FACULTY OF ENGINEERING

Parallel and Distributed Processing

Project1

Floyd Warshall algorithm using pthread

Name	Sama Samara
ID	202111863
Section	1

- ## Introduction

Algorithm choice: Floyd Warshall

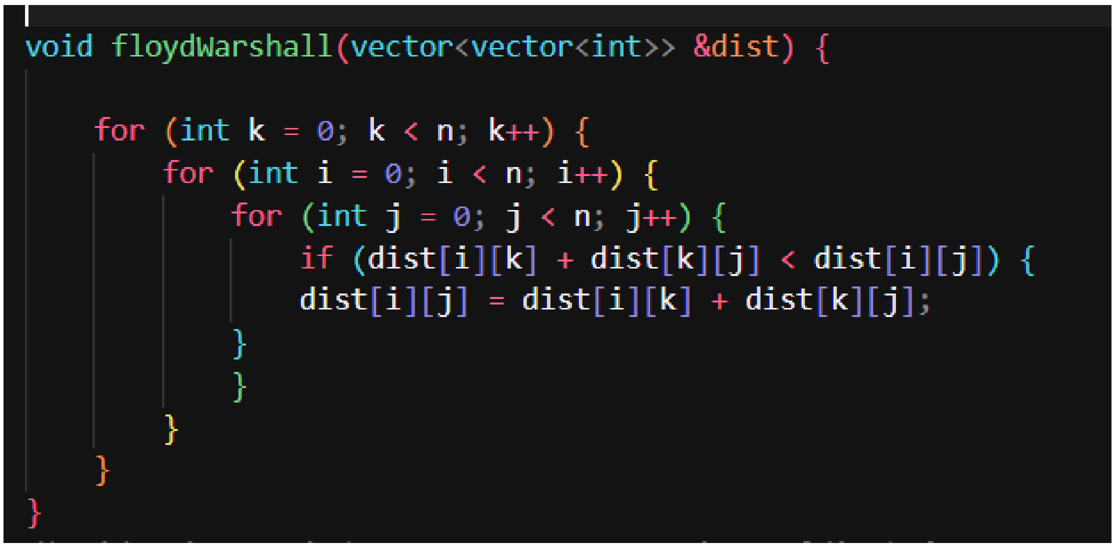
Floyd warshall algorithm is an algorithm used to find the shortest path among all pairs of vertices in a weighted graph. This algorithm can be used for both directed and undirected weighted graphs with negative or positive weighted edge, but not with negative cycles.

The algorithm works by considering each vertex as an intermediate point in the path between any two nodes. It updates the distance matrix by checking if going through the intermediate point (k) offers a shorter path from (i) to (j) than the original direct path.

This algorithm is highly parallelizable, since the updates done on each pair (i)(j) are independent within the same (k).

- ## Sequential Implementation

To implement this algorithm sequentially we need 3 nested loops, as shown:



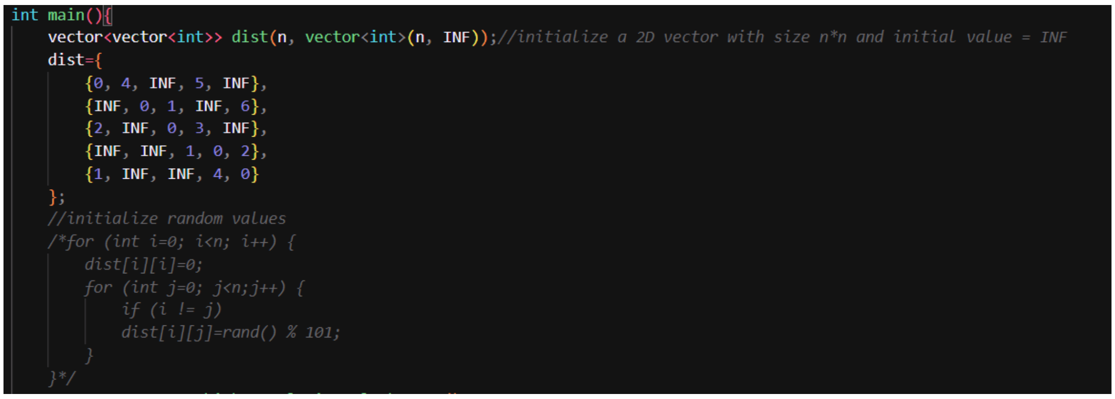
```
void floydWarshall(vector<vector<int>> &dist) {

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
            }
        }
    }
}
```

The first loop is used to move among the intermediate vertices. The second loop is used to go through the rows one by one, which in this case, each (i) represents the source. Meanwhile, each (j) represents the destination and the inner loop is used to go through the matrix column by column.

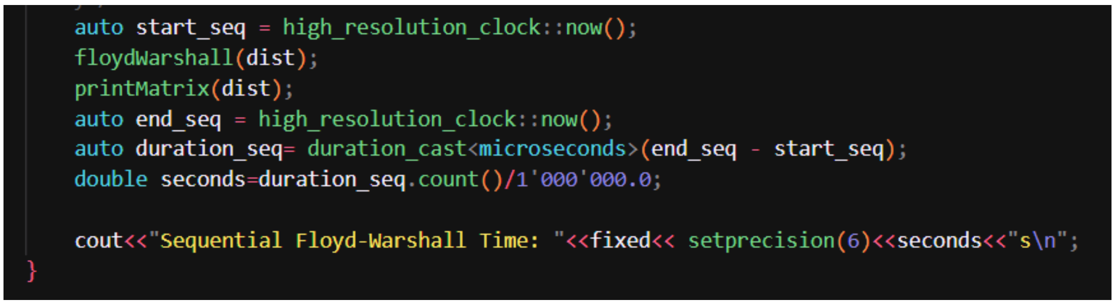
Each time the loop is executed, it checks if the distance between (i) and (j) is further than the distance from (i) to (k) and from (k) to (j). If it is, it will be updated in place.

The sequential function takes a vector with size n*n as an input and it edits it directly since it's sent by reference, not by value.



```cpp
int main(){
    vector<vector<int>> dist(n, vector<int>(n, INF));//initialize a 2D vector with size n*n and initial value = INF
    dist={
        {0, 4, INF, 5, INF},
        {INF, 0, 1, INF, 6},
        {2, INF, 0, 3, INF},
        {INF, INF, 1, 0, 2},
        {1, INF, INF, 4, 0}
    };
    //initialize random values
    /*for (int i=0; i<n; i++) {
        dist[i][i]=0;
        for (int j=0; j<n;j++) {
            if (i != j)
            dist[i][j]=rand() % 101;
        }
    }*/
```

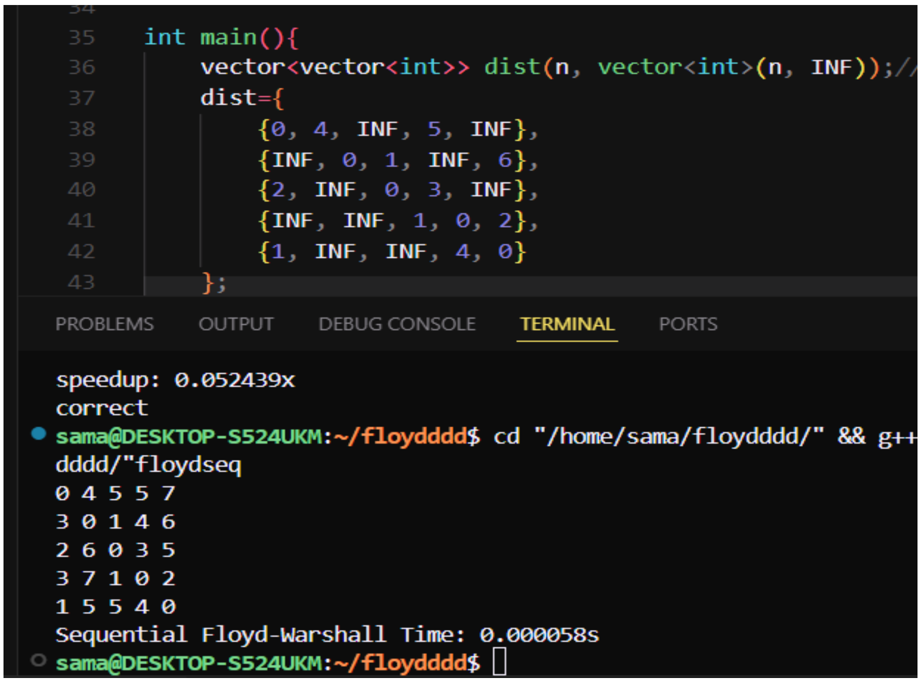
In this code implementation, I tried both values with pre-known answers to check for correctness, and random generated variables to study the change of execution time when changing the number of the vertices (n).



```cpp
    auto start_seq = high_resolution_clock::now();
    floydWarshall(dist);
    printMatrix(dist);
    auto end_seq = high_resolution_clock::now();
    auto duration_seq= duration_cast<microseconds>(end_seq - start_seq);
    double seconds=duration_seq.count()/1'000'000.0;

    cout<<"Sequential Floyd-Warshall Time: "<<fixed<< setprecision(6)<<seconds<<"s\n";
}
```

The time is measured using std::chrono::high_resolution_clock from <chrono> library to record the start and end time of the execution, along with <iomanip> library where I used the manipulators fixed and setprecision(int n) to display the time with specific amount of digits.



```cpp
int main(){
    vector<vector<int>> dist(n, vector<int>(n, INF));//
    dist={
        {0, 4, INF, 5, INF},
        {INF, 0, 1, INF, 6},
        {2, INF, 0, 3, INF},
        {INF, INF, 1, 0, 2},
        {1, INF, INF, 4, 0}
    };
```

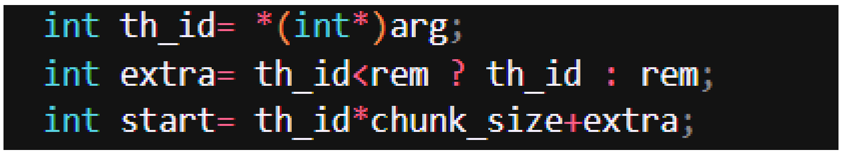
```
speedup: 0.052439x
correct
sama@DESKTOP-S524UKM:~/floydddd$ cd "/home/sama/floydddd/" && g++
dddd/"floydseq
0 4 5 5 7
3 0 1 4 6
2 6 0 3 5
3 7 1 0 2
1 5 5 4 0
Sequential Floyd-Warshall Time: 0.000058s
sama@DESKTOP-S524UKM:~/floydddd$
```

Sequential code exeution

• Parallelization Strategy

To parallelize this algorithm, we can divide the outer loop (over rows i) among multiple threads for each k.

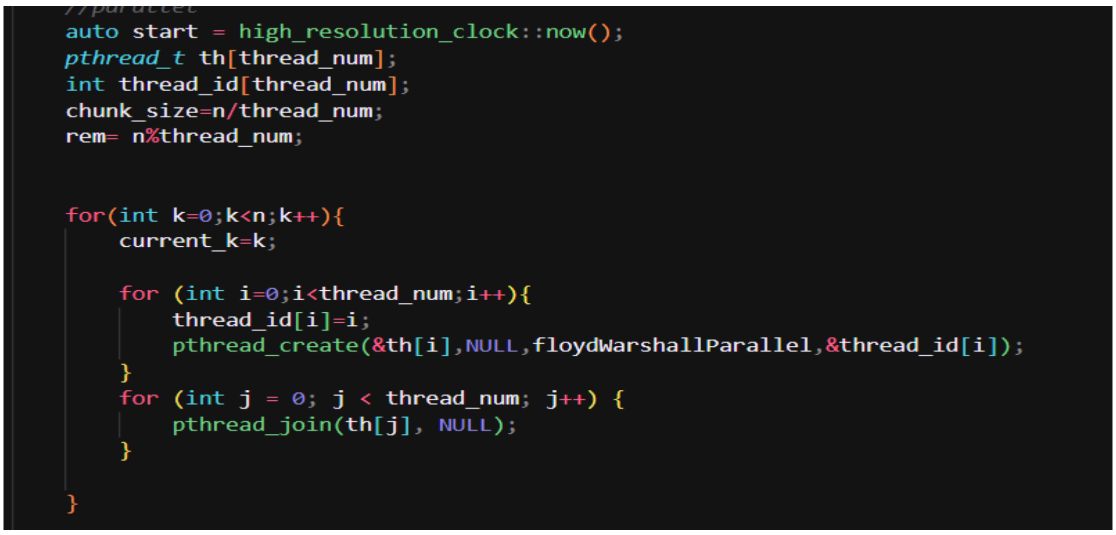
Each thread is responsible for a chunk of rows. To balance the load, the remaining rows of the division are distributed among the first threads.



```
int th_id= *(int*)arg;
int extra= th_id<rem ? th_id : rem;
int start= th_id*chunk_size+extra;
```

This part is responsible to distribute the load among the threads even when the number of rows is indivisible by the number of threads.

The outer loop (k) stays sequential, only the inner loops over (i) and (j) are parallelized. Each time the outer loop runs, new threads are created to process different chunks in parallel. After all threads complete their work for the current (k), the next iteration proceeds.



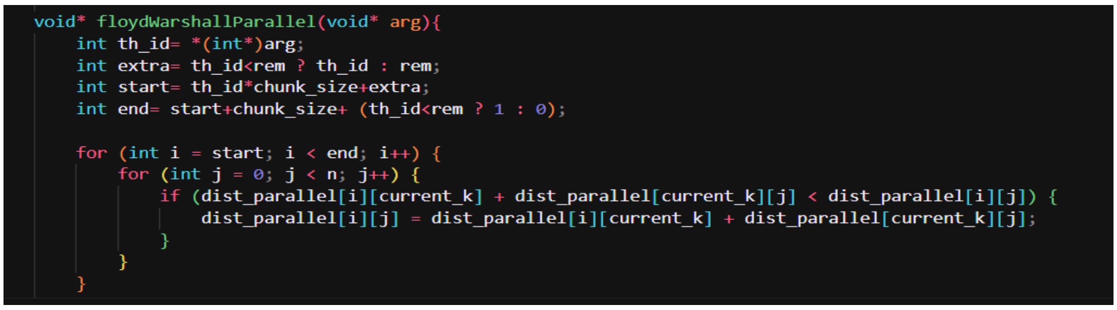
```
//parallel
auto start = high_resolution_clock::now();
pthread_t th[thread_num];
int thread_id[thread_num];
chunk_size=n/thread_num;
rem= n%thread_num;


for(int k=0;k<n;k++){
    current_k=k;

    for (int i=0;i<thread_num;i++){
        thread_id[i]=i;
        pthread_create(&th[i],NULL,floydWarshallParallel,&thread_id[i]);
    }
    for (int j = 0; j < thread_num; j++) {
        pthread_join(th[j], NULL);
    }

}
```

The sequential part inside main

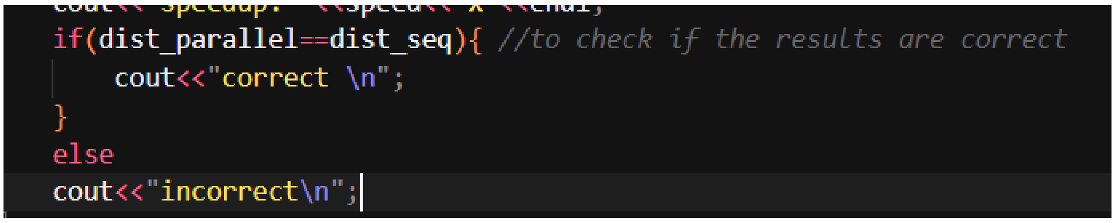


```
void* floydWarshallParallel(void* arg){
    int th_id= *(int*)arg;
    int extra= th_id<rem ? th_id : rem;
    int start= th_id*chunk_size+extra;
    int end= start+chunk_size+ (th_id<rem ? 1 : 0);

    for (int i = start; i < end; i++) {
        for (int j = 0; j < n; j++) {
            if (dist_parallel[i][current_k] + dist_parallel[current_k][j] < dist_parallel[i][j]) {
                dist_parallel[i][j] = dist_parallel[i][current_k] + dist_parallel[current_k][j];
            }
        }
    }
```

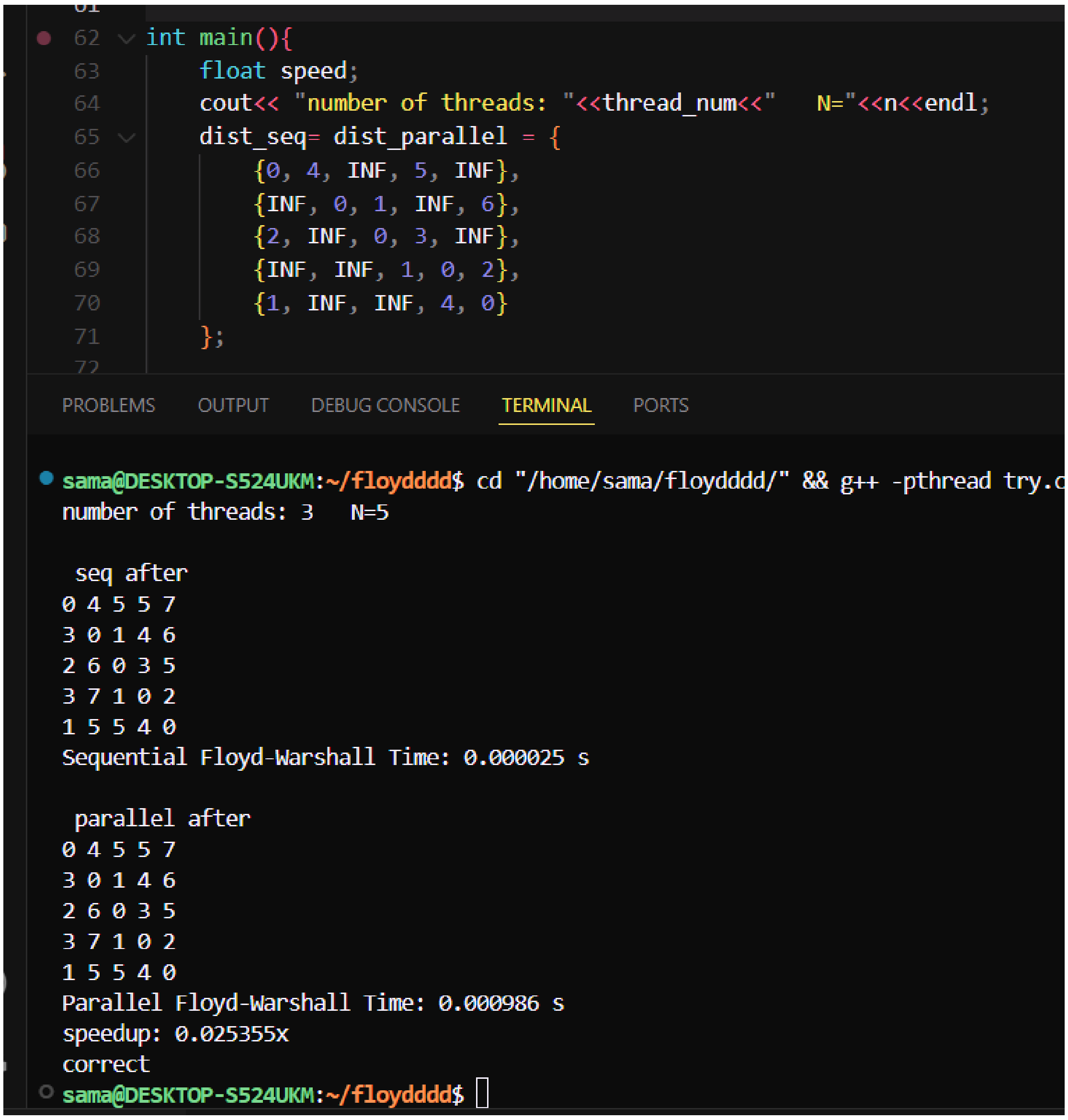
Parallel function executed by threads

To check for correctness, I combined the parallel and sequential version in one code, and compared the results.



```
    if(dist_parallel==dist_seq){ //to check if the results are correct
        cout<<"correct \n";
    }
    else
    cout<<"incorrect\n";
```

results comparison



```
int main(){
    float speed;
    cout<< "number of threads: "<<thread_num<<"    N="<<n<<endl;
    dist_seq= dist_parallel = {
        {0, 4, INF, 5, INF},
        {INF, 0, 1, INF, 6},
        {2, INF, 0, 3, INF},
        {INF, INF, 1, 0, 2},
        {1, INF, INF, 4, 0}
    };
```

```
sama@DESKTOP-S524UKM:~/floydddd$ cd "/home/sama/floydddd/" && g++ -pthread try.c
number of threads: 3    N=5

 seq after
0 4 5 5 7
3 0 1 4 6
2 6 0 3 5
3 7 1 0 2
1 5 5 4 0
Sequential Floyd-Warshall Time: 0.000025 s

 parallel after
0 4 5 5 7
3 0 1 4 6
2 6 0 3 5
3 7 1 0 2
1 5 5 4 0
Parallel Floyd-Warshall Time: 0.000986 s
speedup: 0.025355x
correct
sama@DESKTOP-S524UKM:~/floydddd$
```

Parallel and sequential results

- ## Experiments

Hardware specs:

- CPU: AMD Ryzen 5 5625U.
- Cores: 6 cores, 12 threads.
- OS: Windows 11 pro, version 24H2/ Linux WSL.

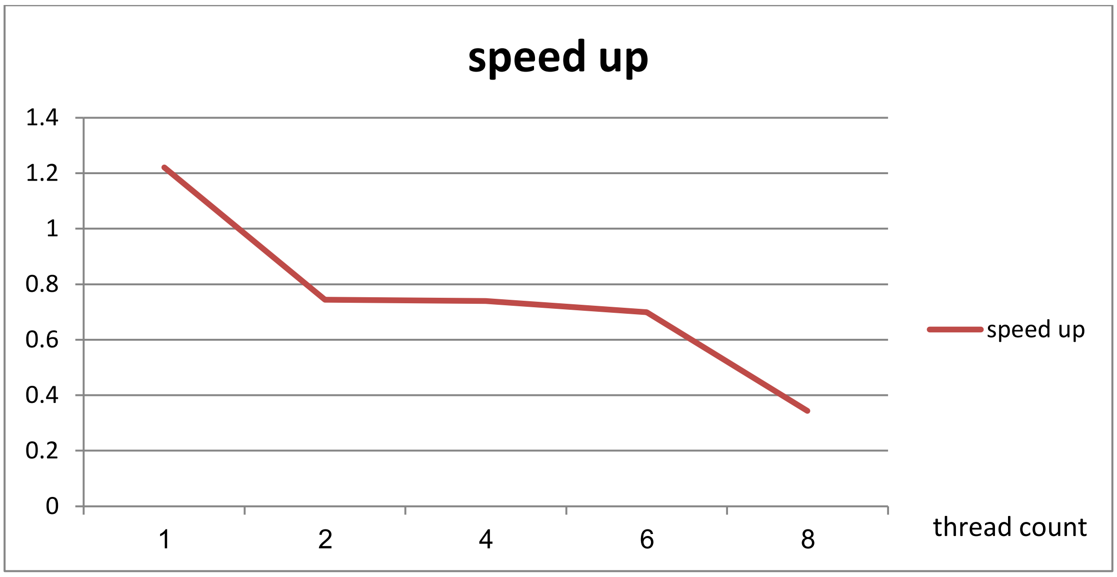
Input sizes:

Tested for n = 100, 300, 500, 600, and 2000 with Thread= 2, 6, 4, 8 for each input.

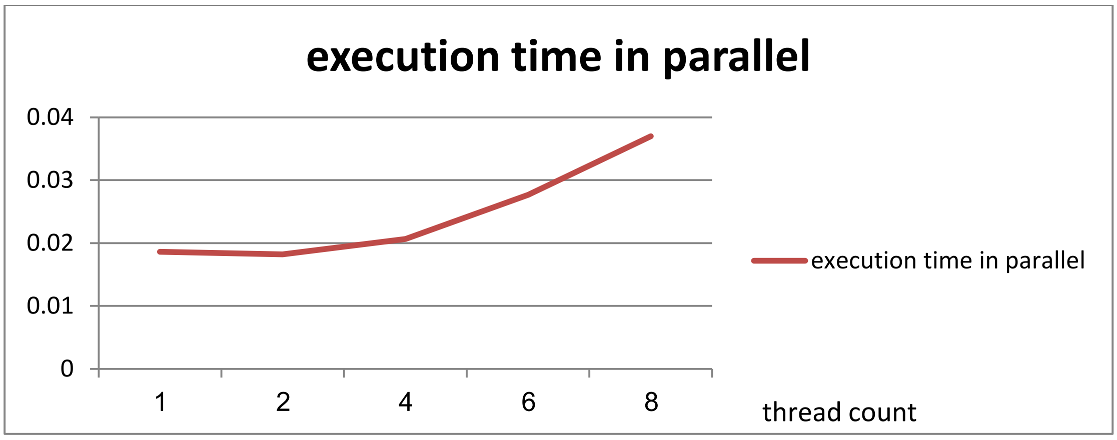
- ## Results

For n= 100

Thread count	Sequential time	Parallel time	Speedup
1	0.0226633	0.0186433	1.2202533x
2	0.0136596	0.0182238	0.744494x
4	0.015108	0.020645	0.739068x
6	0.0193492	0.0276578	0.6990504x
8	0.0127012	0.0369562	0.343792x



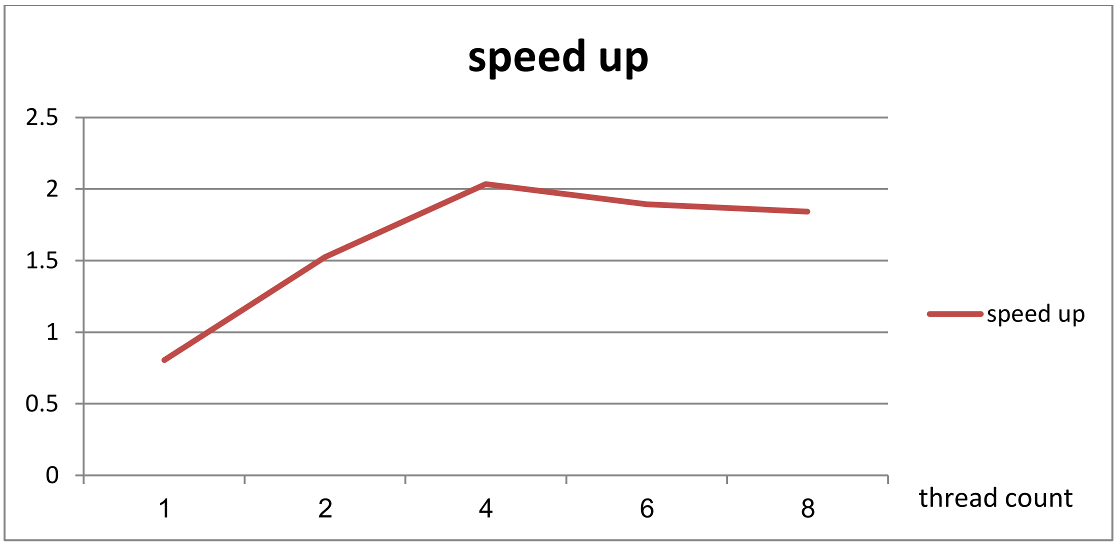
Speedup vs Thread count for n=100



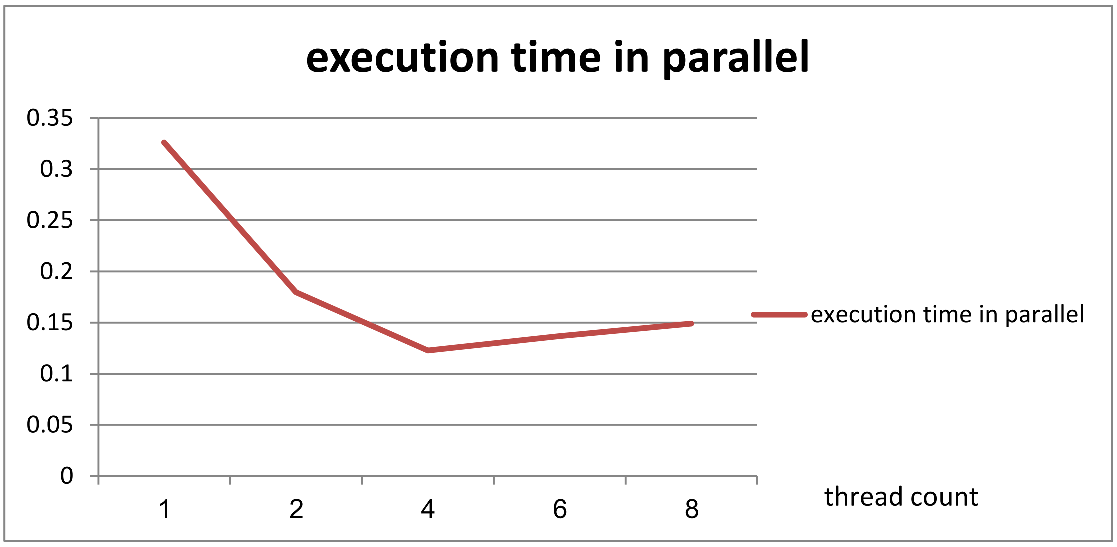
Parallel execution time vs Thread count for n=100

For n= 300

Thread count	Sequential time	Parallel time	Speedup
1	0.26268	0.326023	0.80571
2	0.2731624	0.1795376	1.5238732x
4	0.2492286	0.1228206	2.0353952x
6	0.2589478	0.136816	1.893028x
8	0.2743806	0.1489556	1.8426762x



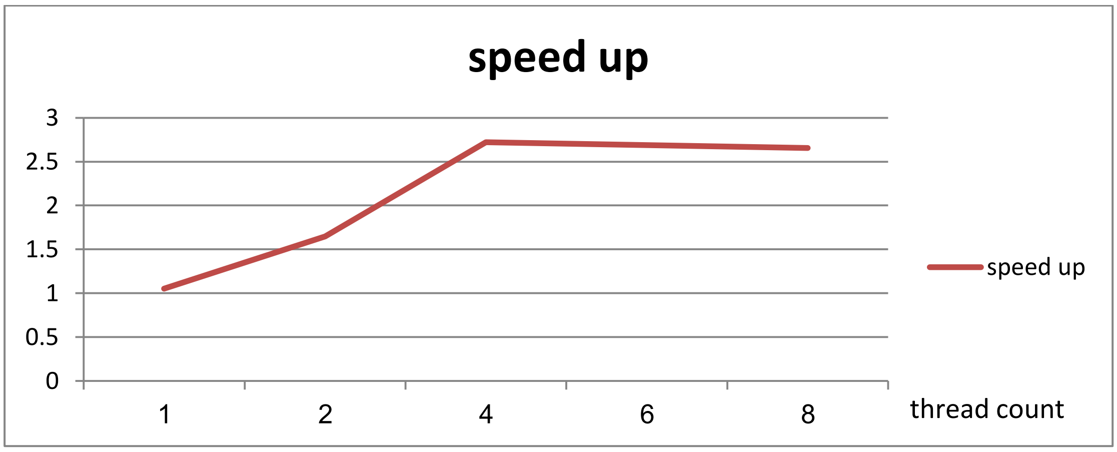
Speedup vs number of threads when n=300



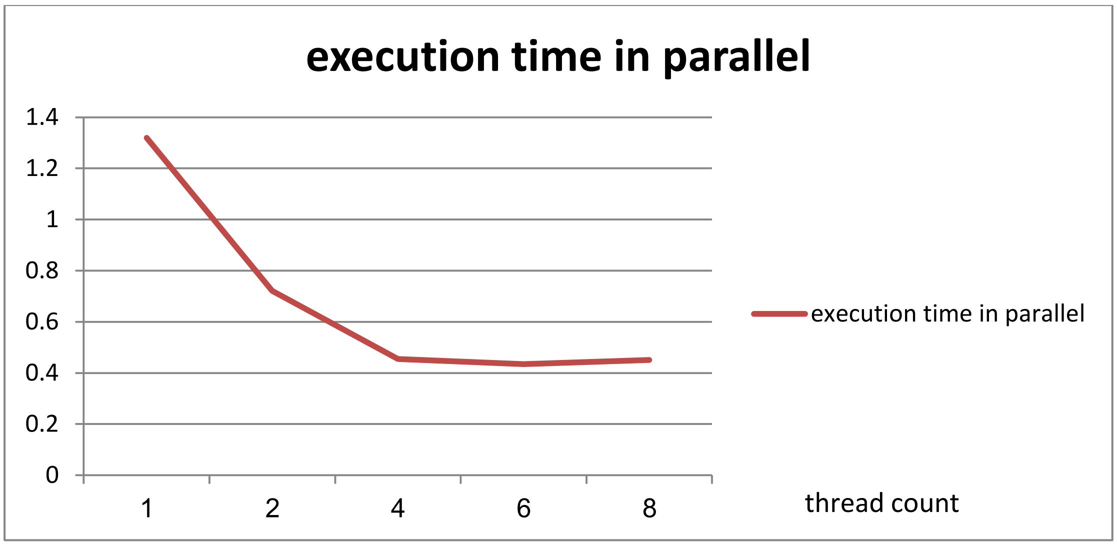
Parallel execution time vs Thread count for n=300

For n= 500

Thread count	Sequential time	Parallel time	Speedup
1	1.387679	1.319443	1.051716
2	1.189132	0.721673	1.648519x
4	1.235756	0.1228206	2.0353952x
6	1.166639	0.454587	2.720159x
8	0.2743806	0.434356	2.687262x



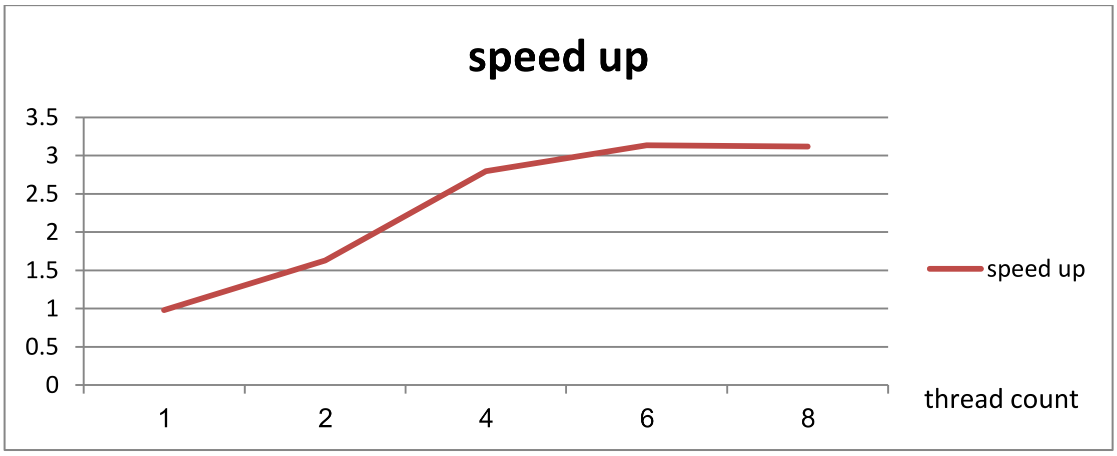
Speed up vs Thread count for n=500



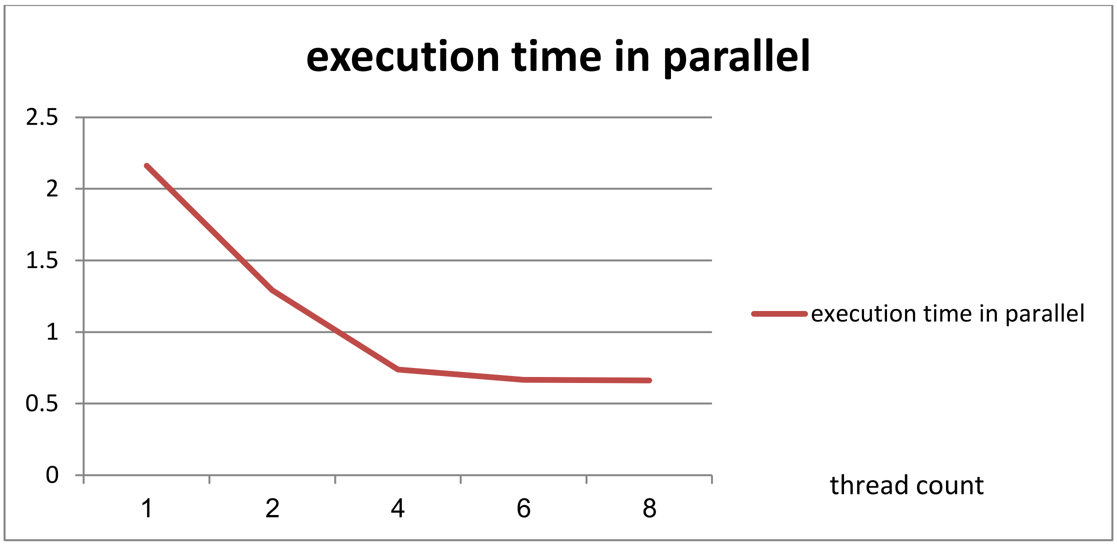
Parallel execution time vs Thread count for n=500

For n= 600

Thread count	Sequential time	Parallel time	Speedup
1	2.11975	2.161607	0.980636x
2	2.099118	1.291893	1.629574x
4	2.058724	0.737375	2.79379x
6	2.089999	0.666279	3.136086x
8	2.067464	0.662953	3.11975x



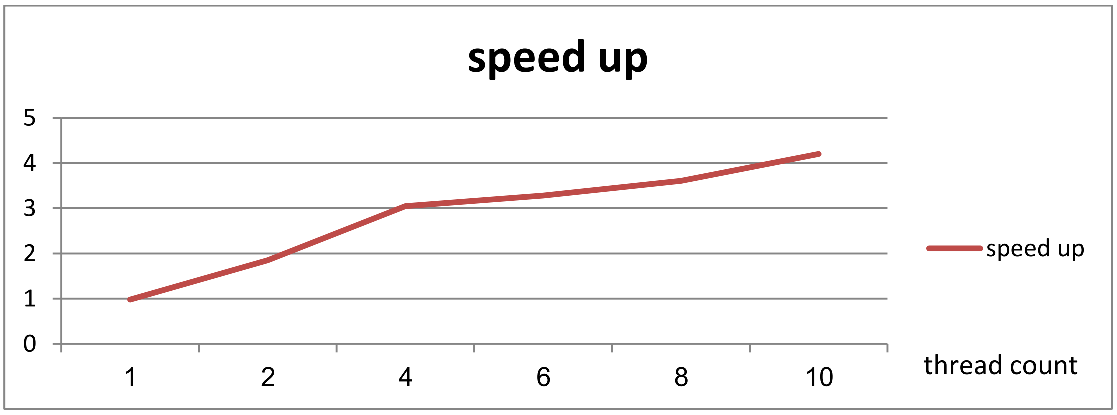
Speed up vs Thread count for n=600



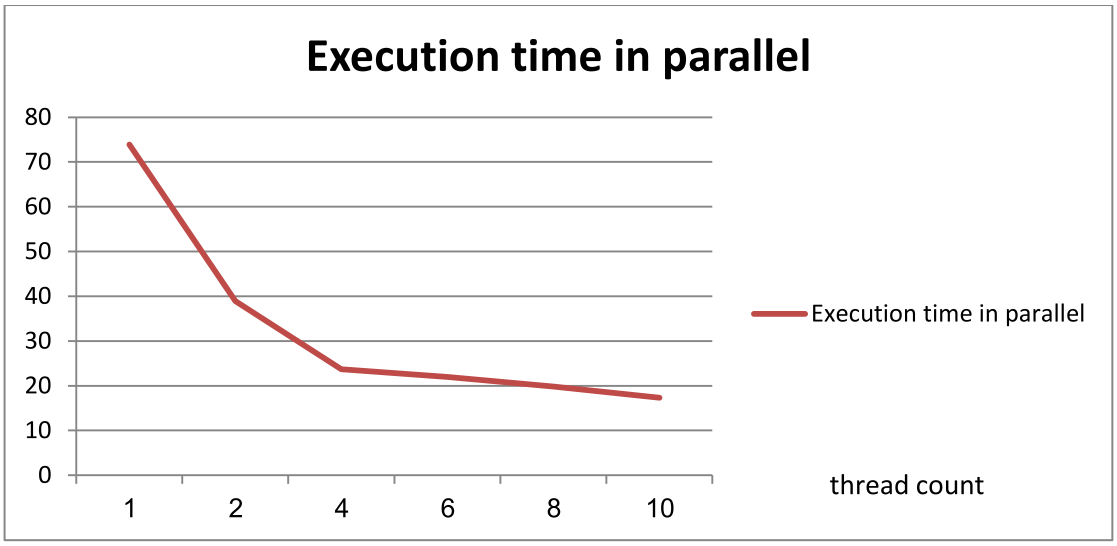
Parallel execution time vs Thread count for n=600

For n= 2000

Thread count	Sequential time	Parallel time	Speedup
1	72.371476	73.868117	0.979739x
2	71.854989	38.909186	1.846736x
4	72.131604	23.716798	3.041372x
6	71.896788	21.955281	3.274692x
8	71.410495	19.823369	3.602339x
10	72.79891	17.341939	4.197853x



Speed up vs Thread count for n=2000



Parallel execution time vs Thread count for n=2000

• Discussion

Why is speedup sublinear?

- **Thread creation and synchronization overhead**: Threads have to wait for each other to finish after every iteration of the outer loop (k), which causes delay.
- **Shared memory**. Although threads in this code operate on different rows, reading common values in the current_k row and column may cause conflicts.
- **Uneven load distribution**: If the division of rows isn't handled correctly specially when (n%threads_num !=0), some threads will do more work than others, which will cause the others to be idle waiting for them to finish.
- When we increase the number of threads, the speed improves a lot at first. Later, adding more threads leads to more delay until we reach a point where adding more threads makes it slower.

Amdahl's Law:

Amdahl's law implies that even with perfect parallelization, some parts of the code remain sequential which causes the speedup to be limited. Accordingly, the maximum speedup depends on the parallelizable portion of the code.

In our case, if we suppose that 70-80% of the code was parallelized, then even with infinite number of threads, the maximum speedup we can get is between (1/0.3) – (1/0.2), which is around 3.33x – 5x.

Conclusion

The Floyd-Warshall algorithm is a good candidate for parallelization because of its independent updates across the matrix. Speedup can be achieved by splitting the word among threads correctly and synchronize them at the right point.

Lessons learned:

- Dividing the work evenly is very important to achieve efficiency.
- Incorrect indexing can cause segmentation faults, which is so easy to run into.
- To get the best results, creation and synchronization overhead must be minimized.
- Always test correctness by comparing the output of parallel version with the output of the sequential one.
- Speedup is not linear, although multithreading improved performance, the speedup was sublinear due to overhead, memory contention, and Amdahl's law. Adding more threads doesn't necessarily leads to faster execution.

Despite the challenges, the parallel version achieved noticeable speedup, which validates the effort to parallelize this algorithm.

References

[1] "Floyd Warshall Algorithm," GeeksforGeeks, 16 April 2025. [Online]. Available: https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/.

[2] "< iomanip > Header in C++," Geeksforg, [Online]. Available: https://www.geeksforgeeks.org/iomanip-in-cpp/.

[3] "Floyd-Warshall Algorithm," Programiz, [Online]. Available: https://www.programiz.com/dsa/floyd-warshall-algorithm.

[4] "Chrono in C++," GeeksforGeeks, 20 January 2023. [Online]. Available: https://www.geeksforgeeks.org/chrono-in-c/.

[5] "Vector of Vectors in C++," GeeksforGeeks, 14 February 2020. [Online]. Available: https://www.geeksforgeeks.org/vector-of-vectors-in-c-stl-with-examples/.

- **Tools used**
  1. VSCod.
  2. Excel.
  3. Canva for graphs.
  4. ChatGPT to debug load distribution.