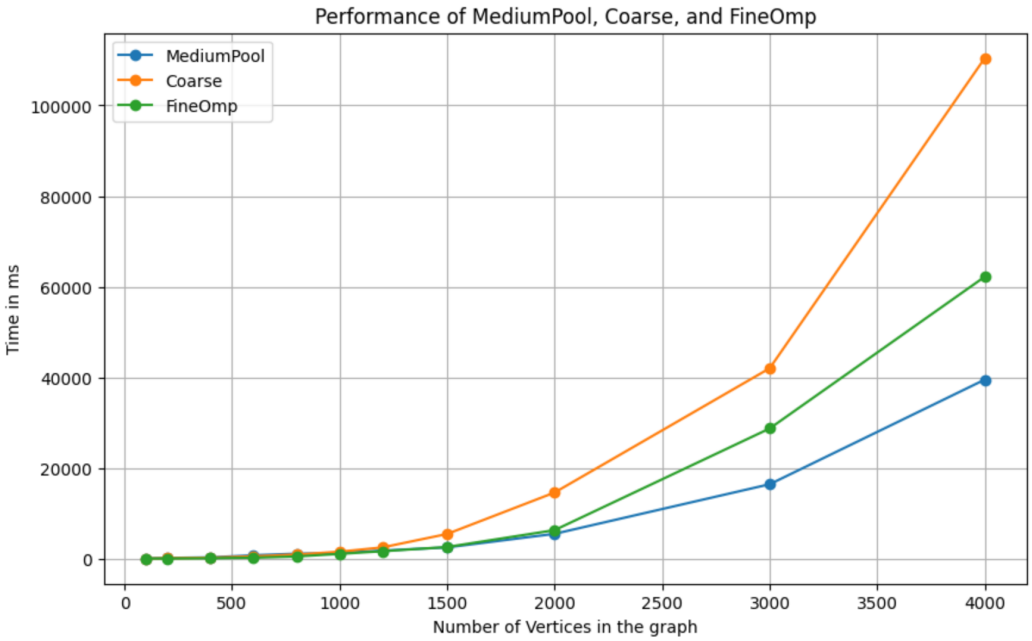


Figure 2: MediumChunk vs OpenMP vs MediumPool vs Seq

- The overhead of ThreadPool was visible in the initial few runs; however, that is not visible in the graph.
- Our Thread Pool is very close to OpenMP in terms of speed up to 2000 vertices but is much faster after that.
- To compare with the sequential algorithm, we are slightly over 3 times faster.
- All the data has been taken for 16 threads.

Results for other Approaches



- The coarse-grained parallelization strategy is the slowest.
- Not only is the coarse-grained strategy slowest, it gets slower with increasing input size.
- The medium-grained approach is faster than fine-grained, but they are somewhat comparable.

Results with varying Threads

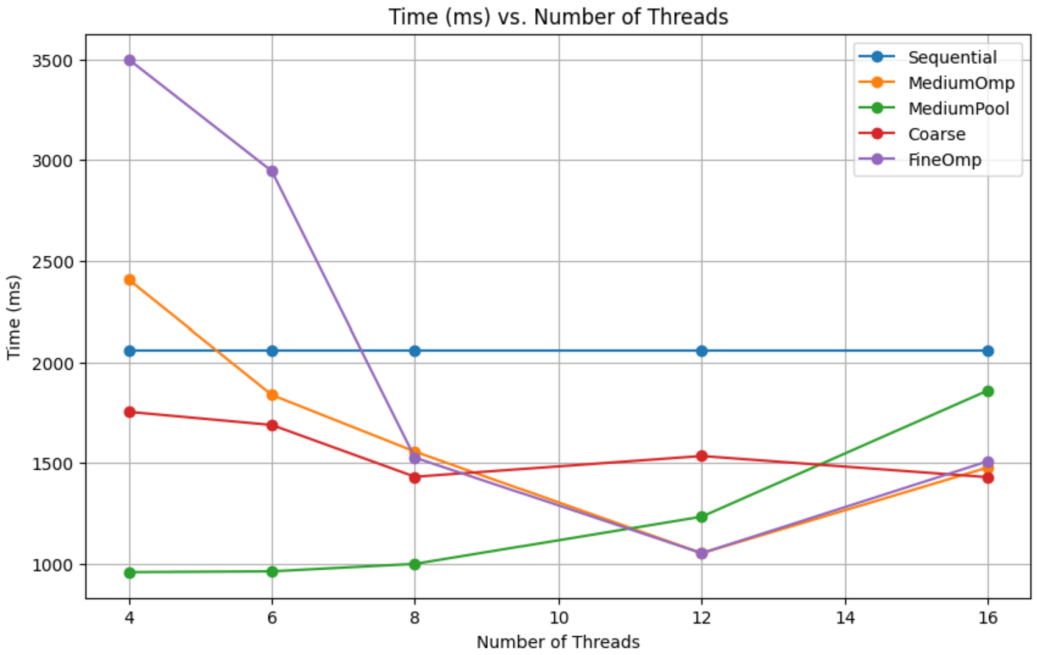


Figure 3: Results with varying threads, 1000 nodes

- Both the OpenMP implementations give very similar results after 8 threads.
- These results use a graph with 1000 vertices.
- The MediumPool performs better with a lower number of threads; this could be because the thread pool implemented is not lock-free.
- Coarse-grained is better than fine-grained for a small number of threads.
- Nearly all approaches are better than sequential starting from 8 threads.

Scope for further work

- We should make the queue used in the thread pool to be lock-free; this could improve performance for more threads.

- We could also integrate a task scheduling algorithm like CILK because threads often need to wait because one thread is slow.
- The results reported are only for fine-grained open-mp because the Thread pool could not perform well when many tasks were inserted, which is the case for fine-grained strategy. Again, a lock-free queue could help here.
- We later realized that our use case of ThreadPool is very similar to barriers, and we could try using that as it is available in C++-20.
- The results displayed are from AMD Ryzen 9 5900HX with Radeon Graphics machine with 12 logical cores. We did not run this on the virtual machine provided because of high fluctuations and contention issues.

References

1. A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality
2. A Faster Algorithm for Betweenness Centrality
3. A Parallel Algorithm for Computing Betweenness Centrality