Rajiv Gandhi University of Knowledge Technologies

Catering the Educational Needs of Gifted Rural Youth of AP

Parallel Implementation and Analysis of All Pairs Shortest Path Algorithm (Floyd-Warshall Algorithm)

SUMMER INTERNSHIP PROJECT REPORT

Submitted by

R131983: THARIKOPPULA BINESH

Under the guidance of

N.SATYANANDARAM

Department of CSE

Rajiv Gandhi University of Knowledge Technologies, RK Valley, Kadapa.

Department of Computer Science and Engineering

Rajiv Gandhi University Of Knowledge Technologies,

AP-IIIT RK Valley, Kadapa, Andhra Pradesh-516330.

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
कंदि - ५०२ २८५, संगारेड्डी, तेलंगाना, भारत.
फोन : +९१-४०-२३०१ ६०३३, फेक्स : +९१-४०-२३०१ ६०३२

Indian Institute of Technology Hyderabad
Kandi - 502 285, Sangareddy, Telangana, INDIA
Phone: (040) 2301 6033; Fax: (040) 2301 6032

CERTIFICATE

*This is to certify that **Mr. Tharikoppula Binesh,** Roll no. R131983, 3rd Year student of CSE Department of AP-IIIT RGUKT RKValley has done an internship from **16th May 2018 to 24th July 2018** under my supervision. As a part of the internship, he studied **Parallel Implementation and Analysis of All Pairs Shortest Path Algorithm (Floyd-Warshall Algorithm)**.*

Dr. Sathya Peri Ph.D
Associate Professor
Department of Computer Science Engineering
Indian Institute of Technology Hyderabad
Kandi, Sangareddy - 502 285 Telangana, India

[illegible]

Dr. Sathya Peri,

Associate Professor,

Dept. of Computer Science & Engineering,

IIT Hyderabad, India, 502285,

Email: sathya_p@iith.ac.in

RGUKT

Rajiv Gandhi University of Knowledge Technologies

Catering the Educational Needs of Gifted Rural Youth of AP

CERTIFICATE

This is to certify that this work entitled "**Parallel Implementation and Analysis of All Pairs Shortest Path Algorithm (Floyd-Warshall Algorithm)**" was successfully carried out by T.BINESH(R131983) in fulfillment of the requirements leading to award the credits for SUMMER INTERNSHIP in the **DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING by RAJIV GANDHI UNIVERSITY OF KNOWLEDGE TECHNOLOGIES, RK VALLEY** during the academic year 2017-2018.

Internal Guide
N.SATYANANDARAM
Department of CSE,
RGUKT, RK Valley,
Kadapa.

ACKNOWLEDGEMENT

To start with ,I express my profound sense of gratitude towards my project internal guide **N.Satyanandaram** Sir Dept.of CSE,Rk Valley for the valuable guidance, constructive support and continuous encouragement towards the successful completion of the project.

I would like to express my sincere gratitude to Dr.Sathya Peri Sir, Department of Computer Science and Engineering , Indian Institute of Technology Hyderabad for imparting his valuable guidance and giving me such a wonderful opportunity to learn something new in this internship. He has been very kind,caring,Patience and helpful throughout the project.He is the One who motivated me towards research side in the area of parallel computing and patiently corrected my mistakes and supports me. And I owe a Big thanks to you sir. I would also like to thank Mr.Mukhthi Ph.D student, for his time and support for clearing my doubts and providing me inputs regarding some areas .

Finally, I owe my gratitude and special thanks to my family and my friends for their constant support and encouragement throughout this project.

Yours Sincerely,

THARIKOPPULA BINESH

ABSTRACT

Floyd-Warshall (FW) algorithm is a classic dynamic programming algorithm that solves the all-pairs shortest path (APSP) problem on directed weighted graphs. So in this work i parallelized the standard FW and also parallelized another version of FW i.e. Tiled FW algorithm using OPENMP.In standard FW Algorithm,we initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which includes the picked vertex as an intermediate vertex in the shortest path.Then i parallelized this standard FW by using OPENMP directive '#pragma omp parallel for' it creates a parallel region and each thread will get particular number of rows of intermediate matrix. And those row values will be updated parallelly.In second version,Parallelized Tiled FW Algorithm, input matrix is divided into tiles of size B. During the k-th block iteration, the algorithm updates the k-th diagonal tile first, then it updates the tiles in the remainder of the k-th block row(East and West Tiles) parallelly and k-th block column(North,South tiles) parallelly and finally it updates the remaining tiles of the matrix (North East, North West, South East, South West tiles) parallelly.Each thread will take one tile of size BXB and applies FW algorithm to execute in parallel. Likewise i have given many input graphs as matrices to two parallelized algorithms to calculate the efficiency of both.But the results were unexpected that Parallelized Standard FW remains at constant runtime with variable number of threads.And Parallelized Tiled FW algorithm shows quite best results(less run time) with the increase in number of threads when block size is fixed. At last i concluded that for considerably large graphs it is best to use parallelized Tiled FW Algorithm to achieve maximum effeciency but we must maintain some threshold number of threads.That's why parallelized Tiled FW Algorithm is asymptotically optimal among all implementations of FW algorithm because it reduces processor-memory traffic by factor of B.

Keywords: FW- Floyd Warshall , OPENMP - Open Multi-Processing

Abbreviations:

1. FWA - Floyd Warshall Algorithm
2. SFWA - Standard Floyd Warshall Algorithm
3. TFWA - Tiled Floyd Warshall Algorithm
4. OPENMP - Open Multi-Processing
5. APSP - All Pairs Shortest Path
6. PSFW - Parallelized Standard Floyd Warshall
7. PTFW - Parallelized Tiled Floyd Warshall
8. FW - Floyd-Warshall

LIST OF CONTENTS

CHAPTER 1

1 INTRODUCTION

1.1 Background

Most of today’s algorithms are sequential, that is, they specify a sequence of steps in which each step consists of a single operation. These algorithms are well suited to today’s computers, which basically perform operations in a sequential fashion. Although the speed at which sequential computers operate has been improving at an exponential rate for many years, the improvement is now coming at greater and greater cost. As a consequence, researchers have sought more cost-effective improvements by building “parallel” computers – computers that perform multiple operations in a single step.In order to solve a problem efficiently on a parallel machine, it is usually necessary to design an algorithm that specifies multiple operations on each step, i.e., a parallel algorithm.

Finding the shortest path between two objects or all objects in graphs is a common task in solving many day to day and scientific problems also. The algorithms for finding the shortest path finds their application in many fields such as social networks, bioinformatics, aviation, routing protocols, google maps etc. Shortest path algorithms can beclassified into two types single source shortest paths and all pairs shortest paths. There are many algorithms for all pairs shortest paths. Some of them are Floyd-Warshall Algorithm and Johnson's algorithm.

Floyd-Warshall algorithm was developed by Robert Floyd in 1962. This algorithm follows the dynamic programming methodology .This algorithm is used for graph analysis and finds the shortest paths between all pairs of nodes or vertices. It requires a directed weighted graph with positive or negative edges.This algorithms is only for returning shortest path lengths between vertices but does not return the shortest path with names of nodes.

Reason for parallelizing is sequential code of this algorithm requires N^3 comparisons for 'N' nodes in a graph.But in real world situations number of nodes will be huge, at that time sequential code takes too much amount of time. So to avoid larger time complexity, we have chosen for parallelization.

1.2 Statement of the Problems

Parallelizing All Pairs Shortest Path Algorithm is a challenging task considering the fact that the problem is dynamic in nature but it can be possible with shared memory multiprocessing environment OPENMP.
Answers to many of the issues arising is parallel computing i.e., accelerating applications by using multiple cores,based on the notion that larger inputs may be divided into disjoint parts which may be solved almost independently. Once all these independent parts are solved, one must subsequently combine their intermediate answers. The key-observation is that we may parallelize those independent works by assigning each subproblem to a different processor or thread and sometimes we need to synchronize some tasks to get the desired result.
Problem is that we cannot parallelize all algorithms because some algorithms might follows sequential nature.But Floyd-Warshall is parallelizable and it is commonly acknowledged that parallelizing applications is neither trivial, nor straightforward. Several algorithms need to be redesigned from scratch in order to be parallelized and effectively utilize vital system resources such as cache memory.
Another problem is that we must have to use the system resources for the purpose of efficiency of our algorithm. If we use all processors/cores in our system along with with the best usage of cache memory then our efficiency will be best and that's why optimizations are required in our algorithm.

1.3 Objectives of the Research

1.3.1 Overall objective

Main objective is to optimize the runtime complexity of floyd-warshall algorithm because real world graph applications like social networks,routing protocols,bioinformatics,road networks contains huge number of nodes/vertices then it will be a difficult task to analyze and find distance between each and every object with sequential FW algorithm.

So performance can be improved with parallel versions of FW.And there are many versions are available for FW algorithm some of them are Standard FW,recursive FW and Tiled FW algorithms.Now my objective is to parallelize the Standard FW algorithm and Tiled FW algorithm.We dont need recursive FW algorithm because it requires lot of recursive function calls for larger graphs means recursion is fairly expensive and it requires the allocation of a new stack frame everytime so it is not a recommended one.

I want to parallelize those two algorithms using OPENMP parallel programming environment.OPENMP is an API for C,C++ and fortran languages which contains a bunch of compiler directives and library routines suitable for multithreading.After parallelizing both SFWA and TFWA, we have to compare the runtime complexities of both and check which one is best for which condition and analyse the characterstices of both algorithms on different platforms with variable number of threads but fixing the size of block size in parallelized Tiled FW algorithm.

1.4 Scope

Scope of Floyd-Warshall algorithm is only for returning shortest path lengths between vertices but does not return the actual shortest path with names of nodes. We need to find efficiently the shortest path from every vertex to every other vertex, and to store this information in a compact form, the naive solution of finding and storing every possible route using DFS or similarly simple graph searches will not yield acceptable results. So that's why we are using adjacency matrix to store shortest distance between each and every vertex.However the scope is not limited to Tiled FW algorithm using OPENMP and we can improve the performance of our algorithms using other parallel programming environments.

CHAPTER 2: LITERATURE REVIEW

The All-Pairs Shortest Paths (APSP) problem is one of the most important, and most studied, algorithmic graph problem. Graph shows a weighted directed graph G = (V,E) where V is the vertices and E is the edges of the graph. The real-world applications including VLSI routing, bioinformatics, social network analysis, and communications. Certain a directed weighted graph, APSP is the problem of finding the shortest distance between every two nodes in the graph. This problem reduces to multiple breadth-first search traversals in the case of an unweighted directed graph. The following figure shows Undirected and Directed graph.

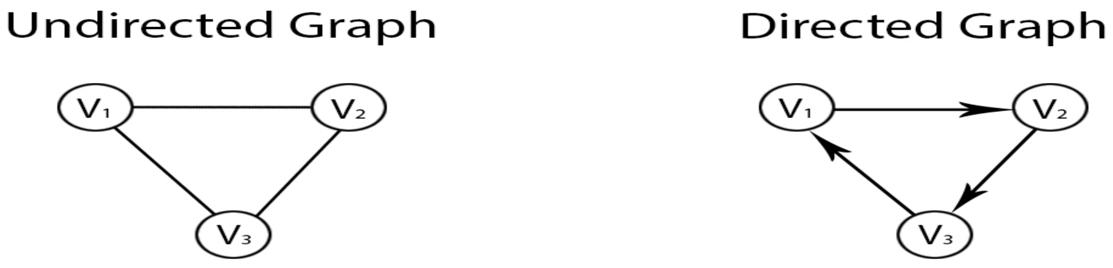


Fig 1. Example for Directed Graph and Undirected Graph

Floyd-Warshall algorithm is a procedure, which is used to find the shorthest path lengths among all pairs of nodes in a graph, which does not contain any cycles of negative length. The main advantage of Floyd-Warshall algorithm is its simplicity.

DESCRIPTION OF FLOYD-WARSHALL ALGORITHM:

Floyd-Warshall algorithm uses a matrix of lengths D_0 as its input. If there is an edge between nodes *i* and *j* , than the matrix D_0 contains its length at the corresponding coordinates. The diagonal of the matrix contains only zeros. If there is no edge between edges *i* and *j*, then the position (i,j) contains positive infinity. In other words, the matrix represents lengths of all paths between nodes that does not contain any intermediate node.

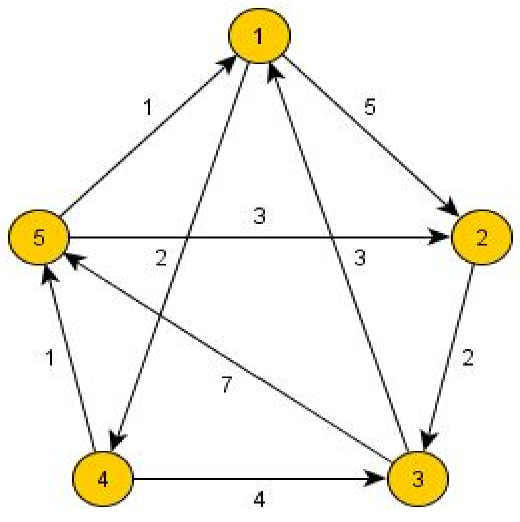


Fig 2. Input Graph with Five Vertices.

$$\begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix}$$

Fig 3. Adjacency Matrix for above Graph(D0)

In each iteration of Floyd-Warshall algorithm this matrix is recalculated, so it contains lengths of paths among all pairs of nodes using gradually enlarging set of intermediate nodes. The matrix D_1, which is created by the first iteration of the procedure, contains paths among all nodes using exactly one (predefined) intermediate node. D_2 contains lengths using two predefined intermediate nodes. Finally the matrix D_n uses *n* intermediate nodes.

This transformation can be described using the following recurrent formula:

$$\mathbf{D_n[i,j] = min(D_{n-1}[i,j] , D_{n-1}[i,k]+D_{n-1}[k,j])}$$

Because this above transformation never rewrites elements, which are to be used to calculate the new matrix, we can also use the same matrix for both D_i and $D_{i+1.}$

And Small Example that how above tranformation(FW Algorithm) is Executing:

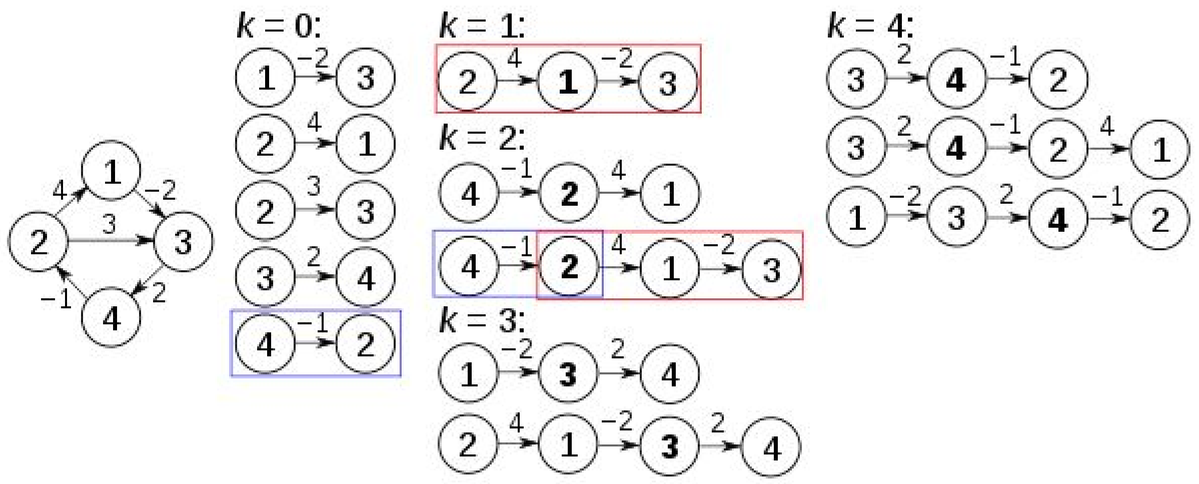
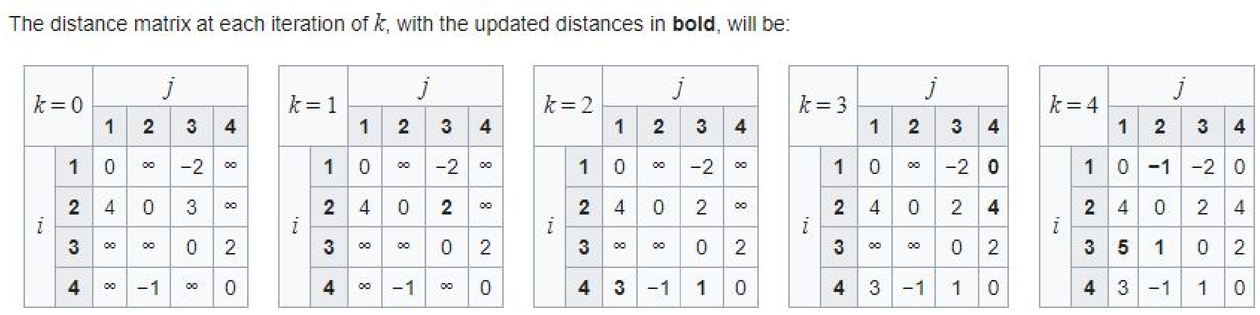


Fig 4. Small Example Shows how Sequential FW is Working



The distance matrix at each iteration of k, with the updated distances in **bold**, will be:

$k=0$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	3	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k=1$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	**2**	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k=2$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	2	∞
	3	∞	∞	0	2
	4	**3**	−1	**1**	0

$k=3$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	**0**
	2	4	0	2	**4**
	3	∞	∞	0	2
	4	3	−1	1	0

$k=4$		j			
		1	**2**	**3**	**4**
i	**1**	0	**−1**	−2	0
	2	4	0	2	4
	3	**5**	**1**	0	2
	4	3	−1	1	0

Fig 5. Distance Matrices for Above Graph and k=4 is the final APSP

Pseudo Code:

```
FloydWarshall(D)
1.for k in 1 to n do
2.     for i in 1 to n do
3.         for j in 1 to n do
4.             if D[i][j] > D[i][k] + D[k][j] then
5.                 D[i][j] = D[i][k] + D[k][j]
```

And the Intermediate Matrices while finding the All Pairs Shortest Path Matrix for D_0 is shown in figure below:

$$D_0 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & 3 & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 5 & 7 & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 7 & 2 & 14 \\ 5 & 0 & 2 & 7 & 9 \\ 3 & 8 & 0 & 5 & 7 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_5 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 2 & 4 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

Fig 6. All Intermediate Matrices and D5 is APSP Matrix.

Asymptotic complexity:

The algorithm consists of three loops over all nodes, and the most inner loop contains only operations of a constant complexity. Hence the aymtotic complexity of the whole Floyd-Warshall algorithm is O(N^3) ,where N is number of nodes of the graph. So parallelization is necessary because FW algorithm requires N^3 comparisons for considerably large graphs with thousands of nodes/vertices it will take huge amount of time that's why it is needed to parallelize this algorithm.

2.1 Summary

Sequential implementation of the Floyd-Warshall algorithm. It utilizes 3 for-loops to compute the minimum distance between all possible connectivities in the graph, a method that's most commonly used to implement the algorithm in sequential.For that purpose i have implemented Sequential FW algorithm by giving Graphs with variable number of nodes and calculated the runtime taken for each graph and noted down the values in below table.

And the Chart for above Table is:

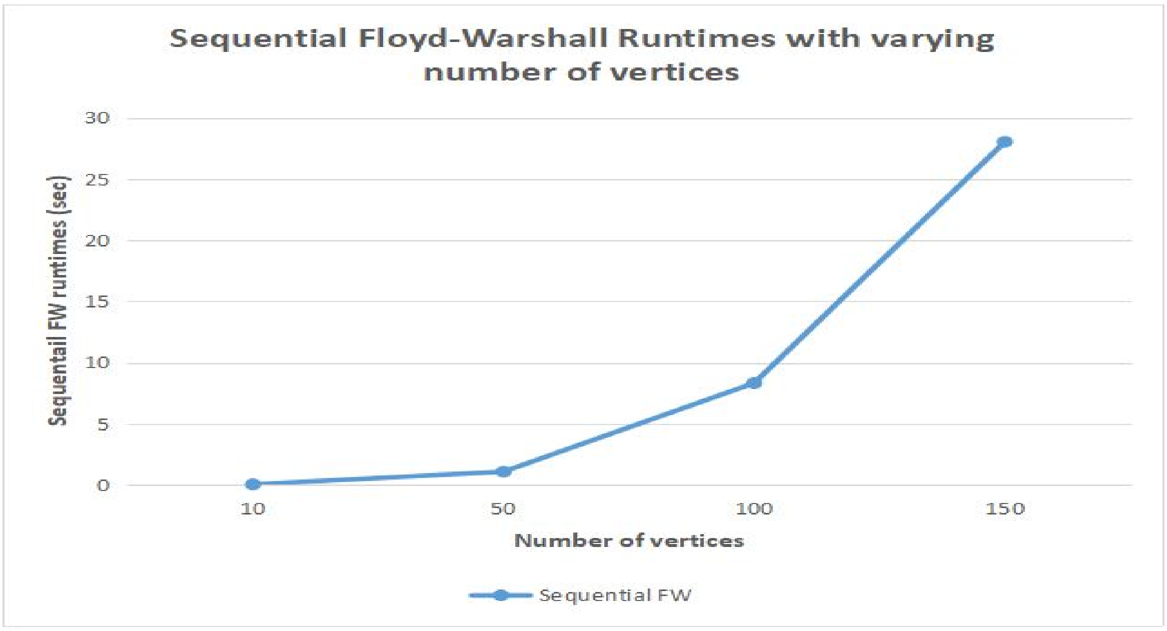


Fig 7. Chart Showing Sequential FW runtimes vs number of nodes

By observing the above chart we can conclude graph is going in an exponential manner and assume for graph with thousand nodes, we can't imagine how much time it might take.So it need to be parallelized otherwise it will be time waste job even if we are having suffiecient resources we are wasting most time on execution only by using limited resources in our system.

And By looking at pseudo code itself we can conclude that i-loop and j-loop both are independent so it is possible for parallelization.But k-loop cannot be Parallelized because we must produce the intermediate matrices in a sequential manner.

And for sufficiently large graphs we can divide that adjacency matrix into tiles and we can also apply FW algorithm on those tiles parallely by maintaining some synchronization techniques to get the desired results.

3 METHODOLOGY

3.1 Concepts

My First concept is to parallelize Standard FW algorithm using OPENMP.OpenMP (Open Multi-Processing) is an application programming interface (API) for parallel programming intended to work on shared-memory architectures. More specifically, it is a set of compiler directives, library routines and environmental variables, which influence run-time behavior. OpenMP enables parallel programming in various languages, such as C, C++ and FORTRAN and runs on most operating systems.

OPENMP API uses the fork-join model of parallel execution. Multiple threads perform tasks defined implicitly or explicitly by OpenMP directives. All OpenMP applications begin as a single thread of execution, called the initial thread. The initial thread executes sequentially until it encounters a parallel construct. At that point,this thread creates a group of itself and zero or more additional threads and becomes the master thread of the new group. Each thread executes the commands included in the parallel region, and their execution may be differentiated, according to additional directives provided by the programmer. At the end of the parallel region, all threads are synchronized.

The runtime environment is responsible for effectively scheduling threads. Each thread, receives a unique id, which differentiates it during execution. Scheduling is performed according to memory usage, machine load and other factors and may be adjusted by altering environmental variables. In terms of memory usage, most variables in OpenMP code are visible to all threads by default. However, OpenMP provides a variety of options for data management, such as a thread-private memory and private variables, as well as multiple ways of passing values between sequential and parallel regions. Additionally,recent OpenMP implementations introduced the concept of tasks, as a solution for parallelizing applications that produce dynamic workloads. Thus, OpenMP is enriched with a flexible model for irregular parallelism, providing parallel while loops and recursive data structures.

I'm using '**#pragma omp parallel for**' compiler directive to parallelize the Standard FW algorithm.

```
#pragma omp parallel
  for(int i=0; i<N; i++)
  { ... }
```

This code creates a parallel region, and each individual thread executes what is in your loop. In other words, you do the complete loop N times, instead of N threads splitting up the loop and completing all iterations just once.

```
#pragma omp parallel
{
#pragma omp for
for(int i=0;i<N;i++)
{….}
}
```

(or)

```
#pragma omp parallel for

{

for(int i=0;i<N;++i)

{ }

  }
```

Above one will create one parallel region (aka one fork/join, which is expensive and therefore you don't want to do it for every loop) and run multiple loops in parallel within that region. Means it divides loop iterations between the spawned threads and executes them in parallel.

And Second concept is to parallelize Tiled Floyd-Warshall Algorithm for best results, Tiling is a commonly used technique to achieve higher data reuse in looped code. The tiled version of FW (FW TILED) works as follows: Initially, the input matrix is divided into tiles of size B. During the k-th block iteration, the algorithm updates the k-th diagonal tile (black, CR tile in Figure 9) first, then it updates the tiles in the remainder of the k-th block row and block column (grey, E, W, N, S tiles) and finally it updates the remaining tiles of the matrix (white, NE, NW, SE, SW tiles). This way, all dependencies are satisfied. FW TILED reduces processor-memory traffic by a factor of B (where B is the order of the cache size) and is asymptotically optimal among all implementations with respect to processor-memory traffic.

And this is the algorithm for Tiled FW implementation:

Algorithm 3: Tiled FW (FW_TILED)

```
1: for $k = 0 \rightarrow n$ step $B$ do
2:     FW(CR);
3:     for tile in E, W, N, S do
4:         FW(tile);
5:     end for
6:     for tile in NE, NW, SE, SW do
7:         FW(tile);
8:     end for
9: end for
```

Fig 8. Tiled Version of FW Algorithm.

Now below figure shows how tiled FW algorithm is working:

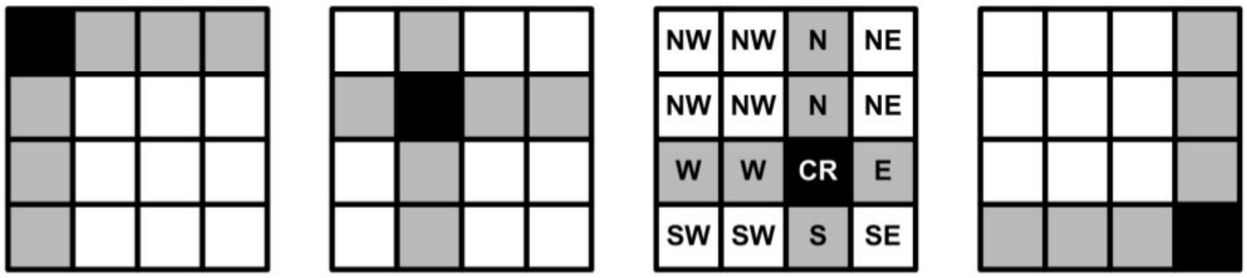


Fig 9. FW_Tiled The Matrix is divided into tiles of size B

So my concept of parallelization is , From the above figure every square is a tile of size BXB and threads run CR first, then all tiles in CR row(E and W) parallely,then all tiles in CR column(N’s and S’s) parallely.And atlast our diagonals tiles(NE,NW,SE,SW) parallely.Then our CR tile points to next position like black tile moved from first diagram to second diagram and then performs updation of its adjacent tiles and soon until CR(main tile) reaches to last diagonal position.

And this parallelization is also done by using OPENMP directive to split each tile and make them to run parallel using '#pragma omp parallel for' directive.And the good thing we dont need to synchronize any task because different tiles are running parallelly not one tile is running parallelly at different positions.

3.2 Methods

Parallelized Standard Floyd-Warshall Algorithm:

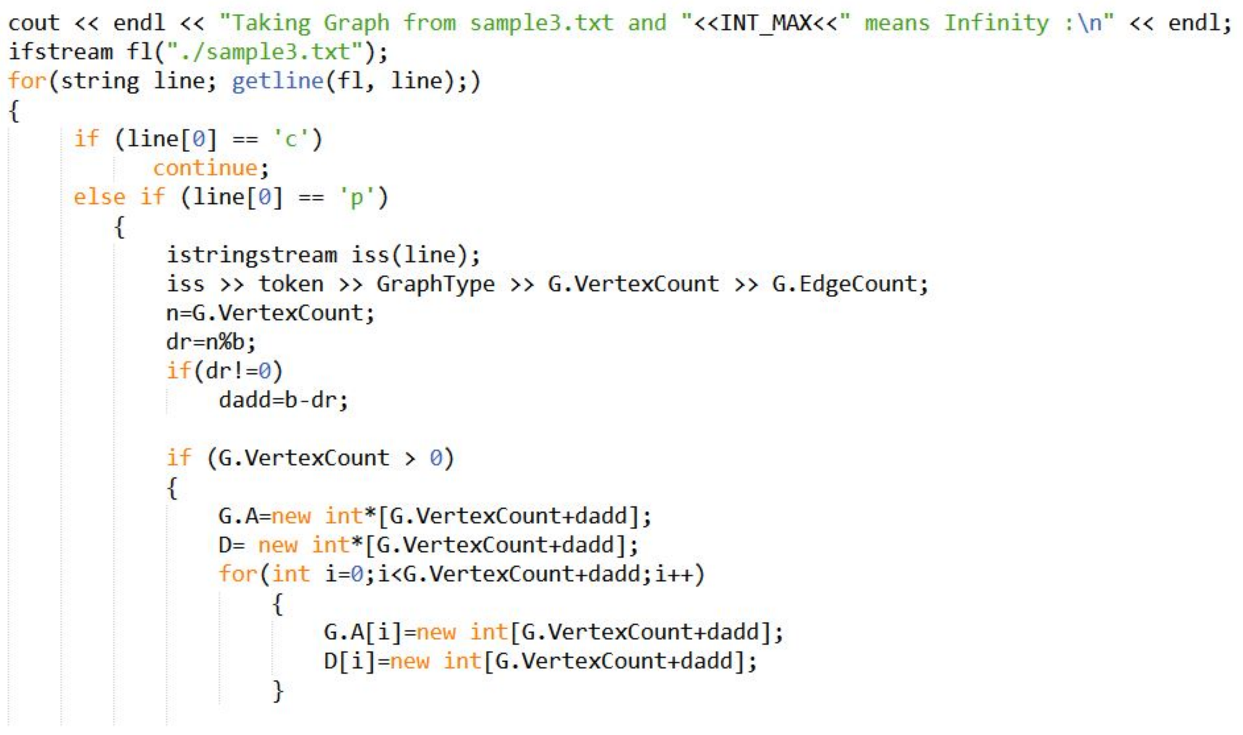
My first method is parallelized Standard FW algorithm using OPENMP directives as we have already discussed that what is the concept behind it.Now we it is the time to discuss about my method.

As i want to take number of threads from user because architecture/number of processors varies from system to system.That's why i'm using compiler directive, . **omp_set_num_threads(t);** //To create atmost number of threads if needed.

After intializing number of threads,then i have created 'Graph' like structure containing 'A' as a double dimension pointer to store the input graph as an adjacency matrix in it. VertexCount and EdgeCount are variable to store the number of vertices and number of edges in the input graph.

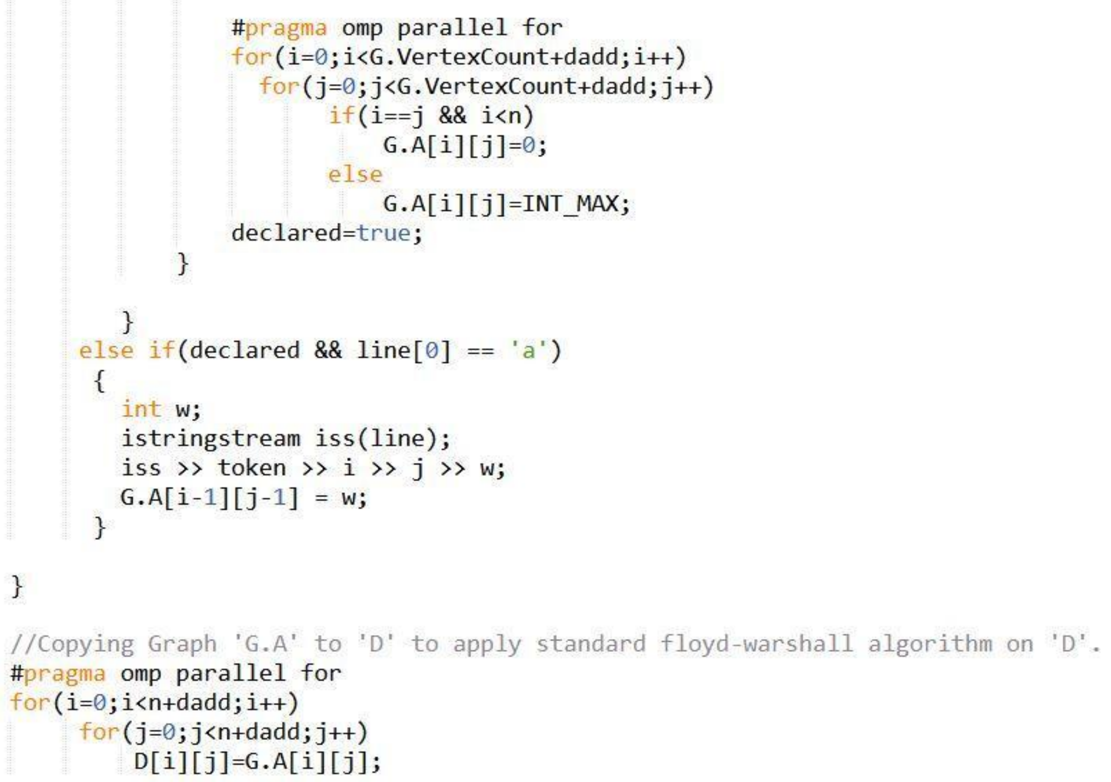
struct Graph { . int **A; int VertexCount; int EdgeCount; };

Storing Graph into G.A[][] as an adjacency matrix from input .txt file and after that copying values from G.A to D:



```
cout << endl << "Taking Graph from sample3.txt and "<<INT_MAX<<" means Infinity :\n" << endl;
ifstream fl("./sample3.txt");
for(string line; getline(fl, line);)
{
    if (line[0] == 'c')
        continue;
    else if (line[0] == 'p')
      {
        istringstream iss(line);
        iss >> token >> GraphType >> G.VertexCount >> G.EdgeCount;
        n=G.VertexCount;
        dr=n%b;
        if(dr!=0)
            dadd=b-dr;

        if (G.VertexCount > 0)
        {
            G.A=new int*[G.VertexCount+dadd];
            D= new int*[G.VertexCount+dadd];
            for(int i=0;i<G.VertexCount+dadd;i++)
                {
                    G.A[i]=new int[G.VertexCount+dadd];
                    D[i]=new int[G.VertexCount+dadd];
                }
```

Fig 10. Taking Input from txt file Part1

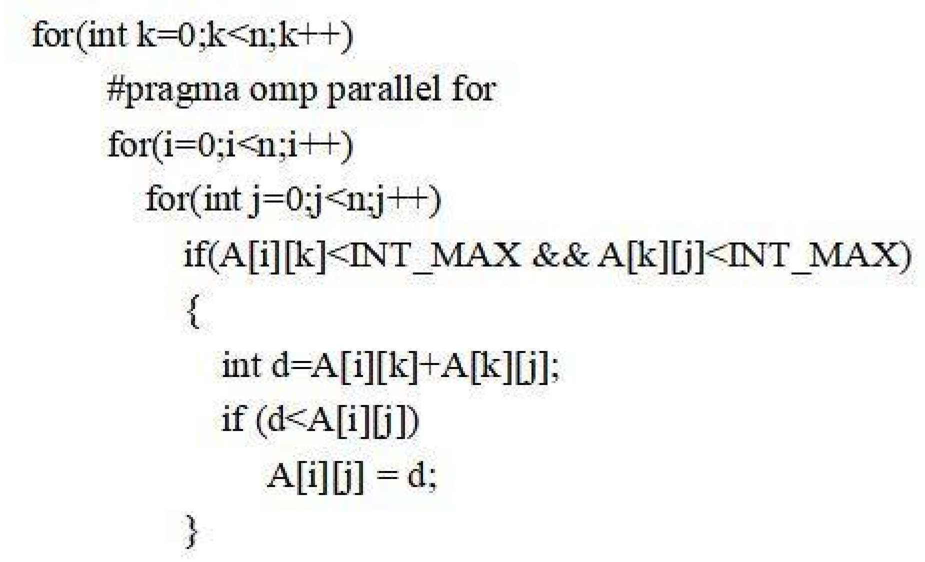


```
                #pragma omp parallel for
                for(i=0;i<G.VertexCount+dadd;i++)
                  for(j=0;j<G.VertexCount+dadd;j++)
                       if(i==j && i<n)
                           G.A[i][j]=0;
                       else
                           G.A[i][j]=INT_MAX;
                declared=true;
            }

        }
    else if(declared && line[0] == 'a')
     {
       int w;
       istringstream iss(line);
       iss >> token >> i >> j >> w;
       G.A[i-1][j-1] = w;
     }

}

//Copying Graph 'G.A' to 'D' to apply standard floyd-warshall algorithm on 'D'.
#pragma omp parallel for
for(i=0;i<n+dadd;i++)
     for(j=0;j<n+dadd;j++)
         D[i][j]=G.A[i][j];
```

Fig 11. Taking Input from txt file Part2

In the below PSFW #pragma omp parallel will divide the iterations of i among the number of threads.And Every thread will select one particular A[i,j] at particular time and executes it in parallel. ex: For below matrix. 1 2 3 4 0 1 3 6 2 If three threads are there then, At a time 3 threads parallelly takes 3 values and tries to update those values by,

if(A[i][k]<INT_MAX and A[k][j]<INT_MAX)
 {
 if(A[i][k]+A[k][j]<A[i][j]) . A[i][j]=A[i][k]+A[k][j];
 }



```
for(int k=0;k<n;k++)
    #pragma omp parallel for
    for(i=0;i<n;i++)
      for(int j=0;j<n;j++)
        if(A[i][k]<INT_MAX && A[k][j]<INT_MAX)
        {
          int d=A[i][k]+A[k][j];
          if (d<A[i][j])
             A[i][j] = d;
        }
```

Fig 12. Parallellized Standard FW Algorithm

So by using the above parallelized algorithm we can be able split the rows and assign them to different threads.

Parallelized Tiled Floyd-Warshall Algorithm:

Second method of work is parallelized Tiled FW algorithm.This algorithm would take graph 'A' as input, 'b' is the block size and it is better to take block size equivalent as cache size of your system, 'n' number of vertices and 't' number of threads.

As like mentioned above we must have to iterate our main tile(CR tile in fig.9) for k times.And in every kth iteration we need to do the following four phases:

Phase 1: updates the k-th diagonal-tile(main tile). Phase 2:updates the tiles which are remained in the main tile row parallelly by passing arguments to updateSubmatrix() method. Phase 3:updates the tiles which are remained in the main tile column parallelly by passing arguments to updateSubmatrix() method. Phase 4:finally it updates the remaining tiles except that main tile row and column of the matrix parallelly by passing arguments to updateSubmatrix() method.

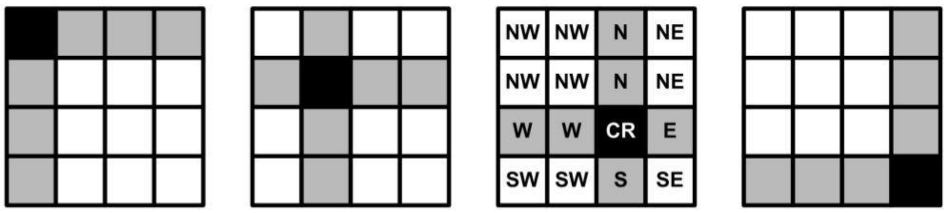
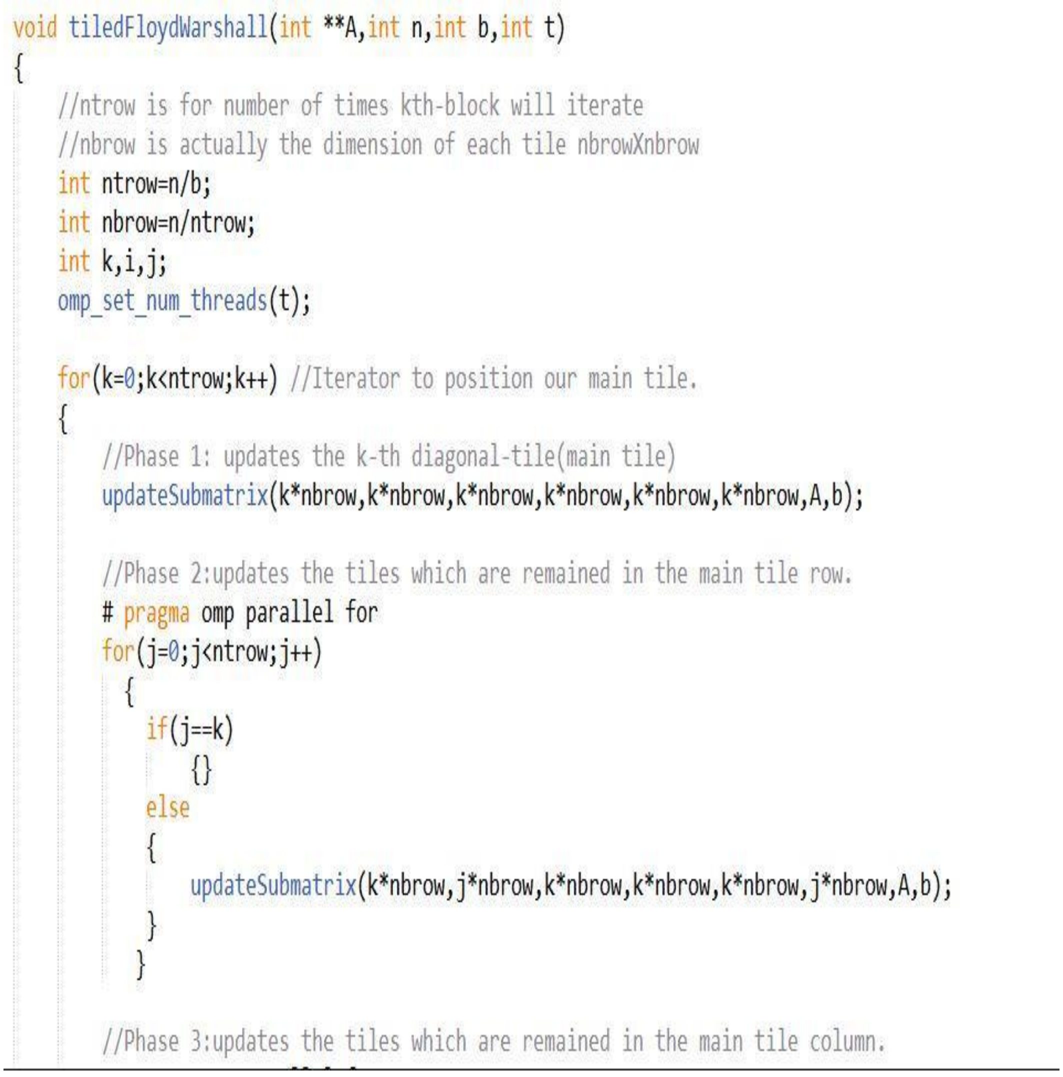


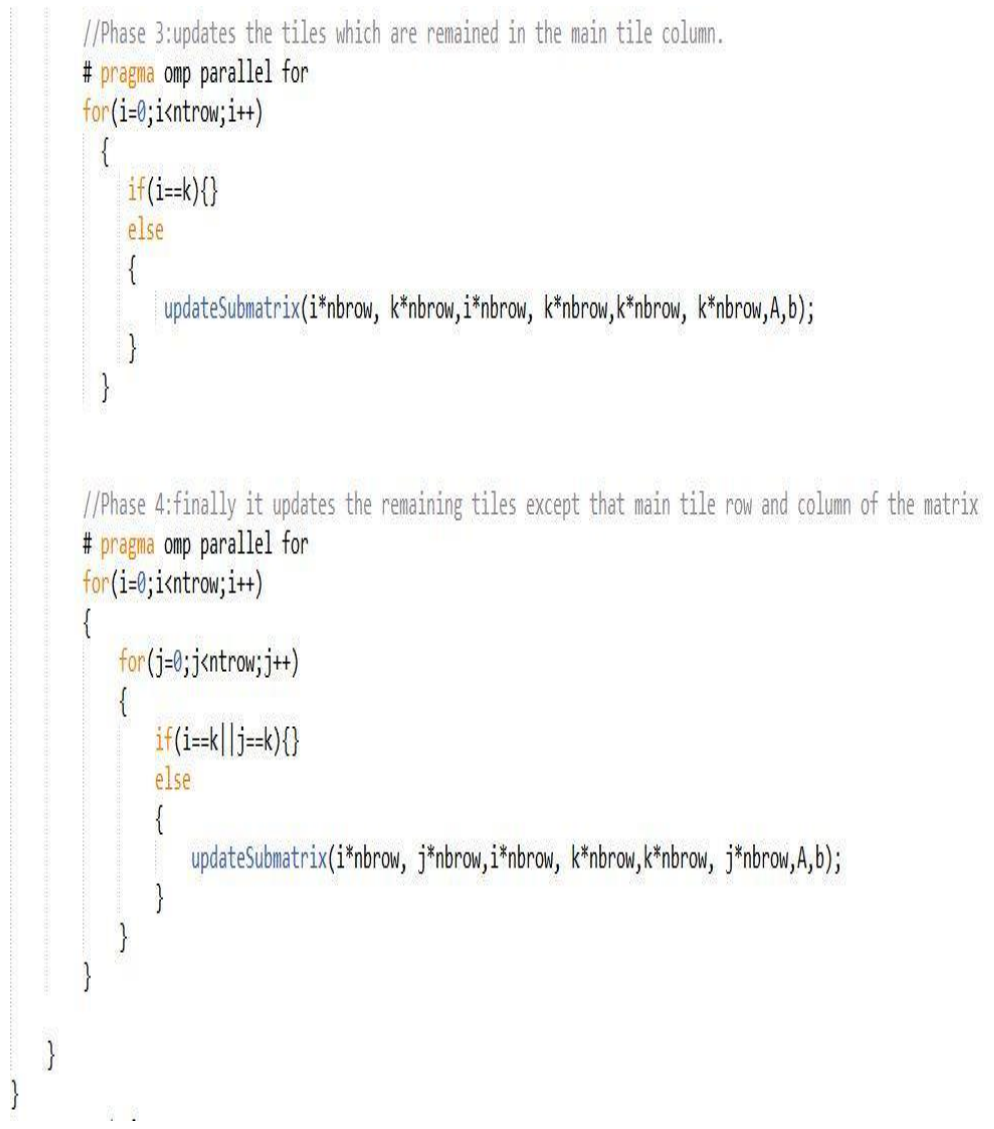
Fig 13. FW_Tiled The Matrix is divided into tiles of size B

So in above diagram every square is a tile of size bXb and threads run CR first, then all tiles in CR row(E and W) parallely,then all tiles in CR column(N’s and S’s) parallely.And atlast our diagonals tiles(NE,NW,SE,SW) parallely.Then our CR tile points to next position like black tile moved from first diagram to second diagram and then performs updation of its adjacent tiles and soon until CR(main tile) reaches to last diagonal position.



```
void tiledFloydWarshall(int **A,int n,int b,int t)
{
    //ntrow is for number of times kth-block will iterate
    //nbrow is actually the dimension of each tile nbrowXnbrow
    int ntrow=n/b;
    int nbrow=n/ntrow;
    int k,i,j;
    omp_set_num_threads(t);

    for(k=0;k<ntrow;k++) //Iterator to position our main tile.
    {
        //Phase 1: updates the k-th diagonal-tile(main tile)
        updateSubmatrix(k*nbrow,k*nbrow,k*nbrow,k*nbrow,k*nbrow,k*nbrow,A,b);

        //Phase 2:updates the tiles which are remained in the main tile row.
        # pragma omp parallel for
        for(j=0;j<ntrow;j++)
          {
            if(j==k)
              {}
            else
            {
              updateSubmatrix(k*nbrow,j*nbrow,k*nbrow,k*nbrow,k*nbrow,j*nbrow,A,b);
            }
          }

        //Phase 3:updates the tiles which are remained in the main tile column.
```

Fig 14. PTFW part1



```
        //Phase 3:updates the tiles which are remained in the main tile column.
        # pragma omp parallel for
        for(i=0;i<ntrow;i++)
          {
            if(i==k){}
            else
            {
              updateSubmatrix(i*nbrow, k*nbrow,i*nbrow, k*nbrow,k*nbrow, k*nbrow,A,b);
            }
          }


        //Phase 4:finally it updates the remaining tiles except that main tile row and column of the matrix
        # pragma omp parallel for
        for(i=0;i<ntrow;i++)
        {
            for(j=0;j<ntrow;j++)
            {
                if(i==k||j==k){}
                else
                {
                    updateSubmatrix(i*nbrow, j*nbrow,i*nbrow, k*nbrow,k*nbrow, j*nbrow,A,b);
                }
            }
        }

    }
}
```

Fig 15. PTFW Part2

Each Tile will be taken over by one thread and it executes updateSubmatrix parallelly.and each updateSubmatrix() method will do FW procedure internally but for small size of tile bXb.

updateSubmatrix():

Each thread would take one updateSubmatrix method to execute in parallel. i.e.. updateSubmatrix(int a_row,int a_col,int b_row,int b_col,int c_row,int c_col,int **A,int b).

where a_row,a_col --- are the starting coordinates of present tile. b_row ---- will be same as a_row. b_col,c_row ---- are the starting coordinates of main tile(kth-diagonal tile or CR tile). c_col ----- will be same as a_col.

That is nothing but Applying Floyd-Warshall Algorithm on each Tile of size bXb in matrix A.

```
updateSubmatrix(int a_row,int a_col,int b_row,int b_col,int c_row,int c_col,int **A,int b)
  {
for(k=0;k<b;k++)
   for(i=0;i<b;i++)
        for(j=0;j<b;j++)
                if(A[b_row+i][b_col+k]< INT_MAX && A[c_row+k][c_col+j] < INT_MAX )
A[a_row+i][a_col+j]=min(A[a_row+i][a_col+j],(A[b_row+i][b_col+k]+A[c_row+k][c_col+j]));
}
```

And below fig shows how Tiled FW is working: Means first blue color tile is send to updateSubmatrix and it will be updated according to FW algorithm.Then all red tiles in that blue tile row will execute parallelly. After that all red tiles in blue tile column executes parallelly. After that all yellow tiles will executes parallelly according to FW algorithm. And there is no need of synchronization required at all because at a time we are working on different tiles. And while updating the distance(length) value in a tile sometimes we might look at the values in other tiles also, it will happen mostly but it wont require synchronization because we are retrieving value from outside tile not updating the value.

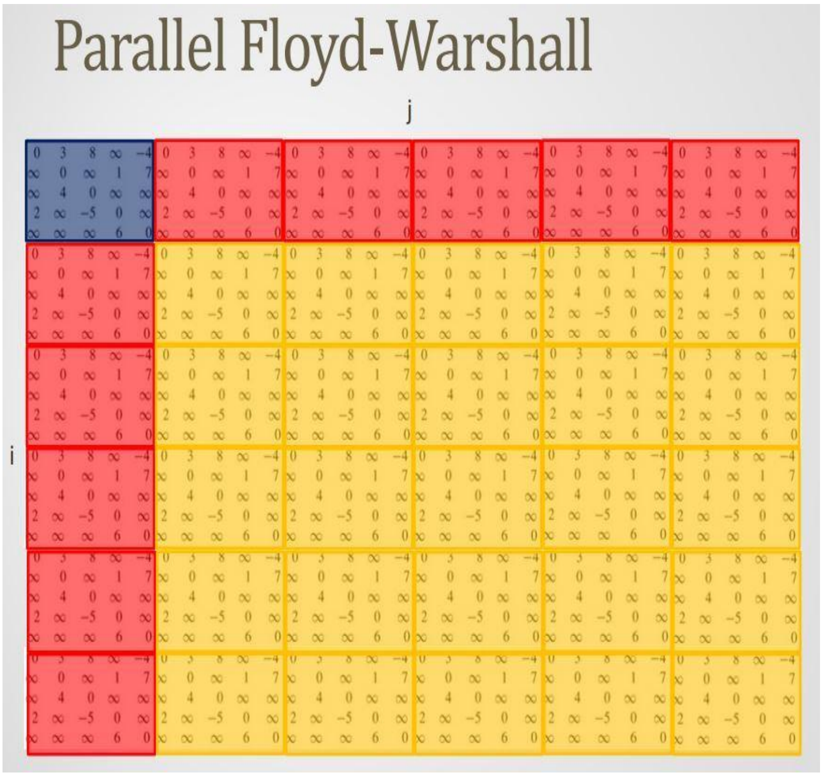


Fig 16. Adjacency Matrix Splitted into Tiles of Size bXb

4 RESULTS AND DISCUSSION

4.1 Purpose

Purpose is i'm going to compare the running time complexities of both Parallelized Standard FW with Parallelized Tiled FW algorithm.And based on the comparison i want to specify for which conditions, which algorithm is suitable.

Standard FW without parallelization is taking exponentail time with increase in adjacency matrix size in figure below:

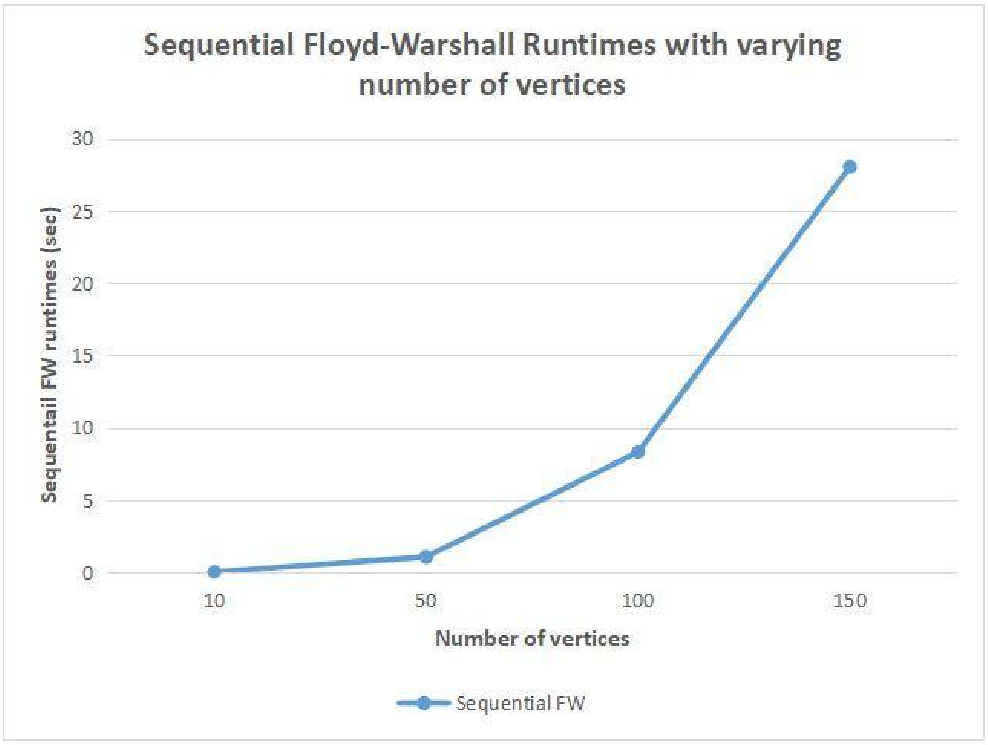
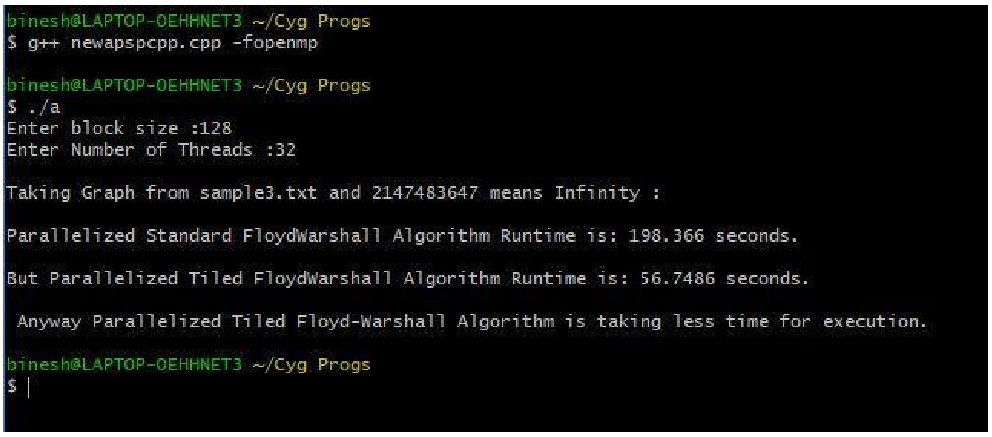


Fig 17. Sequential FW with different Matrix Sizes

Actually my program takes block size and number of threads as input then it will executes both parallel algorithms and prints the time taken for each algorithm to the console.One example is shown in below fig.18

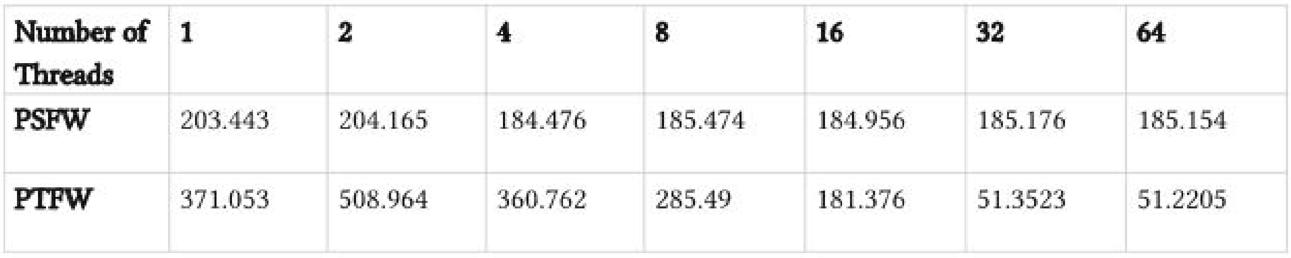


```
binesh@LAPTOP-OEHHNET3 ~/Cyg Progs
$ g++ newapspcpp.cpp -fopenmp

binesh@LAPTOP-OEHHNET3 ~/Cyg Progs
$ ./a
Enter block size :128
Enter Number of Threads :32

Taking Graph from sample3.txt and 2147483647 means Infinity :

Parallelized Standard FloydWarshall Algorithm Runtime is: 198.366 seconds.

But Parallelized Tiled FloydWarshall Algorithm Runtime is: 56.7486 seconds.

 Anyway Parallelized Tiled Floyd-Warshall Algorithm is taking less time for execution.

binesh@LAPTOP-OEHHNET3 ~/Cyg Progs
$ |
```

Sample Output Comparing Runtimes of PSFW and PTFW

The architecture of my system is Intel® Core™ i3-5005U CPU @ 2GHz,2000Mhz, 2 Cores and 4 Logical processors. In my experimentation i have given Input from txt file with 3353 vertices and 8866 edges from sample3.txt with static block_size=128 and variable number of threads. Matrix is 3353X3353 and i executed both Parallelized Standard FW and Parallelized Tiled FW algorithm on same architecture with same graph as input.

Lab data 1 Running Times of PSFW and PTFW with matrix size 3353X3353 with block size=128



Number of Threads	**1**	**2**	**4**	**8**	**16**	**32**	**64**
PSFW	203.443	204.165	184.476	185.474	184.956	185.176	185.154
PTFW	371.053	508.964	360.762	285.49	181.376	51.3523	51.2205

And the chart for above table is:

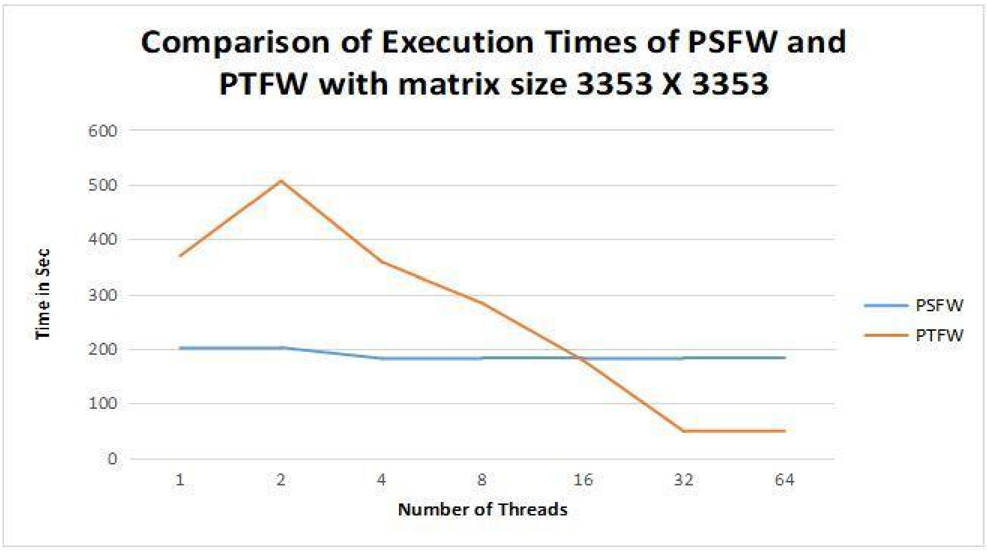


Fig 19. PSFW vs PTFW

From the above chart we can observe that PSFW is independent of number of threads means Parallelized Standard FW remains constant with number of threads.And it is better than Sequential FW algorithm which is having chart like exponential curve(see fig.17) .Because of using OPENMP to parallelize, it reduces the runtimes of Standard FW from exponential curve to linear constant straight line. But However Parallelized Tiled FW algorithm shows quite good results(less runtime) with the increase in number of threads.And the running time complexity of PTFW is O(n^3/b) means it reduces number of computations by factor of b.

5 CONCLUSION AND RECOMMENDATIONS

5.1 Conclusion

In this work we have experimented with the well known Floyd-Warshall algorithm, which solves the all-pairs shortest path problem on directed graphs.First of all our Standard FW algorithm is taking exponential running times based on the number of vertices, but after parallelizing using OPENMP running times of Parllelized Standard FW became optimal than Sequential FW. And From the comparison chart we can conclude that for considerably large graphs it is best to use parallelized Tiled FW Algorithm to achieve maximum effeciency but we must maintain some threshold number of threads. And we should maintain the block-size(Tile size) equal to the cache size of our system to achieve maximum efficiency.However usage of maximum number of threads depends upon the number of cores available and for small graphs it is better to use PSFW. But the real world graphs like social networks,google maps etc. deals with huge number of vertices so our parallelized Tiled FW will be useful evrywhere. And for the systems with good architecture will be capable of spawning many threads , results in best performance of PTFW. That’s why parallelized Tiled FW Algorithm is asymptotically optimal among all implementations of FW algorithm because it reduces processor-memory traffic by factor of b where 'b' is dimension of the tile.

6.REFERENCES

[1] Parallelizing the Floyd-Warshall Algorithm on Modern Multicore Platforms https://people.cs.kuleuven.be/~george.karachalias/papers/floyd-warshall.pdf

[2] Parallelization of Shortest Path Algorithm Using OpenMP and MPI by Rajashri Awari https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8058360&tag=1

[3] C. Papadimitriou, M. Sideri,On the Floyd-Warshall Algorithm for Logic Programs.

[4] Robert D. Blumofe and Charles E. Leiserson, Scheduling Multithreaded Computations by Work Stealing, http://supertech.csail.mit.edu/papers/steal.pdf

[5] OPTIMIZING ALL-PAIRS SHORTEST-PATH ALGORITHM USING VECTOR INSTRUCTIONS Sungchul Han and Sukchan Kang https://www.inf.ethz.ch/personal/markusp/teaching/18-799B-CMU-spring05/material/sungchul-sukc han.pdf

[6] J.S. Park, M. Penner, and V. K. Prasanna, Optimizing Graph Algorithms for Improved Cache Performance IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 15, NO. 9, SEPTEMBER 2004.

[7] Introduction to Parallel Computing, chapter 10 Mikael Rännar after material by Robert Granat, Isak Jonsson och Erik Elmroth https://www8.cs.umu.se/kurser/5DV050/VT10/handouts/F10.pdf

[8] The OpenMP API specification for parallel programming https://www.openmp.org/

[9] Dynamic Programming | Set 16 (Floyd Warshall Algorithm) https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/

[10]OPENMP tutorial https://computing.llnl.gov/tutorials/openMP/