DC PROJECT

PARALLEL BFS

KATZ CENTRALITY ANALYSIS

DISHIT SHARMA (B22CS082)

RESEARCH PAPER USED AS REFERENCE

CONTENT

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.

KATZ CENTRALITY

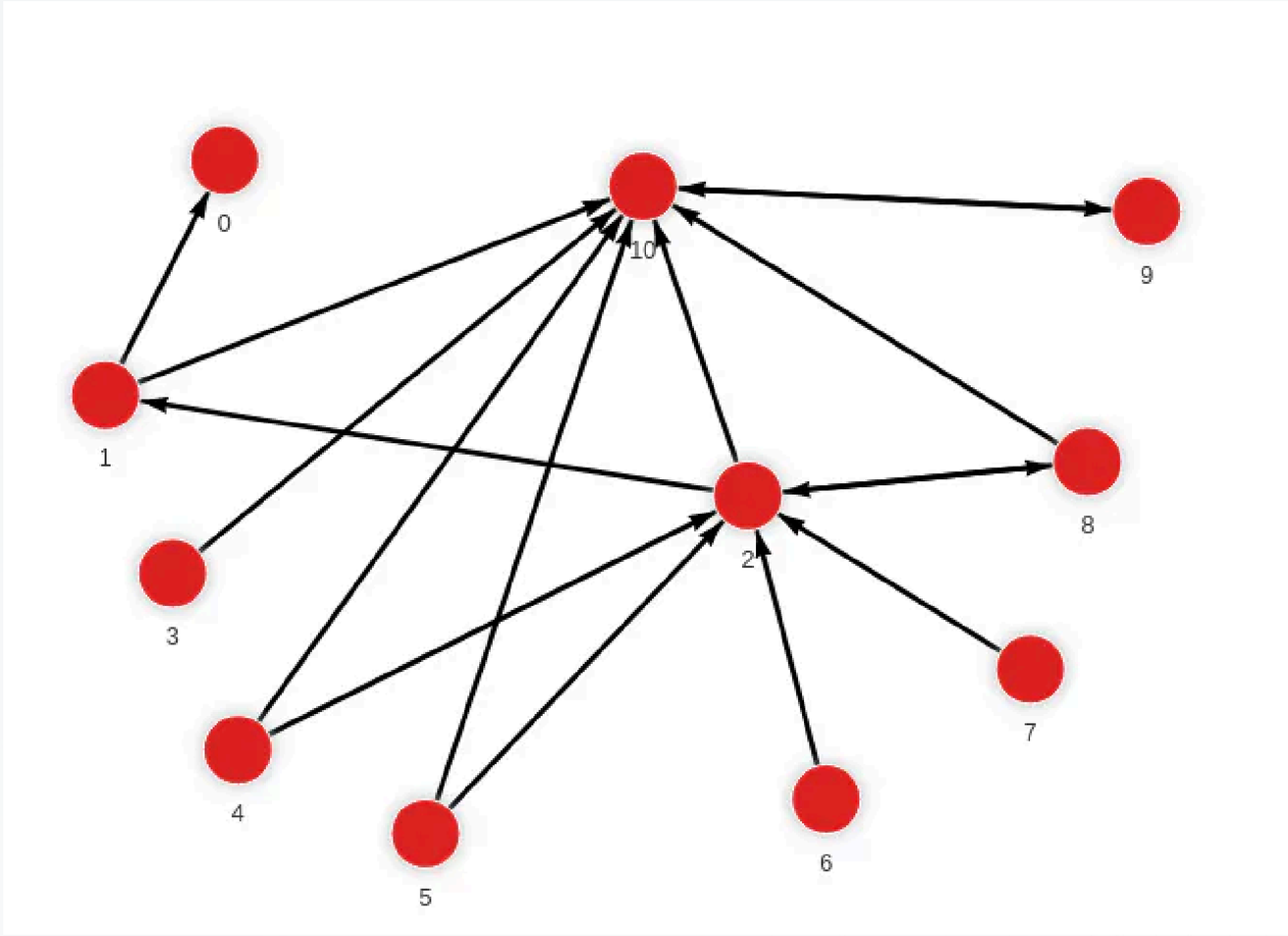
- Computes the relative influence of a node within a network by measuring the number of the immediate neighbors (first degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors.
- Connections made with distant neighbors are, however, penalized by an attenuation factor “alpha”.
- Each path or connection between a pair of nodes is assigned a weight determined by “alpha” and the distance “d” between nodes as “alpha ^ d”.

Theoretically speaking there exists an attenuation factor `(alpha^i)` smaller than 1 which is applied to walks of length `i`. If `w_i(v)` is the number of walks of length `i` starting from node `v`, Katz centrality is defined as:

```
Centrality(v) = sum { w_i(v) * alpha ^ i}
```

KATZ CENTRALITY EXAMPLE

node	rank
(:Node {id: 9})	0.544
(:Node {id: 7})	0
(:Node {id: 6})	0
(:Node {id: 5})	0
(:Node {id: 4})	0
(:Node {id: 3})	0
(:Node {id: 8})	0.408
(:Node {id: 2})	1.08
(:Node {id: 10})	1.864
(:Node {id: 0})	0.28
(:Node {id: 1})	0.408



SERIAL IMPLEMENTATION

Reference

Here, I did BFS from each vertex and maintained two queues which represented the current and the next frontiers. This way, I was able to able to attain the katz centrality for the source vertex by calculating the distance of each node from the source node and raising the attenuation factor to that distance and summing that for all the nodes.

For example, if the attenuation factor is 0.2 (assumed by default), then I added the katz centrality for the source vertex for the all the nodes in the current frontier and then update the attenuation factor to be 0.04 for the next frontier and so on.

CUDA IMPLEMENTATION

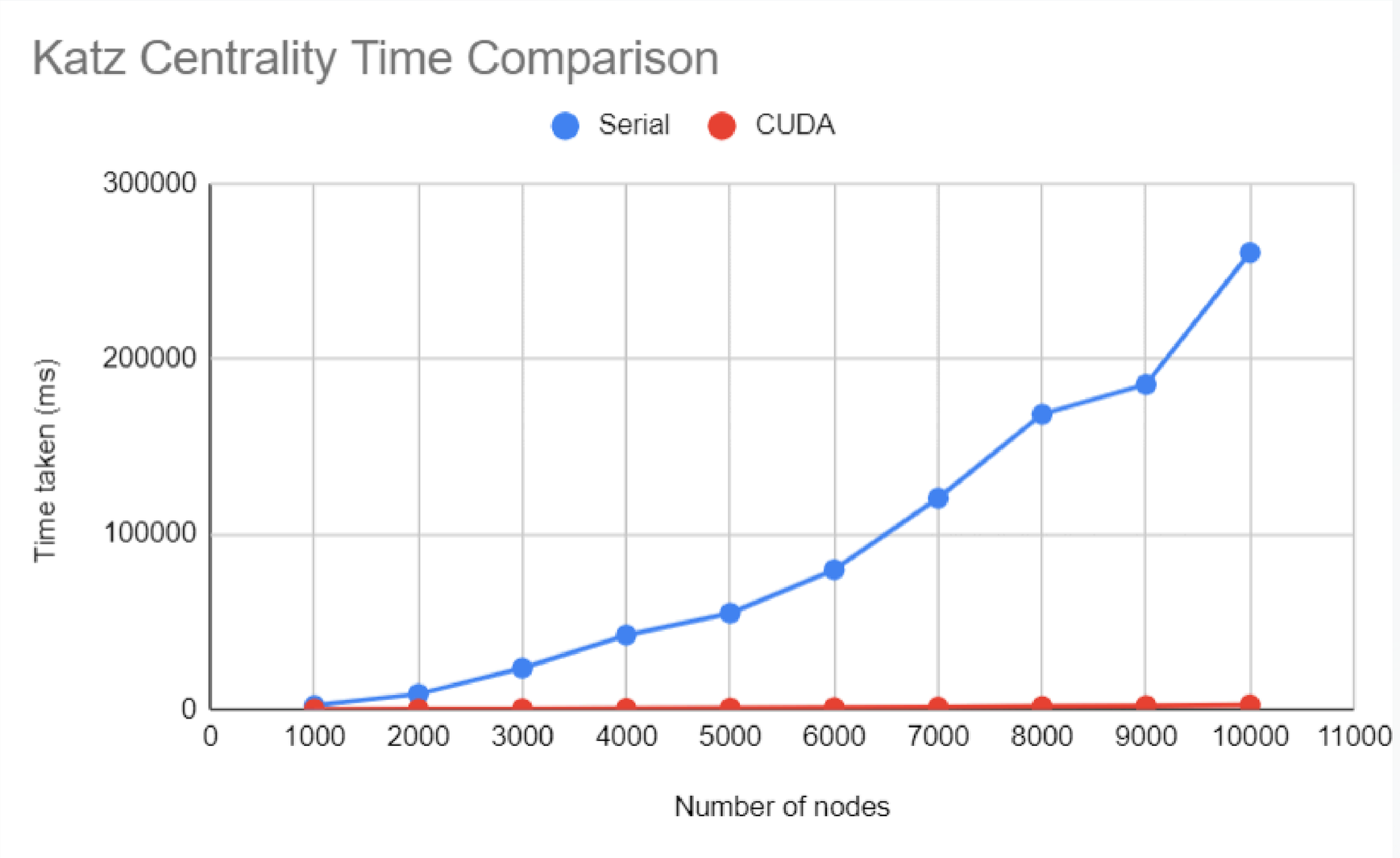
Reference

In this approach, we basically parallelize each BFS individually by assigning a thread for each vertex. Thus, the individual BFS is executed a lot faster.
For the rest part, the approach is similar to the serial implementation in the sense that we still maintain two queues representing the current and the next frontiers.
Again, we use the attenuation factor (0.2 by default) and then we add the katz centrality for the source vertex for the all the nodes in the current frontier and then update the attenuation factor to be 0.04 for the next frontier and so on.

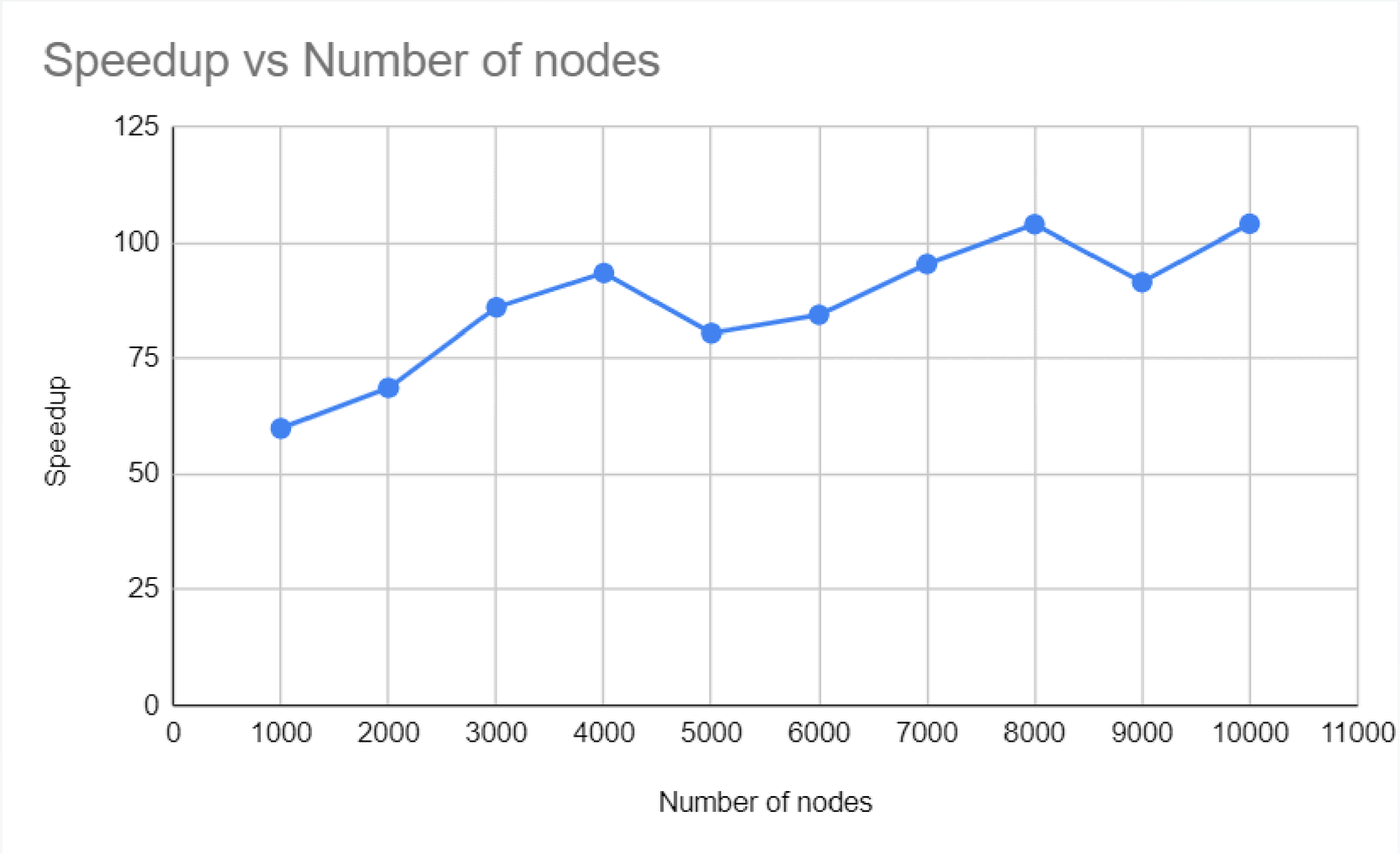
OUTPUT OF THE TWO IMPLEMENTATIONS

		Serial Implementation		Cuda Implementation
Number of nodes	Number of edges	Time (ms)	Time (mm:ss)	Time (ms)
1000	10000	2098	00:02	35
2000	20000	8658	00:08	126
3000	30000	23420	00:23	272
4000	40000	42298	00:42	452
5000	50000	54768	00:54	680
6000	60000	79590	01:20	942
7000	70000	120375	02:00	1260
8000	80000	168259	02:48	1617
9000	90000	185238	03:05	2023
10000	100000	260522	04:21	2501

VISUALIZATION USING GRAPHS



VISUALIZATION USING GRAPHS



FURTHER PARALLELISATION POTENTIAL

- Since a thread is assigned to each of the vertices in the graph. Vertices are stored in CSR format. As mentioned before, only vertices in current frontier in the vertex-frontier, or equivalently, have a distance equivalent to current_depth. The remaining 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, a vertex may have four outgoing edges whereas another vertex may only have three.

THANK YOU