DC PROJECT

PARALLEL BFS

BETWEENESS CENTRALITY ANALYSIS

DISHIT SHARMA (B22CS082)

GITHUB LINK

CONTENT

BETWEENESS CENTRALITY

Reference

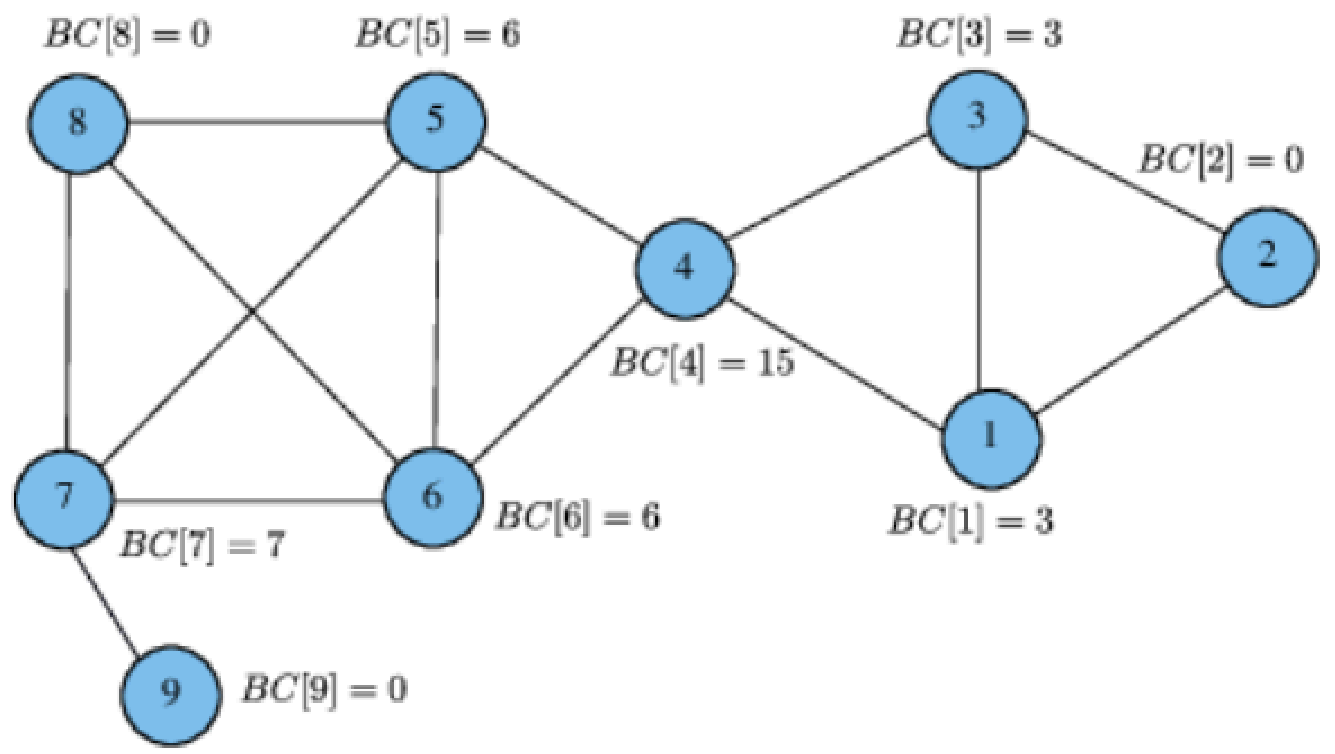


Fig. 1. Example Betweenness Centrality scores for a small graph

Betweenness Centrality determines the importance of vertices in a network by measuring the ratio of shortest paths 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. 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. Consider Figure 1 above. Vertex 4 is the only vertex that lies on paths from its left (vertices 5 through 9) to its right (vertices 1 through 3). Hence vertex 4 lies on all the shortest paths between these pairs of vertices and has a high BC score. In contrast, vertex 9 does not belong on a path between any pair of the remaining vertices and thus it has a BC score of 0.

CSR FORMAT

Widely used method to store graphs efficiently, especially for GPU computations. It leverages two main arrays to represent the graph's structure:

1. Adjacency List Array (Column Indices):
    - This array is a concatenation of the adjacency lists of all vertices in the graph.
    - It has a total length of E elements, where E is the number of edges in the graph.
    - Each entry in this array represents the destination vertex of a corresponding edge.
2. Adjacency List Pointers Array (Row Offset):
    - This array contains V+1 elements, where V is the number of vertices in the graph.
    - It indicates the starting and ending positions of each vertex's adjacency list within the adjacency list array.
    - For a given vertex s, its adjacency list starts at adjPointers[s] and ends at adjPointers[s+1] - 1.

CSR FORMAT

Example

Consider a graph with the following adjacency lists:

- Vertex 0: [1, 2]
- Vertex 1: [0, 2, 3]
- Vertex 2: [0, 3]
- Vertex 3: [1, 2]

In CSR format, the arrays would be:

- Adjacency List Array (Column Indices): [1, 2, 0, 2, 3, 0, 3, 1, 2]
- Adjacency List Pointers Array (Row Offset): [0, 2, 5, 7, 9]

Here, the adjacency list of vertex 0 starts at index 0 and ends at index 1 (adjPointers[0] to adjPointers[1] - 1), vertex 1 starts at index 2 and ends at index 4, and so on.

Advantages

1. Memory Efficiency: The CSR format reduces the memory footprint by storing only non-zero elements (edges) and their respective column indices, avoiding the need to store zero elements.
2. Parallel Processing: It is particularly suited for parallel processing on GPUs, allowing efficient traversal and manipulation of large graphs due to its compact structure.

SERIAL IMPLEMENTATION

- The code calculates the betweenness centrality for each node in a graph using a serial algorithm. Betweenness centrality measures how often a node appears on the shortest paths between other nodes.
- The function betweennessCentrality (Graph *graph) initializes arrays to store the centrality values, predecessors, dependencies, shortest path counts (sigma), and distances.
- For each node, it performs a Breadth-First Search (BFS) to find all shortest paths from that node, updating the distance and sigma arrays as it explores the graph. A queue is used for BFS, and a stack keeps track of the nodes in the order they are processed.

- After BFS, the code calculates dependencies for each node using the stack, ensuring nodes are processed in the reverse order of their distance from the source.
- Dependencies help determine how central a node is by tracking how many shortest paths pass through it.
- The centrality values are adjusted accordingly, and each value is halved because each path is counted twice.
- The main function reads the graph, runs the centrality calculation, measures how long it takes, and can print the results.

PARALLEL VERTEX BASED IMPLEMENTATION

Reference:
https://developer.nvidia.com/blog/accelerating-graph-betweenness-centrality-cuda/

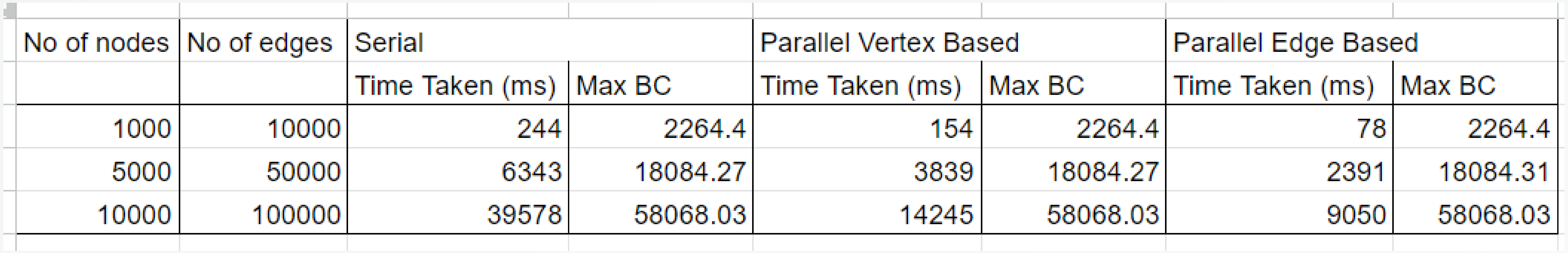
To see the example, go to the reference given above.
The d array of length stores the BFS distance of each node from the root and the sigma array, also of length , represents . We call the distribution of threads to work in the above code the vertex-parallel method, because each thread is assigned to its own vertex. In this toy example, a thread is assigned to each of the 9 vertices in the graph. Vertices are stored in CSR format. As mentioned before, only vertices 1, 3, 5, and 6 are currently in the vertex-frontier, or equivalently, have a distance equivalent to current_depth. The remaining 5 threads are forced to wait as the threads assigned to vertices in the frontier traverse their edge-lists, causing a workload imbalance. An additional workload imbalance can be seen among the threads that are inspecting edges. For example, vertex 5 has four outgoing edges whereas vertex 1 only has three.

PARALLEL EDGE BASED IMPLEMENTATION

Reference:
https://developer.nvidia.com/blog/accelerating-graph-betweenness-centrality-cuda/

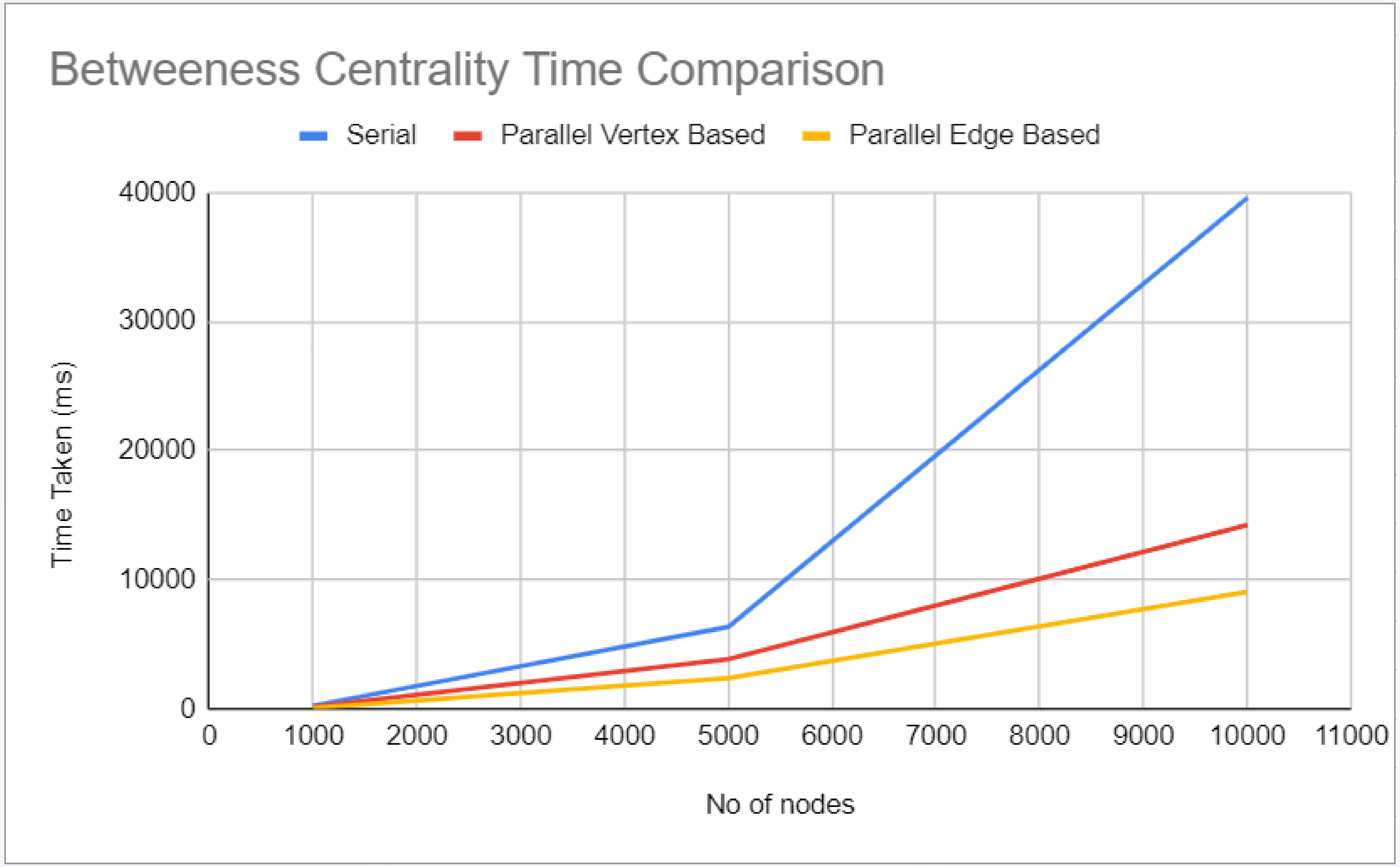
Each thread is assigned to an edge. Treat each undirected edge as two directed edges such that for an edge one thread is assigned to and another thread is assigned to . Since each thread processes one edge rather than an arbitrarily long set of edges, the work among threads is properly balanced. However, both the vertex-parallel and edge-parallel methods assign threads to vertices or edges that don’t belong to the current frontier, leading to unnecessary accesses to memory and severe branch divergence.

OUTPUT OF THE THREE IMPLEMENTATIONS



No of nodes	No of edges	Serial		Parallel Vertex Based		Parallel Edge Based	
		Time Taken (ms)	Max BC	Time Taken (ms)	Max BC	Time Taken (ms)	Max BC
1000	10000	244	2264.4	154	2264.4	78	2264.4
5000	50000	6343	18084.27	3839	18084.27	2391	18084.31
10000	100000	39578	58068.03	14245	58068.03	9050	58068.03

VISUALIZATION USING GRAPHS



THANK YOU