Министерство науки и высшего образования Российской Федерации
Новосибирский Государственный технический университет
Кафедра автоматизированных систем управления

Отчет по лабораторной работе №6

по дисциплине «Параллельное программирование»

«Матрично-векторное умножение в распределенных вычислительных системах.»

Вариант – мм, л2

Выполнили
студенты группы АВТ-813:
Кинчаров Данил
Пайхаев Алексей
Чернаков Кирилл
Преподаватель:
Ландовский Владимир Владимирович,
к.т.н., доцент кафедры АСУ

г. Новосибирск
2020 г.

Содержание

1. Постановка задачи:

Разработать и реализовать с помощью MPI параллельный алгоритм умножения.

2. Задание:

Разработать и реализовать с помощью MPI параллельный алгоритм умножения матриц.

Умножения матриц ленточная схема: каждая подзадача содержит по одной строке перемножаемых матриц.

3. Пример работы программы и описание алгоритма:

Вычисление в нашем случае сводится к поэлементному умножению, имеющихся векторов, матриц A и B. После в каждой подзадаче получается промежуточный результат – строка частичных результатов, необходимая для получения результат для матрицы C.

Строки матрицы идут последовательно и Scatter это разделяет, а после выполнения Gather собирает уже результаты умножения строк матрицы обратно.

В алгоритме используется сначала широковещательная рассылка наших строк матрицы A и B, а в нашем случае это строки, к примеру для матрицы 8x8 равные 1x8 для матрицы A и 1x8 для матрицы B и 8 процессов.

Происходит последовательная передача строк. На каждой итерации данные перемножаются поэлементно и производится суммирование с полученными ранее значениями. Происходит последовательное получение в подзадачах всех строк матрицы B, поэлементное умножение данных и суммирование вновь получаемых значений с ранее вычисленными результатами. После завершения итераций строки собираются в единую матрицу C.

В таблице на рисунке 1 приведены результаты выполнения программы, выполнявшейся с использованием процессора 4/8 (Intel core i7-7700HQ), а также график зависимости времени от количества процессов.

Размер матрицы 2000x2000	
Кол-во потоков	Время, мс
1	43653
2	19142
3	13924
4	10492
5	9332
6	8124
7	7642
8	7142

Размер матрицы 1000x1000	
Кол-во потоков	Время, мс
1	4081
2	2066
3	1397
4	1232
5	1013
6	899
7	835
8	822

4 потока	
Размер матрицы	Время, мс
2000x2000	10492
1000x1000	1232
750x750	135
500x500	128
250x250	24
100x100	3
50x50	1

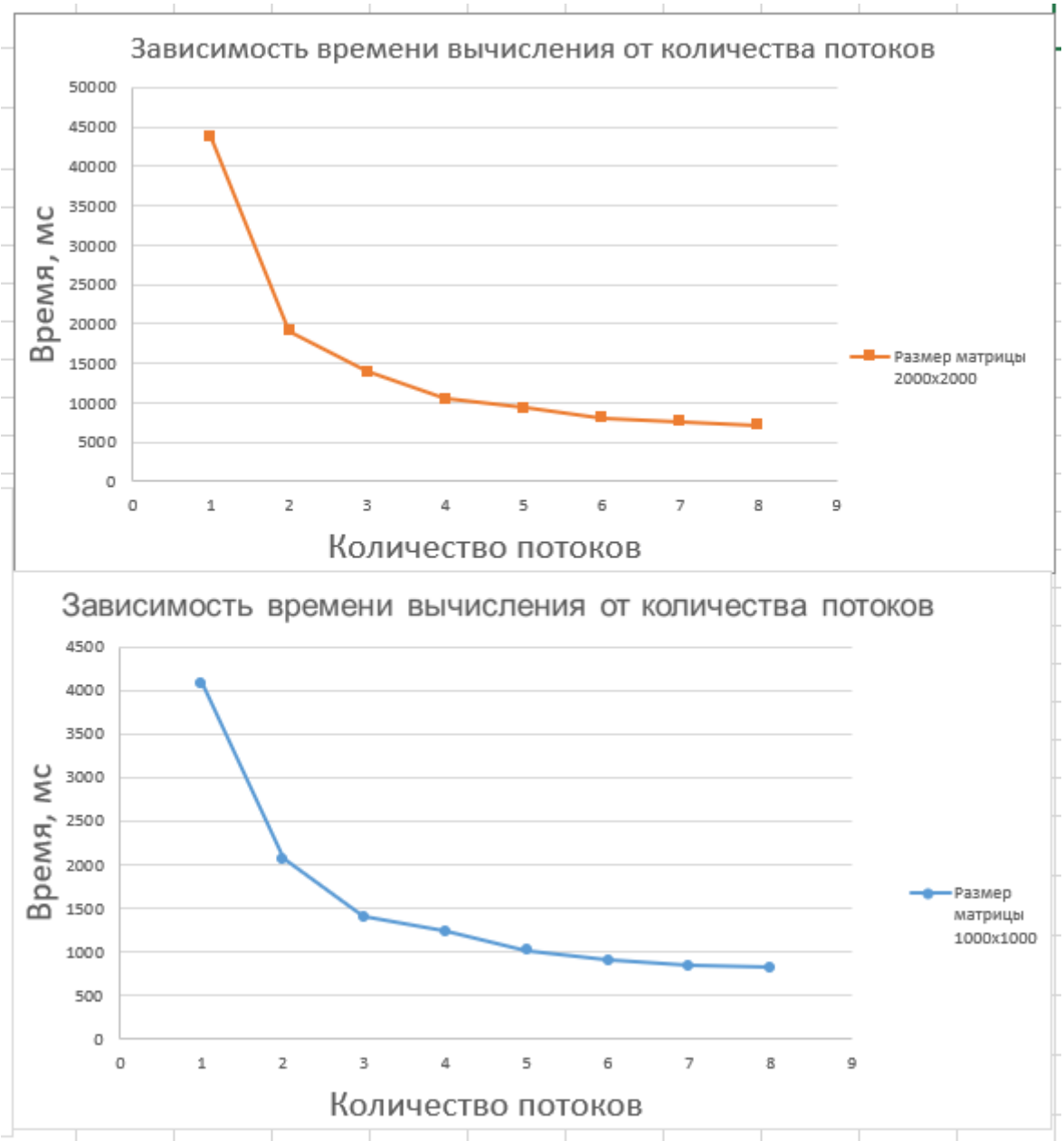

Рисунок 1 – Таблица с полученными данными и графики зависимости времени от количества процессов и от размера матрицы

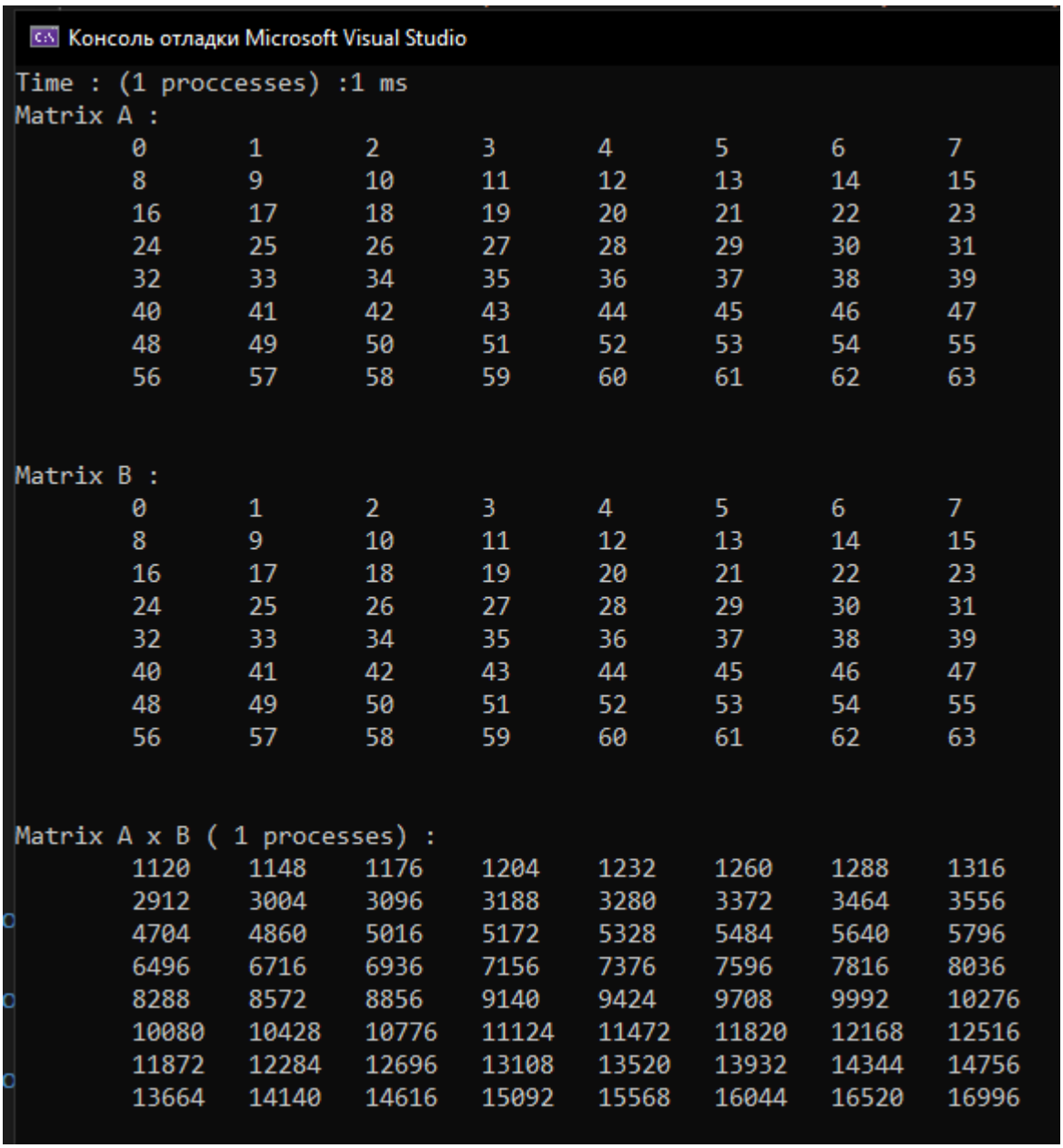

Рисунок 2 – Пример работы программы при использовании 1 процесса

```
D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 1 code.exe
Time : (1 proccesses) :43653 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 2 code.exe
Time : (2 proccesses) :19142 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 3 code.exe
Time : (3 proccesses) :13024 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 4 code.exe
Time : (4 proccesses) :10492 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 5 code.exe
Time : (5 proccesses) :9332 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 6 code.exe
Time : (6 proccesses) :8124 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 7 code.exe
Time : (7 proccesses) :7921 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 8 code.exe
Time : (8 proccesses) :7642 ms
```

Рисунок 3 – Результат работы программы при использовании разного количества процессов 1-8 (матрица размером 2000x2000)

```
D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 1 code.exe
Time : (1 proccesses) :4081 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 2 code.exe
Time : (2 proccesses) :2066 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 3 code.exe
Time : (3 proccesses) :1397 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 4 code.exe
Time : (4 proccesses) :1232 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 5 code.exe
Time : (5 proccesses) :1013 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 6 code.exe
Time : (6 proccesses) :899 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 7 code.exe
Time : (7 proccesses) :835 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>mpiexec -np 8 code.exe
Time : (8 proccesses) :822 ms

D:\Project\AVT_813_5SEM\parallel programming\LAB6\code1\Debug>
```

Рисунок 4 – Результат работы программы при использовании разного количества процессов 1-8 (матрица размером 1000x1000)

5. Вывод:

В ходе лабораторной работы была написана программа, с помощью которой осуществляется многопоточное умножение матриц при помощи MPI и ленточной схемы, где каждая подзадача содержит по одной строке перемножаемых матриц.

Исходя из рисунка 1 (Зависимость затраченного времени от кол-ва потоков), можно сделать вывод о том, что в результате работы программы при увеличении числа процессов уменьшается время работы программы, а после превышения числа доступных для процессора процессов идёт увеличение времени работы программы, так как процессы ожидают освобождения вычислительной мощности процессора.

Исходя из графика (Зависимость затраченного времени от размера матрицы) на рисунке 1, чем больше элементов, тем дольше работает программа. А также продемонстрированные примеры работы программы на рисунках 2-4.

4. Листинг программы:

```
#include <mpi.h>
#include <cmath>
#include <iostream>
#include <ctime>

void insert(int* matrixA, int* matrixB, size_t A, size_t B, size_t C);
void show(int* matrix, size_t A, size_t B);
void MultiMatrixParallel(int* , int AHeght, int AWidth, int* B, int BHeght, int B
Width, int* C, int rank, int numProcesses);

void main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    int rank;
    int numProcesses;

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

    size_t A = 2000;
    size_t B = 2000;
    size_t C = 2000;

    MPI_Status stat;

    int* matrixA = new int[A * B];
    int* matrixB = new int[B * C];
    int* matrixC = new int[A * C];

    insert(matrixA, matrixB, A, B, C);

    clock_t time = clock();
    MultiMatrixParallel(matrixA, A, A, matrixB, B, B, matrixC, rank, numProcesses
);
    time = clock() - time;

    if (rank == 0)
    {
        std::cout << "Time : (" << numProcesses << " proccesses) :" << time << "
ms" << std::endl;
        /*
        std::cout << "Matrix A : " << std::endl;
        show(matrixA, A, B);

        std::cout << "Matrix B : " << std::endl;
        show(matrixB, B, C);
```

```
        std::cout << "Matrix A x B ( " << numProcesses << " processes) :" << std:
:endl;
        show(matrixC, A, C);
        */
    }



    delete matrixA;
    delete matrixB;
    delete matrixC;

    MPI_Finalize();
}

void insert(int* matrixA, int* matrixB, size_t A, size_t B, size_t C)
{
    for (size_t i = 0; i < A * B; i++)
    {
        matrixA[i] = i;
    }

    for (size_t i = 0; i < B * C; i++)
    {
        matrixB[i] = i;
    }
}

void show(int* matrix, size_t A, size_t B)
{
    for (size_t i = 0; i < A * B; i += B)
    {
        for (size_t j = 0; j < B; j++)
        {
            std::cout << '\t' << matrix[j + i];
        }
        std::cout << '\n';
    }

    std::cout << "\n\n";
}

void MultiMatrixParallel(int* A, int AHeght, int AWidth, int* B, int BHeght, int
BWidth, int* C, int rank, int numProcesses)
{
    int i = 0, j = 0;
    int *bufferA, *bufferB, *bufferC;

    int rowA = AHeght / numProcesses;
    int rowB = BHeght / numProcesses;
```

```
    int rowC = rowA;

    int PartA = rowA * AWidth;
    int PartB = rowB * BWidth;
    int partC = rowC * BWidth;

    bufferA = new int[PartA];
    bufferB = new int[PartB];
    bufferC = new int[partC];

    for (i = 0; i < partC; i++)
    {
        bufferC[i] = 0;
    }

    MPI_Scatter(A, PartA, MPI_INT, bufferA, PartA, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Scatter(B, PartB, MPI_INT, bufferB, PartB, MPI_INT, 0, MPI_COMM_WORLD);

    int k = 0, temp = 0;
    int NextProcesses = rank + 1;
    if (rank == numProcesses - 1)
    {
        NextProcesses = 0;
    }
    int PrevProcesses = rank - 1;
    if (rank == 0)
    {
        PrevProcesses = numProcesses - 1;
    }

    MPI_Status Status;

    int PrevDataNum = rank;
    for (int p = 0; p < numProcesses; p++)
    {
        for (i = 0; i < rowA; i++)
        {
            for (j = 0; j < BWidth; j++)
            {
                temp = 0;
                for (k = 0; k < rowB; k++)
                {
                    temp += bufferA[PrevDataNum * rowB + i * AWidth + k] * buffer
B[k * BWidth + j];
                }
                bufferC[i * BWidth + j] += temp;
            }
        }
        PrevDataNum -= 1;
        if (PrevDataNum < 0)
```

```
        {
            PrevDataNum = numProcesses - 1;
        }

        MPI_Sendrecv_replace(bufferB, PartB, MPI_INT, NextProcesses, 0, PrevProce
sses, 0, MPI_COMM_WORLD, &Status);

    }

    MPI_Gather(bufferC, partC, MPI_INT, C, partC, MPI_INT, 0, MPI_COMM_WORLD);

}
```