IT301: Parallel Computing
Parallelization of Floyd-Warshall Algorithm

Gaurang Jitendra Velingkar
191IT113
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
gaurang.191it113@nitk.edu.in

Rakshita Varadarajan
191IT140
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
rakshitajps.191it140@nitk.edu.in

Ritvik Mahesh
191IT246
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
ritvikmahesh.191it246@nitk.edu.in

***Abstract*—Before traveling, it is of utmost priority to determine not just any path to the destination, but to find the optimal or shortest path that can be taken from the source to the destination. The Floyd Warshall Algorithm is used to solve the All Pairs Shortest Path problem in a graph. This paper aims at improving the efficiency of the sequential implementation of the algorithm by parallelizing the same. Based upon the results obtained from 3 different programming environments (namely OpenMP, MPI and Cuda) on varying the size of the graph and the number of threads involved, a solid comparison on the performance inn terms of speedup and run time were made.**
Keywords – all-pairs, shortest path, Floyd-Warshall.

I. Introduction

Shortest Path algorithms are one of the most deeply researched and used algorithms in *Graph Theory*. A *graph* in programming is an object with *vertices*, that are connected to each other via *edges*. Each of these edges may have some *cost* associated with it. The internet or networks in general and digital maps are some examples of entities that can be visualized as graphs. Finding the shortest paths between cities in a map, routers in a network, or in a programming sense, between vertices in a graph, therefore becomes necessary for optimal performance.

The shortest path from one vertex in the graph to another is the one that minimises the sum of cost of all edges along the path. Several algorithms have been designed to find the shortest paths in graphs, such as the Dijkstra's algorithm, Bellman-Ford algorithm and the Floyd-Warshall algorithm to name a few. Each of these work differently and are used in different scenarios. Out of these, the Floyd-Warshall algorithm happens to be the most expensive algorithm with respect to time taken for execution that makes it an interesting subject for research on the effect of parallelization.

The Floyd-Warshall algorithm is a dynamic programming approach at finding the shortest path between all pairs in a weighted graph without negative cycles. It was published in 1962 by Robert Floyd. However, it was the same as the algorithms previously published by Stephen Warshall for finding the transitive closure of graphs. Hence, the algorithm came to be known as the Floyd-Warshall algorithm. The algorithm is immensely used in networks for finding optimal routes between routers.

The algorithm works regardless of whether the graph is directed or undirected but fails when the graph has a negative cycle. Thus, this algorithm is especially useful in finding the optimal distance between a source and a destination. Other usages of the algorithm include inverting real matrices, checking if a graph is bipartite or not, finding similarities between graphs, detecting negative cycles in a graph and more. The algorithm consists of three loops, each of constant complexity. Thus, the Floyd Warshall algorithm has a time complexity of $O(n^3)$, while the space complexity is $O(n^2)$. This turns out to be inefficient considering both performance and power consumed. This inefficiency can be dealt with by paralleling the algorithm using various mechanisms.

The core idea of this paper is to parallelize the Floyd-Warshall Algorithm in three different environments using different mechanisms:

- Shared-Memory Multithreading using OpenMP
- Distributed Computing using MPI, and
- Distributing amongst Graphical Processing Units (GPUs) using CUDA

Further, the paper aims to compare the performance measures obtained from all three mechanisms along with those obtained from the naive algorithm implementation to find the most efficient execution of the algorithm.

II. Literature Survey

Current research on the topic mainly throws light on the applications of the Floyd-Warshall algorithm in networks and maps. There has been research to compare the performance of the Floyd-Warshall Algorithms with other similar shortest-path algorithms. There has also been a study on the parallelized Floyd-Warshall algorithm using TBBs. However, there has been limited research with respect to parallelizing the algorithm using OpenMP, MPI and CUDA. With the algorithm already being expensive in terms of its worst-case time complexity ($O(n^3)$), there is a need to employ methods to improve the runtime of the algorithm. Parallelizing the algorithm can

be one of the methods to improve the runtime. This paper aims to compare the various parallel implementations of the Floyd-Warshall algorithm using OpenMP, MPI and CUDA so that it can be used in various scenarios that uses shortest-path algorithms for graphs with a large number of nodes. A summary of the survey is given in Table I.

III. Methodology

A. Sequential

The Floyd-Warshall algorithm is used to find the shortest path between all pairs of vertices in a weighted directed or undirected graph. However, the algorithm fails in the presence of a negative cycle.

The weighted graph is represented using an adjacency matrix $adj\_mat[][]$, which is a $n \times n$ matrix, where n is the number of vertices. The indices i and j of the matrix represent the vertices of the graph. The value at position $adj\_mat[i][j]$ is the weight of the edge from i^{th} vertex to j^{th} vertex. The diagonal values are 0.

After initializing the adjacency matrix with random values using *rand()%10000*, the *floydWarshall* function is called.

In the *floydWarshall* function, the result is stored in another $n \times n$ matrix named $distances[][]$. The values of this matrix are initialized using the values of the adjacency matrix. The next step involves modifying the $distances[i][j]$ with the shortest distance between vertex i and vertex j. This is done using three loops where k, i, and j are the loop variables. The outermost loop variable k indicates the intermediate vertex, and inner loop variables i and j indicates the vertices between which the shortest path must be found. Here the values of $distances[i][j]$ is modified if the intermediate distance from source to destination through vertex k is less than the direct path from i to j. Hence, the shortest path is found using the following condition,

$$D_k[i,j] = min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$$

Where D_k indicates the distances matrix after the k^{th} iteration. After n iterations, D_n gives the length of the shortest path between i and j. This algorithm is unsuccessful if the diagonals of the distances matrix hold a negative value, indicating the presence of a negative weighted cycle in the graph.

Since the algorithm uses three loops to compute the value of the distances matrix using a dynamic programming approach, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$. This is highly inefficient. Hence it is necessary to introduce parallel techniques to acquire better performance.

Since the algorithm uses the two-dimensional matrix to represent the graph as well as to store the result, the space complexity of the Floyd Warshall algorithm is $O(n^2)$.

Algorithm 1: The Floyd-Warshall algorithm Sequential

```
for (u, v) ∈ Edges do
    D_{u,v} ← weight(u, v)
end

for v = 1 → n do
    D_{v,v} ← 0
end

for k = 1 → 4n do
    for i = 1 → n do
        for j = 1 → n do
            if D_{i,j} > D_{i,k} + D_{k,j} then
                D_{i,j} ← D_{i,k} + D_{k,j}
            end
        end
    end
end
```

B. Parallel Techniques - OpenMP

Open Specification for Multi-Processing (OpenMP) is an application program interface that supports multi-threaded, shared memory parallelism. OpenMP can be used to perform parallelism in languages like C, C++, and Fortran. It includes API components like Compiler Directives, Runtime library routines, and Environmental Variables.

In order to parallelize the sequential algorithm mentioned in the previous section, the *omp parallel* directive is used, which tells the compiler to parallelize the below-chosen block of code. Here the *omp parallel* directive is utilized to parallelize the three loops needed for the algorithm. Along with *omp parallel* directive the $num\_threads()$, $shared()$ and $private()$ clauses are used. The $num\_threads$ clause defines the number of threads to be used in the parallel region, the shared clause declares variables in the list to be shared among various threads in the team, and the private clause declares variables in the list as private to each thread in a team.

In this code, the variable k, the loop variable of the outermost loop, is declared using the private clause, the adj_mat and $distances$ matrices are declared using the $shared$ clause.

In addition to the *omp parallel* directive, the *omp for* directive is used for the inner two loops to distribute loop iterations within the team of threads. The private clause is used along with the *omp for* directive to declare the inner looping variables i and j.

The scheduling clause is also used along with the *omp for* directive, which specifies how different iterations of the for loop are split between the available threads. Here, four types of scheduling, namely *static*, *dynamic*, *guided*, and *runtime*, were used by varying the number of node and threads. Each of their execution times were recorded, and a graph was plotted to compare all types of scheduling, varying the number of nodes with runtimes.

The first type of scheduling used is dynamic scheduling with a chunksize of 20. In this type of scheduling, iterations are divided among the team of threads as and when they request them. Each thread receives chunksize amount of chunk of iteration on request. Every chunk contains chunksize except

TABLE I
SUMMARY OF LITERATURE SURVEY

Ref No.	Study Description	Conclusions/Results
1.	This paper studied two algorithms that find the shortest path between locations. The two algorithms used were the Floyd-Warshall algorithm and a Heuristic method, which is a greedy algorithm that considers local optimum for nodes as compared to the former that checks all nodes within the graph to calculate the shortest paths.	It was found that the greedy method was faster as it just considered nearest routes with the lowest value or local optimums, but wasn't accurate. In contrast, the Floyd-Warshall algorithm took longer to compute the results, but was highly accurate as it went through all nodes in the graph.
2.	This paper implements a parallelized version of the Floyd-Warshall algorithm using Threading Building Blocks (TBB). TBB is a C++ library for parallelization. It performed a detailed analysis of the performance by varying number of threads and number of nodes within the graph.	The results of this paper showed that the parallel programs performs immensely better than the serial implementation and can be used for applications using graphs with a large number of nodes.
3.	This research paper compared the performance of some common shortest path algorithms like Dijkstra, Floyd-Warshall and Bellman-Ford algorithms and the genetic algorithm. The genetic algorithm uses a dynamic approach that results in a different optimal solution everytime it is run.	The results showed that the Dijkstra, Floyd-Warshall and Bellman-Ford algorithms perform optimally and in polynomial time. They are consistent in performance, as opposed to the genetic algorithm that can produce different optimal solutions in different executions. Bellman-Ford has the worst time complexity compared to Dijkstra and Bellman-Ford.
4.	This paper studied CUDA and it's parallel programming model. It was developed by NVIDIA for it's GPUs and is increasingly being used for parallel processing. The paper discussed the working of CUDA and it's various applications in health-care, like MRIs, CBCT, Ultrasound, etc.	The results of this paper showed how CUDA can be used in various real-world scenarios in health-care. It threw light on the fact that CUDA provides direct access to the virtual instruction set of GPUs. It ensures exceptional performance wherever it's used, and especially helps with maintenance of highly customized environments.

the last chunk.

Static scheduling is similar to dynamic scheduling, except that the chunks are statically provided to the threads in round-robin fashion based on the thread number.

In guided scheduling, like dynamic scheduling, iterations are assigned to threads as they request them. However, the chunk of iterations assigned varies. Here the chunk assigned to each thread is the ratio of the remaining number of iterations and the total number of threads. The chunksize specified determines the minimum size of chunk of iterations.

Finally, runtime scheduling is determined during runtime using the *OMP_SCHEDULE* environmental variable.

C. Parallel Techniques - MPI

MPI, or Message Passing Interface, is a message passing library specification or standard. It allows message passing between processes. This standard formulated provides the details (such as the syntax and semantics) needed for the important library routines which allows coding message-passing programs in C, C++, and Fortran. It is based on Single Program, Multiple Data (SPMD) technique. This implementation uses OpenMPI, which is an open source Message Passing Interface implementation.

There are three basic routines from Collective Communication that are used in this parallelized MPI-based implementation. *MPI_Scatter* is one such routine used to send chunks of an array to all the processes in a communicator. In order to facilitate scattering the adjacency matrix among all processes using *MPI_Scatter*, the adjacency matrix is initialized as a one dimensional array of size n^2 (n is the number of vertices) instead of the conventional two dimensional array. Thus, each process will get a portion of the adjacency matrix having a size of $\frac{n^2}{p}$, where p stands for the number of processes.

In this code, the outer loop of the three loops having k as the looping variable runs till the value of k is that of the number of vertices (n). Since the adjacency matrix of the vertices has been scattered, each process has the adjacency matrix of $\frac{n}{p}$ vertices. This k value is used to identify an intermediate vertex and thus needs to access the entire adjacency matrix having all vertices (not just the scattered blocks amongst the processes).

Algorithm 2: The Floyd-Warshall algorithm in OpenMP

```
for (u, v) ∈ Edges do
    D_{u,v} ← weight(u, v)
end

for v = 1 → n do
    D_{v,v} ← 0
end

#pragma omp num_threads(8) private(k) shared(adj_mat,D)
for k = 1 → n do
    #pragma omp for private(i,j) schedule(dynamic,20)
    for i = 1 → n do
        for j = 1 → n do
            if D_{i,j} > D_{i,k} + D_{k,j} then
                D_{i,j} ← D_{i,k} + D_{k,j}
            end
        end
    end
end
```

Therefore, the adjacency matrix values corresponding to the k^{th} vertex are stored in a separate array by the process having the block containing the k^{th} vertex, following which this array is broadcasted to all the other processes by *MPI_Bcast*. *MPI_Bcast* is a collective communication routine used to broadcast a message from one process to all other processes of the group. As this broadcasting happens everytime the value of k changes, a total of k *MPI_Bcast* operations are done. These operations, being collective communication operations, act as points of synchronization among the processes.

The remaining two loops within the outer loop work in the same manner as the original Floyd Warshall Algorithm. However, as the processes each have their own chunk of the adjacency matrix, the second loop runs $\frac{n}{p}$ times, while the third loop runs n number of times.

As all these processes write the updated values of shortest distances in their own local matrix, all these local matrices are gathered to the root process into one resultant matrix at the end of the k^{th} for-loop. This is done utilizing *MPI_Gather*, which is essentially just the reverse of *MPI_Scatter*. This routine collects chunks of data (an array) from all the processes into one single process.

D. Parallel Techniques - CUDA

CUDA, or Compute Unified Device Architecture, is an API that helps software programs use GPUs for general processing. The major difference between a GPU and a CPU is that a GPU has hundreds of cores. CUDA utilizes these cores to execute multiple processes in the same program in a parallelized fashion, thereby following the SPMD (Single Program Multiple Data) processing model.

Algorithm 3: The Floyd-Warshall algorithm in MPI

```
for (u, v) ∈ Edges do
    D_{(u*n)+v} ← weight(u, v)
end

for v = 1 → n do
    D_{(v*n)+v} ← 0
end

div ← (n * n)/p

MPI_Scatter(D, div, MPI_INT, M, div, MPI_INT, 0, MPI_COMM_WORLD)

floydWarshall(M, rank, p)

MPI_Gather(M, div, MPI_INT, Ans, div, MPI_INT, 0, MPI_COMM_WORLD)

Function floydWarshall(M, r, p):
    d ← n/p
    for k = 1 → n do
        pos ← k%d
        rank ← k/d
        if r == rank then
            for x = 1 → n do
                kthRow_x ← M_{(pos*n)+x}
            end
        end
        MPI_Bcast(kthRow, n, MPI_INT, rank, MPI_COMM_WORLD)
        for i = 1 → d do
            for j = 1 → n do
                dist ← M_{(i*n)+k} + kthRow_j
                if M_{(i*n)+j} > dist then
                    M_{(i*n)+j} ← dist
                end
            end
        end
    end
    return
```

CUDA allows defining blocks in one, two or three dimensions as grids. Additionally, threads within blocks can also be defined in up to three dimensions. However, a block can have a maximum of 1024 threads, while a grid supports up to 65536 blocks in each dimension. This paper proposes an algorithm that utilizes this to use a grid of blocks in 2D with each block further having threads in two dimensions.

Due to the limitation on the number of threads per block, the number of threads are defined in every program, and the workload is then split across blocks. This implementation defines a $\frac{n}{t} \times \frac{n}{t}$ grid of blocks, with each having $t \times t$ threads. Much like the previous parallelization techniques,

CUDA is also used to parallelize the two inner loops. The kernel function, *floydWarshallGPU* is run n times. The inner variables, i and j are calculated as

$$i = blockIdx.x * blockDim.x + threadIdx.x$$

$$j = blockIdx.y * blockDim.y + threadIdx.y$$

This ensures that i and j vary from 0 to n, just as required. CUDA requires that arrays need to be copied from the CPU memory to the GPU and vice-versa. To make this easier, one dimensional arrays are defined of the size n^2 and $arr[i][j]$ is accessed as $arr[i*n+j]$. First, the $distance$ array is initialised in CPU with values from the adjacency matrix. Once space is allocated in the GPU for computation, values are copied from the CPU to GPU. Post this, the kernel function is run. Within this, after computing the value of i and j as mentioned above, threads are synced before and after applying the core part of the algorithm -

$$D_k[i,j] = min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$$

The syncing of threads is done using the *__syncthreads* function. This ensures that all threads in the block have the same version of the array. After GPU computation, values are copied back to the CPU memory, after a *cudaDeviceSynchronize* directive, which ensures that all blocks within the GPU are synchronized and are done executing.

Algorithm 4: The Floyd-Warshall algorithm in CUDA

for $(u,v) \in Edges$ **do**
  $D_{(u*n)+v} \leftarrow weight(u,v)$
end

for $v = 1 \rightarrow n$ **do**
  $D_{(v*n)+v} \leftarrow 0$
end

$block \leftarrow \frac{n}{t} \times \frac{n}{t}$

$threads \leftarrow t \times t$

for $k = 1 \rightarrow n$ **do**
  floydWarshallGPU$<<<block, threads>>>(D)$
end

Function `floydWarshallGPU(D)`**:**
  $i \leftarrow (blockIdx.x \times blockDim.x) + threadIdx.x$
  $j \leftarrow (blockIdx.y \times blockDim.y) + threadIdx.y$
  syncthreads()
  if $D_{i,j} > D_{i,k} + D_{k,j}$ **then**
    $D_{i,j} \leftarrow D_{i,k} + D_{k,j}$
  end
  syncthreads()
  return

IV. Results and Analysis

The sequential Floyd-Warshall algorithm was compared with the parallelized versions implemented using OpenMP, MPI, and CUDA. A comprehensive analysis was performed by varying the data size and number of threads for all implementations.

A. OpenMP

The sequential algorithm was analyzed by varying the number of nodes to see how it performs compared to the parallel versions. As expected, it took longer to execute than the parallel versions of the code. The results have been plotted along with the OpenMP parallelized version of the algorithm in Fig. 1 and tabulated in Table II and Table III.

Further, the OpenMP algorithm was analyzed extensively by varying various parameters like:

1) Number of nodes
2) Scheduling type
3) Number of threads

It was found that the parallel versions execute much faster than the sequential when using 8 threads. While *static* scheduling takes longer than its parallel counter-parts when the number of nodes is large, it's comparable to the other scheduling types for smaller graphs. Changing scheduling types have little effect on the time taken to run the algorithm, as shown in Fig. 1.

It is a similar situation on varying the number of threads. The code was executed with *dynamic* scheduling, and the number of threads was varied to analyze the effect. When only one thread is used, it takes a long time to execute. It was found that the effect of the number of threads on the runtime is negligible when the number of threads were varied between 2 and 16, as shown by straight lines parallel to the x axis in Fig. 2. The results have also been tabulated in Table IV.

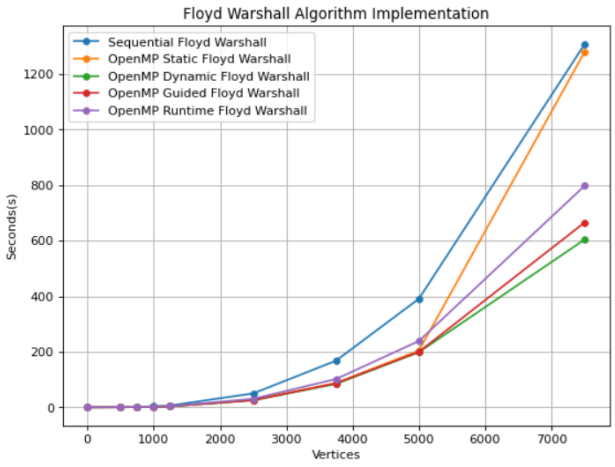

Fig. 1. OpenMP Runtime vs Nodes

TABLE II
EXECUTION TIMES FOR THE SEQUENTIAL ALGORITHM

Algorithm	No. of Nodes							
	500	750	1000	1250	2500	3750	5000	7500
Sequential	0.422094	1.384741	3.222286	6.381931	49.762170	167.688373	390.662876	1307.492269

TABLE III
EXECUTION TIMES FOR THE OPENMP ALGORITHM WITH 8 THREADS & DIFFERENT SCHEDULES

Schedule	No. Of Nodes							
	500	750	1000	1250	2500	3750	5000	7500
Static	0.256139	0.763251	1.753842	3.306170	25.882340	87.366389	203.911654	1279.759253
Runtime	0.288553	0.907263	2.059180	3.945450	30.440848	101.550698	238.506913	797.356790
Guided	0.233105	0.748041	1.700897	3.283199	25.413071	86.269530	199.419663	665.432118
Dynamic	0.237393	0.740559	1.682285	3.231955	25.085005	84.068729	198.316902	664.064237

TABLE IV
EXECUTION TIMES FOR THE OPENMP ALGORITHM WITH DIFFERENT NUMBER OF THREADS

No. of Nodes	No. Of Threads					
	1	2	4	8	16	32
500	0.485256	0.254166	0.278963	0.285666	0.276869	0.300628
1250	7.038478	3.748890	3.709332	3.727504	3.777829	3.801500
2500	56.020662	28.844804	29.145761	28.788539	29.296373	29.588597

TABLE V
EXECUTION TIMES FOR THE MPI ALGORITHM WITH DIFFERENT NUMBER OF PROCESSES

No. of processes	No. Of Nodes							
	500	750	1000	1250	2500	3750	5000	7500
1	0.431461	1.375540	3.182001	6.188979	50.150933	166.003417	412.136310	1325.932049
2	0.212534	0.736809	1.602658	3.171841	24.969531	92.337896	216.585473	738.390621
4	0.163430	0.584663	1.377823	2.574914	19.922221	85.763126	148.464114	488.099018
5	0.560581	0.833428	2.044864	3.726466	28.022778	97.398225	229.157220	769.443911
6	0.225698	0.619068	1.574791	2.950205	27.479623	83.829576	199.947368	648.187426
7	0.174054	0.668151	1.346150	3.154860	25.506249	83.671826	185.290698	620.251866
8	0.097505	0.345942	0.825535	2.130025	19.046290	68.002339	148.380683	499.123987

B. MPI

The MPI implementation was executed on a system with 4 cores, and an *Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz* processor. It was then analyzed by varying the number of processes and the number of nodes within the graph. As expected, the runtime increases with the number of nodes. When compared with the number of processes, it can be seen that the performance is best when the number of processes is 4. In general, however, performance isn't affected much when the number of processes is varied between 2 and 8 for nodes less than 3750. The difference is much more significant for graphs with 5000 or more nodes. This can be seen in Fig. 3.

Another point to be noted, by comparing values in Table II and values for 1 process from Table V, is that the MPI algorithm performs worse than the sequential algorithm, with significant differences for graphs with more than 5000 nodes, negligible otherwise. This can be attributed to the additional processing done for the distribution of data across processes.

A deeper analysis on the effect of varying the number of processes, as shown in Fig. 4, showed that the performance worsens when the number of processes increases from 4 to 5. It again gets better as the number of processes is reduced to 8, which is comparable with the performance of 4 processes. The variation is significant when the number of nodes in the graph is large. This happens because the program is run on a computer with 4 cores. With 4 processes equally distributed across the 4 cores, all cores perform efficiently and give the best performance. When the number of processes

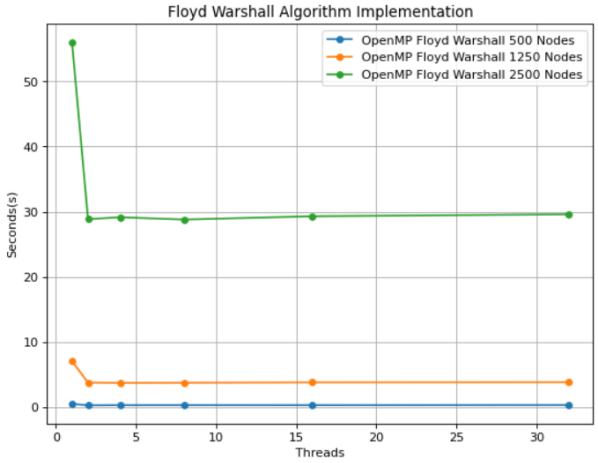

Fig. 2. OpenMP Runtime vs Threads

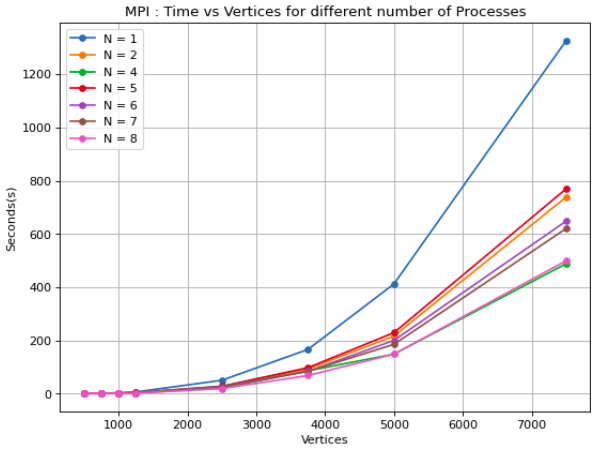

Fig. 3. MPI Runtime vs Nodes

increases to 5, after distribution across the 4 cores, the 5th process runs on one of the cores, reducing its efficiency and affecting overall performance. With more processes, however, the workload is more evenly distributed. Even though the number of processes is greater than the number of cores, the performance isn't affected as much, and it gets better. For 8 processes, the distribution is similar to that of 4, so performance is comparable. A detailed set of observations is given in Table V.

Overall, when the best performances of MPI and OpenMP are considered, MPI performs better than OpenMP.

C. CUDA

The CUDA code was executed on a Google Colab environment with CUDA-9.2. Due to the large number of nodes, the number of threads per block was controlled, and number of blocks varied depending upon the size of the graph. The performance was analyzed by varying the size of the graph and the number of threads per block. As expected, runtime increased with the number of nodes. While 1 thread per block

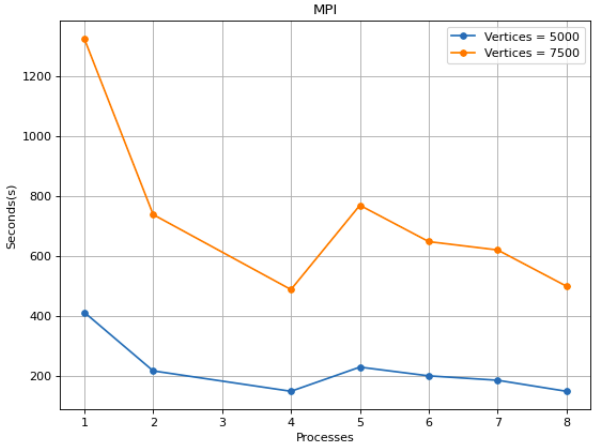

Fig. 4. MPI Runtime vs Processes

performed similar to the sequential algorithm on the Google Colab environment, it improved with more threads per block.

It was found that 64 threads per block performed best, much better than 4 threads or lesser per block. As the number of threads per block in the grid increases, the runtime also steadily, although negligibly, increases. This is due to the synchronization of threads performed before and after the main computation of distances. With more number of threads, each thread waits for all threads to be in sync before proceeding, which causes a bigger delay for a larger number of threads. This can be seen in Fig. 5 and Fig. 6.

However, CUDA performs much better than the other algorithms overall. While the sequential algorithm takes approximately 20 minutes to execute for 7500 nodes, CUDA only takes about 1 minute in the best case. This is a massive improvement, even compared to MPI and OpenMP, which take 8 and 11 minutes respectively, for the same graph size. A detailed set of observations is given in Table VI.

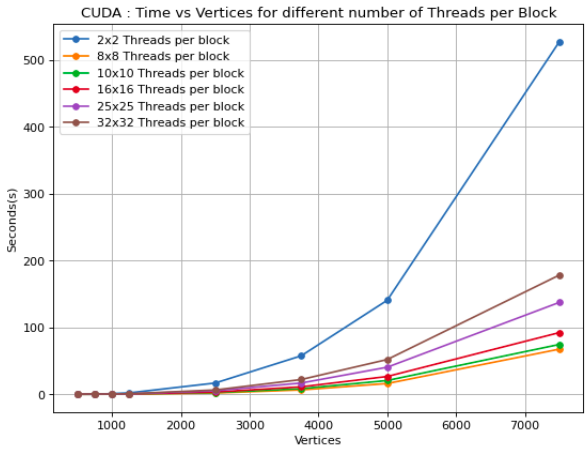

Fig. 5. CUDA Runtime vs Nodes

TABLE VI
EXECUTION TIMES FOR THE CUDA ALGORITHM WITH DIFFERENT NUMBER OF THREADS PER BLOCK

No. of threads per block	No. Of Nodes							
	500	750	1000	1250	2500	3750	5000	7500
1x1	0.84375	1.896837	3.889294	7.454307	58.9163	199.723786	488.1944741	1934.097417
2x2	0.313703	0.0683649	1.174575	2.235608	16.988455	57.694908	140.745801	527.207128
8x8	0.148674	0.216927	0.309651	0.447183	2.050511	6.681803	16.248385	67.646944
10x10	0.152255	0.228705	0.317386	0.588120	2.665412	8.502768	20.744947	74.656530
16x16	0.169377	0.271441	0.406250	0.557588	3.449512	11.285999	26.752994	92.373658
25x25	0.182580	0.332671	0.533418	0.786950	5.346313	16.979300	40.622643	137.546148
32x32	0.217515	0.711268	0.316364	0.949067	6.607430	22.205769	52.127983	178.362754

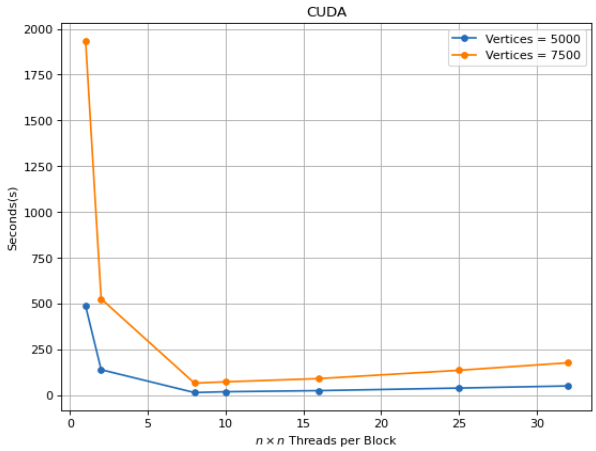

Fig. 6. CUDA Runtime vs Threads per Block

V. CONCLUSION

The sequential implementation has been parallelized in three different programming environments: OpenMP, MPI and CUDA. The results have been analyzed considering the varying number of vertices in every environment, along with varying number of threads in OpenMP, varying number of processes in MPI and varying number of threads per block in CUDA. The analysis of sequential versus parallel implementations revealed all of the parallel implementations to have much lesser execution times for varying number of vertices. However, when OpenMP is run with one thread, MPI for one process and CUDA for one thread per block, the time taken is similar to that of sequential execution as the parallelization aspect of these three environments is not being used in these scenarios.

From the experimental observations, it is clear that OpenMP performed best when *dynamic* scheduling was used, MPI when the load was equally distributed among the machine's cores (number of processes was a multiple of the number of cores) and CUDA when the number of threads per block were 8 x 8, with 9 x 9 and 10 x 10 also having comparable performance times. Overall, CUDA performed much better than the other two environments, followed by MPI and finally OpenMP. OpenMP is relatively easier to implement and thus is better suited for most general purposes. In situations involving memory intensive calculations, OpenMP struggles with efficiency. In such situations, the distributed computation with MPI works well. CUDA is best suited in cases of GPGPU computing and in using Nvidia GPUs to parallelize tasks. In the future, implementation and analysis of hybrid implementations of these parallel environments can be done to see the performance efficacy as compared to the traditional implementations.

REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[2] Azis, H., Mallongi, R. dg., Lantara, D., Salim, *Comparison of Floyd-Warshall Algorithm and Greedy Algorithm in Determining the Shortest Route*[1]

[3] Jian Ma, Ke-ping Li, Li-yan Zhang., *A Parallel Floyd-Warshall algorithm based on TBB* [2]

[4] Magzhan Kairanbay, Hajar Mat Jani, *A Review and Evaluations of Shortest Path Algorithms* [3]

[5] Dehal, R. S., Munjal, C., Ansari, A. A., Kushwaha, A. S., *GPU Computing Revolution: CUDA* [4]

INDIVIDUAL CONTRIBUTION

The work was equally divided among the three team members after studying the required material to ensure sufficient knowledge of the works to be implemented.

- Gaurang Jitendra Velingkar worked on the CUDA implementation and analysis of the results of the same.
- Rakshita Varadarajan worked on the MPI implementation and analysis of the results of the same.
- Ritvik Mahesh worked on the sequential and OpenMP version of the algorithm and the analysis of the results of the same.

Further, we divided the report into equal sections and worked on it together.