**Sequential Implementation**

To calculate the minimum paths between all pairs of a graph with dist N nodes, the Floyd-Warshall algorithm was used. The algorithm compares all possible paths across the graph through a pair of vertices. It has N³ complexity due to the triple loop it implements. Inside nested loops additional if conditions were needed to make sure the algorithm that the compound (i,k,j) exists. It checks, that is, that there is k which is directly joined to i, j respectively. The main function of the algorithm was implemented in the following piece of code.

time(&start);

//Calculate minimum distance paths

//Using the Floyd Warshall algorithm

for(k=0; k<n; k++)

for(i=0; i<n; i++)

for(j=0; j<n; j++)

if ((dist[i][k] \* dist[k][j] != 0) && (i != j))

if(dist[i][j] > dist[i][k] + dist[k][j] || dist[i][j] == 0)

dist[i][j] = dist[i][k] + dist[k][j];

time(&end);

Running the algorithm for nodes n=10, 100, 1000, 5000 respectively, we obtained the following results.

| **Number of Nodes** | **Seconds** |
| --- | --- |
| 10 | 0.00 |
| 100 | 0.00 |
| 1000 | 7.00 |
| 5000 | 768.00 |

**OpenMP & MPI**

OpenMP is a programming platform that allows code parallelism in a homogeneous shared memory system (e.g. a multi-core processor). For example, we could parallel a set of functions in a multi-core processor where cores share memory with each other. This memory includes cache, RAM, hard disk memory, etc. It is also mentioned that communication is easy and relatively cheap.

MPI (based on the message transmission interface) is also a programming platform, but provides the ability to parallelize code through a (non-)homogeneous distributed system (e.g. a supercomputer). For example, it is possible to parallel an entire program through a network of computers or nodes communicating over the same network. Since these nodes are essentially computers, they have their own memory layout and set of kernels (see OpenMP description). Communication between nodes, compared to shared memory systems, can be difficult and usually expensive.

Listed below are the ways implemented in OpenMP and MPI respectively, to run the Floyd Warshall algorithm in parallel.

**OpenMP Implementation**

OpenMP enables us to parallelize code using sequential programming languages. Calling the omp.h library provides routines that parallelize processes that are anointed with high complexity. So they should be used when the sequence code is resource intensive, otherwise the overhead will be fatal. In execution, if we have not defined them somewhere, as many threads are created as there are CPUs of the machine, which is ideal (optimal). The main algorithmic function is listed below.

time(&start);

//Calculate minimum distance paths

//Using omp parallel for, it partitions the loop into the threads (as many as the CPUs) and runs the algorithm

#pragma omp parallel for private(i,j,k) shared(dist)

for(k=0; k<n; k++)

for(i=0; i<n; i++)

for(j=0; j<n; j++)

if ((dist[i][k] \* dist[k][j] != 0) && (i != j))

if(dist[i][j] > dist[i][k] + dist[k][j] || dist[i][j] == 0)

dist[i][j] = dist[i][k] + dist[k][j];

time(&end);

The results produced by the OpenMP operation are shown in the table below.

| **Number of Nodes** | **Seconds** |
| --- | --- |
| 10 | 0.00 |
| 100 | 0.00 |
| 1000 | 3.00 |
| 5000 | 453.0 |

**MPI Implementation**

MPI is an alternative way of parallelizing a sequential code. By calling the mpi.h library, mechanisms are provided for the exchange of messages over a network between the machines of the system. The sequential code is divided into 2 sections.

One segment (rank=MASTER) refers to the master process, which initializes the array of distances and sends it (in MPI\_Send) to the other processes (slaves). It receives the results from each process, until the moment when all processes are finished and inactive. Within this loop for each result he gets, he selects the smallest distance between the result and the previous value of that particular distance and refreshes the array. Its implementation is given below.

if (my\_rank == MASTER) {

double t1,t2;

int disable=0,t=3;

int result[t];

//Initiate the dist with random values from 0-99

for(i=0; i<n; i++)

for(j=0; j<n; j++)

if(i==j)

dist[i][j] = 0;

else

dist[i][j] = rand()%100;

//Print initial distances

showDistances(dist);

t1 = MPI\_Wtime();

for(i=1;i<num\_procs;i++)

MPI\_Send(&(dist[0][0]),n\*n,MPI\_INT,i,WORKTAG,MPI\_COMM\_WORLD); //send the array dist in every machine

//send takes the first argument [0][0] and the size of the array n\*n and fills the array appropriately

do {

MPI\_Recv(&result,t,MPI\_INT,MPI\_ANY\_SOURCE,MPI\_ANY\_TAG,MPI\_COMM\_WORLD,&status);

if (status.MPI\_TAG == DIETAG) //if received DIETAG(0) means that every other slaves has done and finish the whole process

disable++;

else

if (dist[result[1]][result[2]]>result[0]) //do a final check in results. Works like a reduce function

dist[result[1]][result[2]]=result[0];

} while (disable < num\_procs-1);

t2 = MPI\_Wtime();

//print the final distances

showDistances(dist);

printf("Total Elapsed Time %f sec\n", difftime(t2, t1));

}

The second section refers to all slaves processes, which receive the table sent to them by the master. The part that will be separated for parallelization is the outer loop. Each slave process takes a portion of the loop (with a k-index). Calculates the segment using its rank. Calculates all assigned values (finds the shortest distances) and sends each value to the master process. Once finished, he must update the master process so that the whole program can be terminated at the right time. Its implementation is given below.

/\* workers code\*/

else{

int i, j, k,t=3;

int out[t];

MPI\_Recv(&(dist[0][0]),n\*n,MPI\_INT,MASTER,MPI\_ANY\_TAG,MPI\_COMM\_WORLD,&status);

if(my\_rank+1!=num\_procs)

remain=0;

for (k = slice\*(my\_rank-1); k < slice\*(my\_rank-1)+slice+remain; ++k) // slice the k alongside the ranks(slaves)

for (i = 0; i < n; ++i) //use of remain for the last slice

for (j = 0; j < n; ++j)

/\* If i and j are different nodes and if

the paths between i and k and between

k and j exist, do \*/

if ((dist[i][k] \* dist[k][j] != 0) && (i != j))

/\* See if you can't get a shorter path

between i and j by interspacing

k somewhere along the current

path \*/

if ((dist[i][k] + dist[k][j] < dist[i][j]) || (dist[i][j] == 0)){

dist[i][j] = dist[i][k] + dist[k][j];

out[0]=dist[i][j];

out[1]=i;

out[2]=j;

MPI\_Send(&out,t,MPI\_INT,MASTER,0,MPI\_COMM\_WORLD); //send back to master the calculated distance

}

MPI\_Send(0,0,MPI\_INT,MASTER,DIETAG,MPI\_COMM\_WORLD);

}

The results of the algorithm's runtimes are shown below.

| **Number of Nodes** | **Seconds** |
| --- | --- |
| 10 | 0.00 |
| 100 | 0.00 |
| 1000 | 7.00 |
| 5000 | 792.00 |

**Time & Acceleration Comparison**

In order to produce correct results, the same seed was used in all 3 implementations in order to produce the same distance table each time in order to check the correctness of each implementation and to be able to make a fair comparison between cases.

In the implementation of the Sequential algorithm and in the implementation of the parallel algorithm using OpenMP to measure the runtime, time is used. In implementation, however, by using MPI for the correct measurement of the execution time, the MPI\_Wtime is used (to avoid cases of unsynchronized clocks, etc.).

In the case of MPI, attempts were made to perform it on multiple machines in the laboratory of the 2nd. But the machines were not responding. For this reason, only one local machine was executed to determine the correct operation of the algorithm.

The machine used in all 3 cases has 4 threads.

The results of the runtimes for each case are aggregated in the table below.

|  | **n=10** | **n=100** | **n=1000** | **n=5000** |
| --- | --- | --- | --- | --- |
| **Sequential** | 0.0 sec | 0.0 sec | 7.0 sec | 768.0 sec |
| **OpenMP** | 0.0 sec | 0.0 sec | 3.0 sec | 453.0 sec |
| **MPI** | 0.0 sec | 0.0 sec | 7.0 sec | 792.0 sec |

To calculate the acceleration for each case, the formula is used:

S(p) = tₛ/tₚ, where the execution time of the sequential algorithm and the execution time of the parallel. All accelerations are shown in the table below.tₛtₚ

|  | **n=10** | **n=100** | **n=1000** | **n=5000** |
| --- | --- | --- | --- | --- |
| OpenMP | 0 | 0 | 2.33 | 1.70 |
| MPI | 0 | 0 | 1 | 0.97 |

**Conclusion**

From comparisons between runtimes and accelerations we see that there is a significant improvement in performance using OpenMP compared to the sequential algorithm.

In the case of MPI, due to the impossibility of execution on multiple machines, the results are not representative. But we see that there is a slowdown in relation to the sequential program. This overhead is due to the time it takes to exchange messages (containing the data) between processes.