DC PROJECT

PARALLEL BFS

BETWEENESS CENTRALITY ANALYSIS

DISHIT SHARMA (B22CS082)

RESEARCH PAPER USED AS REFERENCE

CONTENT

BETWEENESS CENTRALITY

Reference 1

Reference 2

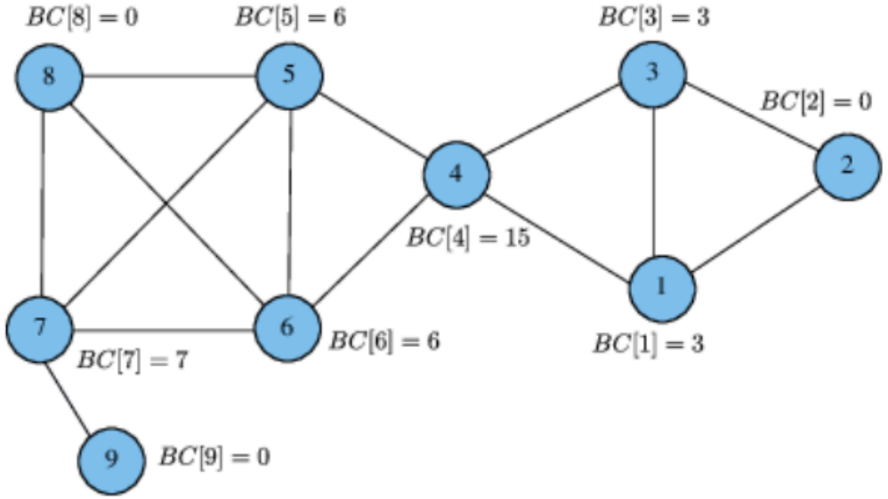


Fig. 1. Example Betweenness Centrality scores for a small graph

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.

PARALLEL WORK EFFICIENT

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

- To ensure that only edges in the current frontier are traversed for each iteration, we need to use explicit queues to store vertices. Since each vertex can only be placed in the queue one time we are sure to perform an asymptotically optimal BFS for each source vertex.
- We use two “queues,” Q and Q2, stored as arrays in global memory on the device. Q contains vertices in the current vertex frontier and Q2 contains vertices to be processed in the following vertex frontier. At the end of each search iteration we transfer the vertices from Q2 to Q and do the appropriate bookkeeping to keep track of Q_len and Q2_len, the shared memory variables that represent their lengths. The graph is stored in CSR format.
- We use an atomicCAS() operation to prevent vertices from being placed in the queue more than once. In practice this restriction can be lifted without affecting program correctness for a potential increase in performance, but that technique is outside the scope of this post.
- A workload imbalance still may exist among threads; however, the unnecessary branch divergence and memory fetches seen by the previous two methods have been eliminated.

PARALLEL WORK EFFICIENT (COARSE GRAIN)

- introduces the concept of BLOCK_COUNT, allowing the kernel to be launched with multiple blocks. This enhances the scalability of the computation, enabling it to handle larger graphs more efficiently by distributing the workload across multiple blocks.
- Optimizes memory usage by calculating BLOCK_COUNT based on available device memory, ensuring the efficient utilization of up to 4 GB (can be changed) of device memory. This prevents memory over-allocation and ensures that the kernel can handle larger data sets without running out of memory.
- Instead of processing one source node at a time, the second code processes multiple source nodes in parallel using different blocks. This leads to a significant reduction in the overall computation time by leveraging more of the GPU's parallel processing capabilities.
- **__syncthreads()** and shared memory usage within each block ensure that dependencies are accurately computed and shared data is consistently managed across threads within a block.
- Uses atomic operations to update the betweenness centrality scores (**atomicAdd**), ensuring that concurrent updates from multiple threads do not result in race conditions, thereby maintaining the correctness of the betweenness centrality calculations.

CODE OF PARALLEL WORK EFFICIENT (COARSE) APPROACH

```
while(currentSource < numNodes - totalBlocks) {
    if(!threadId) {
        currentSource += totalBlocks;

        queue[0 + (blockIdx.x * numNodes)] = currentSource;
        queueLength = 1;
        queueIndices[0 + (blockIdx.x * numNodes)] = 0;
        queueIndices[1 + (blockIdx.x * numNodes)] = 1;
        queueIndicesCount = 1;
    }
    __syncthreads();

    for(int v = threadId; v < numNodes; v += blockDim.x) {
        if(v == currentSource) {
            distances[v + (blockIdx.x * numNodes)] = 0;
            shortestPaths[v + (blockIdx.x * numNodes)] = 1;
        } else {
            distances[v + (blockIdx.x * numNodes)] = INT_MAX;
            shortestPaths[v + (blockIdx.x * numNodes)] = 0;
        }
        dependencies[v + (blockIdx.x * numNodes)] = 0.0;
    }
    __syncthreads();
```

CODE OF PARALLEL WORK EFFICIENT (COARSE) APPROACH

```
// Perform BFS
while(1) {
    __syncthreads();
    for(int k = threadId;
        k < queueIndices[queueIndicesCount + (blockIdx.x * numNodes)];
        k += blockDim.x){

        if(k < queueIndices[queueIndicesCount - 1 + (blockIdx.x * numNodes)])
            continue;
        int v = queue[k + (blockIdx.x * numNodes)];
        for(int r = g->adjacencyListPointers[v]; r < g->adjacencyListPointers[v + 1]; r++) {
            int w = g->adjacencyList[r];
            if(atomicCAS(&distances[w + (blockIdx.x * numNodes)], INT_MAX,
                    distances[v + (blockIdx.x * numNodes)] + 1) == INT_MAX) {
                int t = atomicAdd(&queueLength, 1);
                queue[t + (blockIdx.x * numNodes)] = w;
            }
            if(distances[w + (blockIdx.x * numNodes)] == (distances[v + (blockIdx.x * numNodes)] + 1)) {
                atomicAdd(&shortestPaths[w + (blockIdx.x * numNodes)], shortestPaths[v + (blockIdx.x * numNodes)]);
            }
        }
    }
    __syncthreads();

    if(queueLength == queueIndices[queueIndicesCount + (blockIdx.x * numNodes)]) break;

    if(!threadId){
        queueIndicesCount++;
        queueIndices[queueIndicesCount + (blockIdx.x * numNodes)] = queueLength;
    }
    __syncthreads();
}
__syncthreads();
```

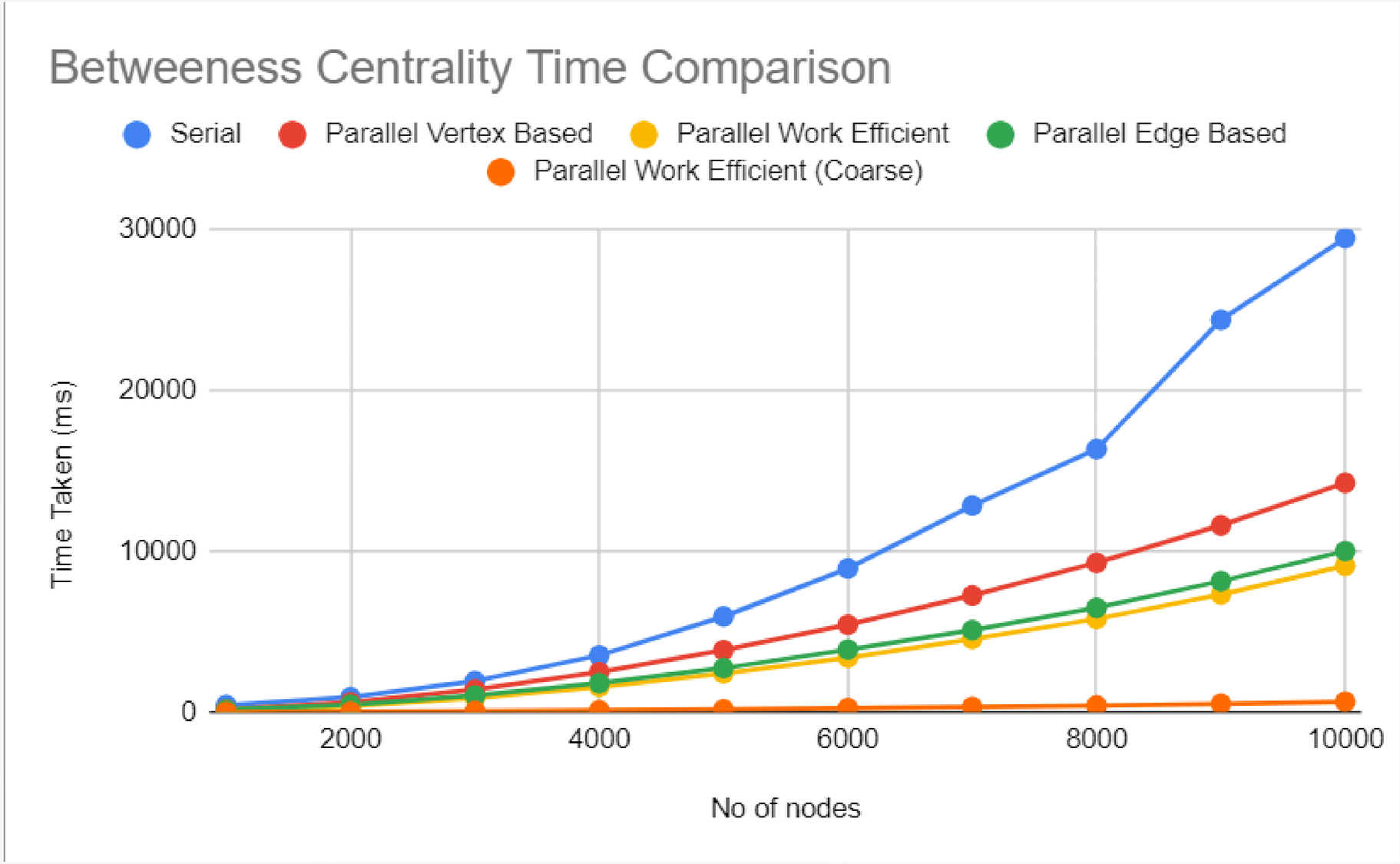
CODE OF PARALLEL WORK EFFICIENT (COARSE) APPROACH

```
// Perform reverse BFS
while(queueIndicesCount > 0) {
    for(int k = threadId; k < queueIndices[queueIndicesCount + (blockIdx.x * numNodes)]; k += blockDim.x) {
        if(k < queueIndices[queueIndicesCount - 1 + (blockIdx.x * numNodes)]) continue;

        int v = queue[k + (blockIdx.x * numNodes)];
        for(int r = g->adjacencyListPointers[v]; r < g->adjacencyListPointers[v + 1]; r++) {
            int w = g->adjacencyList[r];
            if(distances[w + (blockIdx.x * numNodes)] == (distances[v + (blockIdx.x * numNodes)] + 1)) {
                if (shortestPaths[w + (blockIdx.x * numNodes)])
                    dependencies[v + (blockIdx.x * numNodes)] +=
                        (shortestPaths[v + (blockIdx.x * numNodes)] * 1.0 /
                        shortestPaths[w + (blockIdx.x * numNodes)]) *
                        (1 + dependencies[w + (blockIdx.x * numNodes)]);
            }
        }
        if (v != currentSource)
            atomicAdd(centrality + v, dependencies[v + (blockIdx.x * numNodes)] / 2);
    }
    __syncthreads();

    if(!threadId) queueIndicesCount--;

    __syncthreads();
}
```

OUTPUT OF THE THREE IMPLEMENTATIONS

No of nodes	No of edges	Serial		Parallel Vertex Based		Parallel Edge Based		Parallel Work Efficient		Parallel Work Efficient (Coarse)	
		Time Taken (ms)	Max BC	Time Taken (ms)	Max BC	Time Taken (ms)	Max BC	Time Taken (ms)	Max BC	Time Taken (ms)	Max BC
1000	10000	430	2264.4	159	2264.4	81	2264.4	147	2280.04	10	2264.4
2000	20000	916	6061.83	596	6061.83	364	6061.84	452	6055.94	28	6061.84
3000	30000	1917	9466.17	1393	9466.17	874	9466.18	1017	9467.68	63	9466.18
4000	40000	3515	12018.03	2476	12018.03	1562	12018.01	1792	12009.51	111	12018.02
5000	50000	5927	18084.27	3833	18084.27	2400	18084.31	2726	18112.01	161	18084.31
6000	60000	8912	25725.24	5419	25725.24	3378	25725.25	3876	25793.63	242	25725.25
7000	70000	12816	28011.53	7232	28011.53	4539	28011.56	5088	28024.03	309	28011.56
8000	80000	16328	35896.41	9259	35896.41	5779	35896.5	6471	35890.47	403	35896.49
9000	90000	24348	41181.91	11581	41181.91	7298	41181.97	8120	41207.67	509	41181.97
10000	100000	29427	58068.03	14227	58068.03	9079	58068.02	9990	58117.68	637	58068.01

VISUALIZATION USING GRAPHS



HYBRIDIZATION POTENTIAL

- It turns out that there is no “one size fits all” parallelization method that provides peak performance for all graph structures. The work-efficient method tends to work best for graphs that have a large diameter whereas the edge-parallel method tends to work best for graphs that have a small diameter. The diameter of a graph is the greatest distance between any pair of its vertices.
- Traversals of high diameter graphs (such as road networks) tend to require a large number of iterations that each consist of a small amount of work. In contrast, for graphs with a small diameter (such as social networks) traversals require a small number of iterations consisting of a large amount of work.

THANK YOU