Lab 1 report – team 5

All Pairs Shortest Path Problem in MPI

Summary

1. Introduction 3
2. Results 4
2.1. Total time 4
2.1.1. Table 4
2.1.2. Graph 4
2.1.3. Analyse 4
2.2. Speedup 5
2.2.1. Table 5
2.2.2. Graph 5
2.2.3. Analyse 5
3. Discussion 6
3.1. First version 6
3.2. Second version 6
4. Appendix 7
4.1. Main 7
4.2. Parallel program 8
4.3. Calculation lines 8
4.4. Function prototype (MatUtil.h) 9

Christopher BELINGUIER

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.

```
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=1200,2400,4800) which are exactly divisible by the numbers of processors (p=4,6,8).

2.1. Total time

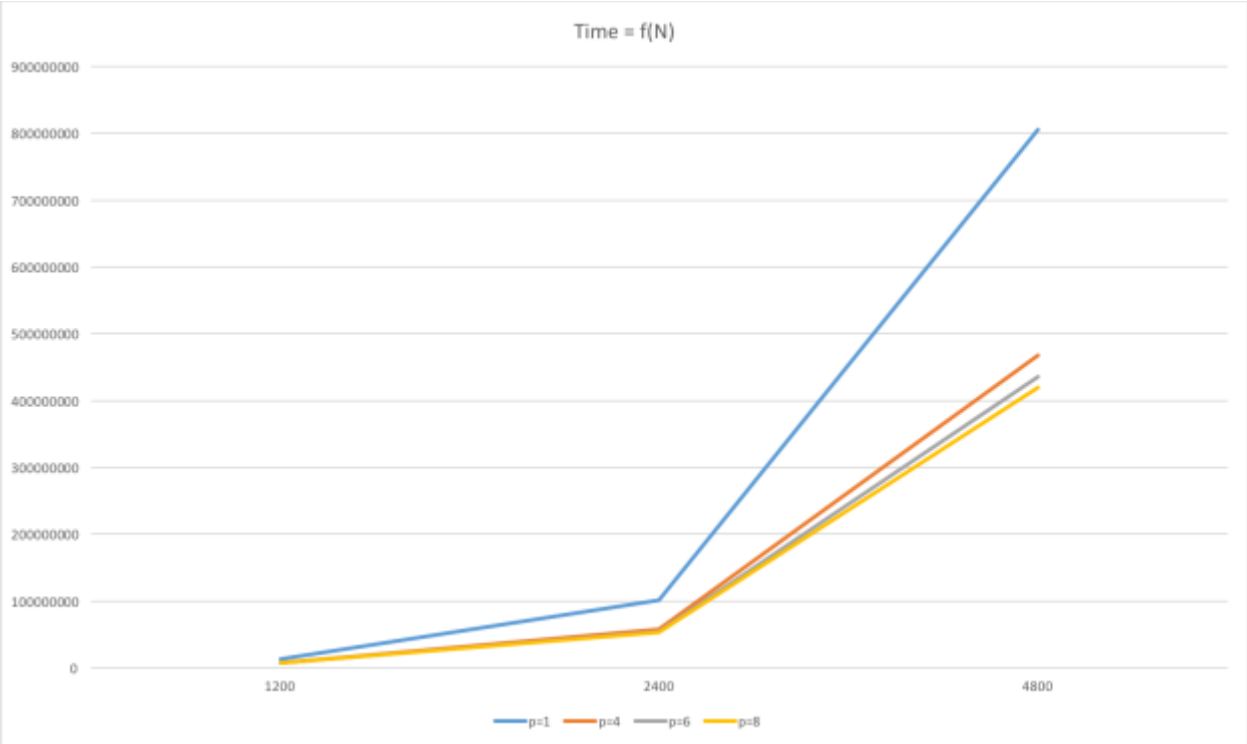
2.1.1. Table

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

Processor/N	1200	2400	4800
1 (sequential)	12591376	101182372	804848342
4	7588517	58089223	467461122
6	7267530	54362854	436098764
8	7002088	52922786	419617762

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 processor.



2.1.3. Analyse

Thanks to this information, we can see that the execution time grows when the matrix size grow and the number of processors decreases.

2.2. Speedup

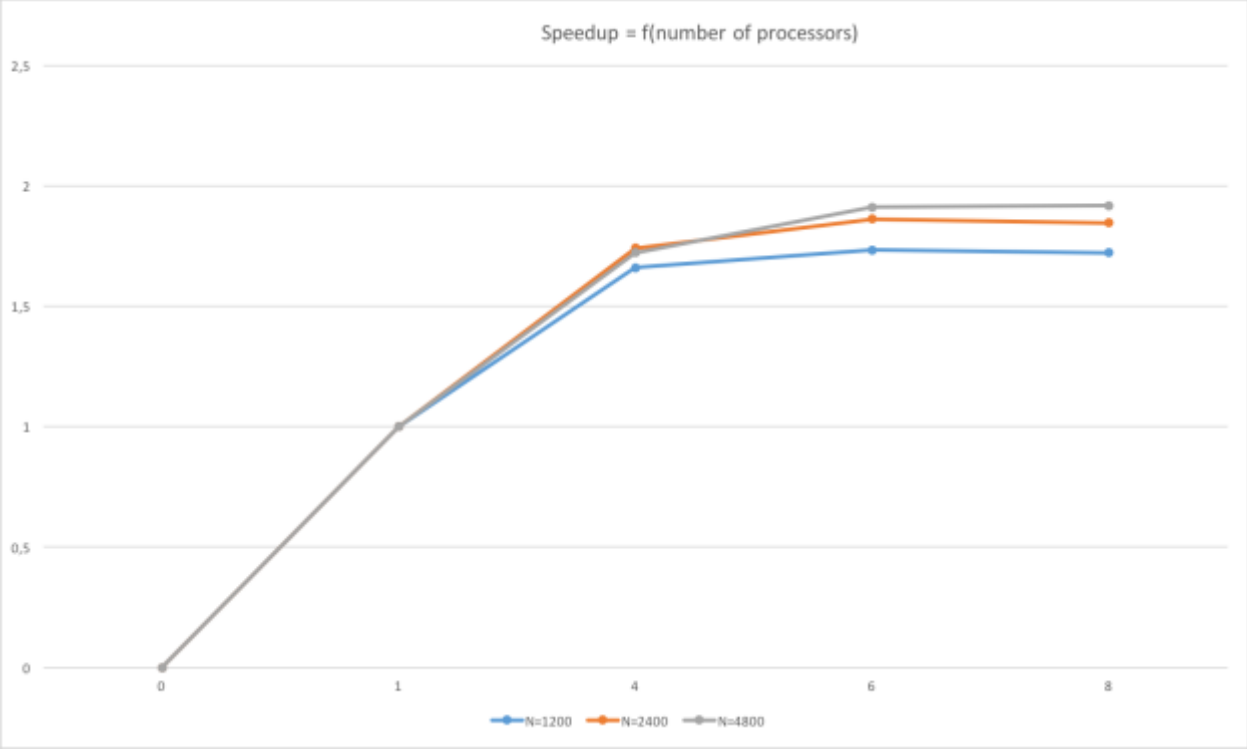
2.2.1. Table

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. In fact : $Speedup = \frac{time\ sequential}{\text{time parallel}}$

Speedup=f(p)	1200	2400	4800
1 (sequential)	1	1	1
4	1,659267021	1,741844128	1,721743914
6	1,732552325	1,861240986	1,845564373
8	1,798231613	1,911886725	1,918051176

2.2.2. Graph

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



2.2.3. Analyse

Thanks to this information, we can see that the gain in execution speed is more important when the matrix size and the number of processors grow.

3. Discussion

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 has to be calculated in sequence but the values of the matrix within that iterations can be calculated in sequence.

3.1. First version

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 was 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.

3.2. Second version

After this first version, we understand that it is not enough to just partition the calculations, the data has to be partitioned aswell. 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 brodcast 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

4.1. Main

```
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);
    if(rank == 0)
    {
        printf("Elasped time sequential = %ld usecs\n",
            (tv2.tv_sec-tv1.tv_sec)*1000000+tv2.tv_usec-tv1.tv_usec);
    }

    //compute your results
    int *result = (int*)malloc(sizeof(int)*N*N);
    memcpy(result, mat, sizeof(int)*N*N);
    //replace by parallel algorithm
    gettimeofday(&tv1, NULL);
    MT_SM_APSP(result, N);
    gettimeofday(&tv2, NULL);
    if(rank == 0)
    {
        printf("Elasped time parallel = %ld usecs\n",
            (tv2.tv_sec-tv1.tv_sec)*1000000+tv2.tv_usec-tv1.tv_usec);
    }

    //compare your result with reference result
    if(rank == 0)
    {
        if(CmpArray(result, ref, N*N))
            printf("Rank = %d Your result is correct.\n", rank);
        else
            printf("Rank = %d Your result is wrong.\n", rank);
    }

    MPI_Finalize();

}
```

4.2. Parallel program

```
/*
Parallel (Multiple Threads) APSP on CPU.
distributing parts of the matrix to different threads
*/
void MT_SM_APSP(int *mat, const size_t N)
{
    int rank;
    int numprocs;
    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 *tmp_row_buf;
    tmp_row_buf = malloc(row_data_size*sizeof(int));
    int *tmp_k_buf;
    tmp_k_buf = malloc(N*sizeof(int));

    //distribute to each processor their own rows
    MPI_Scatter(mat, row_data_size, MPI_INT, tmp_row_buf, row_data_size, MPI_INT, root, MPI_COMM_WORLD);

    //iteration of k in sequential
    for (int k = 0; k < N; k++)
    {
        //put the row k in the "tmp_k_buf" buffer
        if (rank == root)
        {
            for (int i = 0; i < N; i++)
                tmp_k_buf[i] = mat[k*N + i];
        }

        //broadcast current row k (tmp_k_buf) to all processors
        MPI_Bcast(tmp_k_buf, N, MPI_INT, root, MPI_COMM_WORLD);

        //each processor calculate their own rows of the matrix
        computeRows(tmp_row_buf, tmp_k_buf, N, k, row_data_size);

        //collect all the rows in the root processor
        MPI_Gather(tmp_row_buf, row_data_size, MPI_INT, mat, row_data_size, MPI_INT, root, MPI_COMM_WORLD);
    }
}
```

4.3. Calculation lines

```
/*
Compute rows of the matrix with the row k
*/
void computeRows(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;
            if (mat[i1] != -1 && linek[j] != -1)
            {
                int sum = (mat[i1] + linek[j]);
                if (mat[i0] == -1 || sum < mat[i0])
                    mat[i0] = sum;
            }
        }
    }
}
```

4.4. Function prototype (MatUtil.h)

```c
#ifndef MATUTIL_H
#define MATUTIL_H
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#include <mpi.h>

/// Generate a random matrix.
//
// Parameters:
// int *mat - pointer to the generated matrix. mat should have been
//            allocated before callling this function.
// const int N - number of vertices.
void GenMatrix(int *mat, const size_t N);

/// Compare the content of two integer arrays. Return true if they are
// exactly the same; otherwise return false.
//
// Parameters:
// const int *l, const int *r - the two integer arrays to be compared.
// const int eleNum - the length of the two matrices.
bool CmpArray(const int *l, const int *r, const size_t eleNum);

/// Compute APSP using single thread.
//
// Parameters:
// int *mat - the input matrix storing the distance values between vertices.
//            mat should have been allocated before callling this function.
//            The result will be directed stored in mat.
// const int N - the number of vertices.
void ST_APSP(int *mat, const size_t N);

// Parallel (Multiple Threads) APSP on CPU. distributing parts of the matrix to different threads
void MT_SM_APSP(int *mat, const size_t N);

// Compute rows of the matrix with the row k
void computeRows(int *mat, int *linek, const int N, const int K, const int row_data_size);

#endif
```