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

ОТЧЕТ

по лабораторной работе №7

по дисциплине «Системы параллельной обработки данных»

Тема: Умножение матриц

Студент гр. 0303 _____ Калмак Д.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2024

Цель работы.

Целью работы является написание последовательной программы и параллельной программы MPI с умножением матриц, изучение зависимостей времени работы программы от числа процессов и построение сети Петри.

Задание.

1. Выполнить задачу умножения двух квадратных матриц A и B размера m × m, результат записать в матрицу C. Реализовать последовательный и параллельный алгоритм и провести анализ полученных результатов. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.
    - Непараллельный алгоритм умножения матриц
    - Ленточный алгоритм 1 (горизонтальные полосы)
2. Краткое описание выбранного алгоритма решения задачи и листинг программы. Описание выбранного принципа разбиения задачи на параллельные подзадачи. Описание информационных связей и обоснование выбора виртуальной топологии. Описание распределение подзадач по процессам (задача масштабируемости).
3. Нарисовать сеть Петри для MPI программы.
4. Распечатка результатов и времени работы программы на разном количестве процессов.
5. Построить графики зависимости времени выполнения программы от числа запущенных процессов. Теоретическая оценка времени работы алгоритмов и сравнение с экспериментальными данными.
6. Построить графики ускорения/замедления программы.

Выполнение работы.

1-2. Для умножения квадратных матриц написана последовательная программа, работающая на одном процессе. Функция printMatr(const vector<int> &matrix, int K, int n) принимает аргументы с вектором матрицы matrix, количеством столбцов K и количеством строк n, чтобы построчно вывести матрицу. Для последовательного умножения предназначена функция seq_mult_matr(const vector<int> &A, const vector<int> &B, int K), которая принимает вектора матриц A и B, которые необходимо умножить, и размерность матрицы K. Вектор C изначально инициализирован нулями и после прохождения трех циклов for для умножения матриц в результате имеет результат умножения матриц A и B. Функция seq_mult_matr возвращает вектор C. При запуске программы передается параметр K, который сообщает размер матрицы, что позволяет не менять переменную внутри кода. Функция stoi приводит ее к типу int. Для ввода матриц создан цикл for, который работает K раз, то есть соразмерно матрице. Используются функция getline для построчного считывания, а класс stringstream, что представлен в библиотеке sstream, позволяет удобно преобразовать строку матрицы с пробелами к элементам вектора матрицы. Для замера времени используется библиотека chrono и ее функция steady_clock::now(). Программа представлена в листинге 1. Пример работы программы представлен в листинге 2.

Листинг 1.

```
#include <iostream>
#include <vector>
#include <chrono>
#include <sstream>

using namespace std;
using namespace std::chrono;

void printMatr(const vector<int> &matrix, int K, int n) {
    for (int i = 0; i < n*K; i++) {
        if (i % K == 0 && i != 0){
            cout << endl;
        }
```

```
        cout << matrix[i] << " ";
    }
    cout << endl;
}

vector<int> seq_mult_matr(const vector<int> &A, const vector<int> &B, int K){
    vector<int> C(K * K, 0);

    for (int i = 0; i < K; i++) {
        for (int j = 0; j < K; j++) {
            for (int q = 0; q < K; q++) {
                C[j + i * K] += A[q + i * K] * B[j + q * K];
            }
        }
    }

    return C;
}


int main(int argc, char* argv[]) {
    // int K = 4;
    int K = stoi(argv[1]);
    vector<int> A(K*K, 1);
    // vector<int> A(K*K);
    // for (int i = 0; i < K; ++i) {
    //     string line;
    //     getline(cin, line);

    //     stringstream ss(line);
    //     for (int j = 0; j < K; ++j) {
    //         ss >> A[i * K + j];
    //     }
    // }
    vector<int> B(K*K, 2);
    // vector<int> B(K*K);
    // for (int i = 0; i < K; ++i) {
    //     string line;
    //     getline(cin, line);

    //     stringstream ss(line);
    //     for (int j = 0; j < K; ++j) {
    //         ss >> B[i * K + j];
    //     }
    // }
    vector<int> C;

    auto start = steady_clock::now();
    C = seq_mult_matr(A, B, K);
    auto end = steady_clock::now();
    duration<double> duration = end - start;
    cout << "Time " << duration.count() << endl;

    printMatr(C, K, K);

    return 0;
}
```

Листинг 2.

```
cut@cute:~/Документы/cpluscplus/7$ ./main_seq 4
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
25 26 27 28
29 30 31 32
Time 6.933e-06
250 260 270 280
618 644 670 696
986 1028 1070 1112
1354 1412 1470 1528
```

С использованием языка программирования C++ написана параллельная программа MPI для умножения матриц. Для работы с MPI была включена библиотека mpi.h. Программа также имеет функцию printMatr для построчного вывода матрицы. Для инициализации среды MPI используется MPI_Init. MPI_Comm_size позволяет получить общее количество процессов ProcNum, а MPI_Comm_rank – ранг текущего процесса ProcRank. Объявлены структуры sendStatus, recvStatus - MPI_Status, и req – MPI_Request, для использования MPI_Issend и MPI_Recv далее. Переменная K также получает значение параметра запуска. Если размер матрицы делится на количество процессов с остатком, то программа завершает свою работу. Переменная n содержит частное от K и ProcNum, тем самым определяя количество строк матрицы на каждый процесс. Каждый процесс выделяет память под векторы local_A, local_B и local_C размером n * K, а вектор local_C сразу инициализируется нулями. Также объявлен вектор C, который будет содержать результирующую матрицу после умножения, однако его размер определяется только в нулевом процессе, в котором и будут собраны части local_C в одно целое. Ввод матриц A и B производится в процессе с рангом 0 так же, как и последовательной программе.

Алгоритм в данной программе ленточный с горизонтальными полосами: строки матрицы A умножаются на строки матрицы B и получаются частичные результаты строк матрицы C. При суммировании этих частичных результатов

получается результирующие строки матрицы C. То есть каждая строка A определяет при умножении номер строки C: элемент матрицы A под тем же индексом, что и индекс строки матрицы B, умножается на каждый элемент строки матрицы B, таким образом получается частичный результат в строке C. Если умножить каждую строку матрицы A на каждую строку матрицы B, то суммируя эти частичные результаты, строки C станут результатом умножения матрицы A на матрицу B. (Если матрица 4 на 4, то просуммировав 1 * 1 + 1 * 2 + 1 * 3 + 1 * 4 по описанному выше правилу, получится первая строка матрицы C и т.д.) Данный алгоритм позволяет зафиксировать строки A по причине соотвествия номерам строк матрицы C, а строки B двигать между процессами, тем самым мы работаем не с целыми матрицами, а со строками матриц. На каждый процесс распредяется одинаковое число строк матрицы A и матрицы B, равное n, которые обозначаются матрицами local_A и local_B. В каждом процессе по алгоритму каждая строка local_A умножается по правилу на каждую строку local_B, и каждое умножение суммируется с предыдущими в local_C. Затем каждый процесс передает следующему свои строки матрицы local_B, и шаги повторяются столько раз, сколько всего процессов. Поскольку обмен строками происходит между соседями, используется кольцевая топология, что позволяет еще замкнуть первый и последний процессы для лучшего обмена.

Нулевой процесс с помощью коллективной операции MPI_Scatter(A.data(), n * K, MPI_INT, local_A.data(), n * K, MPI_INT, 0, MPI_COMM_WORLD) отправляет каждому процессу n строк матрицы A, или n * K элементов типа MPI_INT, и строки записываются в вектор local_A. Аналогично с помощью коллективной операции MPI_Scatter каждый процесс получает n строк матрицы B, или n * K элементов типа MPI_INT. С помощью функции MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, reorder, &ring_Comm) для всех процессов исходного коммуникатора

MPI_COMM_WORLD задана кольцевая топология размерностью 1, dims, определяет количество процессов, которое равно ProcNum, в измерении. periods[1] = {1}, что означает периодичность вдоль измерения, чтобы связать первый и последний процессы, а reorder = 0 позволяет отключить переупорядочивание процессов, ring_Comm – новый коммуникатор с кольцевой топологией процессов. С помощью функции MPI_Cart_shift(ring_Comm, 0, 1, &left, &right) каждый процесс определяет соседей left и right в виртуальной топологии коммуникатора ring_Comm. Переменная PrevRank необходима для контролирования индексации строк матрицы B и соответственно элемента вектора local_A, и изначально равна ProcRank. Запускается цикл, который работает ProcNum раз. Происходит с помощью трех циклов расчет частичного результата в строках local_C. Первый цикл работает n раз, то есть столько, сколько строк матрицы local_A. Второй цикл работает K раз, то есть столько, сколько символов в строке матриц. Третий цикл работает n раз, то есть столько, сколько строк матрицы local_B. Затем происходит перерасчет переменной PrevRank для контроля индекса строк матрицы B. После этого процесс отправляет следующему соседу right свои строки матрицы local_B, или n * K элементов типа MPI_INT, с помощью функции MPI_Issend(local_B.data(), n * K, MPI_INT, right, 0, ring_Comm, &req), что позволяет провести неблокирующую операцию отправки с гарантией, что процесс-получатель получит данные. Затем процесс получает строки матрицы local_B от своего left соседа с помощью функции MPI_Recv(local_B.data(), n * K, MPI_INT, left, 0, ring_Comm, &recvStatus), а затем выполняется блокирующая функция MPI_Wait(&req, &sendStatus), которая ждет завершения операции отправки. После выполнения всех циклов каждый процесс помечает коммуникатор ring_Comm для удаления. В результате каждый процесс имеет в local_C строки матрицы C, которая является результатом умножения, поэтому, чтобы соединить эти строки в одну матрицу C, используется коллективная операция MPI_Gather(local_C.data(), n *

K, MPI_INT, C.data(), n * K, MPI_INT, 0, MPI_COMM_WORLD), и после этого в нулевом процессе вектор C имеет конечную матрицу умножения. Программа представлена в листинге 3. Пример работы программы представлен в листинге 4.

Листинг 3.

```
#include <iostream>
#include <mpi.h>
#include <vector>
#include <sstream>

using namespace std;

void printMatr(const vector<int> &matrix, int K, int n) {
    for (int i = 0; i < n*K; i++) {
        if (i % K == 0 && i != 0){
            cout << endl;
        }
        cout << matrix[i] << " ";
    }
    cout << endl;
}

int main(int argc, char** argv) {
    int ProcNum, ProcRank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    MPI_Status sendStatus, recvStatus;
    MPI_Request req;

    // int K = 4;
    int K = stoi(argv[1]);

    if (K % ProcNum != 0) {
        if (ProcRank == 0) cout << "Wrong size of matrix!" << endl;
        MPI_Finalize();
        return 0;
    }
    int n = K / ProcNum;

    vector<int> local_A(n*K);
    vector<int> local_B(n*K);
    vector<int> local_C(n*K, 0);
    vector<int> C;

    double start, end;
    if (ProcRank == 0) {
        vector<int> A(K*K, 1);
        // vector<int> A(K*K);
        // for (int i = 0; i < K; ++i) {
        //     string line;
```

```
        //     getline(cin, line);

        //     stringstream ss(line);
        //     for (int j = 0; j < K; ++j) {
        //         ss >> A[i * K + j];
        //     }
        // }
        vector<int> B(K*K, 2);
        // vector<int> B(K*K);
        // for (int i = 0; i < K; ++i) {
        //     string line;
        //     getline(cin, line);

        //     stringstream ss(line);
        //     for (int j = 0; j < K; ++j) {
        //         ss >> B[i * K + j];
        //     }
        // }
        C.resize(K * K);
        start = MPI_Wtime();
        MPI_Scatter(A.data(), n * K, MPI_INT, local_A.data(), n * K, MPI_INT, 0,
MPI_COMM_WORLD);
        MPI_Scatter(B.data(), n * K, MPI_INT, local_B.data(), n * K, MPI_INT, 0,
MPI_COMM_WORLD);
    }
    else {
        MPI_Scatter(nullptr, n * K, MPI_DATATYPE_NULL, local_A.data(), n * K,
MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Scatter(nullptr, n * K, MPI_DATATYPE_NULL, local_B.data(), n * K,
MPI_INT, 0, MPI_COMM_WORLD);
    }

    int dims[1] = {ProcNum};
    int periods[1] = {1};
    int reorder = 0;
    MPI_Comm ring_Comm;

    MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, reorder, &ring_Comm);

    int left, right;
    MPI_Cart_shift(ring_Comm, 0, 1, &left, &right);

    int PrevRank = ProcRank;

    for (int i = 0; i < ProcNum; i++) {
        for (int j = 0; j < n; j++){
            for (int q = 0; q < K; q++){
                for (int s = 0; s < n; s++) {
                    local_C[q + K * j] += local_A[PrevRank * n + s + K * j] *
local_B[q + K * s];
                }
            }
        }
        PrevRank -= 1;
        if (PrevRank < 0){
            PrevRank = ProcNum - 1;
        }
        MPI_Issend(local_B.data(), n * K, MPI_INT, right, 0, ring_Comm, &req);
```

```
        MPI_Recv(local_B.data(),   n   *   K,   MPI_INT,   left,   0,   ring_Comm,
&recvStatus);
        MPI_Wait(&req, &sendStatus);
    }
    MPI_Comm_free(&ring_Comm);

    MPI_Gather(local_C.data(), n * K, MPI_INT, C.data(), n * K, MPI_INT, 0,
MPI_COMM_WORLD);

    if (ProcRank == 0) {
        end = MPI_Wtime();
        cout << "Time " << end - start << endl;
        // printMatr(C, K, K);
    }

    MPI_Finalize();
    return 0;
}
```

Листинг 4.

```
cut@cute:~/Документы/cpluscplus/7$ mpiexec.openmpi -n 4 --oversubscribe ./main 4
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
25 26 27 28
29 30 31 32
Time 0.000689838
250 260 270 280
618 644 670 696
986 1028 1070 1112
1354 1412 1470 1528
```

3. Построена сеть Петри. Сеть Петри представлена на рис. 1 и отражает параллельную программу для двух процессов, в которой происходит умножение матриц.

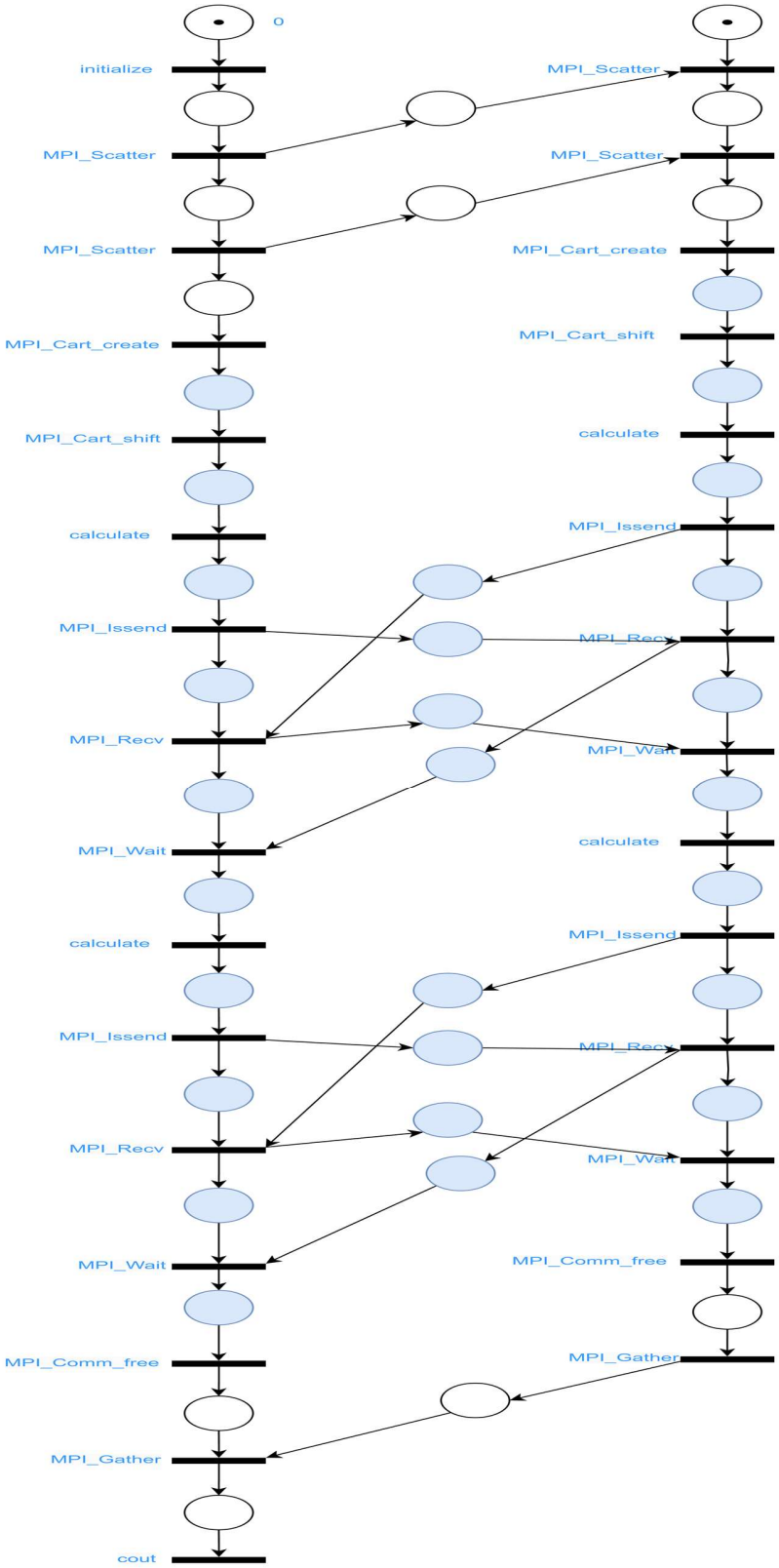

Рисунок 1 – Сеть Петри для параллельной программы с умножением матриц для двух процессов

4. Была запущена последовательная программа с разной размерностью матриц: 16, 64, 256, 512, 1024. Время выполнения программы в зависимости от размерности матрицы представлено в табл. 1. Также была запущена параллельная программа на разных числах процессов: 1, 4, 8, 16, 32, и так же с разной размерностью матриц: 16, 64, 256, 512, 1024. Время выполнения программы в зависимости от количества процессов и размерности матрицы представлено в табл. 1-2. Компиляция последовательной программы и параллельной программы осуществлена с флагом оптимизации –O2.

Таблица 1. Результат работы последовательной программы и параллельной программы на 1, 4 и 8 процессах

Размерность матриц (K)	Последова-тельный алгоритм	1 процесс		4 процесса		8 процессов	
	время	время	ускорение	время	ускорение	время	ускорение
16	4.087e-06	0.000325599	0.01255	0.000596481	0.00684	0.000753357	0.00543
64	0.00018598	0.00052035	0.357	0.00065473	0.284	0.00082049	0.227
256	0.0204744	0.0156494	1.308	0.0049588	4.131	0.00540808	3.785
512	0.604235	0.623817	0.968	0.138505	4.36	0.0404514	14.93
1024	13.446	14.3508	0.936	3.45607	3.89	2.13339	6.31

Таблица 2. Результат работы параллельной программы на 16, 32 процессах

Размерность матриц (K)	16 процесса		32 процесса	
	время	ускорение	время	ускорение
16	0.00299894	0.00136	-	-
64	0.0035753	0.052	0.00935793	0.01988
256	0.0108406	1.89	0.0168053	1.218
512	0.0551608	10.95	0.0658703	9.174
1024	2.5635	5.25	0.685377	19.61

5. На основе полученных результатов, представленных в табл. 1-2, построены графики зависимости времени выполнения программы от числа запущенных процессов: 1, 4, 8, 16, 32, при разных размерностях матрицы: 64, 256, 512, 1024, - которые представлены на рис. 2-5 соответственно.

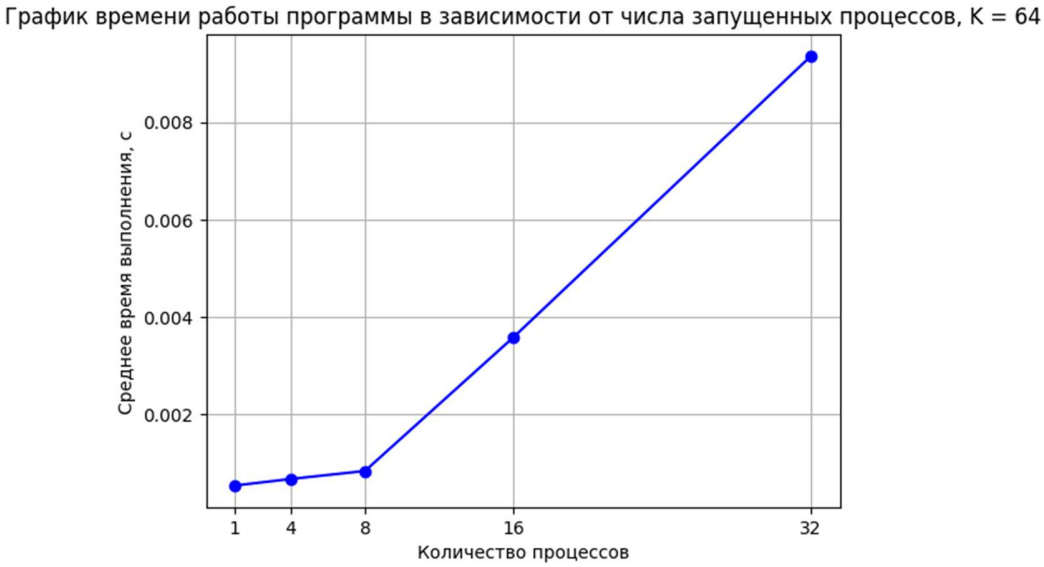

Рисунок 2 – График времени работы программы в зависимости от числа запущенных процессов, K = 64

Рисунок 3 – График времени работы программы в зависимости от числа запущенных процессов, K = 256

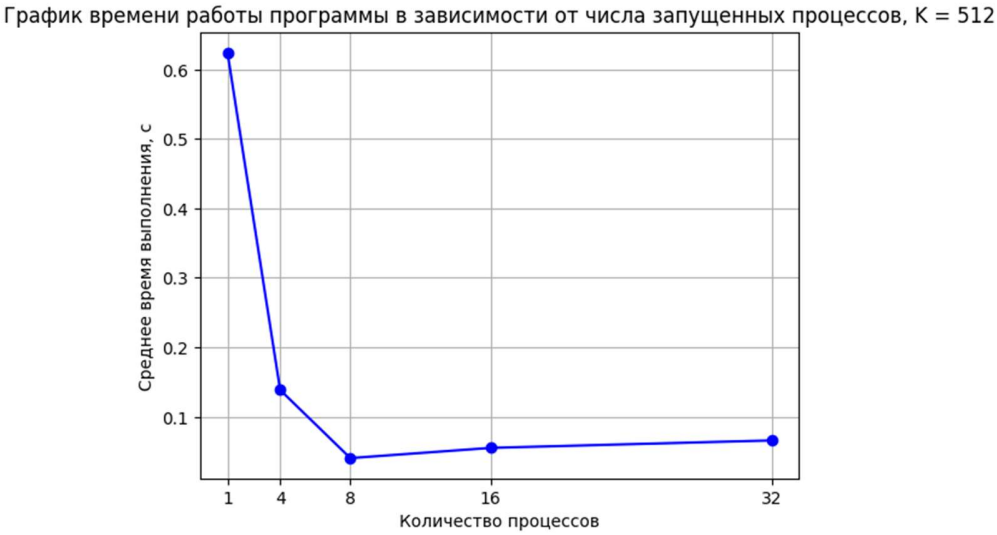

Рисунок 4 – График времени работы программы в зависимости от числа запущенных процессов, K = 512

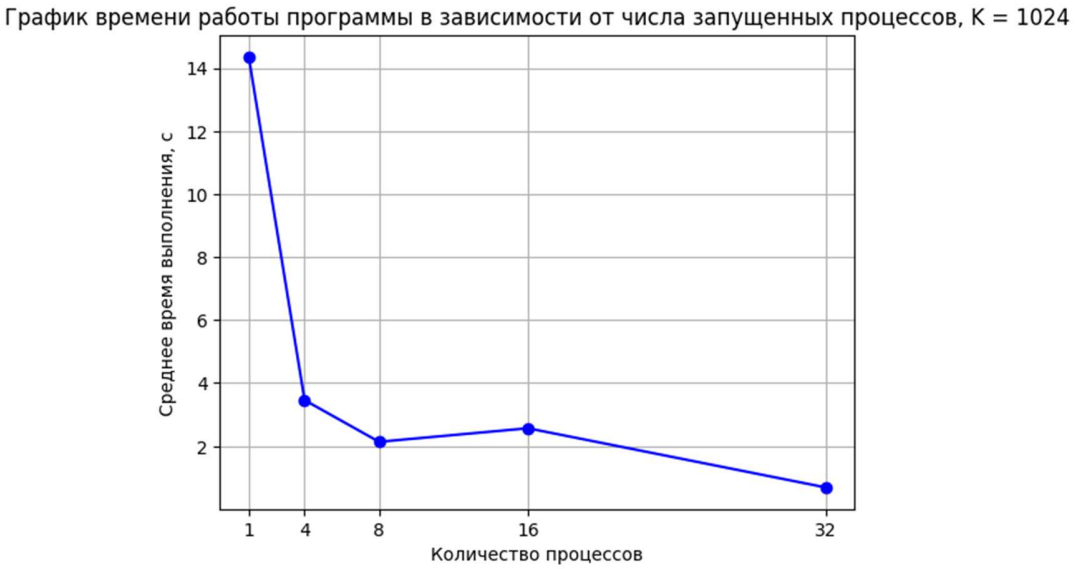

Рисунок 5 – График времени работы программы в зависимости от числа запущенных процессов, K = 1024

Исходя из графиков, представленных на рис. 2-5, на малой размерности матрицы время работы параллельной программы увеличивается, что вызвано нагрузкой коллективных операций рассылки и сбора данных, созданием виртуальной топологии, освобождением нового коммуникатора, а также

передачей данных между процессами при расчетах. На матрицах с размерностью больше замечено ускорение, которое стабильно выше единицы, на параллельной программе на 4, 8, 16, 32 процессах, на 1 процессе ускорение замечено в одном случае, в остальных оно было чуть меньше единицы, что вызвано дополнительными затратами MPI. Ускорение растет до определенного момента: 4-8 процессов, что связано с характеристиками вычислительной машины. Исключением являются 32 процесса при матрице 1024, которые показали наибольшее значение ускорения за все эксперименты, равное 19.61. Ускорение параллельной программы при больших матрицах говорит об эффективном разбиении задачи на параллельные подзадачи, распределении подзадач по процессам и использовании выбранной виртуальной топологии. Время выполнения программы может быть разным при одинаковых условиях, что может быть вызвано затратами на коллективные операции, создание кольцевой топологии, передачей и приемом данных между процессами, пометками удаления нового коммуникатора, а также разным состоянием вычислительной машины.

Определим теоретическое время работы последовательной и параллельной программы. Размерность матрицы n. При последовательном алгоритме $C_{ij} = \sum_{k=1}^{n} A[i][k] * B[k][j]$. Сложность получения одного C_{ij} O(n), а таких элементов в матрице n*n, то есть n^2. Алгоритм проходит тройной цикл с n шагами. Итоговая сложность $O(n^3)$. При параллельном алгоритме процессы работают с полосами матрицы, а эти полосы определены как $\frac{n}{p}$, где p – количество процессов, соотвественно число операций на процессе $\frac{n^3}{p^2}$, при этом алгоритм работает столько же раз, сколько и процессов. Алгоритм проходит цикл p раз, в котором тройной цикл $\frac{n}{p}$ шагов, n шагов и $\frac{n}{p}$ шагов. Итоговая сложность $O(\frac{n^3}{p^2} * p) = O\left(\frac{n^3}{p}\right)$. Получается, что при росте числа процессов время

работы программы сокращается в идеальном случае, а ускорение соотвествует числу процессов. Однако на практике такие результаты получить тяжело из-за дополнительных затрат на передачу и прием данных, а также характеристик вычислительной машины и ее состояния. В проведенных экспериментах лучше всего в соответствии с теорией повела параллельная программа, работающая на 4 процессах.

6. Для построения графика ускорения/замедления работы программы с разным количеством запущенных процессов необходимо разделить время выполнения последовательной программы на время выполнения параллельной программы с 1, 4, 8, 16, 32 процессами соотвественно. Расчитанные данные представлены в табл. 1-2. Графики ускорения/замедления представлены на рис. 6-9.

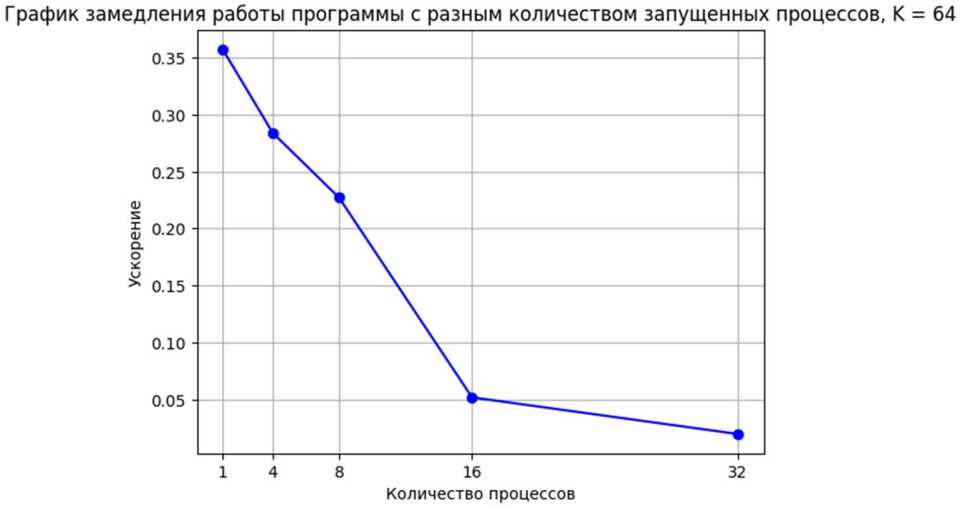

Рисунок 6 – График замедления работы программы с разным количеством запущенных процессов, K = 64

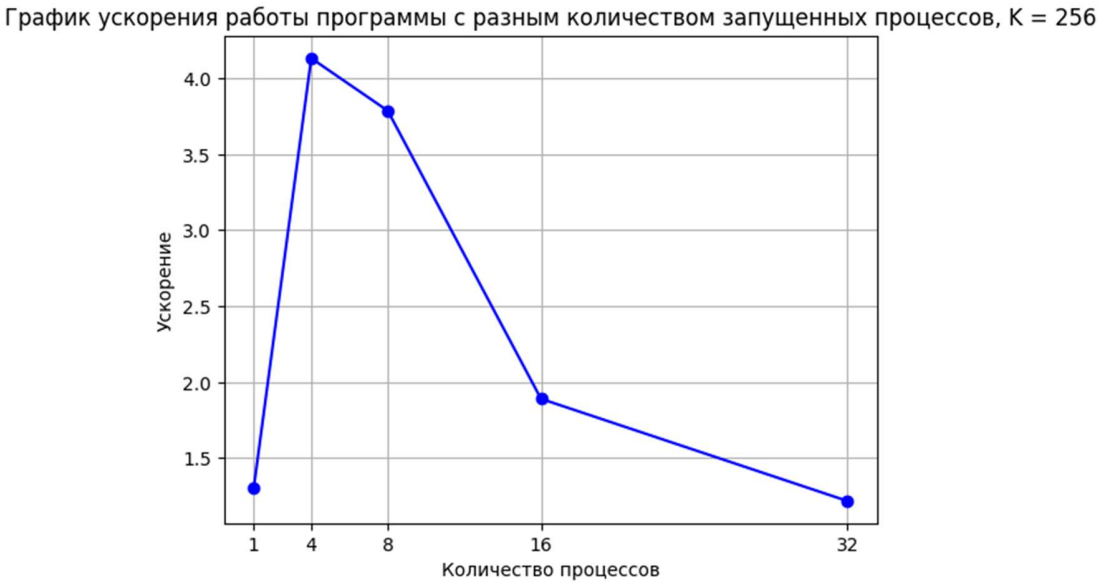

Рисунок 7 – График ускорения работы программы с разным количеством запущенных процессов, K = 256

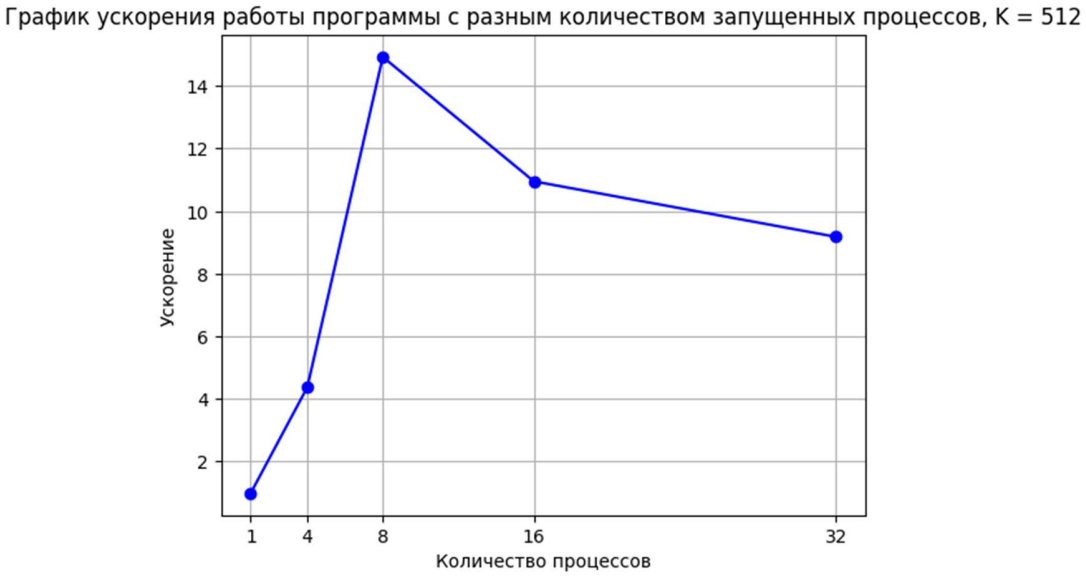

Рисунок 8 – График ускорения работы программы с разным количеством запущенных процессов, K = 512

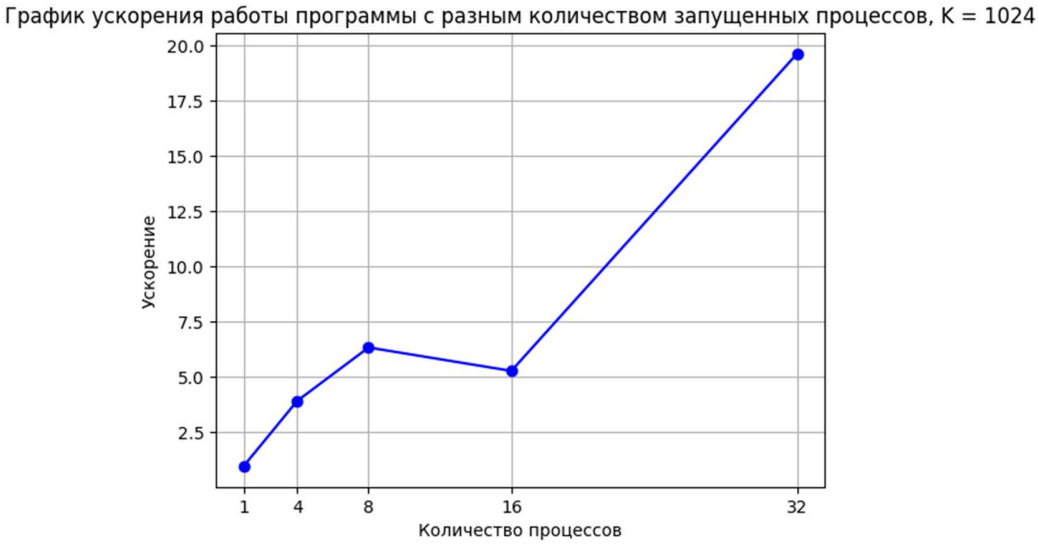

Рисунок 9 – График ускорения работы программы с разным количеством запущенных процессов, K = 1024

Вывод.

Таким образом, было освоено написание параллельных программ для умножения матриц с использованием полученных ранее навыков работы с библиотекой MPI. Для сравнения работы параллельной программы была написана последовательная программа, работающая на одном процессе. В параллельной программе был использован ленточный алгоритм с горизонтальными полосами и кольцевая виртуальная топология.

Были изучены зависимости времени работы параллельной программы от числа запущенных процессов и размерности матрицы и последовательнной программы от размерности матрицы. На малой размерности матрицы время работы параллельной программы возрастает с увеличением количества процессов, что связано с нагрузкой коллективных операций, передачей и приемом данных, созданией топологии и освобождением коммуникатора. На большой размерности матрицы заметно стабильное ускорение по сравнению с последовательной программой кроме работы на одном процессе, в котором только в одном случае было ускорение, что говорит о необходимости

распределения подзадач на процессы. Рост ускорения происходил до 4-8 процессов, что связано как с вычислительной машиной, так и с нагрузками. Лучшее ускорение было получено на 32 процессах и равно 19.61. Проведена теоретическая оценка времени работы последовательного и параллельного алгоритма, в результате которой параллельный алгоритм должен иметь ускорение, соответствующее количеству процессов, однако на практике такой результат получить затруднительно, и в проведенных экспериментах только программа с 4 запущенными процессами показала приблизительные результаты. На основании полученных данных о времени работы программы построены графики времени работы программы в зависимости от числа запущенных процессов и размерности матриц и соответствующие им графики ускорения/замедления работы программы. Для программы была построена сеть Петри.