**Name: M.Hassaan Atif**

**Roll no: 0147-BSCS-19**

**Section: CSA3**

**Title: PDC Project report**

**Algorithm**  
The algorithm of choice is Floyd Warshal’s “**All-pair shortest path**” algorithm.

**Serial solution**

**//The following code has been borrowed from Geeks4Geeks for analysis purposes**

**#include <iostream>**

**#include <chrono>**

**#include "mpi.h"**

**using namespace std::chrono;**

**using namespace std;**

**// defining the number of vertices**

**#define nV 4**

**#define INF 999**

**void printMatrix(int matrix[][nV]);**

**// Implementing floyd warshall algorithm**

**void floydWarshall(int graph[][nV]) {**

**int matrix[nV][nV], i, j, k;**

**for (i = 0; i < nV; i++)**

**for (j = 0; j < nV; j++)**

**matrix[i][j] = graph[i][j];**

**// Adding vertices individually**

**for (k = 0; k < nV; k++) {**

**for (i = 0; i < nV; i++) {**

**for (j = 0; j < nV; j++) {**

**if (matrix[i][k] + matrix[k][j] < matrix[i][j])**

**matrix[i][j] = matrix[i][k] + matrix[k][j];**

**}**

**}**

**}**

**printMatrix(matrix);**

**}**

**void printMatrix(int matrix[][nV]) {**

**for (int i = 0; i < nV; i++) {**

**for (int j = 0; j < nV; j++) {**

**if (matrix[i][j] == INF)**

**printf("%4s", "INF");**

**else**

**printf("%4d", matrix[i][j]);**

**}**

**printf("\n");**

**}**

**}**

**int mainzs(int \*argc, char \*\*argv) {**

**int rank, size;**

**MPI\_Init(argc, &argv);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);**

**double start, end;**

**int graph[nV][nV] = { {0, 3, INF, 5},**

**{2, 0, INF, 4},**

**{INF, 1, 0, INF},**

**{INF, INF, 2, 0}};**

**start = MPI\_Wtime();**

**floydWarshall(graph);**

**end = MPI\_Wtime();**

**if (rank == 0) { /\* use time on master node \*/**

**cout << "Runtime = " << end - start << "\n";**

**}**

**MPI\_Finalize();**

**return 0;**

**}**

**Parallel formulation**

//parallel code + explanation here. Do mention the division of rows in a block row major fashion

***#include "iostream"***

***#include "stdlib.h"***

***#include "mpi.h"***

***#include "math.h"***

***using namespace std;***

***#define INF 10000***

***int main63(int argc, char\*\* argv) {***

***MPI\_Init(&argc, &argv);***

***int size;***

***int rank;***

***MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);***

***MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);***

***const int N = 4;***

***int arr[N \* N] = {0, 3, INF, 5,***

***2, 0, INF, 4,***

***INF, 1, 0, INF,***

***INF, INF, 2, 0};***

***//int m = 4;***

***int rowsPerProc = N / size;***

***int rem = N % size;***

***int\*\* arr2D;***

***int root;***

***arr2D = new int\* [N];***

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

***arr2D[i] = &arr[(i \* N)];***

***}***

***int\* recBuff = new int[N \* (rowsPerProc + rem)];***

***MPI\_Status status;***

***int\* kRow = new int[N];***

***//broadcast to each proc their respective rows***

***if (rank == 0) {***

***for (int i = 1; i < size; ++i) {***

***if (i != size - 1) {***

***MPI\_Send(&arr[i \* N \* rowsPerProc], N \* rowsPerProc, MPI\_INT, i, 0, MPI\_COMM\_WORLD);***

***}***

***else {***

***MPI\_Send(&arr[i \* N \* rowsPerProc], N \* (rowsPerProc + rem), MPI\_INT, i, 0, MPI\_COMM\_WORLD);***

***}***

***}***

***for (int i = 0; i < N \* rowsPerProc; ++i) {***

***//cout << arr[i] << endl;***

***recBuff[i] = arr[i];***

***}***

***}***

***else {***

***if (rank == size - 1) {***

***MPI\_Recv(recBuff, N \* (rowsPerProc + rem), MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);***

***}***

***else {***

***MPI\_Recv(recBuff, N \* rowsPerProc, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);***

***}***

***}***

***/\*if (rank == 0) {***

***int k = 13;***

***cout << "Proc#" << k / size << " is the owner of the " << k << "th" << " row" << endl;***

***}\*/***

***/\*at this point, all processes have the rows in their receive buffer\*/***

***//if (rank == size - 1)***

***// for (int i = 0; i < N ; ++i) {***

***// cout << kRow[i] << endl;***

***// }***

***/\*if (rank == size - 1) {***

***int i = 17;***

***int tmpDiff = i - (rowsPerProc \* rank);***

***for (int j = tmpDiff \* N; j < (tmpDiff \* N) + N; ++j) {***

***cout << recBuff[j] << endl;***

***}***

***}\*/***

***double start, end;***

***MPI\_Barrier(MPI\_COMM\_WORLD);***

***start = MPI\_Wtime();***

***int tmpDiff;***

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

***root = int(i / rowsPerProc);***

***if (root >= size)***

***root = size - 1;***

***if (root == rank) {***

***if (root == size - 1) {***

***int rowStart = root \* rowsPerProc;***

***tmpDiff = i - rowStart;***

***int z = 0;***

***for (int j = tmpDiff \* N; j < (tmpDiff \* N) + N; ++j) {***

***kRow[z++] = recBuff[j];***

***}***

***}***

***else {***

***int rowStart = root \* rowsPerProc;***

***tmpDiff = i - rowStart;***

***int z = 0;***

***for (int j = tmpDiff \* N; j < (tmpDiff \* N) + N; ++j) {***

***kRow[z++] = recBuff[j];***

***}***

***}***

***}***

***MPI\_Bcast(kRow, N, MPI\_INT, root, MPI\_COMM\_WORLD);***

***/\*if (i == 1 && rank == 3) {***

***for (int j = 0; j < N; ++j) {***

***cout << kRow[j] << endl;***

***}***

***}\*/***

***int lim;***

***if (rank == size - 1) {***

***lim = (rowsPerProc + rem);***

***}***

***else {***

***lim = rowsPerProc;***

***}***

***//if (rank == 0) {***

***// if (i == 0) {***

***// for (int j = 0; j < N; ++j) {***

***// cout << kRow[j] << "\t";***

***// }***

***// }***

***//}***

***for (int k = 0; k < lim; ++k) {***

***for (int j = 0; j < N; ++j) {***

***int min;***

***if (recBuff[(k \* N) + j] < (recBuff[(k \* N) + i] + kRow[j])) {***

***min = recBuff[(k \* N) + j];***

***}***

***else***

***min = recBuff[(k \* N) + i] + kRow[j];***

***recBuff[(k \* N) + j] = min;***

***/\*if (rank == 0) {***

***cout << "index : " << kRow[j] << " where k = " << i << " and j = " << j << endl;***

***}\*/***

***}***

***}***

***/\*for (int j = 0; j < N; j++) {***

***int min;***

***if (recBuff[j] < (recBuff[i] + kRow[j]))***

***min = recBuff[j];***

***else***

***min = recBuff[i] + kRow[j];***

***recBuff[j] = min;***

***}\*/***

***}***

***MPI\_Barrier(MPI\_COMM\_WORLD);***

***end = MPI\_Wtime();***

***for (int i = 0; i < size; ++i) {***

***if (rank == i) {***

***if (rank != size - 1) {***

***for (int j = 0; j < rowsPerProc; ++j) {***

***//cout << "Row#" << (j + i);***

***for (int k = 0; k < N; ++k) {***

***cout << recBuff[(j \* N) + k] << "\t";***

***}***

***cout << "\n";***

***}***

***}***

***else {***

***for (int j = 0; j < rowsPerProc + rem; ++j) {***

***for (int k = 0; k < N; ++k) {***

***cout << recBuff[(j \* N) + k] << "\t";***

***}***

***cout << "\n";***

***}***

***}***

***}***

***}***

***delete[] arr2D;***

***delete[] recBuff;***

***delete[] kRow;***

***if (rank == 0) { /\* use time on master node \*/***

***cout << "Runtime = " << end - start << "\n";***

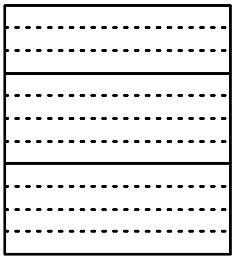
***}***

***MPI\_Finalize();***

***return 0;***

**Design choice(s):**

One design choice, in particular, to highlight here is that the data has been portioned in a “**Rowwise block stripped**” fashion. This is not the ideal way to partition the data and it leads to load imbalancing at times; however, it is one of the simplest ways of portioning the data.

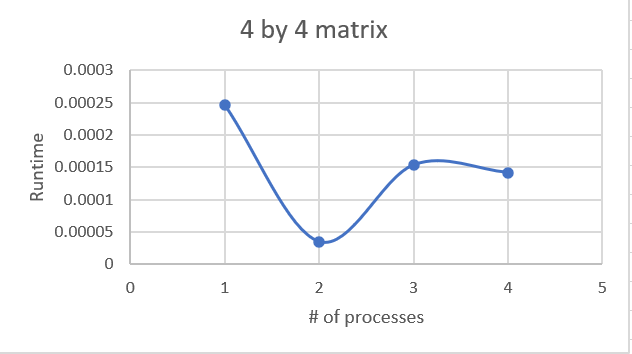


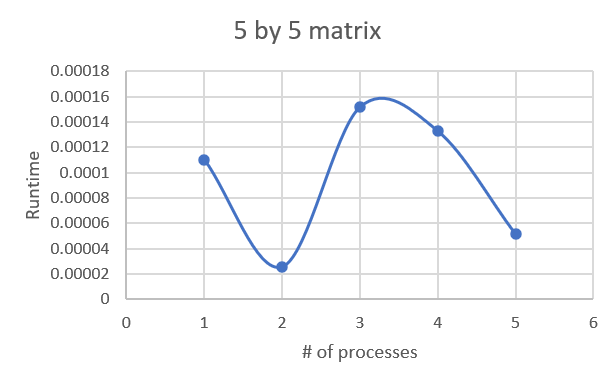
**Major Performance metrics:**

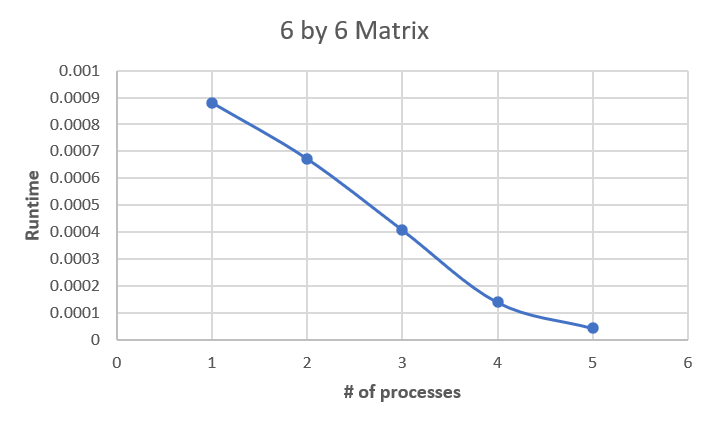
1. **Execution Time**
2. **Speedup**
3. **Efficiency**

**Experimental evaluation**

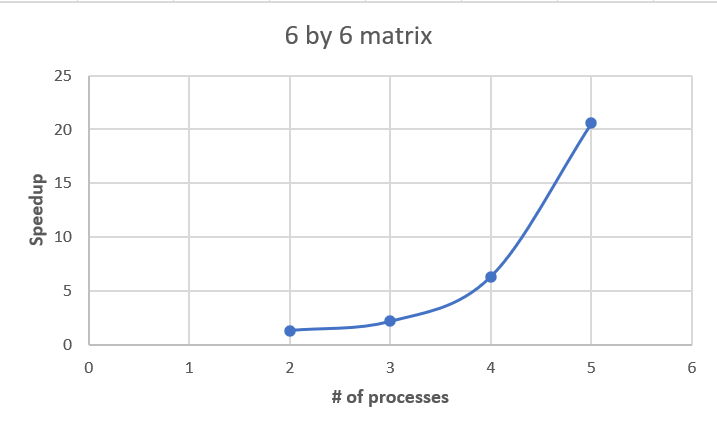
* **Evaluating algorithm on different problem sizes (graphs included)**



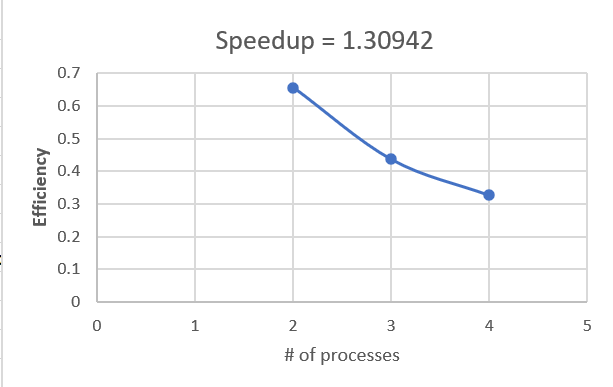


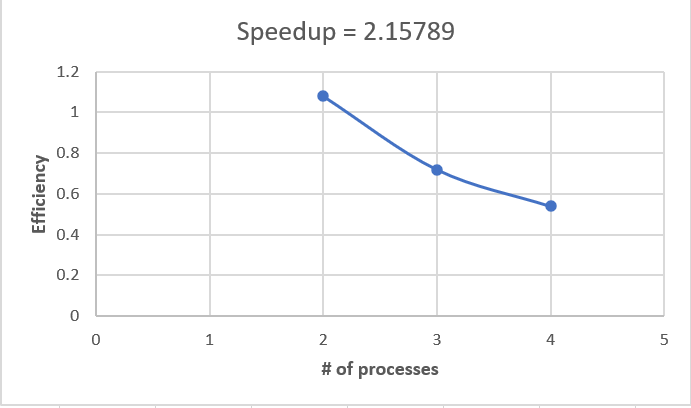


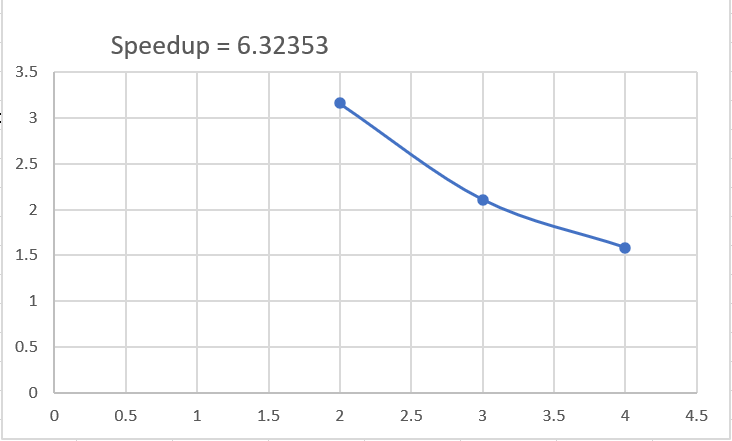
* **Speedup and # of processes for a 6 by 6 matrix (graphs included)**

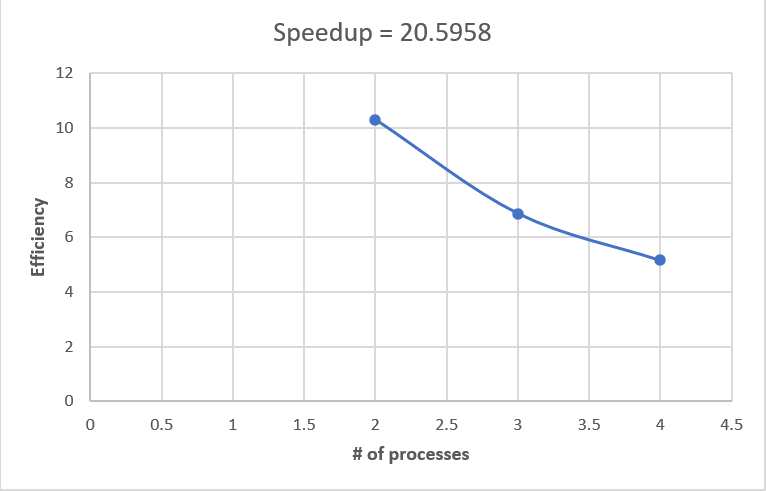


* **Efficiency and speedup (graphs included)**









**Discussion:**

The above performance metrics are indicative of quantitively evaluating of the performance. Firstly, we can observe that there is no linear relationship between the increase in the number of processes and the increase in the speedup for a fixed problem size. This indicates that the performance gains of presented parallel formulation of the algorithm do not scale evenly with the number of processes. This also means that the algorithm is not fully utilizing all the given resources and some of the other factors are limiting its performance as well. One possible issue could be that the load is not being distributed evenly (which is not surprising, considering that the algorithm is using the most basic form 1D data portioning in a “**block row-major fashion**”.

It has also been observed that the algorithm seems to work well on large input-set (for example: a 6 by 6 matrix in the above example). In addition to that, it also seems to perform well when “N” (a single dimension) is an even number. The former case could be because the algorithm is able to take advantage of the additional resources provided by the processes when working on larger inputs, while the later is particular due to the **structure** of the data.   
The increase in the speedup for a large input size is a good sign that the algorithm is able to improve its performance by effectively utilizing its resources.

It can also be observed that increasing the number of processors, for a fixed problem size, can actually end up reducing parallel efficiency. This could be because of the certain overhead associated with using multiple processes (such as communication and synchronization requirements), along with that one big suspect is the division of the problem among the processes – as mentioned above, this algorithm divides the problem into blocks of rows. The division is not always perfect because it doesn’t always account for “**load balancing**”. If the division is imperfect then the last process will have all the extra burden while the rest of the processes may finish up their work and become idle.

**Conclusion:**

While one can argue that in some cases the overhead of managing and coordinating multiple processes can outweigh the benefits of using more processes, resulting in a decrease in parallel efficiency, we have at the same time come to see how the algorithm works well on large input size and how it is performing better than its serialized counterpart, and despite not being able to utilize all the resources effectively, it has still managed to reduce the runtime by a huge degree.