Homework 3

José Javier González Ortiz

October 14, 2015

1 Algorithm

Given the problem in the assignment we can show that although the initial matrix is symmetrical, after the first iteration we will lose the symmetry so we have to calculate all the values in the matrix.

Given more processors we can split the work to be done. Since we want the divisions to be as cohesive as possible to minimize the communication, we can opt for two ways to do the domain decomposition. Both involve *Halo Swapping* as it renders the most effective strategy in this case since every element depends on the ones above, below and to the right.

First the serial time for this algorithm will be SerTime $= \Theta(n^2)$ since we will do a constant amount of iterations during which it will update the n^2 elements of the matrix. Now, an ideal parallel algorithm without communication we would get ParTime $= \Theta(n^2/p)$, however any real algorithm will have to communicate so the SpeedUp won't be p.

If we do the division per processors in one dimension, the Parallel time will be as follows: ParTime $= \Theta(n^2/p) + \Theta(n/p) + \Theta(n) = \Theta(n^2/p) + \Theta(n)$. However, if we do the partition in both dimensions (we are allowed to do it since p is given to be a perfect square) the parallel time will be ParTime $= \Theta(n^2/p) + \Theta(n/\sqrt{p})$ which is clearly better than the naive approach shown before.

2 Analysis

The chosen domain decomposition renders the processor communication in a 2D grid. To better the communication involved there should be communication between the first row of processors and the last row, to make use of the modular properties of the algorithm. Therefore, the sought topology is cylindrical.

Given the partition shown we will have for each processor p with coordinates in the processor grid (x, y). The size of the matrix will be $h \times w$ where h and w are:

$$h = \begin{cases} \lceil \frac{n}{\sqrt{p}} \rceil & \text{if } x < \sqrt{p} - 1 \\ n \mod \lceil \frac{n}{\sqrt{p}} \rceil & \text{if } x = \sqrt{p} - 1 \end{cases} \tag{2.1}$$

$$w = \begin{cases} \lceil \frac{n}{\sqrt{p}} \rceil & \text{if } y < \sqrt{p} - 1 \\ n \mod \lceil \frac{n}{\sqrt{p}} \rceil & \text{if } y = \sqrt{p} - 1 \end{cases} \tag{2.2}$$

Apart from this $h \times w$ matrix, each processor will have to store three halos corresponding to the following values

- **Upper Halo** of size $1 \times w$, corresponding to the lower row of the processor above (note that $(\sqrt{p}-1, y)$) is above $(0, y)$ in a cylindrical topology). This values are needed when $i_{\text{local}} = 0$ and i'' looks for the value above.

- **Lower Halo** of size $1 \times w$, corresponding to the upper row of the processor below. This values will be needed hen $i_{\text{local}} = \lceil \frac{n}{\sqrt{p}} \rceil - 1$ and i' looks for the value below.
- **Right Halo** of size $h \times 1$, corresponding to the leftmost column of the processor to the right. This values will be needed when $j_{\text{local}} = \lceil \frac{n}{\sqrt{p}} \rceil - 1$ and j' will ask for the value to the right. The processors with coordinates $(x, \sqrt{p} - 1)$ will not use this halos since the values of the matrix with coordinates $(i, n-1)$ do not update.

The communication will be composed in three shifts.

1. **Upward Modular Shift** Each processor (x, y) sends its upper row to the processor above $(x - 1 \mod \sqrt{p}, y)$ and receives from the one below $(x + 1 \mod \sqrt{p}, y)$ storing the values in the Lower Halo. Since this is a cylindrical topology this operation involves all the processors sending and receiving at the same time.
2. **Downward Modular Shift** Similarly, this is the reverse operation. Each processor (x, y) sends its lower row to the processor below $(x+1 \mod \sqrt{p}, y)$ and receives from the one above $(x-1 \mod \sqrt{p}, y)$ storing the values in the Upper Halo.
3. **Leftward Shift** This involves every processor (x, y) sending its leftmost column to the one on their left $(x, y - 1)$ side and receiving from the one on their right $(x, y + 1)$ and storing th values in their Right Halo. Note than this an open loop and the processors $(x, 0)$ won't send any information and the processors $(x, \sqrt{p} - 1)$ won't receive any information either.

From this we can gather that the communication time is $O(\max(h, w))$ since we are swapping three halos per processor every time the processors communicate. Since $\max(h, w) = \lceil \frac{n}{\sqrt{p}} \rceil$ the time involved in the communication will be $O(\frac{n}{\sqrt{p}})$.

Since the update involved modifying the whole matrix it will take time $O(h \times w)$ per processor. Since $h, w \leq \lceil \frac{n}{\sqrt{p}} \rceil$ we can express it as $O(h \times w) = O(n^2/p)$.

Finally the whole parallel time will be

$$\text{ParTime} = \Theta(n^2/p) + \Theta(n/\sqrt{p}) \tag{2.3}$$

The SpeedUp will be

$$\text{SpeedUp} = \frac{\text{SerTime}}{\text{ParTime}} = \frac{\Theta(n^2)}{\Theta(n^2/p) + \Theta(n/\sqrt{p})} \tag{2.4}$$

$$\text{SpeedUp} \to p \qquad \text{as} \qquad n \to \infty \tag{2.5}$$

And the efficiency:

$$\text{Efficiency} = \frac{\text{SerTime}}{p \cdot \text{ParTime}} = \frac{\Theta(n^2)}{\Theta(n^2) + \Theta(n\sqrt{p})} \tag{2.6}$$

$$\text{Efficiency} \to 1 \qquad \text{for} \qquad p = O(n^2) \tag{2.7}$$

3 Implementation

To implement the algorithm we have used the partition described above and used a cylindrical virtual topology where the row processors are wrapped around but the column processors are not. Furthermore it has been used `MPI_SENDRECV` for the communication, using a derived datatype for the column communication.

The particular aspects of the whole implementation can be seen in Code A.1

4 Results

The results from the timings are shown in the Table 4.1. The Verification values for the different matrix sizes are shown in the Table 4.2.

n	p	average	min	max
1000	1	5.71552	5.34630	5.83059
2000	1	22.82041	21.18143	23.31813
1000	4	1.46049	1.41766	1.49389
2000	4	6.04183	5.67908	7.35377
1000	16	0.46770	0.38920	0.54539
2000	16	1.77513	1.50135	2.04136
1000	36	0.30457	0.22946	0.37673
2000	36	1.01406	0.68300	1.30123

Table 4.1: Results in seconds

n	sum	min
1000	4283810.871966	-892.718330
2000	17631642.148465	-1800.724437

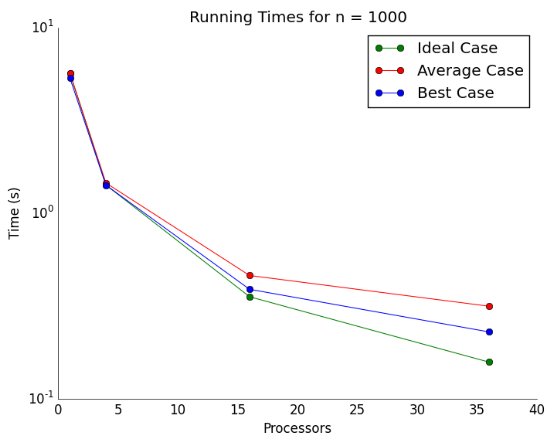
Table 4.2: Verification Values

To ease the understanding of the data we can plot different relations between the data. In figures 4.1a and 4.1b we can see the timing results in a log scale for $n = 1000$ and $n = 2000$. Defining $\text{SerTime}(n) = \text{ParTime}(n, 1)$ and calculating the ideal parallel time as $\text{SerTime}/p$ we get some interesting results. As we can see the curves match up quite nicely, being the minimum time better than the average one, which seems logical. The deviation for greater p seems larger, however we are using a log scale so in fact it is not as big of a difference. Finally, we can see that for $n = 2000$ the minimum case does better $n = 1000$.

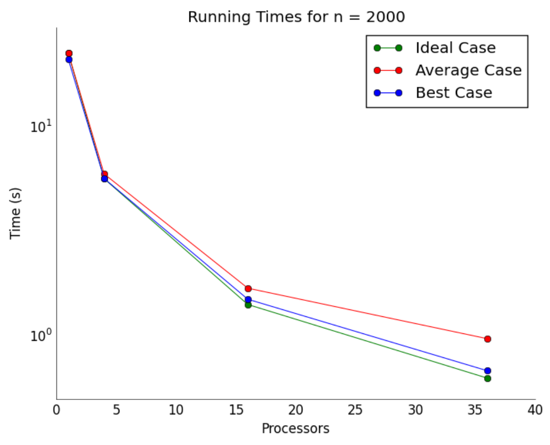
Now, to analyze both SpeedUp and Efficiency, we can see the plots shown in Figures 4.2a and 4.2b, in which we can find the ideal case, for $n = 1000$ and for $n = 2000$. In this plots an average of the minimum cases has been used to avoid deviating the data with longer timings.

Analyzing the Figures we can see that we have achieved almost linear SpeedUp as the Equation (2.5) predicted. As n goes from 1000 to 2000 the SpeedUp improves verifying our predictions. For the Efficiency, Equation (2.7) predicted constant efficiency as n grew larger. Figure 4.2b confirms this, with $n = 2000$ having a behavior much closer to ideal efficiency than $n = 1000$.

Finally, it is important to notice that MPI communication does not behave as the theoretical models and therefore the results will not match perfectly to the model. Nevertheless, we have achieved successful results in the time needed to solve the problem.

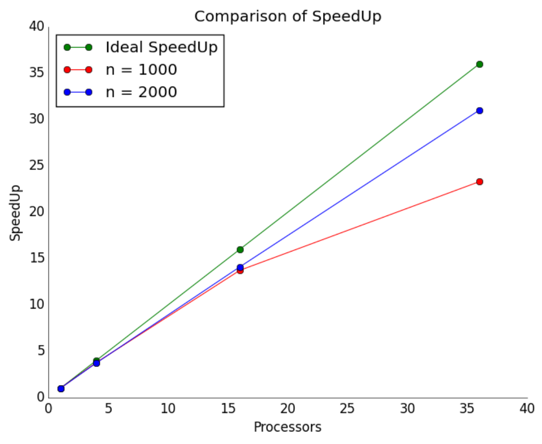


(a) $n = 1000$

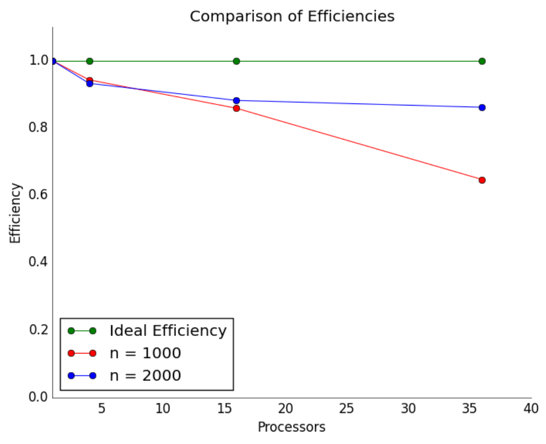


(b) $n = 2000$

Figure 4.1: Logarithmic plot of times for different number of processors



(a) Speed-Up



(b) Efficiency

Figure 4.2: Speed-Up and Efficiency plots for the ideal case, $n = 1000$ and $n = 2000$

A Code

```c
#include <mpi.h>
#include <ctime>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define ITERATIONS 10

#define LEFT_COLUMN A[1][0]
#define RIGHT_HALO A[1][width]
#define UPPER_ROW A[1][0]
#define UPPER_HALO A[0][0]
#define LOWER_ROW A[height][0]
#define LOWER_HALO A[height+1][0]

double** new_matrix(int height, int width);
void delete_matrix(double** matrix);

void initialize_matrix(double** A, int size, int height, int width, int* coords);
void transform_matrix(double** A,double** B, int height, int width, int* coords);
void print_matrix(double** A, int height, int width);

char* gettime(char *buffer);

int main(int argc, char  *argv[])
{
    //////////////// VARIABLES ////////////////
    char *s;

    double **A, **B;
    int n, m, p, sqrt_p, height, width;
    int rank, coords[2];

    MPI_Comm matrix_comm;
    static int dims [2];
    int periods [2] = { 1, 0 };
    int reorder = 1;
    int p_up, p_down, p_right, p_left;

    double local_sum=0.0, local_min;
    double starttime, endtime;

    double global_sum=0.0, global_min;

    char buffer[80];

    ///////////////// INIT  /////////////////

    MPI_Init (&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    sqrt_p = (int)sqrt(p);
    dims[0] = sqrt_p;
    dims[1] = sqrt_p;

    MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,reorder,&matrix_comm);

    MPI_Comm_rank(matrix_comm,&rank);
    MPI_Cart_coords(matrix_comm, rank, 2, coords);

    MPI_Datatype column;
    // The dimensions are reversed becuase MPI_Cart creates the matrix by columns and
we want it by rows
    MPI_Cart_shift(matrix_comm, 1, -1, &p_right, &p_left);
    MPI_Cart_shift(matrix_comm, 0, -1, &p_down, &p_up);

    //////////////// PROGRAM ////////////////

    n = strtol(argv[1], &s, 10);

    m = ceil(float(n) / sqrt_p);
    height = coords[0] != sqrt_p-1 ? m : n - m*(sqrt_p-1);
    width  = coords[1] != sqrt_p-1 ? m : n - m*(sqrt_p-1);

    MPI_Type_vector (height, 1, width+1, MPI_DOUBLE, &column);
    MPI_Type_commit (&column);

    A = new_matrix(height+2,width+1);
    B = new_matrix(height+2,width+1);

    initialize_matrix(A,m,height,width,coords);

    MPI_Barrier(matrix_comm);
    if(rank == 0){
        starttime = MPI_Wtime();
    }

    for (int i = 0; i < ITERATIONS; ++i)
    {
        //// HALO SWAPPING ////
        // Up
        MPI_Sendrecv(&( UPPER_ROW ), width, MPI_DOUBLE, p_up, 0, &( LOWER_HALO ),
width, MPI_DOUBLE, p_down, 0, matrix_comm, MPI_STATUS_IGNORE);
        // Down
        MPI_Sendrecv(&( LOWER_ROW ), width, MPI_DOUBLE, p_down, 0, &( UPPER_HALO ),
width, MPI_DOUBLE, p_up, 0, matrix_comm, MPI_STATUS_IGNORE);
        // Left
        MPI_Sendrecv(&( LEFT_COLUMN ), 1, column, p_left, 0, &( RIGHT_HALO ), 1,
column, p_right, 0, matrix_comm, MPI_STATUS_IGNORE);

        if(coords[1] != sqrt_p-1){
            transform_matrix(A,B,height,width,coords);
        }else{
            transform_matrix(A,B,height,width-1,coords);
        }

    }

    //////////////// VERIFICATION ////////////////

    local_min = A[1+0][0];
    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
                local_sum += fabs( A[1+i][j]) ;
                local_min = A[1+i][j] < local_min ? A[1+i][j] : local_min;
            }
        }

    MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, matrix_comm);
    MPI_Reduce(&local_min, &global_min, 1, MPI_DOUBLE, MPI_MIN, 0, matrix_comm);

    ////////////// FINALIZE ////////////////
    if(rank == 0){
        endtime = MPI_Wtime();
        FILE *f = fopen("Results.txt", "a");
        if(f != NULL)
        {
            fprintf(f, "%s\n", gettime(buffer));
            fprintf(f,"%dx%d matrix\n",n,n);
            fprintf(f,"%d processors\n",p);
            fprintf(f,"Sum : %f\n", global_sum );
            fprintf(f,"Min : %f\n", global_min );
            fprintf(f,"Time : %f seconds\n", endtime-starttime);
            fprintf(f,"\n====================\n");
        }
    }

    delete_matrix(A);
    delete_matrix(B);

    MPI_Finalize();

    return 0;
}

double** new_matrix(int height, int width)
{
    double** matrix;
    matrix = new double*[height];
    matrix[0] = new double[height * width];
    for (int i = 1; i < height; i++)
        matrix[i] = matrix[i-1] + width;
    return matrix;
}

void delete_matrix(double** matrix)
{
    delete[] matrix[0];
    delete[] matrix;
}

void initialize_matrix(double** A, int size, int height, int width, int* coords)
{
    int Origin_x = coords[0]*size;
    int Origin_y = coords[1]*size;
    for (int i_G = Origin_x, i = 0 ; i_G < Origin_x + height ; i_G++, i++)
    {
        for (int j_G = Origin_y, j = 0 ; j_G < Origin_y + width ; j_G++, j++)
        {
            if(coords[0] == coords[1] && i == j){
                A[1+i][j] = i_G*sin(sqrt(i_G));
            }else{
                A[1+i][j] = pow(i_G+j_G,1.1);
            }
        }
    }

    return;
}

void transform_matrix(double** A,double** B, int height, int width, int* coords)
{
    double x;
    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
            if( ! (coords[0] == coords[1] && i == j) )
            {
                x = 0.0;
                for(int k = 1 ; k <= 10 ; k++){
                    x += pow( fabs( 0.5 + A[1+i+1][j] ) , 1.0/double(k));
                    x -= pow( fabs( A[1+i-1][j] ) , 1.0/double(k+1)) * pow( fabs( A[1+
    i][j+1] ) , 1.0/double(k+2));
                }
                x = x < 10.0 ? x : 10.0;
                x = x > -10.0 ? x : -10.0;
                B[1+i][j] = x;
            }else{
                B[1+i][j] = A[1+i][j];
            }
        }
    }
    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
            A[1+i][j] = B[1+i][j];
        }
    }
    return;
}

void print_matrix(double** A, int height, int width)
{
    for (int i = 0; i < height+2; i++)
    {
        for (int j = 0; j < width+1; j++)
        {
            printf("%f ", A[i][j]);
        }
        printf("\n");
    }
}

char* gettime(char* buffer)
{
    time_t rawtime;
    struct tm * timeinfo;

    time (&rawtime);
    timeinfo = localtime(&rawtime);

    strftime(buffer,80,"%Y-%m-%d %I:%M:%S",timeinfo);
    std::string str(buffer);

    return buffer;
}
```

Listing A.1: C++ code for the matrix algorithm