Х`МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №6

по дисциплине «Параллельные алгоритмы»

Тема: Умножение матриц

Студент гр. 1384 _____ Усачева Д. В.

Преподаватель _____ Татаринов Ю. С.

Санкт-Петербург

2023

Цель

Цель работы заключается в изучении работы с виртуальными топологиями, их управлением и функциями в библиотеке MPI.

Задание

Вариант 4. Блочный алгоритм Фокса.

Выполнить задачу умножения двух квадратных матриц A и B размера m × 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.

Непараллельный алгоритм умножения матриц (должен быть реализован во всех вариантах на одном процессе).

Выполнение работы

Алгоритм Фокса.

Предполагается, что все матрицы являются квадратными размера m×m, количество блоков по горизонтали и вертикали являются одинаковым и равным q (т.е. размер всех блоков равен k×k, k=n/q).

При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками. С учетом сказанного определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы C.

Итак, за основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы C и при этом в подзадачах на каждой итерации расчетов располагается только по одному блоку исходных матриц A и B. Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы C, т.е. подзадача (i,j) отвечает за вычисление блока Cij – тем самым, набор подзадач

образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

В соответствии с алгоритмом Фокса в ходе вычислений на каждом процессе с условными координатами в квадратной сетке (i, j) располагается четыре матричных блока:

− блок Cij матрицы C, вычисляемый подзадачей;

− блок Aij матрицы A, размещаемый в подзадаче перед началом вычислений;

− блоки A_ij, B_ij матриц A и B, получаемые подзадачей в ходе выполнения вычислений.

Выполнение параллельного метода включает:

• этап инициализации, на котором каждом процессе (i, j) передаются блоки Aij, Bij и обнуляются блоки Cij;

• этап вычислений, в рамках которого на каждой итерации l, 0≤l <q, осуществляются следующие операции:

− для каждой строки i, 0≤ i <q, блок Aij процесса (i, j) пересылается на все процессы той же строки i решетки;

− полученные в результаты пересылок блоки A_ij, B_ij каждого процесса (i, j) перемножаются и прибавляются к блоку Cij.

− блоки B'ij каждого процесса (i, j) пересылаются подзадачам, являющимися соседями сверху в столбцах решетки процессов (блоки процессов из первой строки решетки пересылаются процессам последней строки решетки).

Для выполнения поставленной задачи написано две программы на языке C, код блочный алгоритм Фокса представлен ниже в листинге 1, а код непараллельный алгоритма умножения матриц в листинге 2.

Для реализации непараллельного алгоритма реализованы функции ввода (предусмотрен ввод матрицы через консоль, по умолчанию матрица заполняется 1), вывода и умножения матриц. В теле главной функции происходит выделение памяти для массивов заданного пользователем

размера, вызываются необходимые функции. В конце программы происходит освобождение выделенной памяти.

Для реализации блочного алгоритма Фокса так же были реализованы функции ввода, вывода, умножения матриц. Функция ввода сохраняет полученную матрицу во временную переменную и преобразует матрицу в формат удобный для пересылки (блочный) между процессами. Для наглядности преобразования приведем пример:

0 1 2 3 4 5		0 1 2 | 3 4 5
6 7 8 9 10 11		6 7 8 | 9 10 11
12 13 14 15 16 17	(кол-во процессов = 9)	12 13 14| 15 16 17
18 19 20 21 22 23	➔	18 19 20| 21 22 23
24 25 26 27 28 29		24 25 26| 27 28 29
30 31 32 33 34 35		30 31 32| 33 34 35

➔ 0 1 2 4 7 8 12 13 14 3 4 5 9 10 11 15 16 17 18 19 20 24 25 26 30 31 32 21 22 23 27 28 29 33 34 35. Далее в таком виде будут храниться матрицы A, B, и C.

Функция вывода реализована для вывода матриц блочного формата.

Функция умножения представляет собой функцию для перемножения блоков (в отличие от обычной функции перемножения матриц, эта функция суммирует результат ранее перемноженных блоков с нынешними).

Для реализации алгоритма использовалась виртуальная топология декартовой решетки. Для ее создания использовалась функция MPI_Cart_create.

Для наилучшей работы алгоритма, считаем, что количество процессов – полный квадрат. Размерность матрицы должна нацело делиться на корень из количества процессов (если не делится, то дополняем нулями входную матрицу до ближайшей подходящей). Сначала происходит этап инициализации – отправка необходимых блоков матрицы каждому процессу. Далее идет нулевой шаг алгоритма, он находится вне цикла, так как нет

необходимости в пересылке матрицы B_ij, а также в нем определяются номера процессов, содержащих диагональные блоки матрицы. Номер столбца и строки процесса определяется при помощи функции MPI_Cart_shift. Отправка блоков матрицы A в каждой строке идет слева направо, начиная с диагонали. Процесс рассылает свой блок Aij всем остальным процессам строки. После в цикле реализованы остальные шаги алгоритма. Сначала происходит пересылка матрицы B_ij. Происходит передача сообщений по кольцу столбца, на каждом шаге процесс пересылает матрицу своему соседу сверху. Определения рангов соседей сверху и снизу происходит при помощи функции MPI_Cart_shift. Далее идет пересылка Aij и их перемножение.

После выполнения алгоритма при помощи коллективной операции MPI_Gather нулевой процесс собирает все блоки матрицы C и выводит результат. В конце программы происходит освобождение выделенной памяти.

Ниже представлена сеть Петри основной части алгоритма (см. рис 1).

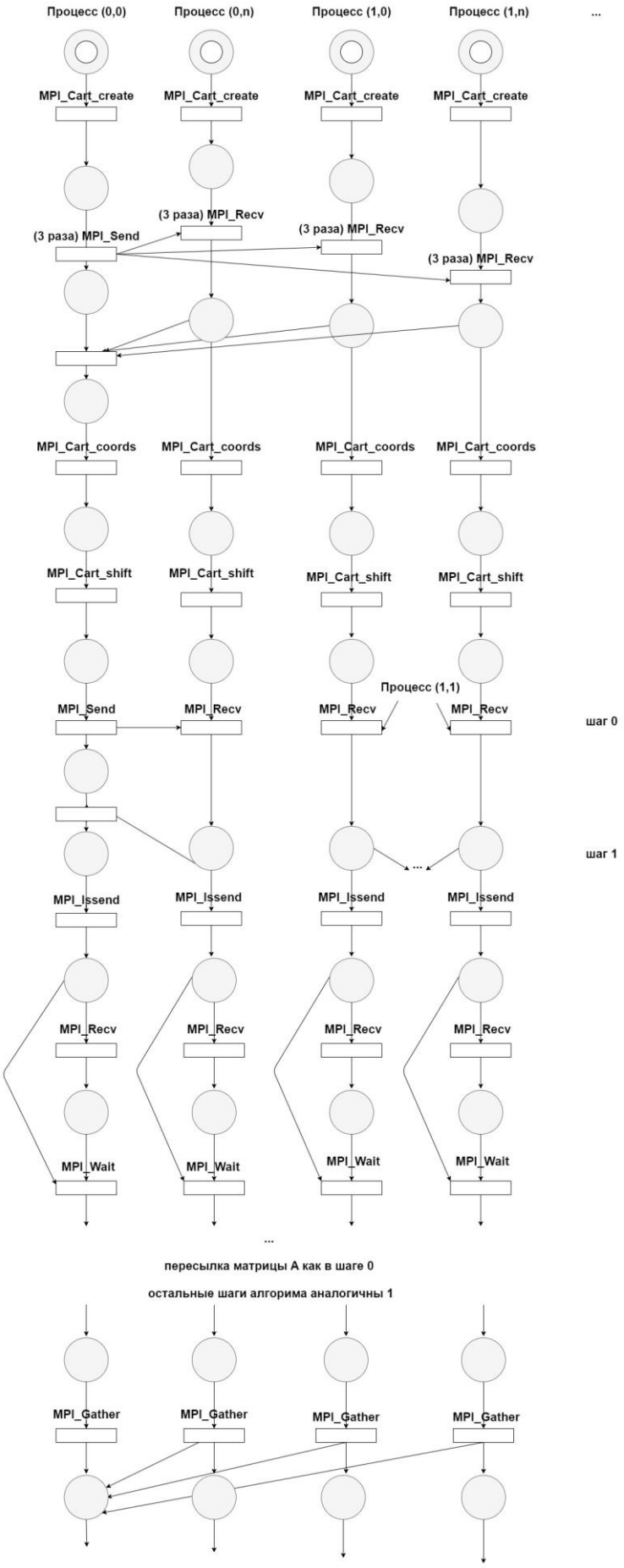

Рисунок 1 — Сеть Петри основной части алгоритма

Листинг 1 — Код программы lab6_1.c

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

void printRes(int* matrix, int k, int q) {
    printf("\nРезультат умножения матриц:\n");
    for (int i = 0; i < q ; i ++) { //строка по блокам
        for (int l = 0; l < k; l++) { // строка по элементам блока
            for (int j = 0; j < q; j++) { // столбец по блокам
                for (int n = 0; n < k; n++) { // столбец по элементам
блока
                    printf("%d ",matrix[n + l * k + j * k * k + i * k *
k * q]);
                }
            }
            printf("\n");
        }
    }
}

void inputMatrix(int* matrix, int m, int procNum, int k, int r){
    int** data = (int**)malloc(m * sizeof(int*));
    for (int i = 0; i < m; i++) {
        data[i] = (int*)malloc(m * sizeof(int*));
    }
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            if (i > r - 1 || j > r - 1)
                data[i][j] = 0;
            else
                data[i][j] = i * m + j;
        }
    }
    int count = 0;
    int column = 0;
    int row = 0;
    for (int num = 0; num < procNum; num++){
        for (int i = 0; i < k; i++){
            for (int j = 0; j < k; j++){
                matrix[count] = data[i+row][j+column];
                count++;
            }
        }
        column += k;
        if (column == m){
            column = 0;
            row += k;
        }
    }
    for (int i = 0; i < m; i++) {
        free(data[i]);
    }
    free(data);
}

void multiply(int* matrix1, int* matrix2, int* result, int m) {
    int row;
    int column;
    for (int i = 0; i < m * m; i++) {
        column = i % m;
        for (int j = 0; j < m; j++) {
```

```
                if (i % m == 0 ) row = i / m;
                result[i]+= matrix1[j + m * row] * matrix2[column + j *m];
            }
        }
    }

    int main(int argc, char** argv) {
        int procRank, procNum;
        int m, q, k;
        int* A;
        int* B;
        int* C;
        double start;
        int dims[2];
        int periods[2] = {1, 1};
        int reorder = 1;
        int mycoords[2];
        MPI_Init(&argc, &argv);
        MPI_Comm cartComm;
        MPI_Request request;
        MPI_Status send_status, rec_status;
        MPI_Comm_size(MPI_COMM_WORLD, &procNum);
        q = sqrt(procNum);
        dims[0] = dims[1] = q;
        if (q * q != procNum) {
            printf("Количество процессов должно быть полным квадратом.\n");
            MPI_Finalize();
            return 0;
        }
        MPI_Cart_create(MPI_COMM_WORLD,   2,   dims,   periods,   reorder,
&cartComm);
        MPI_Comm_rank(cartComm, &procRank);
        if (procRank == 0) {
            int r = 0;
            printf("Размерность матриц должна быть не меньше корня из
количества процессов.\n");
            while (r < q) {
                printf("Введите размерность матриц:\n");
                scanf("%d", &r);
            }
            m = r;
            while(m % q != 0) {
                m++;
            }
            k = m / q ;
            A = (int*)malloc(k * k * procNum * sizeof(int));
            B = (int*)malloc(k * k * procNum * sizeof(int));
            C = (int*)malloc(k * k * procNum * sizeof(int));
            inputMatrix(A, m, procNum, k, r);
            inputMatrix(B, m, procNum, k, r);
            start = MPI_Wtime();
            for (int i = 1; i < procNum; i++) {
                MPI_Send(&m, 1, MPI_INT, i, 0, cartComm);
                MPI_Send(&A[i * k * k], k * k, MPI_INT, i, 0, cartComm);
                MPI_Send(&B[i * k * k], k * k, MPI_INT, i, 0, cartComm);
            }
        }
        else {
            MPI_Recv(&m, 1, MPI_INT, 0, 0, cartComm, MPI_STATUS_IGNORE);
            k = m / q ;
        }
        int* Aij;
        int* A_ij;
        int* B_ij;
```

```
            int* Cij;
            Aij = (int*)malloc(k * k * sizeof(int));
            A_ij = (int*)malloc(k * k * sizeof(int));
            B_ij = (int*)malloc(k * k * sizeof(int));
            Cij = (int*)malloc(k * k * sizeof(int));
            for (int i = 0; i < k * k; i++) {
                Cij[i] = 0;
            }
            if (procRank == 0) {
                for (int i = 0; i < k * k; i++) {
                    Aij[i] = A[i];
                    B_ij[i] = B[i];
                }
            }
            else{
                MPI_Recv(&Aij[0],   k   *   k,   MPI_INT,   0,   0,   cartComm,
MPI_STATUS_IGNORE);
                MPI_Recv(&B_ij[0],   k   *   k,   MPI_INT,   0,   0,   cartComm,
MPI_STATUS_IGNORE);
            }
            MPI_Cart_coords(cartComm, procRank, 2, mycoords);
            int row = mycoords[0];
            int column = mycoords[1];
            int sendRankB;
            int recvRankB;
            int sendRankA;
            MPI_Cart_shift(cartComm, 0, 1, &sendRankB, &recvRankB);
            if (row == column){
                for (int i = 0; i < q ; i++) {
                    if (i != row){
                        MPI_Send(&Aij[0],  k  *  k,  MPI_INT,  row  *  q  +  i,  0,
cartComm);
                    }
                }
                multiply(Aij, B_ij, Cij, k);
                sendRankA = (procRank + 1) % q + row *q;
            }
            else{
                sendRankA = row * q + row;
                MPI_Recv(&A_ij[0],  k  *  k,  MPI_INT,  sendRankA,  0,  cartComm,
MPI_STATUS_IGNORE);
                multiply(A_ij, B_ij, Cij, k);
                sendRankA = (sendRankA + 1) % q + row *q;
            }
            for (int step = 1; step < q; step++) {//количество итераций
                MPI_Issend(&B_ij[0],  k  *  k,  MPI_INT,  sendRankB,  0,  cartComm,
&request);
                MPI_Recv(&B_ij[0],  k  *  k,  MPI_INT,  recvRankB,  0,  cartComm,
&rec_status);
                MPI_Wait(&request, &send_status);
                if (procRank == sendRankA){
                    for (int i = 0; i < q; i++) {
                        if (row * q + i != procRank){
                            MPI_Send(&Aij[0], k * k, MPI_INT, row * q + i, 0,
cartComm);
                        }
                    }
                    multiply(Aij, B_ij, Cij, k);
                    sendRankA = (sendRankA + 1) % q + row *q;
                }
                else{
                    MPI_Recv(&A_ij[0], k * k, MPI_INT, sendRankA, 0, cartComm,
MPI_STATUS_IGNORE);
                    multiply(A_ij, B_ij, Cij, k);
```

```
                sendRankA = (sendRankA + 1) % q + row *q;
            }

        }
        MPI_Gather(&Cij[0], k * k, MPI_INT, C, k * k, MPI_INT, 0, cartComm);
        if ( procRank == 0) {
            // printRes(C, k, q);
            printf("Время работы программы: %f\n",1000*(MPI_Wtime() -
start));
            free(A);
            free(B);
            free(C);
        }
        free(Aij);
        free(A_ij);
        free(B_ij);
        free(Cij);
        MPI_Comm_free(&cartComm);
        MPI_Finalize();
        return 0;
    }
```

Ниже представлен вывод программы lab5.c

Листинг 2 — Код программы lab6_2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void inputMatrix(int** matrix, int m){
    if (m <= 10) printf("Введите элементы матрицы:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            if (m <= 10) scanf("%d", &matrix[i][j]);
            else matrix[i][j] = 1;
        }
    }
}

void multiply(int** matrix1, int** matrix2, int** result, int m) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            result[i][j] = 0;
            for (int k = 0; k < m; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}

void printMatrix(int** matrix, int m) {
    printf("Результат перемножения матриц:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    clock_t start;
    int m;
    printf("Введите размерность матриц:\n");
```

```
        scanf("%d", &m);
        int** A = (int**)malloc(m * sizeof(int*));
        int** B = (int**)malloc(m * sizeof(int*));
        int** C = (int**)malloc(m * sizeof(int*));
        for (int i = 0; i < m; i++){
            A[i] = (int*)malloc(m * sizeof(int));
            B[i] = (int*)malloc(m * sizeof(int));
            C[i] = (int*)malloc(m * sizeof(int));
        }
        inputMatrix(A, m);
        inputMatrix(B, m);
        start = clock();
        multiply(A, B, C, m);
        // printMatrix(C, m);
        printf("Время работы программы: %f\n", (double)(clock() - start) /
CLOCKS_PER_SEC);
        for (int i = 0; i < m; i++) {
            free(A[i]);
            free(B[i]);
            free(C[i]);
        }
        free(A);
        free(B);
        free(C);
        return 0;
    }
```

Листинг 3 — Вывод программы lab6_1.c для 4 и 64 процессов для матрицы 16*16

```
    Размерность матриц должна нацело делиться на корень из количества
процессов.
    Введите размерность матриц:
    16
    Время работы программы: 0.000853

    Размерность матриц должна нацело делиться на корень из количества
процессов.
    Введите размерность матриц:
    16
    Время работы программы: 0.013235
```

Листинг 4 — Вывод программы lab6_2.c для матрицы 16*16

```
    Введите размерность матриц:
    16
    Время работы программы: 0.000024
```

Рассмотрим зависимость времени работы программы от количества процессов и размерности матрицы. Время выполнения программы и ускорение приведены ниже в таблице 1.

Теоретическое время работы непараллельного алгоритма это m^3, где m – размерность матрицы.

Теоретическое время работы параллельного алгоритма это – m^3/p, где p – количество процессов.

Общий анализ сложности дает идеальные показатели эффективности параллельных вычислений. Ускорение работы программы при использование параллельного алгоритма теоретически равно количеству процессов.

Таблица 1 – Результаты работы программы на разном количестве процессов.

Размерность матриц (m)	Последовательный алгоритм	4 процесса		16 процессов	
	время	время	ускорение	время	ускорение
10	0,009	0,792	0,01136364	4,453	0,00202111
50	0,748	1,210	0,61818182	4,044	0,18496538
100	4,148	3,554	1,16713562	5,379	0,77114705
200	34,597	13,345	2,59250656	17,680	1,95684389
500	543,244	194,906	2,78721024	188,853	2,87654419
1000	8293,17	2001,480	4,1435188	1554,278	5,33570571
2000	81082,732	17485,834	4,6370526	12529,100	6,47155279

Размерность матриц (m)	25 процессов		64 процессов	
	время	ускорение	время	ускорение
10	13,179	0,0006829	69,453	0,00013
50	6,474	0,11553908	38,774	0,019291
100	7,170	0,57852162	32,547	0,127446
200	15,915	2,17386114	47,661	0,725897
500	202,596	2,68141523	252,470	2,151717
1000	1351,083	6,13816472	2195,115	3,778012
2000	15816,026	5,12661853	11616,591	6,979908

Ниже указаны графики зависимостей времени выполнения и ускорения (см. рисунки 2-3).

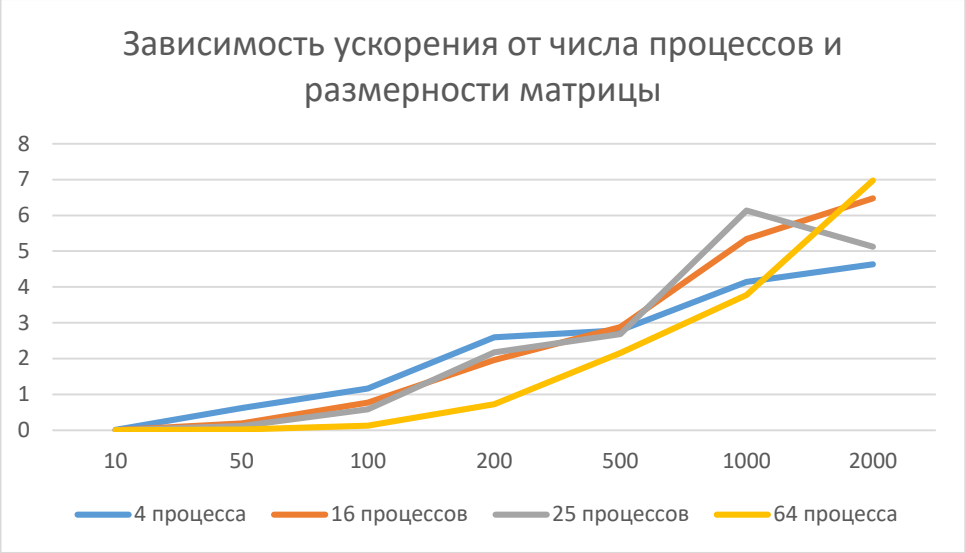

Рисунок 2 — График зависимости ускорения программы от размерности матриц

Теоретическая оценка времени в таблице представляет собой время выполнения задачи одним процессом, деленное на количество процессов.

Таблица 2 – Сравнение экспериментальных и теоретических оценок времени.

Размерность матриц (m)	Экспериментальное время работы	Теоретическое время работы
10	0,792	0,00225
50	1,210	0,187
100	3,554	1,037
200	13,345	8,64925
500	194,906	135,811
1000	2001,480	2073,293
2000	17485,834	20270,68

Выводы

В ходе выполнения лабораторной работы был реализован блочный алгоритм Фокса перемножения двух матриц. Он представляет собой параллельное перемножение блоков исходных матриц.

Полученные экспериментальные данные позволяют сделать вывод, что данный алгоритм наиболее эффективен для матриц больших размеров. Если же размер матрицы достаточно мал, использование параллельной программы менее эффективно, чем последовательный алгоритм.

В ходе сравнения экспериментального и теоретического времени работы параллельной программы очевидно, что мы не достигаем идеального ускорения, равного количеству процессов. Это связано с тем, что помимо вычислительной сложности, так же необходимо учитывать время на пересылку данных каждым процессом. Также различие может быть связано с техническими возможностями устройства, на котором проводилось тестирование.

По графику прослеживается увеличение ускорения для матриц больших размеров и замедление для небольших матриц при большом количестве процессов.