

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ
по лабораторной работе №5
по дисциплине «Системы параллельной обработки данных»
Тема: Умножение матриц

Студент гр. 1310

Комаров Д.Е.

Преподаватель

Татаринов Ю.С.

Санкт-Петербург

2025

Цель работы

Целью выполнения лабораторной работы является создание программы для многопоточного умножения матриц помощи библиотеки MPI.

Постановка задачи

Требуется выполнить задачу умножения двух квадратных матриц A и B размера $m \times m$, результат записать в матрицу C . Реализовать последовательный и параллельный алгоритм, одним из перечисленных ниже способов и провести анализ полученных результатов. Выбор параллельного алгоритма определяется индивидуальным номером задания. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

Вариант 1: Ленточный алгоритм 1 (горизонтальные полосы).

Выполнение работы

Описание используемой виртуальной топологии

Для выполнения лабораторной работы будет использоваться декартова топология. Размер сетки будет выбран так, чтобы количество ее строк было равно количеству столбцов. Также будет осуществлена поддержка только количества процессов, равному степени числа 2 (1,2,4,8 и так далее). В случае, если количество процессов равно четной степени числа 2, то длина и ширина решетки будет равна квадратному корню из числа процессов. Так для 1 процесса будет создана решетка размерностью (1, 1), для 4 – (2, 2), для 16 – (4, 4) и так далее. В случае, если количество процессов равно нечетной степени числа 2, то количество строк решетки будет в 2 раза больше количества столбцов. Так, для 2 процессов будет создана решетка размерностью (2, 1), для 8 – (4, 2), для 32 – (8, 4) и так далее.

Описание принципа разбиения матрицы между процессами

Обозначим умножаемые матрица как A и B . Обе матрицы имеют одинаковое количество строк и столбцов, равное m . Размерность решетки-коммуникатора зададим как $(gridRow, gridCol)$. Матрица A будет разделена по строкам между всеми строками решетки-коммуникатора, а матрица B будет

разделена по строкам между всеми столбцами решетки-коммуникатора. Для того, чтобы умножение матриц выполнялось параллельно, m должно быть больше или равным $gridRow$ и $gridCol$. Таким образом каждый процесс получит одну или несколько строк из матриц A и B . Количество строк, получаемое одним процессом, может быть получено путем деления m на $gridRow$ и $gridCol$. В случае, если m не делится нацело на $gridRow$ и $gridCol$, остаток от деления достается последним процессам в решетке. Приведем пример. Зададим матрицы A и B как

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

$$B = \begin{pmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{pmatrix}.$$

Пусть количество процессов равно 4. Следовательно размерность решетки-коммуникатора равна (2, 2). Тогда процесс (0, 0) получит следующие строки матриц

$$A_{(0,0)} = (1 \quad 2 \quad 3),$$

$$B_{(0,0)} = (10 \quad 11 \quad 12).$$

Процесс (0, 1) получит

$$A_{(0,1)} = (1 \quad 2 \quad 3),$$

$$B_{(0,1)} = \begin{pmatrix} 13 & 14 & 15 \\ 16 & 17 & 18 \end{pmatrix}.$$

Процесс (1, 0) получит

$$A_{(1,0)} = \begin{pmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

$$B_{(1,0)} = (10 \quad 11 \quad 12).$$

Процесс (1,1) получит

$$A_{(1,1)} = \begin{pmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

$$B_{(1,1)} = \begin{pmatrix} 13 & 14 & 15 \\ 16 & 17 & 18 \end{pmatrix}.$$

Описание принципа параллельного умножения матриц

Каждый процесс умножает полученные им строки матрицы и передает их предыдущему процессу в строке решетки коммутатора. Получая умноженные строки, каждый процесс суммирует их со своим результатом умножения и пересылает далее. Когда все умноженные строки достигнут первого столбца решетки-коммуникатора умножение будет закончено. В первом столбце умноженные строки передаются строкам коммуникатора-решетки выше и объединяются в итоговую матрицу. Таким образом, процесс (0,0) получит результат умножения. Продолжим ранее начатый пример. Пусть матрица C – результат умножения A и B . В результате умножения своих строк процесс (0, 0) получит

$$C_{(0,0)} = (10 \quad 11 \quad 12).$$

Процесс (0, 1) получит в результате умножения

$$C_{(0,1)} = (74 \quad 79 \quad 84).$$

Процесс (1, 0) получит в результате умножения

$$C_{(1,0)} = \begin{pmatrix} 40 & 44 & 48 \\ 70 & 77 & 84 \end{pmatrix}.$$

Процесс (1, 1) получит в результате умножения

$$C_{(1,1)} = \begin{pmatrix} 161 & 172 & 183 \\ 248 & 265 & 282 \end{pmatrix}.$$

Процессы (0, 1) и (1,0) передают свои умноженные строки в процессы (0, 0) и (1,0) соответственно. Там полученные строки суммируются со своими результатами умножения. В результате в процессе (0,0) будет

$$C_{(0,0)} = (84 \quad 90 \quad 96).$$

В процессе (1,0) будет

$$C_{(1,0)} = \begin{pmatrix} 201 & 216 & 231 \\ 318 & 342 & 366 \end{pmatrix}.$$

Затем процесс (1,0) передает свои строки процессу (0,0) где формируется итоговый результат умножения

$$C = \begin{pmatrix} 84 & 90 & 96 \\ 201 & 216 & 231 \\ 318 & 342 & 366 \end{pmatrix}.$$

Алгоритм параллельного умножения матриц в виде сети Петри представлен на рисунке 1.

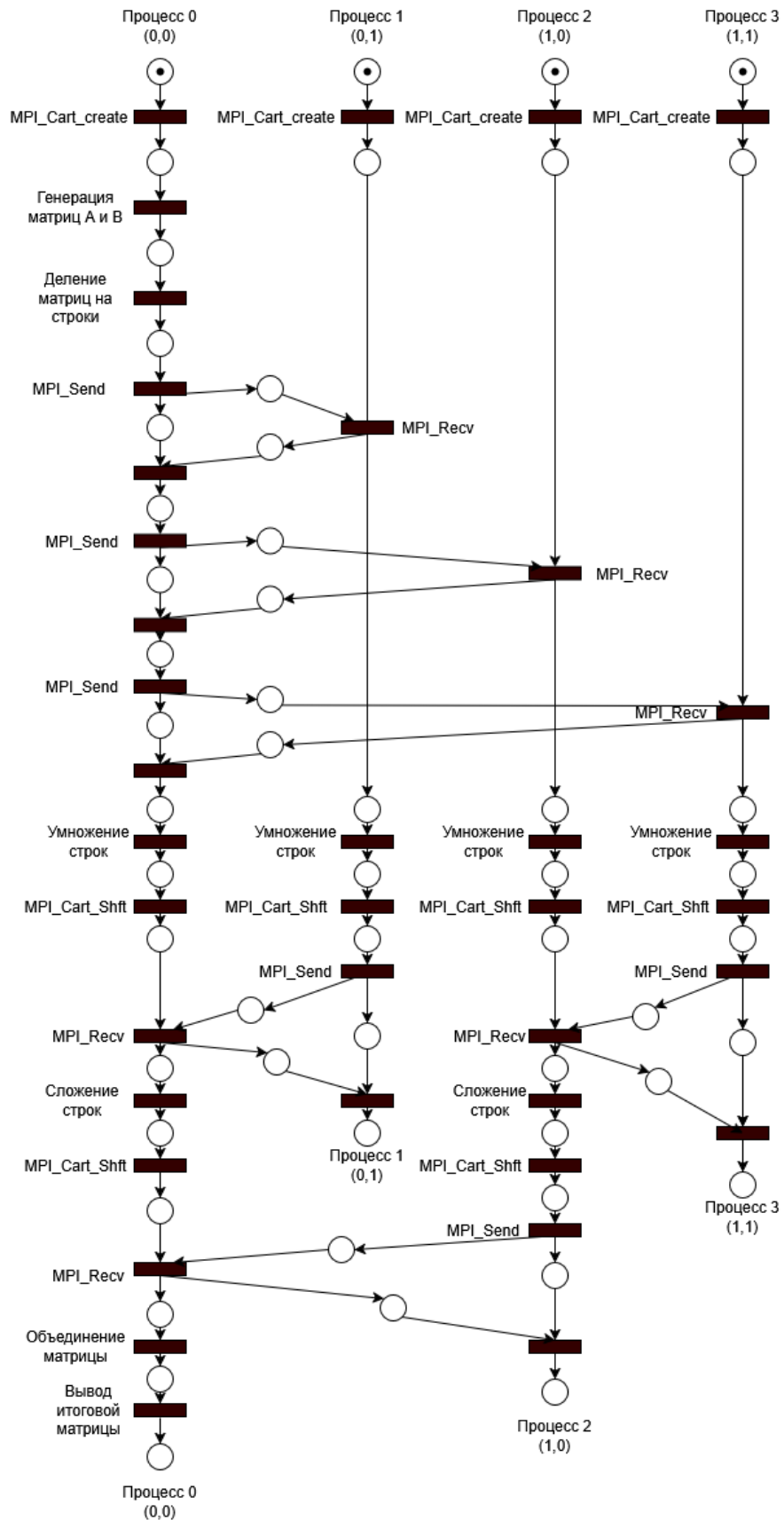


Рисунок 1 – Сеть Петри

Тестирование

Протестируем написанную программу, при количестве процессов, равном 4, на матрицах их приведённого примера. Код программы представлен в приложении А. Результат представлен на рисунке 2. Как можно заметить, теоретический результат совпал с результатом работы программы.

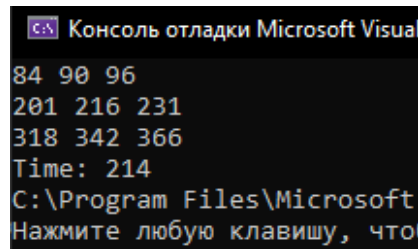


Рисунок 2 – Демонстрация работы программы

Для сравнения производительности последовательного и параллельного алгоритмов также была написана программа последовательного умножения матриц. Ее код представлен в приложении Б.

Проведем замеры времени выполнения алгоритма для различного количества процессов и различной размерности матрицы. Замеры проводились на процессоре, имеющем 4 ядра и 4 потока. Отсюда следует, что ожидаемое ускорение для 2 процессов равно 2, а для 4 – 4. Также ожидается, что для фиксированного числа процессов время выполнения будет расти пропорционально кубу размерности матрицы. Результаты измерений времени выполнения последовательного и параллельного алгоритмов представлены в таблице 1. Как можно заметить, для матриц с размерностью 10, 50, 100 последовательный алгоритм оказался производительнее параллельного, поскольку распараллеливание задачи занимало больше времени, чем ее выполнение. Для больших матриц параллельный алгоритм давал ускорение не более, чем в 4 раза. Также увеличение количества процессов более 4 не давало большего ускорения, поскольку процессор может параллельно выполнять только 4 задачи.

Таблица 1 – Измерения производительности алгоритмов

Размерность матрицы (m)	Последовательный алгоритм	2 процесса		4 процесса		8 процессов		16 процессов		32 процесса		64 процессов	
	время	время	ускорение	время	ускорение	время	ускорение	время	ускорение	Время	Ускорение	Время	Ускорение
10	4 мкс	45 мкс	0.08	277 мкс	0.01	1071 мкс	0.004	3327 мкс	0.001	8027 мкс	0.0005	20695 мкс	0.0002
50	787 мкс	528 мкс	1.49	783 мкс	1.01	1395 мкс	0.58	4057 мкс	0.19	7766 мкс	0.1	30077 мкс	0.03
100	4 мс	3 мс	1.33	2 мс	2	4 мс	1	4 мс	1	10 мс	0.4	34 мс	0.12
200	35 мс	20 мс	1.75	12 мс	2.92	12 мс	2.92	17 мс	2.06	25 мс	1.4	80 мс	0.44
500	516 мс	286 мс	1.8	162 мс	3.18	192 мс	2.67	201 мс	2.57	217 мс	2.38	360 мс	1.43
1000	4099 мс	2172 мс	1.88	1277 мс	3.21	1190 мс	3.44	1247 мс	3.29	1693 мс	2.42	1884 мс	2.18
10000	67 мин	35 мин	1.9	19 мин	3.52	19 мин	3.52	19 мин	3.52	19 мин	3.52	19 мин	3.52

На рисунке 3 представлен график зависимости времени выполнения алгоритма от количества процессов.

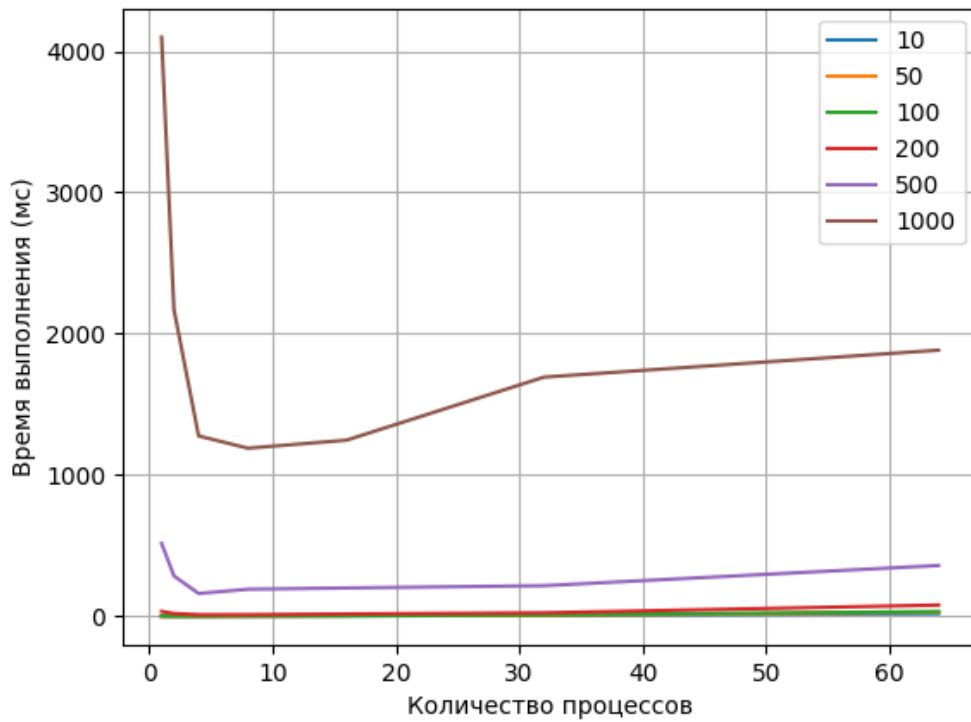


Рисунок 3 – График зависимости времени выполнения алгоритма от количества процессов

На рисунке 4 представлен график зависимости времени выполнения от размерности матрицы.

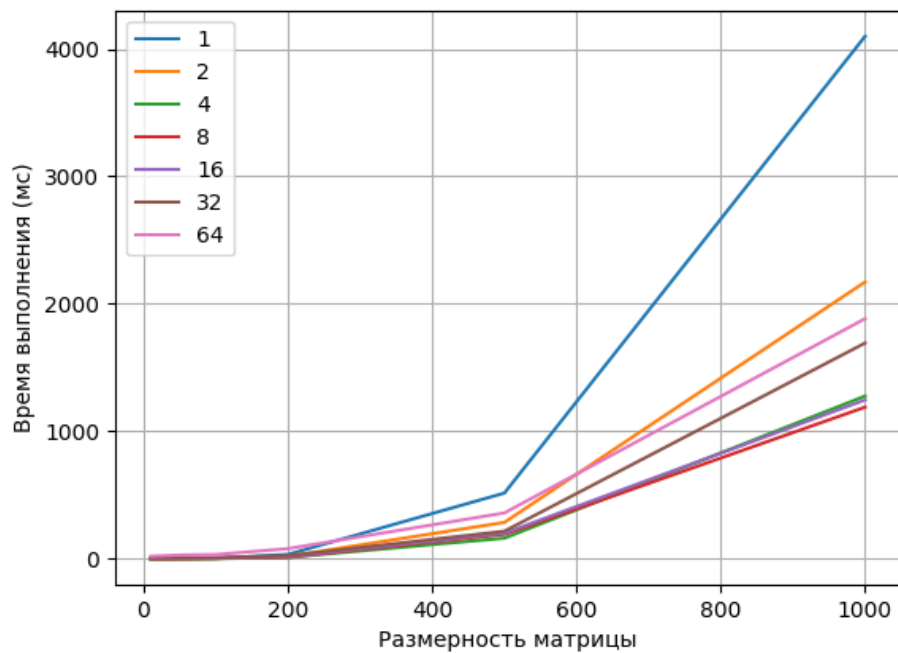


Рисунок 4 – График зависимости времени выполнения алгоритма от Размерности матрицы

На рисунке 5 представлен график ускорения выполнения алгоритма в зависимости от количества процессов.

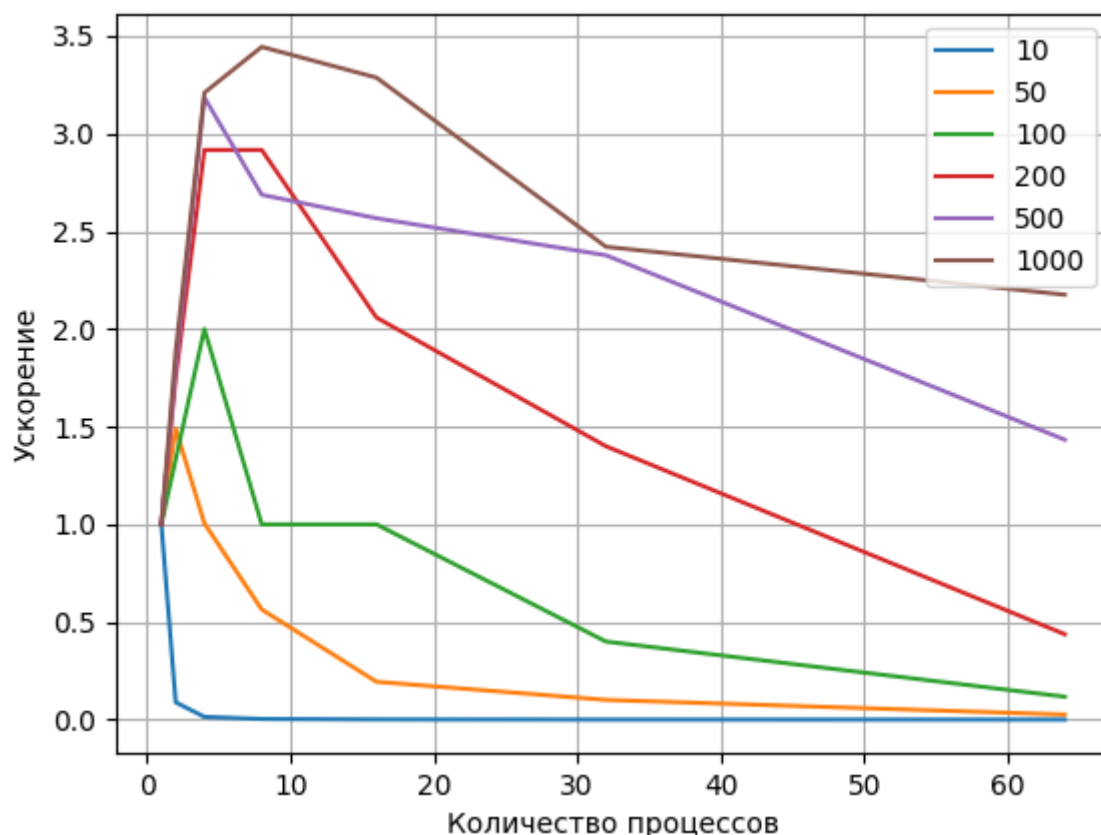


Рисунок 5– График зависимости ускорения времени выполнения алгоритма от количества процессов

Выводы

В ходе выполнения лабораторной работы была создана программа для многопоточного умножения матриц помощи библиотеки MPI.

Программа была протестирована на различном количестве процессов и для различной размерности матрицы. Время выполнения было сравнено с последовательным алгоритмом. Замеры времени выполнения проводились на четырёхъядерном четырехпоточном процессоре. Было выяснено, что для матриц маленькой размерности параллельный алгоритм не дает преимуществ в производительности, поскольку в таком случае время, затраченное на распараллеливание задачи, оказывалось больше времени умножения матриц. Для матриц большой размерности параллельный алгоритм давал большую производительность, чем последовательный. Для двух процессов ускорение было почти в 2 раза, а для 4 процессов почти в 4, что соответствует

ожиданиям. Дальнейшее увеличение количества процессов не давало преимуществ в производительности, поскольку процессор может одновременно выполнять только 4 задачи, а следовательно оптимальное количество процессов является 4.

Также было выяснено, что для фиксированного числа процессов, время выполнения растет пропорционально кубу размерности матрицы, что соответствует ожиданиям.

Код программы параллельного умножения матриц представлен в приложении А.

Код программы последовательного умножения матриц представлен в приложении Б.

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ ПАРАЛЛЕЛЬНОГО УМНОЖЕНИЯ МАТРИЦ

matrix_funcs.h:

```
#pragma once
#include <iostream>

double** newMatrix(int m,int n);
double** createRowBuff(double **matr, int rowSt,int rowNum,int rowLen);
void delMatrix(double** matr, int m,int n);
void printMatrixTo(double** matr, int m,int n, std::ostream& stream);
double **readMatrixFrom(int m,int n, std::istream& stream);
double **randomizeMatrix(int m,int n);
void fillMatrix(double** matr, int m,int n, double num);
double** multiplyRows(double** rowBuffA, double** rowBuffB, int rowANum, int rowBNum, int rowLen, int colAStart);
void sumRows(double** rowBuffA, double **rowBuffB,int rowNum, int rowLen);
double** uniteMatrix(double** rowsA, double** rowsB, int rowANum, int rowBNum, int rowLen);
```

matrix_funcs.cpp:

```
#include "matrixFuncs.h"

double** newMatrix(int m, int n) {
    double** matr = new double* [m];
    for (int i = 0;i < m;i++) {
        matr[i] = new double[n];
    }
    return matr;
}

double** createRowBuff(double** matr, int rowSt, int rowNum,int rowLen) {
    double** rowBuff = new double* [rowNum];
    for (int i = 0;i < rowNum;i++) {
        rowBuff[i] = new double[rowLen];
        for (int j = 0;j < rowLen;j++)
            rowBuff[i][j] = matr[i + rowSt][j];
    }
    return rowBuff;
}

void delMatrix(double** matr, int m, int n) {
    for (int i = 0;i < m;i++)
        delete[] matr[i];
    delete[] matr;
}

void printMatrixTo(double** matr, int m, int n, std::ostream& stream) {
    for (int i = 0;i < m;i++) {
        for (int j = 0;j < n;j++)
            stream << matr[i][j] << " ";
        stream << std::endl;
    }
}

double **readMatrixFrom(int m, int n, std::istream& stream){
    double** matr = newMatrix(m, n);
    for (int i = 0;i < m;i++)
        for (int j = 0;j < n;j++)
            stream >> matr[i][j];
    return matr;
}
```

```

double **randomizeMatrix( int m,int n){
    double** matr = newMatrix(m, n);
    for (int i = 0;i < m;i++){
        for (int j = 0;j < n;j++){
            matr[i][j] = (double)rand() / RAND_MAX;
        }
    }
    return matr;
}

void fillMatrix(double** matr, int m,int n, double num){
    for (int i = 0;i < m;i++){
        for (int j = 0;j < n;j++){
            matr[i][j] = num;
        }
    }
}

double** multiplyRows(double** rowBuffA, double** rowBuffB, int rowANum,int
rowBNum, int rowLen,int colAStart){
    double** res = new double* [rowANum];
    for (int i = 0;i < rowANum;i++) {
        res[i] = new double[rowLen];
        for (int j = 0;j < rowLen;j++){
            res[i][j] = 0;
            for (int j = 0;j < rowBNum;j++) {
                for (int k = 0;k < rowLen;k++) {
                    res[i][k] += rowBuffA[i][colAStart+j] *
rowBuffB[j][k];
                }
            }
        }
    }
    return res;
}

void sumRows(double** rowBuffA, double** rowBuffB, int rowNum, int rowLen) {
    for (int i = 0;i < rowNum;i++){
        for (int j = 0;j < rowLen;j++){
            rowBuffA[i][j] += rowBuffB[i][j];
        }
    }
}

double** uniteMatrix(double** rowsA, double** rowsB, int rowANum, int
rowBNum, int rowLen) {
    double** res = newMatrix(rowANum + rowBNum, rowLen);
    for (int i = 0;i < rowANum;i++){
        for (int j = 0;j < rowLen;j++){
            res[i][j] = rowsA[i][j];
        }
    }
    for (int i = 0;i < rowBNum;i++){
        for (int j = 0;j < rowLen;j++){
            res[i+rowANum][j] = rowsB[i][j];
        }
    }
    return res;
}

```

communicate_funcs.h:

```

#pragma once
#include <mpi.h>

void createGridComm(MPI_Comm oldComm, int dims[], int cords[], MPI_Comm*
gridComm);
void sendRowsBuffToCords(double** rows, int rowsNum, int rowsLen, int*
destCords, MPI_Comm gridComm);
double** receiveRowsBuffFromCords(int rowsNum, int rowsLen, int* srcCords,
MPI_Comm gridComm);

```

```
double** receiveRowsBuffFromRank(int rowsNum, int rowsLen, int srcRank,
MPI_Comm gridComm);
void sendRowsBuffToCords(double** rows, int rowsNum, int rowsLen, int*
destCords, MPI_Comm gridComm);
void sendRowsBuffToRank(double** rows, int rowsNum, int rowsLen, int
destRank, MPI_Comm gridComm);
```

communicate_funcs.cpp:

```
#include "communicateFuncs.h"
#include "matrixFuncs.h"
#include <cmath>
double **receiveRowsBuffFromCords(int rowsNum, int rowsLen, int* srcCords,
MPI_Comm gridComm) {
    int srcRank;
    MPI_Cart_rank(gridComm, srcCords, &srcRank);
    return receiveRowsBuffFromRank(rowsNum, rowsLen, srcRank, gridComm);
}
double** receiveRowsBuffFromRank(int rowsNum, int rowsLen, int srcRank,
MPI_Comm gridComm) {
    double** recvBuff = newMatrix(rowsNum, rowsLen);
    MPI_Status status;
    for (int i = 0; i < rowsNum; i++) {
        MPI_Recv(recvBuff[i], rowsLen, MPI_DOUBLE, srcRank, 0, gridComm,
&status);
    }
    return recvBuff;
}
void sendRowsBuffToCords(double** rows, int rowsNum, int rowsLen, int*
destCords, MPI_Comm gridComm) {
    int destRank;
    MPI_Cart_rank(gridComm, destCords, &destRank);
    sendRowsBuffToRank(rows, rowsNum, rowsLen, destRank, gridComm);
}
void sendRowsBuffToRank(double** rows, int rowsNum, int rowsLen, int
destRank, MPI_Comm gridComm) {
    for (int i = 0; i < rowsNum; i++) {
        MPI_Send(rows[i], rowsLen, MPI_DOUBLE, destRank, 0, gridComm);
    }
}
void createGridComm(MPI_Comm oldComm, int dims[], int cords[], MPI_Comm*
gridComm) {
    int procNum, procRank;
    MPI_Comm_size(MPI_COMM_WORLD, &procNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    int pw = 0;
    while (!(procNum & 1)) {
        pw++;
        procNum >>= 1;
    }
    if (pw % 2 == 0) {
        dims[0] = dims[1] = (int)pow(2, pw / 2);
    }
    else {
        dims[1] = (int)pow(2, (pw - 1) / 2);
        dims[0] = (int)pow(2, (pw - 1) / 2 + 1);
    }
    int periods[2] = { 1, 1 };
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, gridComm);
    MPI_Cart_coords(*gridComm, procRank, 2, cords);
}
}
```

main.cpp:

```

#include <mpi.h>
#include <iostream>
#include <fstream>
#include <cmath>
#include "matrixFuncs.h"
#include "communicateFuncs.h"
#include <chrono>
#define MATRIX_SIZE 9

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm gridComm;
    int dims[2];
    int cords[2];
    createGridComm(MPI_COMM_WORLD, dims, cords, &gridComm);
    double** rowAbuff=NULLptr;
    double** rowBbuff=NULLptr;
    int rowLen = MATRIX_SIZE;
    int m = MATRIX_SIZE;
    int rowANum= dims[0]-1 != cords[0] ? (MATRIX_SIZE / dims[0]) :
MATRIX_SIZE - (dims[0]-1) * (MATRIX_SIZE / dims[0]);
    int rowBNum= dims[1]-1 != cords[1] ? (MATRIX_SIZE / dims[1]) :
MATRIX_SIZE - (dims[1]-1) * (MATRIX_SIZE / dims[1]);
    std::chrono::steady_clock::time_point begin,end;
    if (cords[0] == 0 && cords[1] == 0) {
        double** matrA = randomizeMatrix(m, m);
        double** matrB = randomizeMatrix(m, m);
        begin = std::chrono::steady_clock::now();
        for(int i=0; i<dims[0]; i++)
            for (int j = 0; j < dims[1]; j++) {
                int rowsANumToSend= dims[0] - 1 != i ? (MATRIX_SIZE /
dims[0]) : MATRIX_SIZE - (dims[0] - 1) * (MATRIX_SIZE / dims[0]);
                int rowsBNumToSend = dims[1] - 1 != j ? (MATRIX_SIZE /
dims[1]) : MATRIX_SIZE - (dims[1] - 1) * (MATRIX_SIZE / dims[1]);
                double** rowABuffToSend = createRowBuff(matrA, i *
rowANum, rowsANumToSend, m);
                double** rowBBuffToSend = createRowBuff(matrB, j *
rowBNum, rowsBNumToSend, m);
                if (i == 0 && j == 0) {
                    rowAbuff = rowABuffToSend;
                    rowBbuff = rowBBuffToSend;
                }
                else {
                    int destCords[2];
                    destCords[0] = i;
                    destCords[1] = j;
                    sendRowsBuffToCords(rowABuffToSend,
rowsANumToSend, rowLen, destCords, gridComm);
                    sendRowsBuffToCords(rowBBuffToSend,
rowsBNumToSend, rowLen, destCords, gridComm);
                    delMatrix(rowABuffToSend, rowsANumToSend,
rowLen);
                    delMatrix(rowBBuffToSend, rowsBNumToSend,
rowLen);
                }
            }
        delMatrix(matrA, m, m);
        delMatrix(matrB, m, m);
    }
    else {
        int srcCords[2];
        srcCords[0] = 0;
    }
}

```

```

        srcCords[1] = 0;
        rowAbuff = receiveRowsBuffFromCords(rowANum, rowLen, srcCords,
gridComm);
        rowBbuff = receiveRowsBuffFromCords(rowBNum, rowLen, srcCords,
gridComm);
    }
    int rowCNum = rowANum;
    double** rowCbuff = multiplyRows(rowAbuff, rowBbuff, rowANum, rowBNum,
rowLen, cords[1]* (m / dims[1]));
    delMatrix(rowAbuff, rowANum, rowLen);
    delMatrix(rowBbuff, rowBNum, rowLen);
    int destRank;
    int srcRank;
    MPI_Cart_shift(gridComm, 1, -1, &srcRank, &destRank);
    if (cords[1] == 0) {
        if (dims[1] != 1) {
            double** recvCbuff = receiveRowsBuffFromRank(rowCNum,
rowLen, srcRank, gridComm);
            sumRows(rowCbuff, recvCbuff, rowCNum, rowLen);
            delMatrix(recvCbuff, rowCNum, rowLen);
        }
        int destRank;
        int srcRank;
        MPI_Cart_shift(gridComm, 0, -1, &srcRank, &destRank);
        if (cords[0] == dims[0] - 1)
            sendRowsBuffToRank(rowCbuff, rowCNum, rowLen, destRank,
gridComm);
        else{
            double** matrC = nullptr;
            int recvRowsCNum = m - (cords[0] + 1) * rowCNum;
            double** recvCbuff = receiveRowsBuffFromRank(recvRowsCNum,
rowLen, srcRank, gridComm);
            matrC = uniteMatrix(rowCbuff, recvCbuff, rowCNum,
recvRowsCNum, rowLen);
            delMatrix(recvCbuff, recvRowsCNum, rowLen);
            if (cords[0] == 0) {
                end= std::chrono::steady_clock::now();
                printMatrixTo(matrC,m, m, std::cout);
                std::cout << "Time: " <<
std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count() <<
std::endl;;
            }
            else {
                sendRowsBuffToRank(matrC, rowCNum+
recvRowsCNum, rowLen, destRank, gridComm);
            }
            delMatrix(matrC, rowCNum + recvRowsCNum, rowLen);
        }
    }
    else if (cords[1] == dims[1] - 1) {
        sendRowsBuffToRank(rowCbuff, rowCNum, rowLen, destRank, gridComm);
    }
    else {
        double** recvCbuff = receiveRowsBuffFromRank(rowCNum, rowLen,
srcRank, gridComm);
        sumRows(rowCbuff, recvCbuff, rowCNum, rowLen);
        delMatrix(recvCbuff, rowCNum, rowLen);
        sendRowsBuffToRank(rowCbuff, rowCNum, rowLen, destRank, gridComm);
    }
    delMatrix(rowCbuff, rowCNum, rowLen);
    MPI_Comm_free(&gridComm);
    MPI_Finalize();
    return 0;

```

}

ПРИЛОЖЕНИЕ Б

КОД ПРОГРАММЫ ПОСЛЕДОВАТЕЛЬНОГО УМНОЖЕНИЯ МАТРИЦ

```
#include <iostream>
#include <vector>
#include <random>
#include <fstream>
#include <chrono>
#define MATR_SIZE 9
double** newMatrix(int rows, int columns);
void delMatrix(int rows, int columns, double** matr);
void printMatrixTo(int rows, int columns, double** matr, std::ostream&
stream);
void randomizeMatrix(int rows, int columns, double** matr);
bool multiplyMatrix(int rowsA, int columnsA, double** matrA, int rowsB, int
columnsB, double** matrB, int rowsC, int columnsC, double** matrC);
void readMatrixFrom(double** matr, int m, int n, std::istream& stream);

int main() {
    int m = MATR_SIZE;
    srand(time(NULL));
    double** A=newMatrix(m, m);
    double** B=newMatrix(m, m);
    double** C=newMatrix(m, m);
    randomizeMatrix(m, m, A);
    randomizeMatrix(m, m, B);
    std::chrono::steady_clock::time_point begin =
std::chrono::steady_clock::now();
    multiplyMatrix(m, m, A, m, m, B, m, m, C);
    std::chrono::steady_clock::time_point end =
std::chrono::steady_clock::now();
    printMatrixTo(m, m, C, std::cout);
    std::cout<<"Time: "<<
std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count();
    delMatrix(m, m, A);
    delMatrix(m, m, B);
    delMatrix(m, m, C);
    return 0;
}

double** newMatrix(int rows, int columns) {
    double **matr = new double* [rows];
    for (int i = 0;i < rows;i++) {
        matr[i] = new double[columns];
    }
    return matr;
}

void printMatrixTo(int rows, int columns, double** matr, std::ostream& stream)
{
    for (int i = 0;i < rows;i++) {
        for (int j = 0;j < columns;j++)
            stream << matr[i][j] << " ";
        stream << std::endl;
    }
}

void readMatrixFrom(double **matr,int m, int n, std::istream& stream) {
    for (int i = 0;i < m;i++)
        for (int j = 0;j < n;j++)
```

```

        stream >> matr[i][j];
    }

void randomizeMatrix(int rows, int columns, double** matr)
{
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < columns; j++)
            matr[i][j] = (double)rand() / RAND_MAX;
}

void delMatrix(int rows, int columns, double** matr) {
    for (int i = 0; i < rows; i++)
        delete[] matr[i];
    delete[] matr;
}

bool multiplyMatrix(int rowsA, int columnsA, double** matrA, int rowsB, int
columnsB, double** matrB, int rowsC, int columnsC, double** matrC)
{
    if (columnsA != rowsB)
        return false;
    if (rowsA != rowsC || columnsB != columnsC)
        return false;
    for (int i = 0; i < rowsC; i++) {
        for (int j = 0; j < columnsC; j++)
            matrC[i][j] = 0;
        for (int j = 0; j < rowsB; j++) {
            for (int k = 0; k < columnsC; k++) {
                matrC[i][k] += matrA[i][j] * matrB[j][k];
            }
        }
    }
    return true;
}

```