Parallelising Betweenness Centrality using Brande's Algorithm with CUDA

Akashdeep S
181IT203
Information Technology
National Institute of Technology
Karnataka, Surathkal
akashdeep.181it203@nitk.edu.in

Sujan Reddy A
181IT147
Information Technology
National Institute of Technology
Karnataka, Surathkal
sujan.181it147@nitk.edu.in

Harshvardhan R
181IT217
Information Technology
National Institute of Technology
Karnataka, Surathkal
harshvardhanr.181it217@nitk.edu.in

Abstract—**Graphs that model COVID-19 transmission, social networks, and the structure of the Internet are enormous and cannot be manually inspected. A popular metric used to analyze these networks is betweenness centrality. The fastest algorithm to calculate betweenness centrality runs in quadratic time. Hence the examination of large graphs poses an issue. To improve the performance we can parallelise the algorithm using CUDA.**

Keywords—CUDA, Parallel Computing, Graph Analysis, Betweenness Centrality.

I. Introduction

Graph analysis is a fundamental tool for diverse domains like social networks, machine learning, computational biology and many more. Betweenness Centrality is a popular analytic that determines vertex influence in a graph. It is a measure of centrality in a graph based on shortest paths. For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices such that the number of edges that the path passes through is minimized for undirected graphs. The betweenness centrality for each vertex is the number of these shortest paths that pass through the vertex.Naive implementations of Betweenness Centrality can be solved with the all-pairs shortest-paths problem using the $O(n^3)$ Floyd-Warshall algorithm [1]. Brande's algorithm runs in O(mn) time for unweighted graphs [2] where m is the number of edges , n is the number of vertices. It has applications in analysing COVID-19 transmission. It reflects the role a patient plays in creating a bridge of infectious transmission between patients who would not have had direct contact with each other. Betweenness Centrality also has applications in community detection, power grid contingency analysis, and the study of the human brain, finding the best location of stores within cities.

II. Literature survey

Betweenness Centrality (BC) attempts to distinguish the most influential 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. This determines how well a vertex connects the pairs of other vertices in the same graph. Mathematically Between Centrality of a vertex (v) can be defined as

$$BC(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (1)$$

where σ_{st} is the number of shortest paths between vertices s and t and σ_{st} (v) is the number of those shortest paths that pass through v. Brandes improved upon the naive approach of using Floyd-Warshall algorithm. In Brande's algorithm we reuse the already calculated shortest distances by using partial dependencies of shortest paths between pairs of vertices for calculating BC of a given vertex with respect to a fixed root vertex. [3]. Brandes's algorithm splits the betweenness centrality calculation into two major steps:

1. Find the number of shortest paths between each pair of vertices ie. partial dependency $\delta_s(v)$

$$\delta_s(v) = \sum_{w : v \in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)) \quad (2)$$

2. Sum the dependencies for each vertex

$$BC(v) = \sum_{s \neq v} \delta_s(v) \quad (3)$$

Jia et al. [4] compared two types of fine-grained parallelism based on the distribution of threads to the graph entities: vertex-parallel and edge-parallel. The vertex-parallel method assigns a thread to each vertex of the graph and that thread traverses all of the outgoing edges from that vertex and, the edge-parallel approach assigns a thread to each edge of the graph and that thread traverses that edge only. Since there will be more number of vertices and edges than the number of threads, the edges or vertices will be sequentially processed by the threads. Both these approaches have a time complexity of $O(n^2 + m)$.

The problem with these two algorithms is that vertices or edges that need not be assigned a thread for a given source vertex are assigned a thread which leads to wasted work. This has been addressed by Adam et al.[3] who came up with a work efficient method with fine-grained and coarse-grained parallelism. Running a parallel BFS using optimal work distribution strategies was used for achieving fine grained parallelism. Coarse grained parallelism was achieved by using multiple blocks in a grid in CUDA for parallelising the partial dependencies for the respective independent sources.

III. METHODOLOGY

A. *Graph input*

a. *Compressed Sparse Row (CSR) format*: The graph is represented using two arrays:

- Adjacency List array where each adjacency list is concatenated to form a single array.
- Adjacency List Pointers array which is an array which points to where each adjacency list starts and ends within the adjacency list array.

b. *Cooperative (COO) format*: The vertices at the two ends of the edge are represented using two lists:

- Edge List 1 has the source vertices.
- Edge List 2 has the destination vertices.

An undirected graph with 5 vertices and 5 edges is shown in Figure 1.
The CSR and COO format of the given graph is shown in Table 1.
The Betweenness Centrality of the given graph is shown in Table 2 where it can be observed that Vertex 1 has the maximum Betweenness Centrality.

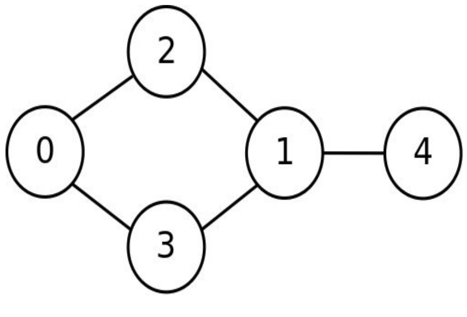


Figure 1: Example of an undirected graph

Adjacency List	2 3 3 2 4 0 1 1 0 1
Adjacency List Pointers	0 2 5 7 9 10
Edge List 1	0 0 1 1 1 2 2 3 3 4
Edge List 2	2 3 3 2 4 0 1 1 0 1

Table 1: Graph representation in CSR and COO format

Vertex	**Betweenness Centrality**
0	0.5
1	3.5
2	1
3	1
4	0

Table 2: Betweenness Centrality for each vertex

B. *Serial Algorithm*

The algorithm can be split into 3 steps.

a. *Initialisation Step*: For every source vertex we initialise the dependency, sigma, distance pointers to their respective default values.

b. Shortest Path Calculation Step: Each vertex is iterated over the algorithm as the source vertex. The source vertex is pushed into a queue and as long as the queue is not empty we dequeue a vertex, push it to the stack and update the distance, sigma and predecessor arguments for its neighbours.

c. Dependency Accumulation Step: The vertices are popped in the decreasing order of their distance from the source vertex. For each of the predecessors of the popped vertex the dependency is updated. The Betweenness centrality is found by calculating the summation of all the dependencies.

The algorithm is shown in Figure 2.

C. *Intuition of Parallelizing the Algorithm*

Since the graph traversal and shortest path accumulations of each vertex are independent, we can process all vertices in parallel. We take advantage of this coarse-grained parallelism by assigning vertices to each Streaming Multiprocessor (SM) of the GPU. Additionally, we can leverage fine-grained parallelism by having threads process the shortest path calculations and dependency accumulations cooperatively by executing breadth-first searches. Threads explore the neighbors of the source vertex and then explore the neighbors of those vertices and so on until there are no remaining vertices to explore. We call each set of inspected neighbors as a vertex frontier and we call the set of outgoing edges from a vertex frontier an edge frontier. The three steps of the algorithm are performed in the kernel.

```
$C_B[v] \leftarrow 0, v \in V$;
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
```

Figure 2: Serial Implementation of Brandes' Algorithm [2]

D. Vertex Parallel Algorithm

The vertex parallel algorithm assigns a thread to each vertex of the graph and that thread traverses all of the outgoing edges from that vertex. There are 3 shared variables: *s* for identifying the source vertex, *current_depth* for keeping track of the depth of the discovered and a flag variable *done* which is used to check if the shortest path calculation is completed. CUDA provides __syncthreads() method to synchronize threads. When the method is encountered in the kernel, all threads in a block will be blocked at the calling location until each of them reaches the location. This function is used at appropriate parts of the program to synchronise the threads. In the shortest path calculation, the loop control variable (which actually denotes the vertex being processed) is updated by the number of threads in the block in the x direction. The neighbors (*w*) of the vertex (*v*) chosen in the outer loop are processed and the number of shortest paths between vertices *s* and *w* are updated by the number of shortest paths between vertices *s* and *v* as seen in Figure 3 . For the dependency accumulation step the same method is followed and the dependency of the immediate neighbours of a vertex are updated along with the Betweenness Centrality of the vertex.

```
1: for $v \in V$ on wavefront in parallel do
2:     for $w \in N(v)$ do
3:         if $d_s(w) = \infty$ then $d_s(w) \leftarrow d + 1$
4:         if $d_s(w) = d + 1$ then $\sigma_s(w) \leftarrow \sigma_s(w) + \sigma_s(v)$
5:     end for
6: end for
```

Figure 3: Shortest Path calculation of the Vertex parallel algorithm [4]

E. Edge Parallel Algorithm

The edge-parallel algorithm assigns a thread to each edge of the graph and that thread traverses that edge only. There are 3 shared variables: *s* for identifying the source vertex, *current_depth* for keeping track of the depth of the discovered and a flag variable *done* which is used to check if the shortest path calculation is completed. The COO representation of the graph is used in this algorithm as we iterate over the edges rather than the vertices. In the shortest path calculation, the loop control variable (which actually denotes the index of the edge being processed) is updated by the number of threads in the block in the x direction. Only the edges which have the same depth as the current depth are chosen to prevent unnecessary allocation of threads to edges which are not going to be processed in the current iteration. The distance and number of shortest paths of the vertices at the two ends of the edge are updated as seen in Figure 4 . For the dependency accumulation step the same method is followed and the dependency of the immediate neighbours of a vertex are updated using atomicAdd() function. The Betweenness Centrality is calculated by summing up the partial dependencies of all the vertices.

```
1: for edge $(v, w)$ incident to wavefront in parallel do
2:     if $d_s(w) = \infty$ then $d_s(w) \leftarrow d + 1$
3:     if $d_s(w) = d + 1$ then $\sigma_s(w) \leftarrow \sigma_s(w) + \sigma_s(v)$
4: end for
```

Figure 4: Shortest Path calculation of the Edge parallel algorithm [4]

F. Work Efficient Algorithm

To ensure that only edges in the current iteration are traversed for each iteration, we need to use explicit queues to store vertices as shown in Figure 5. 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 for this approach one for maintaining the current vertex and the other which has the vertices to be processed in the next iteration. The length of both the queues are stored as a shared variable so that at the end of each iteration the vertices from one queue are transferred to the other. The atomicCAS() function is used to prevent duplicate vertices from being pushed to the queue as shown in Figure 6. In the dependency accumulation stage, rather than traversing the predecessors directly, all of the neighbors of a vertex are instead traversed. The successors are checked rather than the predecessors which are followed in the serial implementation as shown in Figure 7. The dependency accumulation for the source vertex is also ignored as it is naturally 0. the algorithm assigns threads only to the vertices that need to accumulate their dependency values. We prevent the assigning of threads to each vertex or edge and checking to see if that vertex or edge belongs to the current depth. This strategy prevents unnecessary branch overhead and accesses to global memory that are made by previous implementations

```
for $v \in V$ do in parallel
    if $v = s$ then
        $d[v] \leftarrow 0$
        $\sigma[v] \leftarrow 1$
    else
        $d[v] \leftarrow \infty$
        $\sigma[v] \leftarrow 0$
    $\delta[v] \leftarrow 0$
$Q_{curr}[0] \leftarrow s$ ; $Q_{curr\_len} \leftarrow 1$
$Q_{next\_len} \leftarrow 0$
$S[0] \leftarrow s$ ; $S_{len} \leftarrow 1$
$ends[0] \leftarrow 0$ ; $ends[1] \leftarrow 1$ ; $ends_{len} \leftarrow 2$
shared $depth \leftarrow 0$
```

Figure 5: Work-efficient Betweenness Centrality Local Variable Initialization [3]

```
Stage 1: Shortest Path Calculation
while $true$ do
    for $v \in Q_{curr}$ do in parallel
        for $w \in neighbors(v)$ do
            if $atomicCAS(d[w], \infty, d[v]+1) = \infty$ then
                $t \leftarrow atomicAdd(Q_{next\_len}, 1)$
                $Q_{next}[t] \leftarrow w$
            if $d[w] = d[v] + 1$ then
                $atomicAdd(\sigma[w], \sigma[v])$
    $barrier()$
    if $Q_{next\_len} = 0$ then
        $depth \leftarrow d[S[S_{len} - 1]]$ - 1
        break
    else
        for $tid \leftarrow 0 \dots Q_{next\_len} - 1$ do in parallel
            $Q_{curr}[tid] \leftarrow Q_{next}[tid]$
            $S[tid + S_{len}] \leftarrow Q_{next}[tid]$
        $barrier()$
        $ends[ends_{len}] \leftarrow ends[ends_{len} - 1] + Q_{next\_len}$
        $ends_{len} \leftarrow ends_{len} + 1$
        $Q_{curr\_len} \leftarrow Q_{next\_len}$
        $S_{len} \leftarrow S_{len} + Q_{next\_len}$
        $Q_{next\_len} \leftarrow 0$
        $barrier()$
```

Figure 6: Work-efficient Betweenness Centrality Shortest Path Calculation [3]

```
Stage 2: Dependency Accumulation
while $depth > 0$ do
    for $tid \leftarrow ends[depth] \dots ends[depth+1] - 1$ do
    in parallel
        $w \leftarrow S[tid]$
        $dsw \leftarrow 0$
        $sw \leftarrow \sigma[w]$
        for $v \in neighbors(w)$ do
            if $d[v] = d[w] + 1$ then
                $dsw \leftarrow dsw + \frac{sw}{\sigma[v]}(1 + \delta[v])$
        $\delta[w] \leftarrow dsw$
    $barrier()$
    $depth \leftarrow depth - 1$
```

Figure 7: Work-efficient Betweenness Centrality Dependency Accumulation [3]

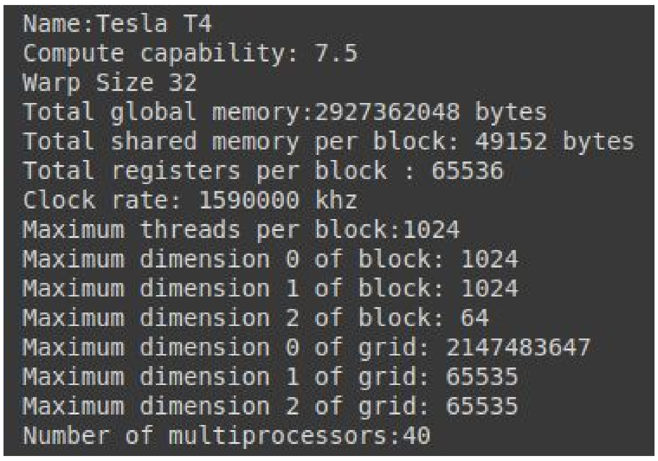
G. *Work Distributed Algorithm*

The Work Efficient method solves the problem of unnecessary allocation of threads. But to make the maximum use of CUDA we can also divide the computations using multiple blocks. The number of blocks is calculated based on the maximum memory allowed. We divide the maximum memory by the number of vertices and edges to calculate the number of blocks. The Work efficient code is modified to use the multiple blocks available. This optimises the previous algorithm by multiple folds and hence improves the execution time as discussed in the results.

IV. Results

The parallel and serial algorithms were executed on Google Collab notebook GPU using the nvcc plugin with the following specifications:

- Number of multiprocessors:40
- Clock rate: 1.59 Ghz
- Total global memory: 3 GB
- Tesla GPU : Tesla T4

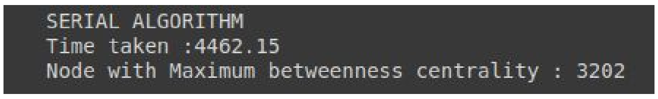


```
Name:Tesla T4
Compute capability: 7.5
Warp Size 32
Total global memory:2927362048 bytes
Total shared memory per block: 49152 bytes
Total registers per block : 65536
Clock rate: 1590000 khz
Maximum threads per block:1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 2147483647
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Number of multiprocessors:40
```

Figure 8: Hardware specifications

A random graph was generated and stored in the input file in CSR format and also converted to COO format. The graph contains 5000 vertices and 5000 edges. The number of threads per block was set to 1024.

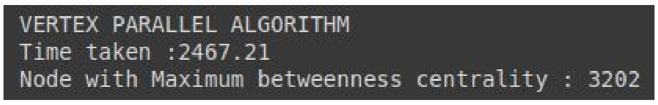
a. The Serial Algorithm took 4462.15 ms to complete its execution Figure 9.



```
SERIAL ALGORITHM
Time taken :4462.15
Node with Maximum betweenness centrality : 3202
```

Figure 9: Serial algorithm results

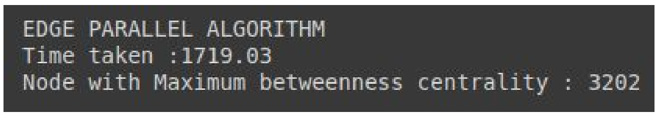
b. The Vertex Parallel Algorithm took 2467.21 ms to complete its execution Figure 10.



```
VERTEX PARALLEL ALGORITHM
Time taken :2467.21
Node with Maximum betweenness centrality : 3202
```

Figure 10: Vertex Parallel algorithm results

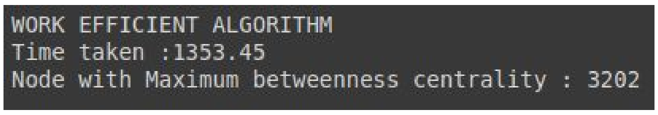
c. The Edge Parallel Algorithm took 1719.03 ms to complete its execution Figure 11.



```
EDGE PARALLEL ALGORITHM
Time taken :1719.03
Node with Maximum betweenness centrality : 3202
```

Figure 11: Edge Parallel algorithm results

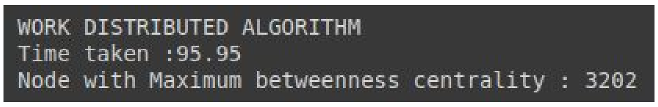
d. The Work Efficient Algorithm took 1353.45 ms to complete its execution Figure 12.



```
WORK EFFICIENT ALGORITHM
Time taken :1353.45
Node with Maximum betweenness centrality : 3202
```

Figure 12: Work Efficient algorithm results

e. The Work Distributed Algorithm took 95.95 ms to complete its execution Figure 13.



```
WORK DISTRIBUTED ALGORITHM
Time taken :95.95
Node with Maximum betweenness centrality : 3202
```

Figure 13: Work Efficient algorithm results

V. Analysis

1. The Vertex Parallel algorithm provides an optimisation over the serial implementations by using multiple threads provided by CUDA but it does not prevent imbalanced allocation of threads.
2. The Edge Parallel algorithm is better load balanced, but it also requires more memory accesses. As generally the number of edges are more generally compared to the number of vertices there is more memory access. This approach does not completely solve unnecessary thread allocation.
3. The Work efficient algorithm solves the problem of unnecessary thread allocation and improves the execution time.
4. The Work Distributed algorithm makes use of a dynamic number of blocks dependent on the size of the graph to make proper use of the advantages provided by CUDA.

We made comparisons of the above algorithms in different ways as mentioned below.

A. Based on Graph Size

CUDA is mainly used in cases where Data Level Parallelism is required and hence the algorithms were run on graphs with different numbers of vertices and vertices. Graphs with a specified number of vertices ranging from 500 to 50000 were generated and stored in the CSR and COO format in an input file. The number of threads for each block was set to 1024 to maximise the computation power provided. For a graph with a low number (500, 1000) of vertices, the vertex parallel, edge parallel and work efficient algorithms took more time to execute than the serial algorithm because of cost of initialization of threads and their communication overheads. But the Work Distributed algorithm had a better execution time than the serial algorithm. By increasing the number of vertices and edges the parallel algorithms show to have a lower runtime as seen in Table 3. The work distributed algorithm was consistently showing faster results when compared to the serial algorithm. For example, for a graph with 50,000 vertices and edges the Serial algorithm took 8.9 minutes to execute but the Work Distributed algorithm finished its execution in 15 seconds.

No of vertices and edges	**Execution time (milliseconds)**				
	Serial	**Vertex**	**Edge**	**Work Efficient**	**Work Distributed**
500	40.35	95.31	55.59	114.30	24.41
1000	174.18	233.81	178.06	266.45	25.73
2000	696.3	529.07	403.69	439.73	34.55
3000	1598.29	1019.63	670.56	684.58	49.15
4000	2740.26	1527.83	1069.9	908.60	66.66
5000	4462.15	2467.21	1719.03	1353.45	95.95
10000	18020.1	8782.2	6734.76	3717.3	269.26
50000	536997	207194	174811	65036.8	15660.6

Table 3: Comparison based on number of vertices and edges with 1024 threads per block

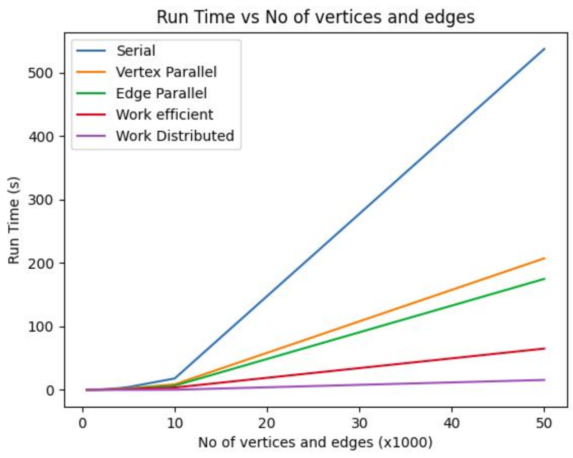


Figure 14: Plot for comparison based on number of vertices and edges with 1024 threads per block

B. Based on Number of Threads per Block

The maximum number of threads per block were varied and the comparison was made for each algorithm with a graph of 5000 vertices and a fixed number of blocks. Since the serial algorithm utilises only one thread the execution time was constant for different numbers of threads and was used as a threshold to make the comparisons for each parallel algorithm. CUDA works well only when the number of threads are in order of hundreds, so the threads were set to powers of 2 over 100 till the maximum limit provided. It was observed that for a low number of threads the Vertex parallel and the Edge parallel algorithms performed worse than the Serial algorithm due to cost of initialization of threads and their communication overheads consuming more time. But the Work efficient and the Work Distributed algorithms proved to be faster even with a low number of threads. With the increase in number of threads the execution times decreased except for the Work Distributed parallel algorithm. It was observed that the execution time for Work Distributed parallel algorithm decreased with an increase in the number of threads till it reached a minima and started increasing again. The time for 512 threads is 74.527 ms and for 1024 threads is 95.95 ms. This proves that more the number of threads for each block does not assure the maximum utilisation of the GPU.

No of threads per block	Execution time (milliseconds)				
	Serial	Vertex	Edge	Work Efficient	Work Distributed
128	4462.15	6987.55	6385.65	2328.56	134.329
256	4468.15	4104.33	3548.33	1690.65	80.017
512	4462.15	2828.54	2241.53	1418.06	74.527
1024	4462.15	2467.21	1719.09	1353.45	95.95

Table 4: Comparison based on number of threads per block

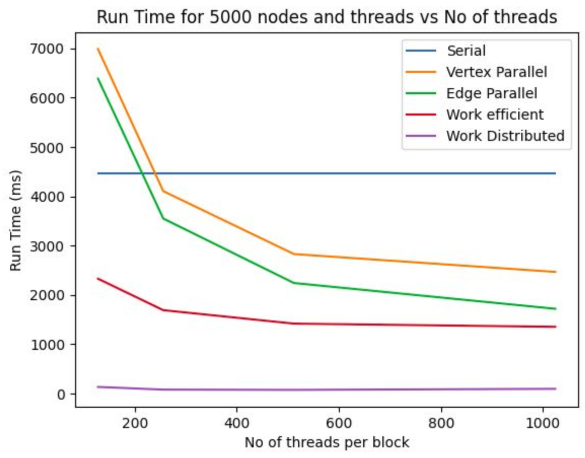


Figure 15: Plot for comparison based on number of threads per block

C. Performance Gain

We have also used the performance gain approach to compare our results. Performance gain is defined as the follows.

$$PG = \frac{(ST - PT) * 100}{ST} \tag{4}$$

where ST is the execution time for the serial algorithm and PT is the execution time for the parallel algorithm. For a low number of vertices and edges the parallel algorithms (except for Work Distributed) have a negative PG. With large number of vertices and edges the Work Distributed algorithm gave a performance gain of over 95%

No of vertices and edges	Performance Gain (%)			
	Vertex	Edge	Work Efficient	Work Distributed
500	-136.20	-37.77	-183.26	39.50
1000	-34.25	-2.22	-52.97	85.22
2000	24.01	42.02	36.84	95.03
3000	36.20	58.04	57.16	96.92
4000	44.24	60.95	66.84	97.56
5000	44.70	61.47	69.66	97.84
10000	51.26	62.62	79.37	98.50
50000	61.41	67.44	87.88	97.08

Table 5: Comparison based on Performance Gain

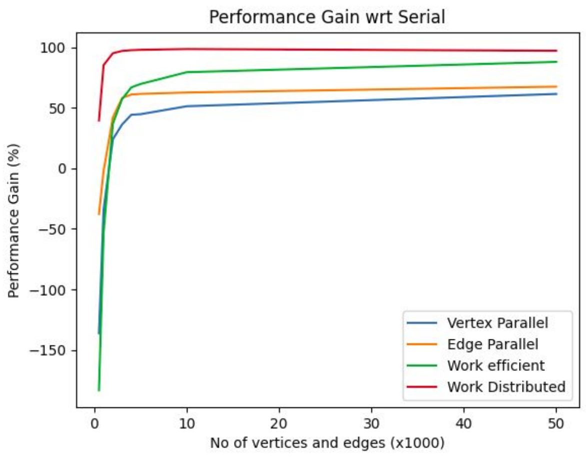


Figure 16: Plot for comparison based on Performance Gain

VI. Conclusion

In this project we compared various methods to calculate the Betweenness centrality of graphs using CUDA. Since the structure of real-world graphs can vary tremendously, no single decomposition of threads to tasks provides peak performance for all input. For high-diameter graphs, using asymptotically efficient algorithms is paramount to obtaining high performance, whereas for low-diameter graphs it is preferable to maximize memory throughput, even at the cost of unnecessary memory accesses.

VII. Future Work

The work efficient approach assigns threads to units of useful work, but the distribution of edges to threads is also dependent on the structure of the graph. Large graphs with more uniform distribution of edges and vertices are more suitable for this algorithm. But for small graphs the high cost of initialization of threads and their communication overheads outweigh the runtime for the actual calculation.
For these cases the load balance and high memory-throughput of the edge-parallel method is preferable to the work-efficient method. So a Hybrid algorithm can be used to efficiently recognise the right method to use to perform the computations.

VIII. Individual contribution

Harshvardhan R (181IT217) : Work Distributed algorithm and Graph Representation.

Akashdeep S (181IT103) : Serial Implementation, Work Efficient algorithm.

Sujan Reddy A(181IT147) : Vertex and Edge Parallel algorithm.

IX. References

[1] R. W. Floyd, "Algorithm 97: Shortest path," Commun. ACM, vol. 5, no. 6, pp. 345–, Jun. 1962.

[2] U. Brandes, "A Faster Algorithm for Betweenness Centrality," Journal of Mathematical Sociology, vol. 25, pp. 163–177, 2001.

[3] Adam McLaughlin, David A. Bader (2014) Scalable and High Performance Betweenness Centrality on the GPU.

[4] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart, "Edge v. Node Parallelism for Graph Centrality Metrics," GPU Computing Gems, vol. 2, pp. 15–30, 2011

[5] CUDA C Programming Guide, October 2018. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-general-purpose-parallel-computing-architecture