Saint Petersburg National Research University of Information Technologies,
Mechanics and Optics (ITMO University)
Faculty of Informational Technologies and Programming

Report

about laboratory work № 2

«Matrix multiplication»

Student

Voronin Victor J4132c

(Surname, initials) Group

Saint-Petersburg, 2020

1 GOAL OF LABORATORY WORK

Calculate multiply of two matrixes like follow:

$$c_{i,j} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}\,, 0 \le i \le m, 0 \le j \le l$$

where m and l are some integer numbers.

2 TASK DEFINITION

- Implement serial version of the matrix multiplication;
- Implement the parallel algorithm#1 of matrix multiplication;
- Implement the parallel algorithm#2 of matrix multiplication using MPI Derived Data Types (define and use columns;
- Define speedup;

3 BRIEF THEORY

To solve this problem it is used serial and parallel implementations with OpenMP and MPI (Message Passing Interface).

Time of execution is measured five time with std::chrono::high_resolution_clock from library <chrono> to get average time spent for 5 attempts.

Formula showed above was realized in as parallel so serial versions programs.

Serial version of program was enacted applying hush from third loop “for”. Code’s details are presents in appendix 1.

To calculate with several threads, it uses MPI library. With that, it divides the first matrix into separate rows – evenly between worker and it will send to each worker from the root equal part of the first matrix. The second matrix is send entire. The variable "offset" is used to indicate the beginning of each transmitted part.

Each process make calculation on its part of the matrix and as fast as it can sent back piece of result matrix to the root. These piece is joined according to value of variable “offset”. To reason, each worker can calculate not only one row of the first matrix, when forming the result matrix the variable “rows” is used also. Code is presented in appendix 2.

Additional, it can use MPI Derived Data Types. Determine vector MPI Data type with following parameter: count – N that is equal to number of columns of the first matrix; blockleight – N that is equal to number of rows of the first matrix; 0 – strida.

After perform a few corrections of code from appendix 2, it get version program with using Derived Data Types. Its code can be following in appendix 3.

4 RESULT AND EXPERIMENTS

The use of all of the above methods has been summarized in following table (see table 1-4).

Table 1 – Time measurement for every versions of program (2 threads)

Order of matrix	Avr time spent for Serial version, µs	MPI		MPI with Derived data types	
		Avr time, µs	Speedpup	Avr time, ms	Avr time, µs
10	200.12	345	0.58	690.3	0.50
60	799.82	1621.6	0.49	3830.0	0.42
110	4600.26	6787.4	0.68	8636.6	0.79
160	14000.8	16794.0	0.83	22210.5	0.76
210	28399.3	39960.6	0.71	46920.7	0.85
260	63956.8	67975.8	0.94	88264.4	0.77
310	100882	114468.6	0.88	152472.7	0.75

360	159680	169976.0	0.94	233516.0	0.73
410	232900	261506.4	0.89	451602.3	0.58
500	473432	440790.8	1.07	821194.0	#REF!
600	815721	837323.7	0.97	1320426.7	1.02
700	1343760	1295580.0	1.04	2127990.0	0.98
800	2376680	1921893.3	1.24	3050973.3	0.90
900	3580550	2770576.7	1.29	4436093.3	0.91
1000	5374280	3690210.0	1.46	6310460.0	0.58

Table 2 – Time measurement for every versions of program (4 threads)

Order of matrix	Avr time spent for Serial version, μs	MPI		MPI with Derived data types	
		Avr time, μs	Speedpup	Avr time, ms	Avr time, μs
10	200.12	200.12	948.7	0.33	614.0
60	799.82	799.82	2384.0	0.19	4182.7
110	4600.26	4600.26	5822.0	0.79	9139.3
160	14000.8	14000.8	17420.0	0.80	31650.3
210	28399.3	28399.3	37287.0	0.76	65200.7
260	63956.8	63956.8	58205.7	1.10	97917.3
310	100882	100882	96505.0	1.05	135771.7
360	159680	159680	113010.5	1.41	200059.3
410	232900	232900	187841.3	1.24	288276.8
500	473432	473432	309460.7	1.53	470503.0
600	815721	815721	491318.7	1.66	725338.3
700	1343760	1343760	754386.7	1.78	1230380.0
800	2376680	2376680	1039985.3	2.29	1693763.3
900	3580550	3580550	1466810.0	2.44	2393613.3
1000	5374280	5374280	2025920.0	2.65	3218690.0

Table 3 – Time measurement for every versions of program (8 threads)

Order of matrix	Avr time spent for Serial version, μs	MPI		MPI with Derived data types	
		Avr time, μs	Speedpup	Avr time, μs	Speedpup
10	200.12	1095.3	0.18	1109.0	0.18
60	799.82	2615	0.31	3275.7	0.24
110	4600.26	4981.3	0.92	8861.7	0.52
160	14000.8	10780.3	1.30	15579.3	0.90
210	28399.3	23238.3	1.22	28910.0	0.98
260	63956.8	39177.0	1.63	49241.7	1.30
310	100882	36176.7	2.79	76774.3	1.31
360	159680	55731.0	2.87	106250.7	1.50
410	232900	77129.3	3.02	161603.0	1.44
500	473432	135742.7	3.49	267039.7	1.77
600	815721	236288.3	3.45	463486.0	1.76
700	1343760	368063.3	3.65	716954.7	1.87
800	2376680	580893.0	4.09	1054696.7	2.25
900	3580550	847220.0	4.23	1518450.0	2.36
1000	5374280	1174010.0	4.58	1957230.0	2.75

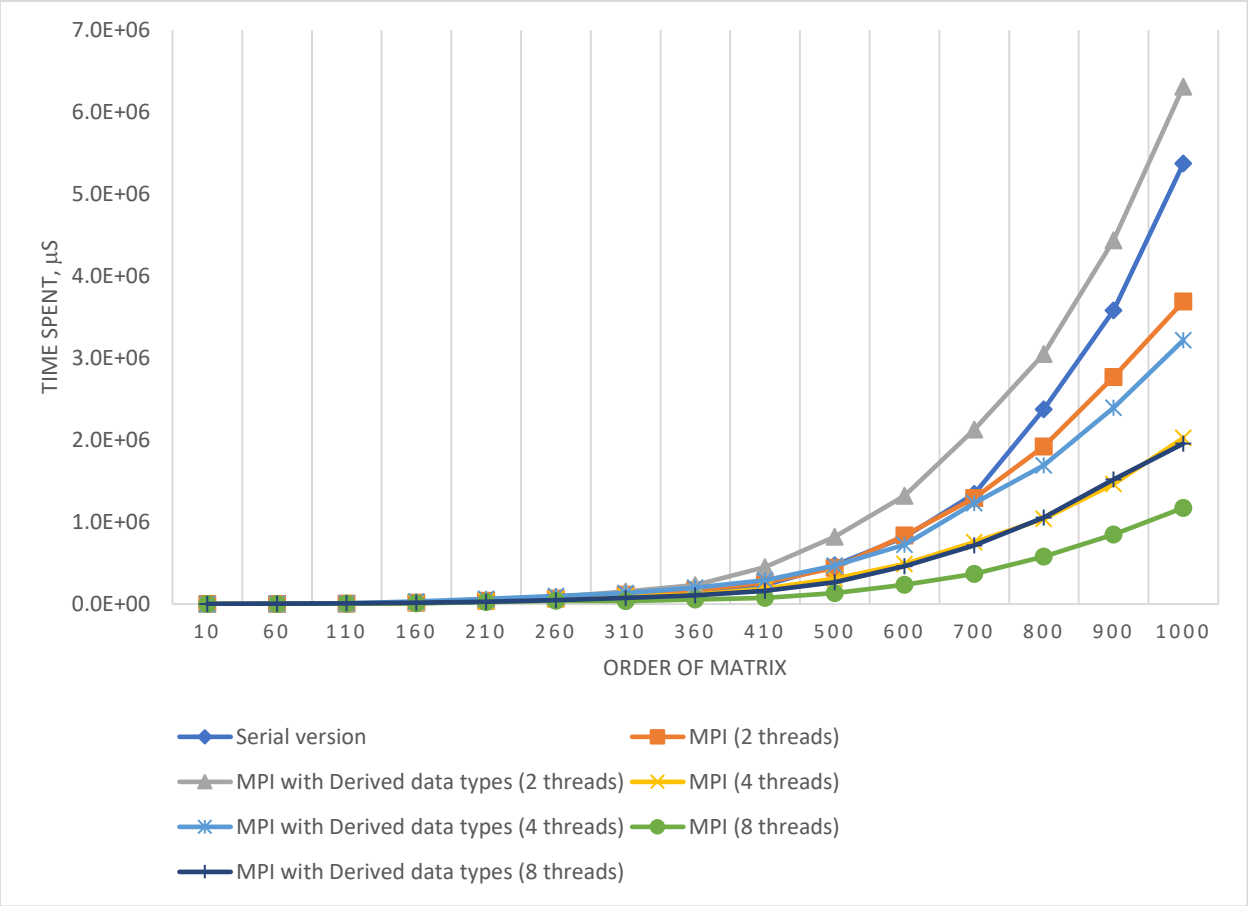

Figure 1 Distribution of time spent

As seen from figure 1, performance from using parallel implementations gains do not start to appear immediately. For a long time, the serial version of the program spends about the same time as parallel versions. It may be explained with fact that create threads also requires time and if matrix has not high order speedup may be not arise because increase overhead cost for create of threads. Number of threads affect very much. Increase number of threads from 2 to 8 cuts time spent almost in four time under order of matrix is equal to 1000.

Using MPI data type theoretically doesn't supposed to gain any performance, because in fact, it transits the same volume of memory just with using useful wrapper. Result obtained even shows some slow-up.

5 CONCLUSION

During the execution of the task, message passing interface (MPI) was used to calculate multiply of two matrixes . The speed up obtained from the use of concurrency is determined. The results obtained were analyzed.

APPENDIX 1

```
#include<iostream>
#include<stdlib.h>
#include<time.h>
#include <chrono>
#include <stdio.h>
using namespace std;
static int n=415;
```

```
int main(int argc, char **argv)
{
    srand(time(0)); //set state of generate depends on current time
    int firstMatrix[n][n], secondMatrix[n][n], mult[n][n];
    int r1, c1, r2, c2, i, j, k;
    r1=n;
    c1 = n;
    r2 = c1;
    c2=n;
    // Storing elements of first matrix.
    for(i = 0; i < r1; ++i){
        for(j = 0; j < c1; ++j)
        {
            firstMatrix[i][j] = rand() %10;
        }
    }
    // Storing elements of second matrix.
    for(i = 0; i < r2; ++i){
        for(j = 0; j < c2; ++j)
        {
            secondMatrix[i][j] = rand() %10;
        }
    }
    // Initializing elements of matrix mult to 0.
    for(i = 0; i < r1; ++i)
        for(j = 0; j < c2; ++j)
        {
            mult[i][j]=0;
        }
    double cntIteration = 5;
    double totalTime = 0.0;
    for (int q=0; q<cntIteration; q++)
    {
        auto t1 = std::chrono::high_resolution_clock::now();
        for(i = 0; i < r1; ++i)
        for(j = 0; j < c2; ++j)
            for(k = 0; k < c1; ++k)
            {
                mult[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
            }
        auto t2 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::micro> fp_ms = t2 - t1;
        totalTime += fp_ms.count();
    }
    double timeSpentSerial = totalTime/cntIteration;
    cout << endl << "timeSpentSerial: " << timeSpentSerial;
    return 0;
}
```

APPENDIX 2

```
#include<iostream>
```

```
#include<stdlib.h>
#include<time.h>
#include <chrono>
#include <stdio.h>
#include "mpi.h"
using namespace std;
const int N = 900;
const int root = 0;
MPI_Status status;

double firstMatrix[N][N], secondMatrix[N][N], resultMatrix[N][N];

int main(int argc, char** argv)
{
    double totalTime = 0.0;

    int countThreads, rank, countWorker, source, targetWorker, rows, offset, i, j, k;

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

    countWorker = countThreads - 1;

    /*---------------------------- master ----------------------------*/
    if (rank == root) {
        srand(time(0));

        //root thread generate two matrixes
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                firstMatrix[i][j] = rand() % 10;
                secondMatrix[i][j] = rand() % 10;
            }
        }

        auto t1 = std::chrono::high_resolution_clock::now();

        /* send matrix data to the worker tasks */
        rows = N / countWorker; // how many rows will be processed by the one worker
        offset = 0;

        for (targetWorker = 1; targetWorker <= countWorker; targetWorker++)
        {
            MPI_Send(&offset, 1, MPI_INT, targetWorker, 1, MPI_COMM_WORLD);
            MPI_Send(&rows, 1, MPI_INT, targetWorker, 1, MPI_COMM_WORLD);
            MPI_Send(&firstMatrix[offset][0], rows * N, MPI_DOUBLE, targetWorker, 1,
MPI_COMM_WORLD);
            MPI_Send(&secondMatrix, N * N, MPI_DOUBLE, targetWorker, 1, MPI_COMM_WORLD);
            offset = offset + rows;
        }

        /* wait for results from all worker tasks */
        for (source = 1; source <= countWorker; source++)
        {
            MPI_Recv(&offset, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
            MPI_Recv(&rows, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
            MPI_Recv(&resultMatrix[offset][0], rows * N, MPI_DOUBLE, source, 2,
MPI_COMM_WORLD, &status);
        }

        auto t2 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::micro> fp_ms = t2 - t1;
        totalTime = fp_ms.count();
```

```
        cout <<"Time spent for order "<<N<<"|\t"<< totalTime <<" ms";
    }

    /*---------------------------- worker----------------------------*/
    if (rank > 0) {
        MPI_Recv(&offset, 1, MPI_INT, root, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, root, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&firstMatrix, rows * N, MPI_DOUBLE, root, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&secondMatrix, N * N, MPI_DOUBLE, root, 1, MPI_COMM_WORLD, &status);

        /* Matrix multiplication */
        for (k = 0; k < N; k++)
            for (i = 0; i < rows; i++) {
                resultMatrix[i][k] = 0.0;
                for (j = 0; j < N; j++)
                    resultMatrix[i][k] = resultMatrix[i][k] + firstMatrix[i][j] *
secondMatrix[j][k];
            }


        MPI_Send(&offset, 1, MPI_INT, root, 2, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, root, 2, MPI_COMM_WORLD);
        MPI_Send(&resultMatrix, rows * N, MPI_DOUBLE, root, 2, MPI_COMM_WORLD);
    }

    MPI_Finalize();
}
```

APPENDIX 3

```
#include<iostream>
#include<stdlib.h>
#include<time.h>
#include <chrono>
#include <stdio.h>
#include "mpi.h"
using namespace std;
const int N = 1000;
const int root = 0;
MPI_Status status;
MPI_Datatype coltype;
MPI_Datatype rowtype;

double firstMatrix[N][N], secondMatrix[N][N], resultMatrix[N][N];
int main(int argc, char** argv)
{
    double totalTime = 0.0;

    int countThreads, rank, countWorker, source, targetWorker, rows, offset, i, j, k;

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

    // create new type data
    MPI_Type_vector(N, N, 0, MPI_INT, &rowtype);
    MPI_Type_commit(&rowtype);

    MPI_Comm_size(MPI_COMM_WORLD, &countThreads);

    countWorker = countThreads - 1;

    /*---------------------------- master ----------------------------*/
    if (rank == root) {
```

```
        srand(time(0));

        //root thread generate two matrixes
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                firstMatrix[i][j] = rand() % 10;
                secondMatrix[i][j] = rand() % 10;
            }
        }

        auto t1 = std::chrono::high_resolution_clock::now();

        /* send matrix data to the worker tasks */
        rows = N / countWorker; // how many rows will be processed by the one worker
        offset = 0;
        for (targetWorker = 1; targetWorker <= countWorker; targetWorker++)
        {
            MPI_Send(&offset, 1, MPI_INT, targetWorker, 1, MPI_COMM_WORLD);
            MPI_Send(&rows, 1, MPI_INT, targetWorker, 1, MPI_COMM_WORLD);
            MPI_Send(&firstMatrix[offset][0], rows, rowtype, targetWorker, 1,
MPI_COMM_WORLD);
            MPI_Send(&secondMatrix, N * N, MPI_DOUBLE, targetWorker, 1, MPI_COMM_WORLD);
            offset = offset + rows;
        for (source = 1; source <= countWorker; source++)
        {
            MPI_Recv(&offset, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
            MPI_Recv(&rows, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
            MPI_Recv(&resultMatrix[offset][0], rows * N, MPI_DOUBLE, source, 2,
MPI_COMM_WORLD, &status);
        }
        auto t2 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::micro> fp_ms = t2 - t1;
        totalTime = fp_ms.count();
        cout <<"Time spent for order "<<N<<"|\t"<< totalTime <<" ms";
    }

    /*---------------------------- worker----------------------------*/
    if (rank > 0) {
        MPI_Recv(&offset, 1, MPI_INT, root, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, root, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&firstMatrix, rows, rowtype, root, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&secondMatrix, N * N, MPI_DOUBLE, root, 1, MPI_COMM_WORLD, &status);

        /* Matrix multiplication */
        for (k = 0; k < N; k++)
            for (i = 0; i < rows; i++) {
                resultMatrix[i][k] = 0.0;
                for (j = 0; j < N; j++)
                    resultMatrix[i][k] = resultMatrix[i][k] + firstMatrix[i][j] *
secondMatrix[j][k];
            }


        MPI_Send(&offset, 1, MPI_INT, root, 2, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, root, 2, MPI_COMM_WORLD);
        MPI_Send(&resultMatrix, rows * N, MPI_DOUBLE, root, 2, MPI_COMM_WORLD);
    }

    MPI_Finalize();
}
```