DA2
All Pairs Shortest Path Problem in MPI

-CED17I023
J.Veeren Chandrahas

Table of Contents

1. Introduction:

The shortest path problem is about finding a path between two nodes in a graph such that the path cost is minimized. One example of this problem could be finding the fastest route from one city to another by car, train or airplane.
The Floyd-Warshall algorithm is an algorithm that solves this problem. It works for weighted graphs with positive or negative weights but not for graphs with negative cycles.
It works by comparing all possible paths between all vertex pairs in the graph. A version of the algorithm implemented in the C language can be seen in the figure below.

```c
void ST_APSP(int *mat, const size_t N)
{
    for (int k = 0; k < N; k++)
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
            {
                int i0 = i * N + j;
                int i1 = i * N + k;
                int i2 = k * N + j;
                if (mat[i1] != -1 && mat[i2] != -1)
                {
                    int sum = (mat[i1] + mat[i2]);
                    if (mat[i0] == -1 || sum < mat[i0])
                        mat[i0] = sum;
                }
            }
}
```

The notations used in this figure; "k", "i", "j", "N" and "mat[...]" will be used throughout the report.
For each k all of the current values (mat[i*N,j]) of the matrix is compared to the sum of two other values in the matrix:

mat[k*N,j] + mat[i*N,k]. Once the outer loop has run N times all paths between all vertex pairs have been compared.
The purpose of the lab was to parallelize this algorithm with the use of MPI. This report will show the results of the parallel version that was implemented and explain the design of the parallel algorithm.

2. Results:

To measure the performance of our algorithm, we performed various tests by varying the number of processors and the size of the initial matrix. For simplicity, we have use values of N (N=1200) which are divisible by the numbers of processors (p=2,4,6,8,10,12,14,16,18,20,24).

2.1. Total time:

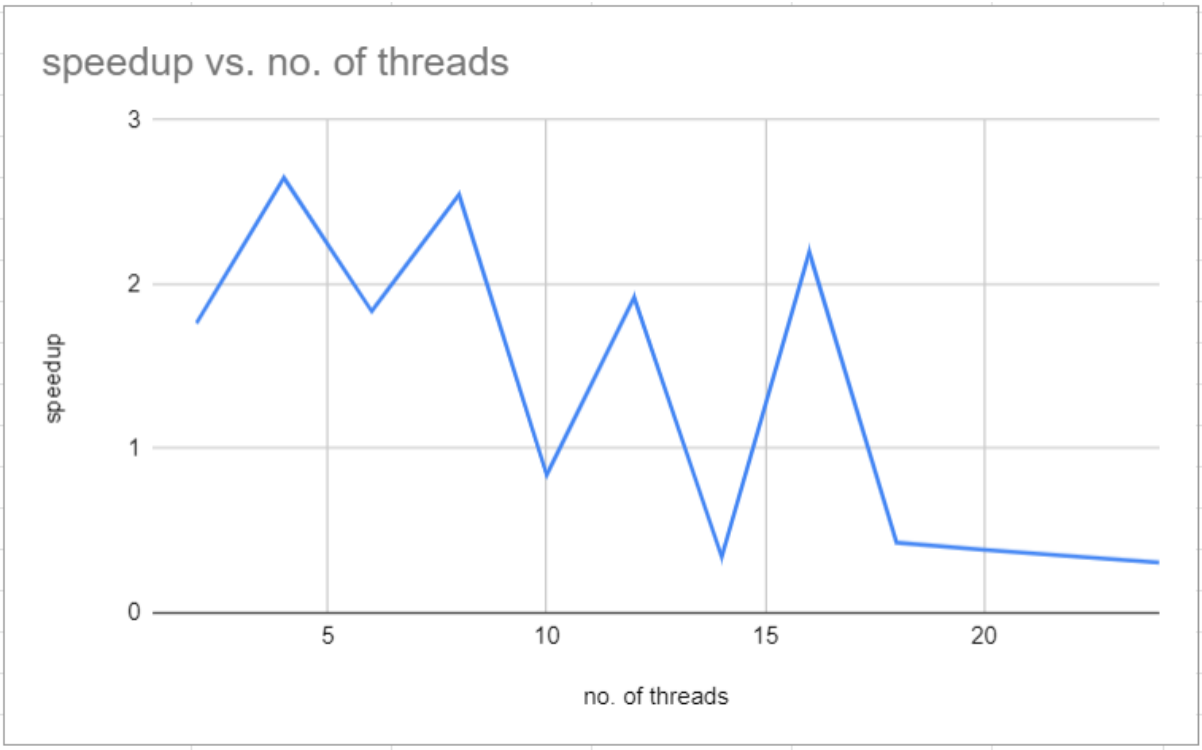
2.1.1. Table:

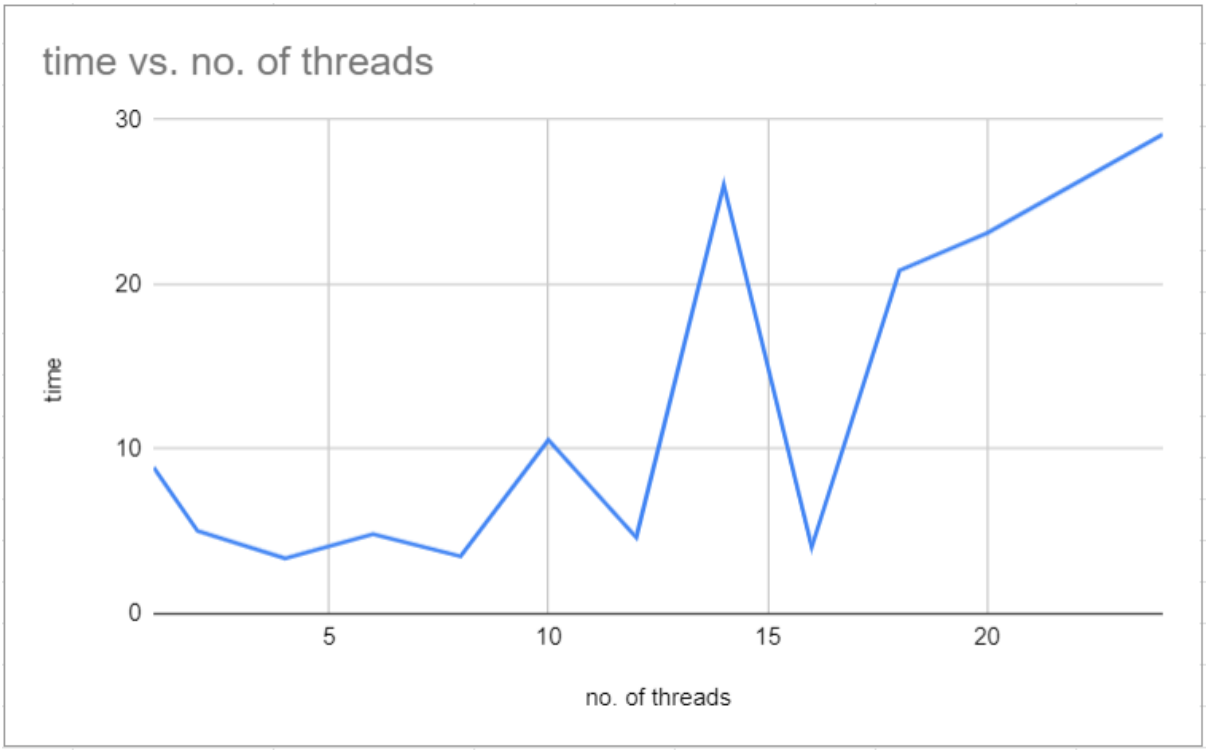
This table summarizes the total execution time of sequential and parallel programs based on the numbers of processors.

no. of processor	time	speedup	parallel fraction
1	8.877685		
2	5.03247	1.764081058	0.8662652482
4	3.352524	2.648060088	0.8298200864
6	4.833789	1.836589268	0.546614934
8	3.488006	2.545203477	0.693833262
10	10.561427	0.8405762782	-0.2107333662
12	4.624904	1.919539303	0.5225909068
14	26.04021	0.3409221738	-2.081930056
16	4.042166	2.196269277	0.5809945874
18	20.83339	0.4261277209	-1.425932748
20	23.114514	0.3840740498	-1.688067981
24	29.097484	0.3051014651	-2.376624164

2.1.2. Graph:

This graph shows the total execution time of programs depending on the size of the initial matrix, N. Moreover, each curve is associated with a different number of processors.





2.1.3. Analyse:

Thanks to this information, we can see that the execution time fluctuates as the number of processors increases.

2.2. Speedup:

Speed Up is defined as the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors.

2.2.1. Table:

This table summarizes the speedup of parallel programs based on the numbers of processors and the size of the initial matrix. This allows us to compare the speed of execution
of a parallel program with a sequential program.

Time taken for 1 thread= 8.877685
Time taken for 4 thread= 3.352524
Speedup =T (1)/T (4) = 2.648060088

Parallelization Factor (f) = (1 - T (4)/T (1))/ (1 - (1/4)) = 0.8298200864

3. Observations:

For each certain iteration (k) of the algorithm the calculations are independent. Calculating the value of mat [i*N, j] will not affect the calculation of mat [i'*N, j'] for a specific k. But the values calculated for the following iterations of k will depend upon the previous iterations. This means that the iterations of k have to be calculated in sequence but the values of the matrix within that iterations can be calculated in sequence.

The implementation was done in two steps. The first version broadcasted the matrix to all processors for each iteration of k. The rows of the matrix were then split up in equal parts and was assigned to different processors. This was distribution was based on the number of processors and their internal

numbering (rank). When all processors had calculated their rows the different matrices from the different processors was merged into one matrix by comparing the different matrices and taking the smallest of each position. This implementation was not much faster than the sequential solution however. This is probably because of the fact that it is expensive to copy the entire matrix to four different processors.

we understand that it is not enough to just partition the calculations, the data has to be partitioned as well. In this new version, we decided to send to each processor only lines that must be calculated by himself and the line « k » which is essential to the calculation.
To do this, we start by calculating the number of lines that will be assigned to each processor by dividing the number of rows of the matrix by the number of processors. Then, we distribute all these lines to different processors thanks to the "MPI_SCATTER" function.
Next, for each iteration (k) of the algorithm, we broadcast the line « k » to all processors, so they can calculate the new values of their lines. When all processors have finished their calculations, we collect all results in the « root » processor thanks to the « MPI_GATHER » function. And we reiterate this program N times.

4.Appendix:

4.a. Execution:

```
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 1 ./a.out 1200
Elasped time = 9161171 usecs
Elasped Parallel time = 8877685 usecs
Rank = 0 Your result is correct.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 2 ./a.out 1200
Elasped time = 9658256 usecs
Elasped Parallel time = 5032470 usecs
Rank = 0 Your result is correct.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 4 ./a.out 1200

Elasped time = 10799809 usecs
Elasped Parallel time = 3352524 usecs
Rank = 0 Your result is correct.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 6 ./a.out 1200
Elasped time = 14592910 usecs
Elasped Parallel time = 4833789 usecs
Rank = 0 Your result is correct.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 8 ./a.out 1200
Elasped time = 18752531 usecs
Elasped Parallel time = 3488006 usecs
Rank = 0 Your result is correct.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 10 ./a.out 1200
Elasped time = 18839562 usecs
Elasped Parallel time = 10561427 usecs
Rank = 0 Your result is correct.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 12 ./a.out 1200
Elasped time = 31383712 usecs
Elasped Parallel time = 4624904 usecs
Rank = 0 Your result is correct.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 14 ./a.out 1200
Elasped time = 19067461 usecs
Elasped Parallel time = 26040210 usecs
ERROR: l[1428000] = 23, r[1428000] = 0
Rank = 0 Your result is wrong.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 16 ./a.out 1200
Elasped time = 41645527 usecs
Elasped Parallel time = 4042166 usecs
Rank = 0 Your result is correct.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 18 ./a.out 1200
Elasped time = 32483127 usecs
Elasped Parallel time = 20833390 usecs
ERROR: l[1425600] = 26, r[1425600] = 0
Rank = 0 Your result is wrong.
veeren@VeerenJ:/mnt/e/ubuntu/wsl/HPC/DA2$ mpirun --mca btl_vader_single_copy_mechanism none -np 20 ./a.out 1200
Elasped time = 42098154 usecs
Elasped Parallel time = 23114514 usecs
```

4.b. Main:

```c
int main(int argc, char **argv)
{

    struct timeval tv1, tv2;
    int rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (argc != 2)
    {
        printf(" Usage: test {N}\n");
        exit(-1);
    }

    //generate a random matrix.
    size_t N = atoi(argv[1]);
    int *mat = (int *)malloc(sizeof(int) * N * N);

    GenMatrix(mat, N);

    //compute the reference result.
    int *ref = (int *)malloc(sizeof(int) * N * N);
    memcpy(ref, mat, sizeof(int) * N * N);
    gettimeofday(&tv1, NULL);
    ST_APSP(ref, N);
    gettimeofday(&tv2, NULL);
```

4.c. Parallel Code:

```c
void MT_APSP(int *mat, const size_t N)
{
    int rank;

    const int root = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int *tmp_buf;
    tmp_buf = malloc(N * N * sizeof(int));

    for (int k = 0; k < N; k++)
    {
        if (rank == root)
        {
            //tmp_k = k;
            for (int i = 0; i < N * N; i++)
                tmp_buf[i] = mat[i];
        }

        //printf("Rank %d broadcasting current matrix...\n", rank);
        MPI_Bcast(tmp_buf, N * N, MPI_INT, root, MPI_COMM_WORLD);

        //printf("Rank %d calculating current matrix...\n", rank);
        functionName(tmp_buf, N, k);

        //printf("Rank %d collecting data with reduce...\n", rank);
        MPI_Reduce(tmp_buf, mat, N * N, MPI_INT, MPI_MIN, root, MPI_COMM_WORLD);
    }
}
```

```c
void MT_SM_APSP(int *mat, const size_t N)
{

    int rank;
    int numprocs;
    int owner;
    const int root = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    int lb = (N / numprocs) * rank;
    int ub = (N / numprocs) * (rank + 1);
    int row_data_size = (ub - lb) * N;
    int nbline = row_data_size / N;
    //printf("Rank : %d row_data_size = %d N= %d numprocs= %d nbline %d \n", rank, row_data_size, N, numprocs, nbline);

    int *tmp_row_buf;
    tmp_row_buf = malloc(row_data_size * sizeof(int));
    int *tmp_k_buf;
    tmp_k_buf = malloc(N * sizeof(int));

    //each processor has their own row
    MPI_Scatter(mat, row_data_size, MPI_INT, tmp_row_buf, row_data_size, MPI_INT, root, MPI_COMM_WORLD);

    for (int k = 0; k < N; k++)
    {
        //with gather
        if (rank == root)
        {
            for (int i = 0; i < N; i++)
                tmp_k_buf[i] = mat[k * N + i];
        }

        //printf("Rank %d broadcasting current row k = %d ...\n", rank, k);
        MPI_Bcast(tmp_k_buf, N, MPI_INT, root, MPI_COMM_WORLD);

        //printf("Rank %d calculating current matrix...\n", rank);
        functionName2(tmp_row_buf, tmp_k_buf, N, k, row_data_size);

        //printf("Rank %d gather...\n", rank);
        //update row k by the owner simplifier apres juste update pas refaire la matrice
        MPI_Gather(tmp_row_buf, row_data_size, MPI_INT, mat, row_data_size, MPI_INT, root, MPI_COMM_WORLD);
    }
```

4.d. Useful Functions:

```c
void functionName(int *mat, const int N, const int K)
{

    int rank, numprocs;

    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int lb = (N / numprocs) * rank;
    int ub = (N / numprocs) * (rank + 1);
    //printf("rank = %d K = %d N = %d lb = %d ub = %d \n", rank, K, N, lb, ub);

    for (int i = lb; i < ub; i++)
    {
        for (int j = 0; j < N; j++)
        {
            int i0 = i * N + j;
            int i1 = i * N + K;
            int i2 = K * N + j;
            if (mat[i1] != -1 && mat[i2] != -1)
            {
                int sum = (mat[i1] + mat[i2]);
                if (mat[i0] == -1 || sum < mat[i0])
                    mat[i0] = sum;
            }
        }
    }
}

void functionName2(int *mat, int *linek, const int N, const int K, const int row_data_size)
{

    int rank, numprocs;
    int nbline = row_data_size / N;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (int i = 0; i < nbline; i++)
    {
        for (int j = 0; j < N; j++)
        {
            int i0 = i * N + j;
            int i1 = i * N + K;
            //int i2 = K*N + j;
            //int i2 = j;
            if (mat[i1] != -1 && linek[j] != -1)
            {
                int sum = (mat[i1] + linek[j]);
                if (mat[i0] == -1 || sum < mat[i0])
                    mat[i0] = sum;
            }
        }
    }
}
```