Parallelizing Floyd Warshall algorithm to find all-pair shortest paths using OpenMP

Rutvik Shah and Shiv Khattar

New York University, New York, New York
rutvik.shah@nyu.edu
shivkhattar@nyu.edu

Abstract. Finding the shortest path between all pairs of vertices in an undirected edge weighted graph is one of the most researched problem in graph theory. There are several algorithms that have been designed to solve this problem, the most popular ones being the Dijkstra's, Bellman Ford and Floyd-Warshall algorithms. In this paper, we aim to parallelize the Floyd-Warshall algorithm to find all pair shortest paths between all pairs of vertices in a graph using OpenMP. A block-based technique which divides the adjacency matrix representation of the input graph into smaller sub-matrices is employed in this paper. The parallel block-based FW algorithm executes in three different phases. We run experiments on different parameters such as - the number of vertices in the graph, the block size of sub-matrices and the number of threads running in parallel, to find out which parameters give us the maximum speedup for the parallel FW algorithm with respect to the sequential version.

Keywords: OpenMP, Floyd-Warshall algorithm, All pairs shortest path, Graph theory, APSP

1 Introduction

In graph theory, one of the most crucial problems is finding the path between two vertices or nodes in a graph, such that the sum of weights of its constituent edges is minimum. We have several real-life applications where shortest path algorithms come in handy, such as - Social Network Analysis, Navigation in Maps, Network Traffic Management etc. These problems are often modelled as graphs where the vertices represent an entity and the edges represent any form of interaction between them.

According to the final vertex and the characteristics of the path, the shortest path problem can be divided into four types: The shortest path between two vertices, the shortest path among all vertices, the K shortest path and the real-time shortest path. This paper mainly discusses the second type of shortest path problem - to find shortest path between all vertices.

The single source shortest path algorithm was conceived by Edsger W. Dijkstra in 1956 and is widely used in road networks. This algorithm however only worked on graphs where edge weights were non-negative. Later, Richard Bellman and Lester Ford published the Bellman-Ford algorithm in 1958 and 1956,

which was significantly different from the Dijkstra's algorithm, in that it could be used with graphs which contained negatively weighted edges. This was a massive breakthrough, albeit with the condition that the graph should not contain any negative weight cycles reachable from the source vertex [8]. In 1962, the Floyd-Warshall algorithm was published which could find shortest paths in a weighted graph with positive or negative edge weights which follows our main problem type - all pairs shortest path.

Since the Floyd-Warshall algorithm is one of the most widely used and versatile graph algorithm to find all pairs shortest paths between vertices, this paper aims to parallelize the algorithm using OpenMP and evaluate different configurations for which one would achieve maximum speedup. The overall asymptotic time complexity of the Floyd-Warshall algorithm is $O(N^3)$, where N is the number of vertices in the graph. Since, the FW algorithm is most suitable for dense graphs (i.e. the adjacency matrix is dense), it is important to be able to parallelize this algorithm and reduce the actual running time.

In applications such as Network Traffic Management and Vehicle Navigation, even a small improvement in the time required to find the shortest path, between two routers or city transit points, could prove to be important as the scale of the problem increases. In applications such as Social Network Analysis, a few less seconds required in finding connections or recommending friends could prove to be a significant improvement in the user-experience. And thus, by reducing the overall time required to find the all pairs shortest path and parallelizing the Floyd-Warshall algorithm, this paper could prove instrumental in making graph-based problems more efficient.

Algorithm 1 Floyd Warshall Algorithm

```
1: for k = 0 → n − 1 do
2:     for i = 0 → n − 1 do
3:         for j = 0 → n − 1 do
4:             if A[i, k] + A[k, j] > A[i, j] then
5:                 A[i, j] = A[i, k] + A[k, j]
6:             end if
7:         end for
8:     end for
9: end for
```

The general idea in the Floyd-Warshall algorithm is to - across n iterations, where n is the width and height of the adjacency matrix graph input - pick all of the vertices as intermediates in the shortest paths. It then tests if the picked vertex is in the shortest path. If it is an intermediate in the path, the distance is updated, otherwise the original distance remains unchanged.

The general idea used to parallelize the Floyd-Warshall algorithm in this paper is derived from parallelization of the Matrix-Multiplication problem. This paper models the input graph as the adjacency matrix representation, where each

element adj_{ij} in the 2-d matrix gives us the length of the shortest path from the vertex i to vertex j in the graph. The parallelization takes place by dividing the adjacency matrix into smaller blocks to calculate the all-pairs shortest path of each such smaller block, and then using the values computed within each block to update the values in other blocks.

2 Literature Survey

Parallelization of the Floyd-Warshall algorithm can be divided into two main categories: Block-based and non block-based methods. In block-based methods, the adjacency matrix is divided into smaller blocks and the solution is computed for the blocks iteratively in different phases. The order of execution and computation of solutions for the sub-matrix blocks is crucial in parallelizing the FW algorithm. For non block-based methods, different researchers have carried out row-based and column-based parallelization techniques. However, the non block-based techniques requires significant communication between individual worker threads, which leads to smaller speedup compared to block-based techniques [11]. In this section we first analyze block-based techniques followed by non-block based techniques implemented in recent years.

2.1 Block-based Methods

In [7] Katz and Kider describe a block based method where the adjacency matrix is divided into blocks as done with parallel matrix-multiplication problems. Each block is processed in one of three phases. The blocks along the diagonal of the adjacency matrix are computed in order from the top to bottom in the first phase. In the second phase the sub-matrices lying on the same row and column as the diagonal block processed in the first phase are evaluated. In the third phase all the remaining blocks are computed. This method maximizes data parallelization between the threads and minimizes communication between the sub-solutions. This method first proposed by Katz and Kider is used in other block-based research works albeit using other parallelization software. In this paper, the authors use CUDA to parallelize the algorithm on GPUs. The main benefit of the shared memory cache efficient algorithm proposed in [7] is that it generalizes to graphs whose sizes are larger than the GPU memory.

The author in [10] used the block-based parallelization method proposed by Katz and Kider, and evaluates the speedup on the Intel C++ Threading Building Blocks (TBB) library [9]. Like OpenMP, TBB library provides generic algorithms such as parallel_for, parallel_scan and parallel_reduce which prevent the programmer from having to write extensive code for their multi-threaded applications using the pthread library.

In [12], Venkataraman et. al evaluate and give proof of correctness of the block-based method described above. They evaluate their method on the Sun Ultra Enterprise 4000/5000 machine and were able to achieve a speedup between 1.6 and 1.9 for graphs of sizes 480 to 3200 vertices.

2.2 Non block-based Methods

Non block-based methods are not highly researched for parallelizing the Floyd-Warshall algorithm since they do not exploit the cache locality or data prallelism obtained from block-based methods. In [11] the authors describe a non block-based method implemented using the CUDA API. In this technique, each CUDA block is assigned an equal amount of the solution matrix's elements (512). Each block is then assigned 512 CUDA threads, thus each thread operates on one single element in the solution matrix. The block that owns a particular row k, broadcasts the row to the outer blocks. The outer blocks that receive row k then relax each entry in their own row with respect to the node k. The solution is obtained after all rows have been processed. The algorithm processes each row in parallel and thus offers significant speedup over the sequential variant. However, there is a limit of 512 threads because of the limitation of CUDA. This algorithm also has poor cache performance because it does not take into account the expense of individual read/write operations in memory. It is also limited by the GPUs memory limit and cannot process the APSP for graphs which exceed the memory of the GPU.

Based on these research works, it is evident that the block-based parallelization of the Floyd-Warshall algorithm offers far superior performance improvement over the non block-based techniques. Moreover, it is also important to experiment with the block-based method proposed in [7] using the OpenMP library. Most of these research works describe the parallelization method followed and their comparison with other parallel implementations of other All Pairs Shortest Path algorithms such as Dijkstra and Bellman-Ford algorithm. However, these do not explore the optimal block sizes for the block-based parallel FW algorithm which would result in the maximum speedup. Nor do these papers discuss the effect that increasing or decreasing the number of threads will have on the speedup. In the following sections we outline a detailed explanation of our proposed idea and the results and analysis of parallelizing the Block-based Floyd-Warshall algorithm.

3 Proposed Idea

3.1 Serial Floyd-Warshall Algorithm

The Floyd-Warshall algorithm computes the all pairs shortest paths between each pair of vertices in a given graph G by iteratively improving on the current shortest path between two vertices until an optimal solution has been reached [3]. For a particular pair of vertices, the FW algorithm determines the shortest path between them by adding the weights of all intermediate edges. The algorithm has been outlined in Algorithm 1 above.

The first outer loop iterates over all vertices one by one and adds the vertices to the set of vertices considered as intermediate. The second loop determines the source vertex and the third inner-most loop picks each vertex as the destination for the previously picked source vertex. If the vertex k is intermediate to the

shortest path between the vertices i and j, then the weight of the path between i and j gets updated, i.e. the edge (i, j) is relaxed with respect to vertex k.

3.2 Parallel Floyd-Warshall Algorithm

By looking at the psuedo code in Algorithm 1, it is observed that after the k^{th} iteration of the outer for loop, the solution matrix contains the shortest path that only uses the first k vertices as intermediaries [3].

The FW algorithm depends on the invariant that following k iterations of the algorithm, each edge is relaxed with respect to the first k vertices since these have already been computed to be the intermediaries in the outer loop [4].

The FW algorithm depends on successive relaxation on an increasing set of intermediate vertices as the algorithm progresses, and thus the order in which the three nested loops can be run is constrained. While dividing the adjacency matrix into blocks, after the k^{th} iteration of the outer loop is run, the result matrix must contain shortest paths which only use the first k vertices as intermediate vertices. Thus we cannot use the divide and conquer approach used in matrix multiplication [5]. Since the values for every element of the adjacency matrix are modified by relaxing the values with respect to every other element in the adjacency matrix, we cannot partition the data.

To avoid this issue, we follow the algorithm proposed by Katz and Kider [7] which aims to minimize communication between individual blocks by ordering the processing of sub-matrices in a particular way.

The adjacency matrix is divided into sub-matrices of size $B \times B$ where B is called the 'Blocking Factor'. Thus, we obtain $(\frac{N}{B})^2$ blocks. where N is the number of vertices in our graph. The modified Floyd-Warshall algorithm can now perform B iterations to obtain the APSP for the vertices in the blocks of size B [7]. A primary block is selected for each stage of the algorithm.

The primary blocks are selected along the diagonal of the adjacency matrix, starting with block $[0, 0]$. The primary block contains elements from $[p_{start}, p_{start}]$ to $[p_{end}, p_{end}]$ of the adjacency matrix. p_{start} and p_{end} are given by:

$$p_{start} = \frac{(primary\ block\ number) \cdot N}{number\ of\ primary\ blocks'} \tag{1}$$

$$p_{end} = p_{start} + \frac{N}{number\ of\ primary\ blocks} - 1, \tag{2}$$

where N is the number of vertices in the graph, and therefore the length and breadth of the adjacency matrix.

Each iteration of the algorithm is divided into three phases, as follows:

In the first phase the APSP values for the primary block are computed, i.e. the APSP between all vertices within the primary block on the diagonal are computed. This phase cannot be parallelized since the computed values are not independent of each other.

The second phase consists of computing values for the blocks that are dependent on the primary block which has already been computed in the first phase.

By looking at the FW algorithm closely, we can conclude that all blocks on the same row and column as the primary block are dependent on the primary block. That is to say, the values of elements in the blocks on the same row and column as the primary block depend only on the values within that block (intermediate vertex within the same block) and the values in the primary block (intermediate vertex within the primary block). Thus, we can parallelize this phase and assign a different thread for each separate dependent block, since each dependent block values can be calculate independently based on the values within itself and in the pre-computed primary block on the diagonal.

The third phase is used to compute the values of all the remaining blocks. For each block in the third phase, the block values depend on two blocks: the block with the row of the current block and the column of the primary block, and the block with the column of the current block and row of the primary block. Since both of these blocks have been computed in phase two, we can compute these independently for each block in the third phase. Thus, we can carry out the third phase in parallel for each of the remaining blocks. Figure 1 displays

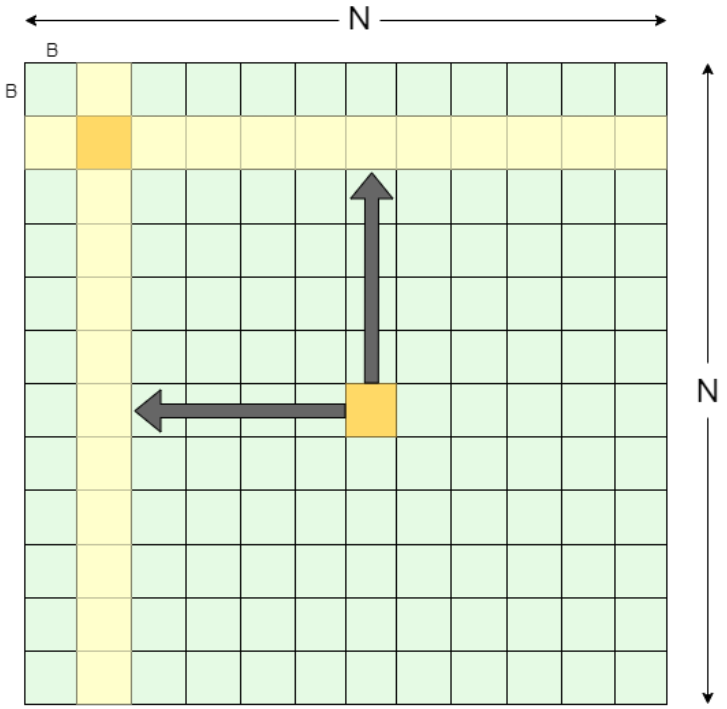

Fig. 1. Phase 3 of Block-based Parallel Floyd-Warshall algorithm

the third phase of the algorithm.

Algorithm 2 Floyd Warshall Per Block

```
1: function FW(A, startBlock, endBlock, blockSize)
2:     for mid = 0 → blockSize − 1 do
3:         for start = 0 → blockSize − 1 do
4:             for end = 0 → blockSize − 1 do
5:                 if startBlock[start, mid] + endBlock[mid, end] > A[start, end] then
6:                     A[start, end] = startBlock[start, mid] + endBlock[mid, end]
7:                 end if
8:             end for
9:         end for
10:    end for
11: end function
```

After the three phases are completed, the elements in the APSP solution matrix are calculated for the values ranging from p_{start} to p_{end}. In the next phase for the next value of k for the outer loop, the primary block is moved down along the diagonal and the three phases are repeated. The algorithm ends when the primary block has reached the end of the diagonal, i.e. $p_{end} = N - 1$, and the APSP has been found for the input graph G. Thus, we realize there is

Algorithm 3 Blocked Floyd Warshall

```
1: function BLOCKED_FW(A, n, blockSize)
2:     blocks = n/blockSize
3:     for k = 0 → blocks − 1 do
4:         fw(A[k, k], A[k, k], A[k, k], blockSize)
5:         for j = 0 → blocks − 1 do
6:             if j ≠ k then
7:                 fw(A[k, j], A[k, k], A[k, j], blockSize)
8:             end if
9:         end for
10:        for i = 0 → blocks − 1 do
11:            if i ≠ k then
12:                fw(A[i, k], A[i, k], A[k, k], blockSize)
13:                for j = 0 → blocks − 1 do
14:                    if j ≠ k then
15:                        fw(A[i, j], A[i, k], A[k, j], blockSize)
16:                    end if
17:                end for
18:            end if
19:        end for
20:    end for
21: end function
```

one major dependency while making the FW algorithm parallel. The iterations of the outer-most loop for the FW algorithm cannot be carried out independently

since the $k + 1^{th}$ iteration depends on the previous k iterations. Moreover, the three phases - the dependent, the partially dependent and independent phases, must be carried out in order to obtain the correct solution. Besides these two dependencies of the algorithm, all of the inner loops can be processed in parallel.

The blocked Floyd-Warshall algorithm increases the locality of the program since the data required for each phase is confined to smaller blocks. Since the FW algorithm is primarily memory bound, taking advantage of the locality of the blocks is extremely important. Besides the dependencies between iterations and phases, the single mathematical operation is taking the minimum of two loaded values and potentially storing it into a third memory location. Getting quick memory reads and writes generates the most benefit for this application.

Since the inner workings of the phases can be done in parallel, OpenMP is a strong method of parallelism, as iterations of the loops can be split into chunks amongst threads.

The pseudo code for the parallel blocked Floyd-Warshall algorithm is given in algorithm 2 and 3.

4 Experimental Setup

We have implemented the sequential and parallel versions of Floyd Warshall algorithm and we've developed various experiments to test the speedup we received by using the parallel version which using blocking. We ran our tests on the crunchy3 machine on CIMS[2]. This machine contains four AMD CPUs which share one main memory module(128GB). Each CPU contains 8 cores(AMD Opteron 6136 @ 2.4 GHz). Each core has 512KB 2-way set associative instruction caches and 512KB 2-way set associative data caches at Level 1, 4MB 16-way set associative exclusive caches at level 2 and 12MB of shared exclusive cache at level 3, with a cache latency of 3 at Level 1.[1] We used C programming language and OpenMP to write our code. We used gcc-9.2 to compile our code.

We generated graphs using a random selection based on the expected density of edges. The distance between vertices was selected uniformly randomly between 1 and 100. We use the same randomly generated graph for the sequential version and parallel version with different thread counts when we compare the speedup. We ran each experiment 5 times and removed the minimum and maximum outliers and took an average of the remaining execution times to keep things fair. It was very important to validate the correctness of the results produced by the parallel version so we validate the results produced by each run of the parallel version and compare it with the results from the sequential version. For measuring the timing of the experiments, we used omp_get_wtime() to capture the start and end time of each experiment. The timing provided in the results contains the time spent in performing the computation and not generating the random graph. Finally, we automated these experiments using various arguments and options that can be provided while running the experiment. Different variables like number of vertices, block size, thread count and probability that an edge

will exist can be manipulated using these arguments. The experiments we ran are mentioned below:

1. We wanted to see how the parallel version performs when we increase the number of threads keeping the vertices and the edges same. We generated a random graph with n = 256, 512, 1024 vertices and a probability of edge between vertices fixed at 0.5. This resulted in a n x n matrix where *[i,j]* indicates the difference between the vertex i and i. We used *INT_MAX/2* as ∞ where the vertices were not connected. We scaled the number of threads exponentially from 1 to 256 and recorded the execution time and the speedup.

2. We wanted to test if the speedup increases when the graph grows. When the data is very small, more time is spent in creating the data rather than doing the computation itself. We generated random graphs with n = 64, 128, 256, 512, 1024 and 2048 vertices and a probability of edge between vertices fixed at 0, 0.5 and 1. For this experiment, we fixed the block size, $b = \sqrt{n}$ and number of threads, $t = min(n/b, 32)$. We recorded the time taken by the sequential version and parallel version and calculated the speedup for each graph.

3. Another experiment that we ran was to see the impact of block size on the speedup. Since the block size eventually dictates the number of iterations that are run in parallel, we wanted to test the impact and find out what the optimal block size is. For this experiment, we generated a random graphs with n = 256, 512 and 1024 vertices and an edge probability of 0.5 and fixed the number of threads, $t = min(n/b, 32)$. We then ran the sequential version and parallel version using block sizes starting from 4 and increasing exponentially till 128.

5 Experiments and Analysis

5.1 Scaling the number of threads

The first experiment that was conducted by us was to collect speedup as the number of threads increase keeping the number of vertices, n = 256, 512 and 1024. We fixed the edge probability between two vertices at 0.5.

As we can see in Figure 2, we can see an increase in the speedup as the number of threads increase. This is because, when the number of threads increase, the parallel iterations are divided amongst the threads and each thread can run parallel and do the work independently and finish the work.

There were three tests that we ran in this experiment.

1. **Number of vertices,** $n = 256$ **and block size,** $b = 16$
   In this test, we see that there is a spike in speedup when the number of threads, $t = 16$. The algorithm is such that the for loop runs for n/b iterations, and when $t = 16$, each iteration is divided amongst each thread

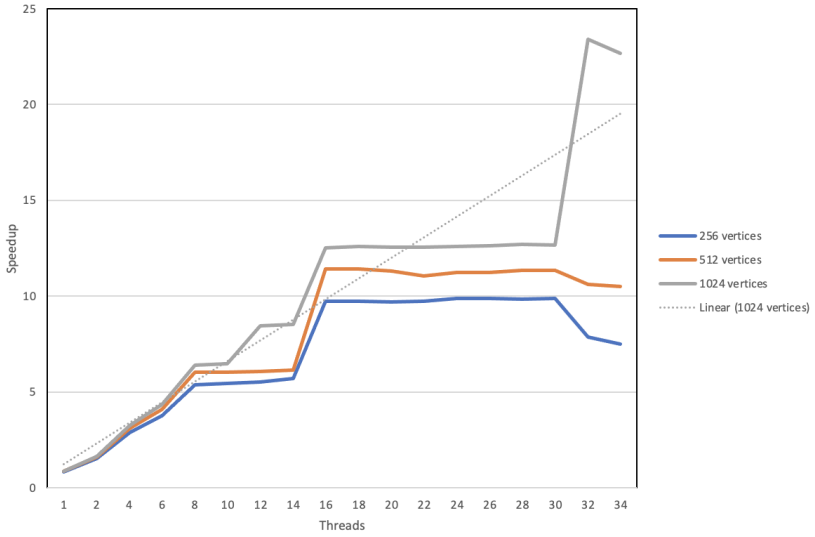

Fig. 2. Speedup vs Number of threads

equally, thus, the code runs at it's full potential when number of threads, t = 16. This is the most optimal case, and the highest speedup is noticed at this case. As the number of threads increase after this, we see that is no further improvement in the process and when t = 32, we see a decline in the speedup. When the number of threads are way more than the number of iterations, we see a decline since more time is spent in actually creating the threads and the costs for synchronization, OS costs, etc are more than the actual computation. Each thread has very little work at this stage.

2. **Number of vertices,** $n = 512$ **and block size,** $b = 32$
   In this test, we see a spike in speedup when the number of threads, $t = 16$. This is because at $t = 16$, the number of threads is equal to the number of iterations being run by the for loop. A decline is seen at $t = 32$ because of the same reasons mentioned in test 1.

3. **Number of vertices,** $n = 1024$ **and block size,** $b = 32$
   A spike in speedup is noticed at $t = 16$ and a bigger spike is noticed at $t = 32$. This is because when $n = 1024$, there are 32 iterations that need to be run. At this point, since the machine contains 32 cores, the maximum parallelism is achieved as each iteration is running in parallel and we're using the full capability of the CPU.

Figure 3 shows how execution time(in secs) decreases as the number of threads increase when the number of vertices, $n = 1024$ and block size, $b = 32$. As we can notice, the block-based parallel Floyd Warshall algorithm performs

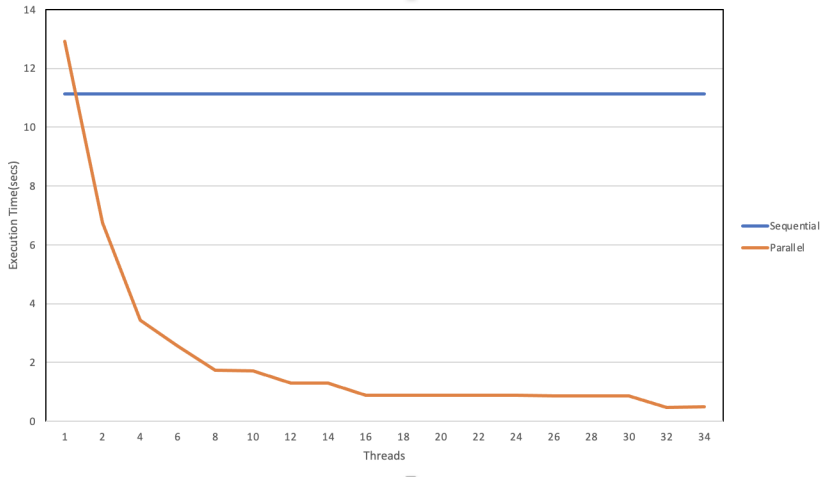

Fig. 3. Execution Time(in secs) vs Number of Threads

worse when the number of threads, $t = 1$. This is because we're spending more time in creating the blocks, and without any actual parallelism, this algorithm will perform worse than the sequential version of the Floyd Warshall algorithm. We can see a decline in the execution time as the number of threads increase and the best performance is seen at $t = 32$ because we're utilizing the full potential of the machine, and each threads gets equal work.

5.2 Scaling the number of vertices in the graph

In this experiment, we generated random graphs with number of vertices, $n = 64, 128, 256, 512, 1024, 2048$ and a probability of edge between vertices fixed at $0, 0.5, 1$. The block size for each test, $b = \sqrt{n}$ and number of threads, $t = min(n/b, 32)$. We saw the maximum speedup at these values and decided to use them for this experiment.

As we see in Figure 4, there is an increase in the speedup as the number of vertices increase. This is because, when the number of vertices increase, the sequential version will take a longer time to execute the computation for all the iterations, whereas in the case of parallel version, the iterations can be divided in the threads and thus there is a speedup.

We can see that there is bigger increase at $n = 256$ and $n = 1024$. This is because, at these values, the value for $b = \sqrt{n}$ and $t = n/b$ are equal, and thus the work is divided amongst the threads efficiently.

1. When the block size is greater than the number of threads, $b > t$, then each thread is doing a lot of work, and hence we're not using the system efficiently.
2. When the block size is smaller than the number of threads, $b < t$, then each thread is doing too little work and there's overhead for creating more threads.

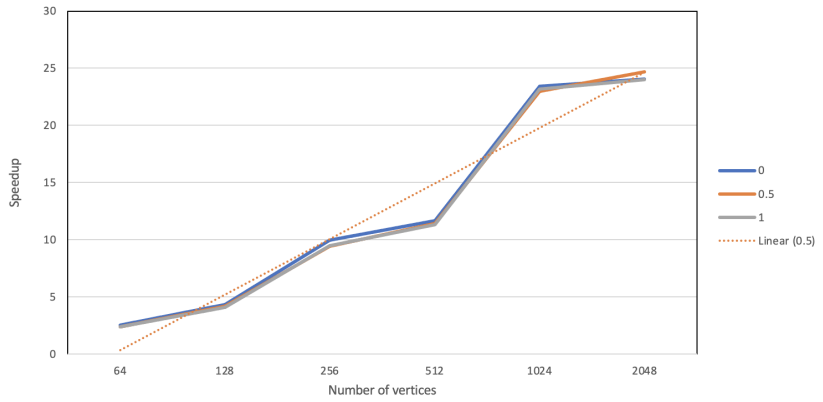

Fig. 4. Speedup vs Number of Vertices

3. When the block size is equal to the number of threads, $b == t$, then each thread is doing equal work and we're creating the most efficient number of threads. In this case, each thread is neither doing too much work, nor too little work.

Another thing to note from 4 is that the number of edges don't impact the speedup. This is because, in each case, the number of iterations to be run are the same. This is unlike Johnson's algorithm[6], which is another algorithm that is used to find all-pair shortest paths, the algorithm only compares the vertices that are actually connected. In Floyd Warshall algorithm, even if the vertices are not actually connected, the comparisons are still run on those vertices the distance between those vertices continues to be ∞.

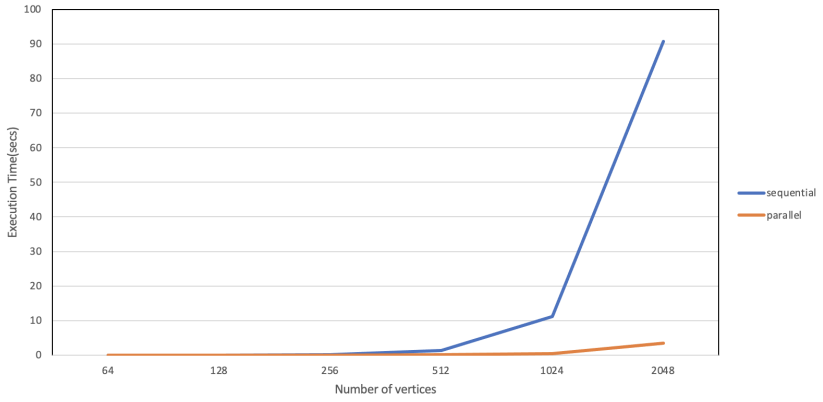

Fig. 5. Execution Time(in secs) vs Number of Vertices

Figure 5 shows how the execution time changes with the number of vertices, keeping the block size, $b = \sqrt{n}$, number of threads, $t = min(n/b, 32)$ and the edge probability, $e = 0.5$. As expected, the sequential time increases when the number of vertices increase. As the number of vertices increase, there is a quadratic increase in the the size of distance matrix and thus, there is a quadratic increase in the execution time when the algorithm is run sequentially. When the parallel version of the FW algorithm is run, we notice that the computations are divided efficiently amongst the cores and thus the execution time doesn't increase.

5.3 Scaling the number of blocks

In this experiment, we wanted to understand the impact of block size on the overall performance of the parallel version of FW algorithm. We also aimed to find the optimal size of the block which produces the best speedup. For this experiment, we fixed the number of vertices, $n = 256, 512, 1024$, the probability of connection between edges, $e = 0.5$ and number of threads, $t = min(n/b, 32)$. We restricted the maximum number of threads to be 32 since our machine has 32 cores and that is the maximum parallelism we can achieve. If we allow more threads, that will reduce the speedup, since each thread will do very little work, and hence it will not produce the correct results. We ran the sequential and parallel versions of FW on block sizes, $b = 4, 8, 16, 32, 64, 128$.

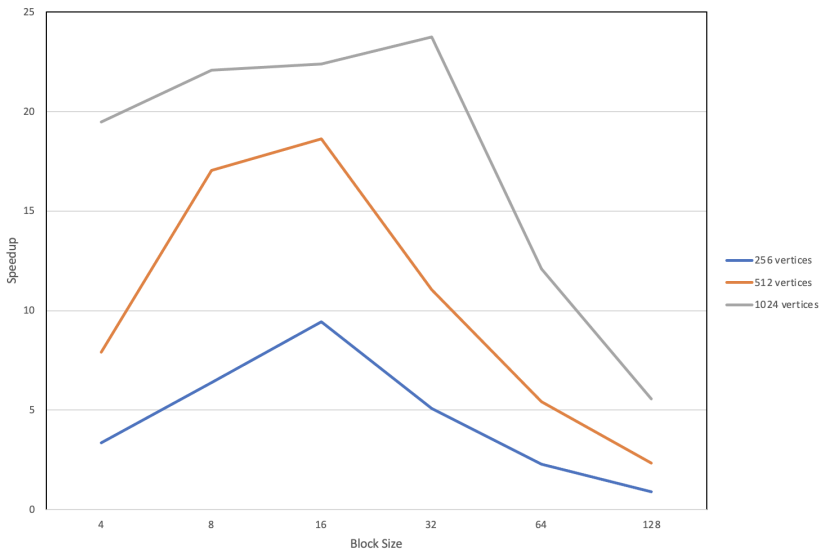

Fig. 6. Speedup vs Block Size

Figure 6 shows the change in speedup with the change in block size. As we see from the figure, we can easily divide the change in speedup into three sections.

1. **Increase in Speedup**
   We notice for the number of vertices, $n = 256, 512, 1024$, we see an increase of speedup in the first phase of increase in the block size. When the block size is too small, number of threads, $t = n/b$ is too big and each thread is performing very less work and the overhead to create the threads is more. Therefore, when we increase the block size, then the work assigned to each thread increases, and the number of threads decrease, resulting in an increase of speedup.

2. **Maximum Speedup**
   After some time, as the speedup keeps increasing with the block size, we reach a point where the speedup is the maximum before it starts decreasing again. This point, as we can see from the graph in Figure 6 is 16 for 256 vertices and 32 for 1024 vertices. Thus, the maximum speedup is seen when the block size, $b = \sqrt{n}$. This is because, when $b = \sqrt{n}$, the number of threads, $t == b$. And as we see from Algorithm 3, we see that the number of iterations being run are n/b and in this case that is equal to the number of threads. Thus, in this case, we're using the machine to it's full capacity and each thread is doing equal work and we see the maximum speedup.

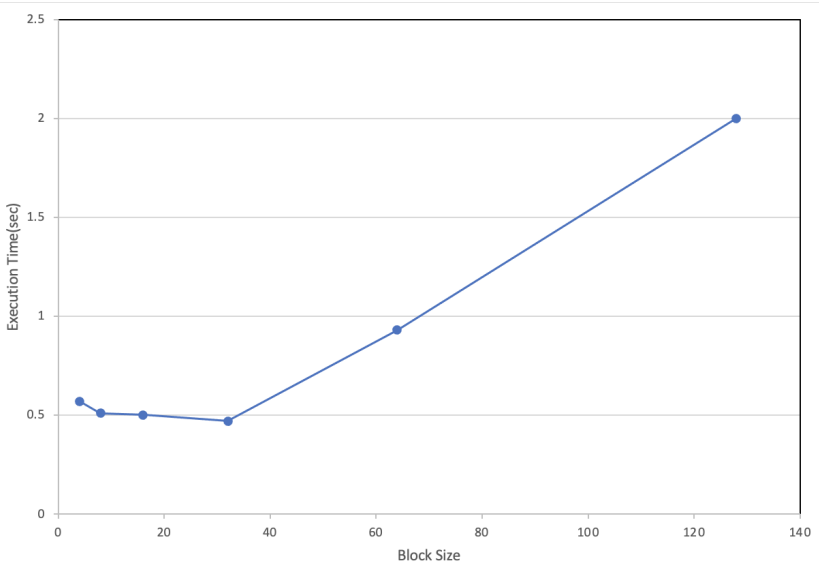

Fig. 7. Execution Time(in secs) vs Block Size

3. **Decrease in Speedup**
   If we keep increasing the block size after $b = \sqrt{n}$, we see a decline in the

speedup for all the cases. This is because, now the work being assigned to each thread is a lot, and the number of threads, $t = n/b$ decrease. Thus, we're assigning a lot of work to each thread, and thus we're not parallelizing optimally, resulting in a decrease of performance.

Figure 7 shows the comparison of execution time with the block size when the number of vertices, $n = 1024$. The execution time decreases until block size, $b = \sqrt{n}$ and then increases again.

From these results, we can conclude that the optimal size of a block is $\sqrt{n}$, as it produces the most efficient parallelism.

6 Conclusion

In this paper, we see how the Floyd Warshall algorithm to find all-pair shortest paths can be parallelized. The sequential Floyd Warshall algorithm can not be parallelized directly, as each iteration k of the outer loop depends on the previous iteration, $k - 1$'s results and when the loop is parallelized directly, it produces non-deterministic results. We saw the block-based approach to parallelize the Floyd Warshall algorithm. We break the distance matrix into smaller blocks with block size b. All the blocks are not completely independent and thus we must run the algorithm in three phases. In each phase, we consider that there path goes through k^{th} diagonal block. the In the first phase, the dependant phase, the k^{th} diagonal block is processed. This phase must be done serially. In the second phase, the partially dependent phase, blocks in the k^{th} row and k^{th} column are processed. And in the third phase, the independent phase, all the remaining blocks are processed.

We ran various experiments after developing this algorithm. We saw that the speedup increases when the number of threads increase, until the point where the number of threads, $t = n/b$, where n is the number of vertices and b is the size of the block. The speedup decreases after this because the overhead of creating the threads is more than the actual computation done by the threads. We saw that the speedup increases when the graph grows, i.e., number of vertices of the graph increase. This is because when the number of vertices increase, the execution time of the sequential version increases, whereas the parallel version still performs well. We also saw that the number of edges between vertices doesn't cause any change in the overall speedup since we compare the distance even if they are ∞. In the third experiment we found out the optimal size of a block, $b = \sqrt{n}$. When the block size is $\sqrt{n}$, we achieve the maximum parallelism, since neither is each thread doing too much work, nor is each thread doing very less work. The optimal number of threads for all cases is, $t = n/b$, since there are n/b iterations for each loop.

Thus, with these experiments we find that the number of vertices, the number of threads and block size have an overall impact on the performance improvement of the parallel version of Floyd Warshall algorithm.

References

1. *AMD Opteron 6136 specifications.* https://www.cpu-upgrade.com/CPUs/AMD/Opteron_6100_series/6136.html.
2. *CIMS Compute Servers.* https://cims.nyu.edu/webapps/content/systems/resources/computeservers.
3. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* 2009.
4. Daniel P Dougherty, Eric A Stahlberg, and Wolfgang Sadee. Network analysis using transitive closure: New methods for exploring networks. *Journal of Statistical Computation and Simulation*, 76(6):539–551, 2006.
5. Steven Huss-Lederman, Elaine M Jacobson, Jeremy R Johnson, Anna Tsao, and Thomas Turnbull. Strassen's algorithm for matrix multiplication: Modeling, analysis, and implementation. In *In Proceedings of Supercomputing*, volume 96, pages 9–6. Citeseer, 1996.
6. Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, January 1977.
7. Gary J Katz and Joseph T Kider. All-pairs shortest-paths for large graphs on the gpu. 2008.
8. Tianrui Li, Luole Qi, and Da Ruan. An efficient algorithm for the single-source shortest path problem in graph theory. In S Li, T Li, and Da Ruan, editors, *2008 3rd International conference on intelligent system and knowledge engineering, vols 1 and 2*, pages 152–157. IEEE, 2008.
9. James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.
10. Peiyi Tang. Rapid development of parallel blocked all-pairs shortest paths code for multi-core computers. In *IEEE SOUTHEASTCON 2014*, pages 1–7, 2014.
11. Edvin Teskeredžić, Kenan Karahodžić, and Novica Nosović. Comparison of the non-blocked and blocked floyd-warshall algorithm with regard to speedup and energy saving on an embedded gpu. In *2020 19th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–5, 2020.
12. Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *ACM J. Exp. Algorithmics*, 8:2.2–es, December 2004.