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

ОТЧЕТ

по лабораторной работе №7
по дисциплине «Параллельные алгоритмы»
Тема: Умножение матриц

Студентка гр. 2384 _____ Соц Е.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2024

Цель работы

Целью данной лабораторной работы является реализация алгоритма умножения двух квадратных матриц с использованием последовательного и параллельного подходов с использованием библиотеки MPI. В частности, в параллельной реализации используется блочный алгоритм Фокса. Необходимо провести анализ полученных результатов, включая временные затраты и особенности реализации.

Задание

Вариант 4

Выполнить задачу умножения двух квадратных матриц A и B размера m × m, результат записать в матрицу C. Реализовать последовательный и параллельный алгоритм – блочный алгоритм Фокса. Провести анализ полученных результатов. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

Выполнение работы

Последовательный подход

В последовательной реализации для умножения двух матриц используется тройной вложенный цикл, где для каждой строки одной матрицы производится умножение на каждый столбец второй матрицы. Этот метод является интуитивным, но обладает рядом ограничений:

- Высокие временные затраты O(n^3), что делает алгоритм медленным для больших матриц.
- Ограничения по памяти и производительности на одиночных вычислительных узлах при увеличении размеров матриц.

Параллельный подход

В данной работе используется алгоритм Фокса, который оптимизирует умножение матриц за счет:

1. **Разбиения** исходных матриц на подблоки.
2. **Распределения** подблоков между процессами.
3. **Параллельного вычисления** локальных произведений блоков.
4. **Обмена данными** между процессами.

Благодаря блочному разбиению и передаче только необходимых данных, алгоритм Фокса позволяет минимизировать объем передаваемой информации и повысить производительность.

Этапы алгоритма:

1. Все процессы получают подблоки матриц A и B.
2. На каждом этапе один из процессов (в строке) передает свой блок A остальным процессам в строке.

3. Процессы выполняют локальное умножение блока A на блок B и обновляют свои локальные части результирующей матрицы C.
4. Блоки B передаются между процессами в столбце для следующего этапа умножения.
5. После завершения всех этапов локальные блоки C собираются в главном процессе для формирования итоговой матрицы.

Виртуальная топология

Для эффективной коммуникации между процессами используется двумерная топология, 2D-сетка:

- Размерность сетки определяется как dim×dim, где dim = sqrt(size), a size — общее число процессов.
- Карта сети создается с помощью функции MPI_Cart_create, обеспечивая:
    - Логическую структуру для пересылки данных внутри строк (коммуникатор row_comm).
    - Аналогичную структуру для столбцов (col_comm).

Выбор 2D-топологии обусловлен:

1. Минимизация затрат на коммуникацию: данные передаются только между соседними процессами.
2. Параллельность обмена данными: пересылка происходит одновременно по строкам и столбцам.

Реализация программы:

В рамках данной задачи выполнено умножение двух квадратных матриц с использованием параллельного алгоритма Фокса. Программа разработана на языке C++ с применением библиотеки MPI. Основное внимание уделено построению декартовой топологии процессов, разбиению

матриц на блоки, обеспечению передачи данных между процессами и выполнению вычислений в каждом процессе.

Для организации вычислений в параллельной среде используется функция MPI_Cart_create, формирующая двумерную топологию процессов. Размер решетки определяется как квадратный корень от числа процессов. Коммуникатор grid_comm координирует взаимодействие между процессами внутри сетки, а функция MPI_Cart_coords предоставляет координаты текущего процесса в этой топологии.

Затем топология разделяется на строки и столбцы с помощью MPI_Cart_sub, что позволяет создать два подкоммуникатора.

На первом этапе главный процесс генерирует исходные матрицы A и B размера n×n. Матрицы разбиваются на подблоки, размер которых определяется как block_size = n / dim, где dim — количество процессов в строке или столбце сетки. Далее с использованием функции MPI_Scatter (отвечает за распределение блоков данных между процессами) блоки передаются соответствующим процессам.

Описание алгоритма Фокса

Алгоритм разбит на несколько итераций, на каждом шаге процессы выполняют вычисления и обмениваются данными:

Процессы в строке организуют передачу текущего блока матрицы A с помощью функции MPI_Bcast, чтобы сделать данные доступными всем процессам строки.

После получения блока A каждый процесс перемножает его с локальным блоком матрицы B, добавляя результат к блоку матрицы C.

Локальные блоки матрицы B циклически перемещаются между процессами столбца с использованием функции MPI_Sendrecv_replace,

которая отправляет данные от одного процесса к другому и заменяет их в месте назначения.

В каждом процессе выполняются локальные вычисления, включающие тройной цикл для умножения блока A на блок B и накопления результата в блоке C.

На завершающем этапе алгоритма блоки матрицы C, рассчитанные в каждом процессе, передаются в главный процесс с использованием функции MPI_Gather (выполняет сбор всех блоков данных в один массив на главном процессе). Главный процесс объединяет их в итоговую матрицу и выводит результат.

Программа завершается вызовом функции MPI_Finalize, что освобождает ресурсы, связанные с MPI, и корректно завершает все процессы.

Особенности реализации

Топология:

- Создана 2D-сетка процессов.
- Функции MPI_Cart_sub обеспечивают разделение топологии на строки и столбцы.

Коммуникации:

- Широковещательная передача и обмен блоками минимизируют накладные расходы на коммуникацию.

Локальные вычисления:

- Выполняются независимо для каждого процесса, что снижает вероятность блокировок.

Код реализованной программы представлен в приложении А.

Запуск программы:

Программа была запущена с различным числом процессов с помощью команды:

```
mpirun -np P --oversubscribe ./main --matrix-size N
```

где P — количество процессов, N — размер матрицы.

Пример на двух процессах матрицы 2x2:

```
Enter matrix A (2x2):
1 2 3 4
Enter matrix B (2x2):
1 2 3 4
Result matrix C (2x2)
7 10
15 22
```

Исходя из листинга, видно, что программа работает корректно, результат умножения правильный.

График зависимости времени выполнения программы от числа процессов:

Для оценки производительности алгоритма проведены эксперименты с различными размерами матриц и числом процессов. Время выполнения замерялось для чистого времени алгоритма (без учета генерации матриц).

Обновленный код программы и скрипт представлены в приложении А.

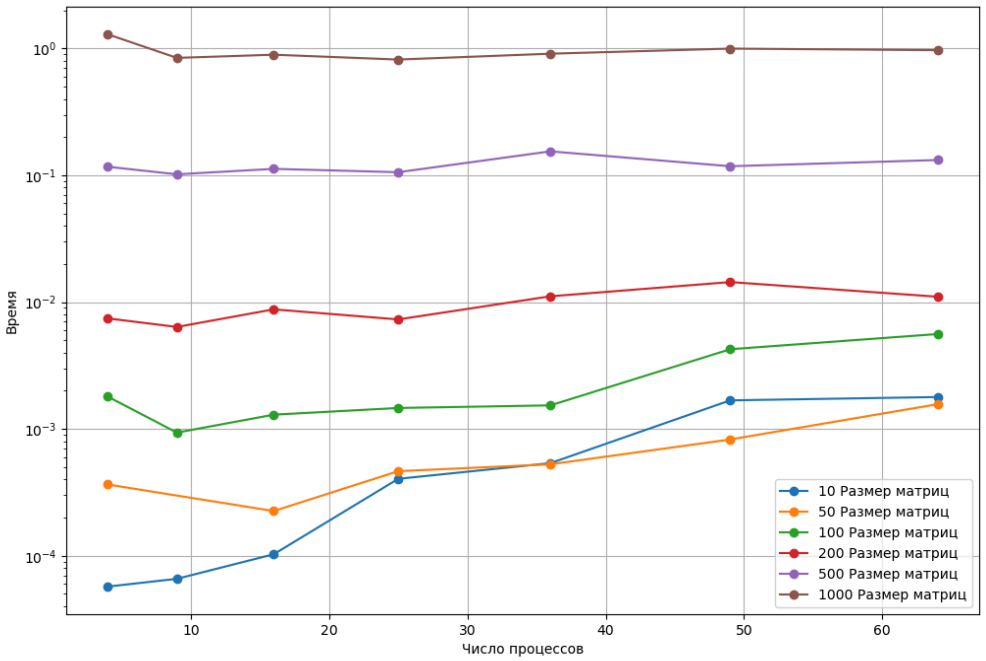


Рисунок 1 – Зависимость времени работы программы от количества процессов

На рис. 1 видно, что как правило графики расположены горизонтально и не имеют явного роста или спада, что говорит о том, что увеличение процессов незначительно влияет на уменьшение времени выполнения или не придает ощутимый прирост скорости. Такое поведение связано с накладными расходами на коммуникацию между процессорами.

Также стоит отметить расположение графика относительно друг друга: графики умножения матриц большей размерности находятся четко над графиками меньшей размерности. Такое поведение очевидно, ведь с ростом размерности матрицы увеличивается количество вычислений, что приводит к росту времени работы программы.

Для маленьких матриц (n=10) наблюдается снижение эффективности из-за высокого влияния накладных расходов, о чем сигнализирует наклон графика и возрастание времени выполнения умножения.

График ускорения работы программы:

Представленный график (Рис. 2) демонстрирует зависимость $T\_1(n)/T\_p(n)$, характеризующую замедление программы. За *T_1(n)* взято время работы последовательной программы. За $T\_p(n)$ взято время работы параллельной программы из p процессов на различных объемах данных n.

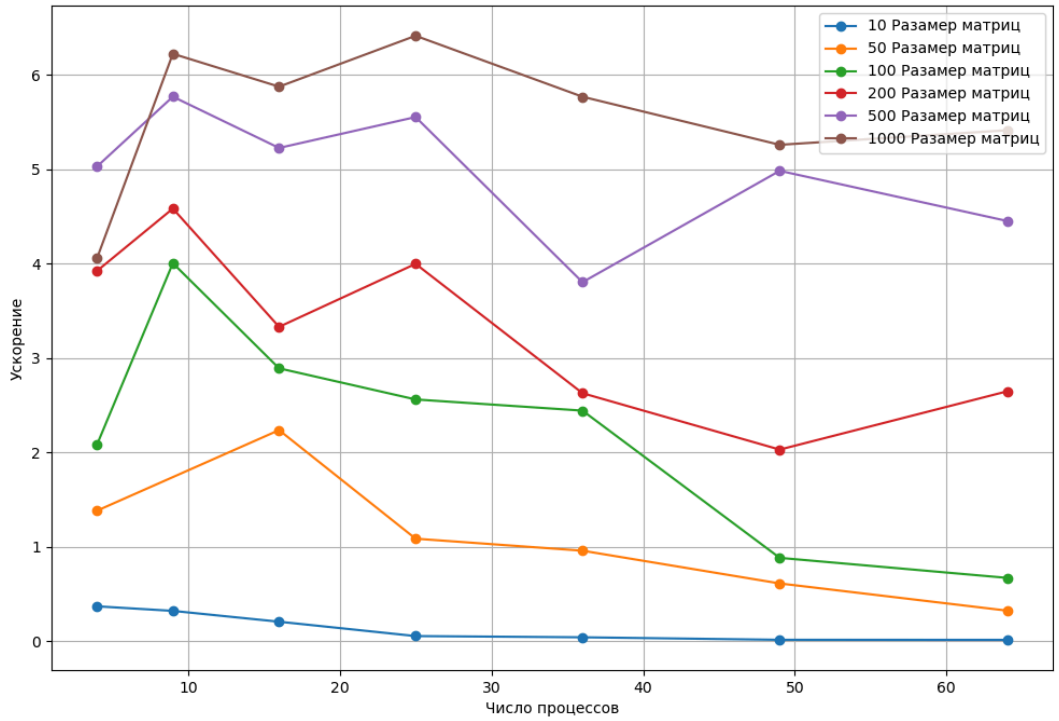


Рисунок 2 – График ускорения программы

Видно, как графики большей размерности находятся вне диапазона первого отрезка, что говорит о кратном росте ускорения. Так, ускорение параллельной программы выше для больших матриц, так как затраты на вычисления доминируют над затратами на коммуникацию.

Но также стоит отметить, что для матриц малых размеров (n=10, n=50) параллелизация не дает преимущество над последовательной программой, а наоборот проигрывает по времени. Таким образом, для малых матриц параллелизация может быть неэффективной из-за доминирования накладных расходов.

Исходные данные для построения графиков см. в Таблице 1, а также в приложении Б.

Таким образом, оптимальное количество процессов зависит от размера матрицы: слишком большое число процессов для малых матриц приводит к снижению эффективности из-за накладных расходов.

Таблица №1

N	Последовательный	4 процесса		9 процессов		16 процессов	
	время	время	ускорение	время	ускорение	время	ускорение
10	0,0000	0,0001	0,3809	0,0001	0,2893	0,0001	0,1852
50	0,0005	0,0004	1,3942	0,0002	2,9060	0,0002	2,2561
100	0,0037	0,0018	2,0678	0,0009	3,9785	0,0013	2,8714
200	0,0291	0,0074	3,9124	0,0064	4,5715	0,0088	3,3211
500	0,4870	0,1167	4,1717	0,1018	4,7856	0,1123	4,3349
1000	5,0902	1,2927	3,9376	0,8425	6,0416	0,8925	5,7032

25 процессов		36 процессов		49 процессов		64 процесса	
время	ускорение	время	ускорение	время	ускорение	время	ускорение
0,0004	0,0471	0,0005	0,0354	0,0017	0,0113	0,0018	0,0107
0,0005	1,0945	0,0005	0,9659	0,0008	0,6156	0,0016	0,3244
0,0015	2,5435	0,0015	2,4258	0,0043	0,8755	0,0056	0,6643
0,0073	3,9871	0,0111	2,6198	0,0144	2,0233	0,0110	2,6397
0,1057	4,6065	0,1544	3,1545	0,1178	4,1338	0,1319	3,6925
0,8175	6,2268	0,9090	5,5995	0,9971	5,1050	0,9687	5,2547

Теоретические оценки

В параллельном алгоритме Фокса матрицы разбиваются на блоки, и каждый процессор обрабатывает свой блок. Время выполнения на p процессорах можно оценить как:

T_par = O((n^3)/p) + O(T_comm),

где O((n^3)/p) — время выполнения вычислений, а O(T_comm) — время, затрачиваемое на коммуникацию между процессорами.

Есть следующие данные:

- Размер матрицы (n): 1000

- Количество процессоров (p): 9

Теоретическое оценка выполнения на 9 процессорах:
T_par = (1000^3) / 9 + T_comm = 111 111 111 + T_comm
Экспериментальное оценка выполнения на 9 процессорах:
T_exp = 600 041 600

Наблюдается расхождение между теоретическим и экспериментальным временем, что связано, прежде всего, с отсутствием точной формулы для вычислений из-за сложности задачи. Однако это расхождение не превышает порядок. Следовательно, полученный результат можно считать удовлетворительным.

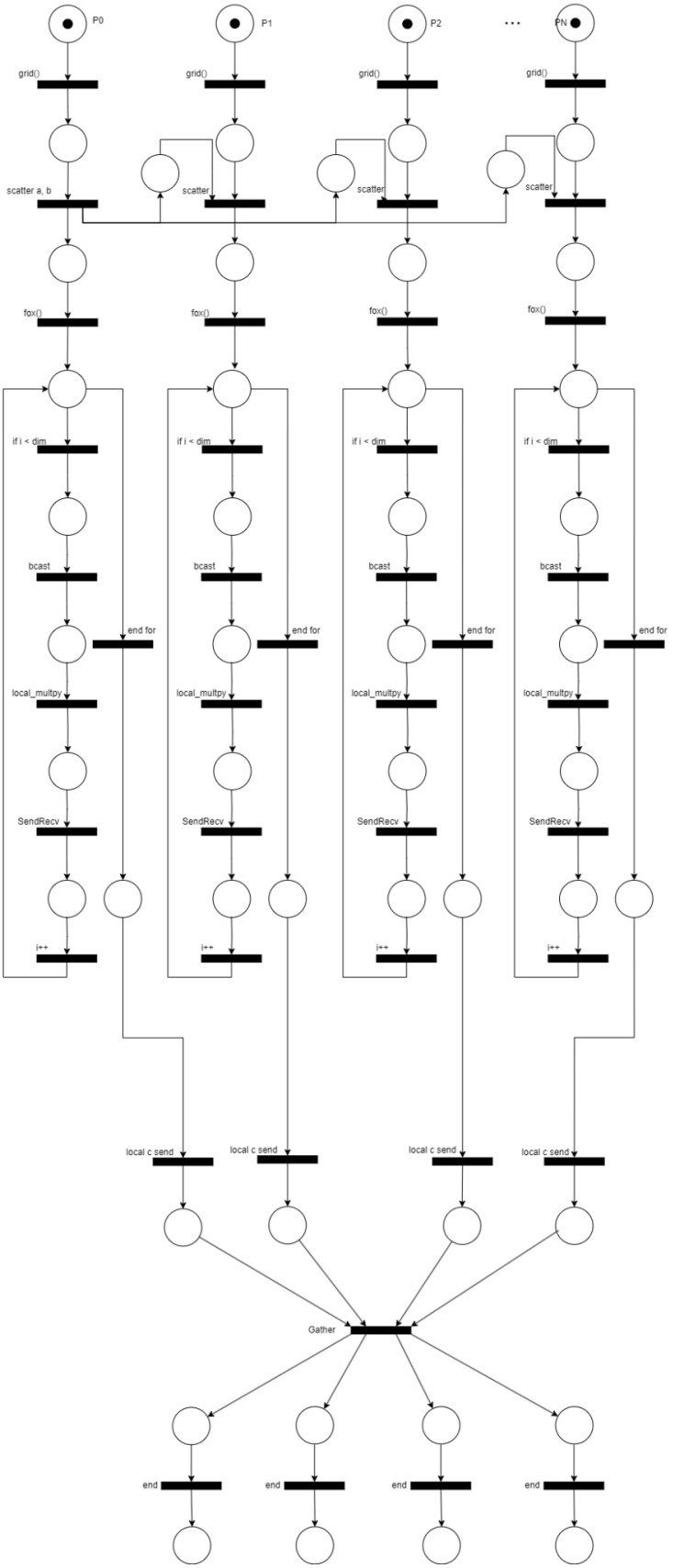
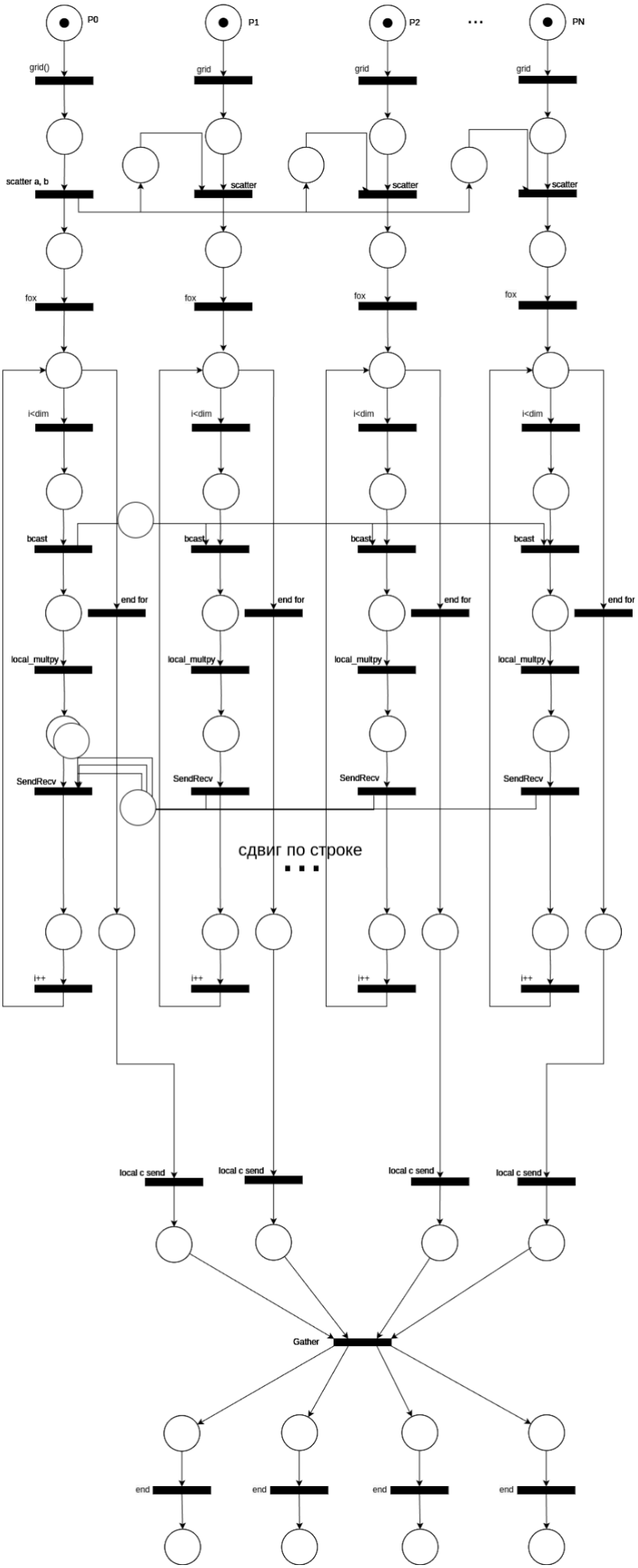


Рисунок 3 – Модель сети Петри для программы


P0
P1
P2
PN
grid()
grid
scatter a, b
scatter
fox
i<dim
bcast
end for
local_multpy
SendRecv
сдвиг по строке
i++
local c send
Gather
end

Выводы.

В ходе выполнения данной лабораторной работы был изучен параллельный алгоритм умножения матриц, а также написана соответствующая программа для реализации как последовательного, так и параллельного подходов. Реализация этих алгоритмов позволила не только освоить основы работы с MPI, но и ознакомиться с принципами блочного разбиения матриц.

Алгоритм Фокса продемонстрировал свою высокую эффективность при тестировании, особенно при работе с большими матрицами. Он эффективно распараллеливает процесс умножения, что позволяет значительно ускорить вычисления. Однако для малых матриц можно наблюдать снижение производительности, связанное с накладными расходами на коммуникацию между процессами.

При оптимизации работы программы важно учитывать размер матриц и количество используемых процессоров. Для каждого размера матрицы существует оптимальное число процессоров, превышение которого не всегда ведет к пропорциональному ускорению. Кроме того, для достижения максимальной производительности необходимо обеспечить равномерное распределение нагрузки между процессами, что позволяет избежать простаивания и достичь максимальной эффективности работы программы.

Приложение А

Название файла: main.cpp

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

void grid(int &dim, int &row, int &col, MPI_Comm &grid_comm,
MPI_Comm &row_comm, MPI_Comm &col_comm) {
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    dim = static_cast<int>(std::sqrt(size));
    int dims[2] = {dim, dim};
    int periods[2] = {0, 0};
    int reorder = 1;

     MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
&grid_comm);
    MPI_Comm_rank(grid_comm, &rank);
    int coords[2];
    MPI_Cart_coords(grid_comm, rank, 2, coords);
    row = coords[0];
    col = coords[1];

    int remain_dims_row[2] = {0, 1};
    int remain_dims_col[2] = {1, 0};

    MPI_Cart_sub(grid_comm, remain_dims_row, &row_comm);
    MPI_Cart_sub(grid_comm, remain_dims_col, &col_comm);
}

void local(const std::vector<int> &a, const std::vector<int>
&b, std::vector<int> &c, int block_size) {
    for (int i = 0; i < block_size; ++i) {
        for (int j = 0; j < block_size; ++j) {
            int temp = 0;
            for (int k = 0; k < block_size; ++k) {
                    temp += a[i * block_size + k] * b[k *
block_size + j];
            }
            c[i * block_size + j] += temp;
        }
    }
}

void fox(int n, std::vector<int> &a, std::vector<int> &b,
std::vector<int> &c, int dim, int row, int col, MPI_Comm row_comm,
MPI_Comm col_comm) {
    int block_size = n / dim;
    std::vector<int> temp_a(block_size * block_size, 0);

    int src = (row + 1) % dim;
```

```
        int dst = (row + dim - 1) % dim;

        for (int stage = 0; stage < dim; ++stage) {
            int root = (row + stage) % dim;

            if (root == col) {
                        MPI_Bcast(a.data(), block_size * block_size,
MPI_INT, root, row_comm);
                local(a, b, c, block_size);
            } else {
                   MPI_Bcast(temp_a.data(), block_size * block_size,
MPI_INT, root, row_comm);
                local(temp_a, b, c, block_size);
            }

             MPI_Sendrecv_replace(b.data(), block_size * block_size,
MPI_INT, dst, 0, src, 0, col_comm, MPI_STATUS_IGNORE);
        }
    }

    void read_matrix(std::vector<int> &matrix, int size) {
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                std::cin >> matrix[i * size + j];
            }
        }
    }

    void print_matrix(const std::vector<int> &matrix, int size) {
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                std::cout << matrix[i * size + j] << " ";
            }
            std::cout << std::endl;
        }
    }

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

        int rank, size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        if (argc < 2) {
            if (rank == 0) {
                        std::cerr << "Usage: " << argv[0] << "
--matrix-size <size>" << std::endl;
            }
            MPI_Finalize();
            return 1;
        }

        int matrix_size = std::stoi(argv[2]);

        if (matrix_size % static_cast<int>(std::sqrt(size)) != 0) {
```

```
            if (rank == 0) {
                 std::cerr << "Error: Matrix size must be divisible
by the square root of the number of processes." << std::endl;
            }
            MPI_Finalize();
            return 1;
        }

        int dim, row, col;
        MPI_Comm grid_comm, row_comm, col_comm;
        grid(dim, row, col, grid_comm, row_comm, col_comm);

        int block_size = matrix_size / dim;

        std::vector<int> local_a(block_size * block_size, 0);
        std::vector<int> local_b(block_size * block_size, 0);
        std::vector<int> local_c(block_size * block_size, 0);

        std::vector<int> a, b;
        if (rank == 0) {
            a.resize(matrix_size * matrix_size);
            b.resize(matrix_size * matrix_size);

              std::cout << "Enter matrix A (" << matrix_size << "x"
<< matrix_size << "):" << std::endl;
            read_matrix(a, matrix_size);

              std::cout << "Enter matrix B (" << matrix_size << "x"
<< matrix_size << "):" << std::endl;
            read_matrix(b, matrix_size);
        }

        MPI_Barrier(MPI_COMM_WORLD);
           MPI_Scatter(a.data(), block_size * block_size, MPI_INT,
local_a.data(), block_size * block_size, MPI_INT, 0, MPI_COMM_WORLD);
           MPI_Scatter(b.data(), block_size * block_size, MPI_INT,
local_b.data(), block_size * block_size, MPI_INT, 0, MPI_COMM_WORLD);

         fox(matrix_size, local_a, local_b, local_c, dim, row, col,
row_comm, col_comm);

        std::vector<int> result;
        if (rank == 0) {
            result.resize(matrix_size * matrix_size, 0);
        }

             MPI_Gather(local_c.data(), block_size * block_size,
MPI_INT, result.data(), block_size * block_size, MPI_INT, 0,
MPI_COMM_WORLD);

        if (rank == 0) {
             std::cout << "Result matrix C (" << matrix_size << "x"
<< matrix_size << "):" << std::endl;
            print_matrix(result, matrix_size);
        }
```

```
        MPI_Finalize();
        return 0;
    }
```

Название файла: time.cpp

```
    #include <mpi.h>
    #include <iostream>
    #include <cmath>
    #include <vector>
    #include <fstream>

    void grid(int &dim, int &row, int &col, MPI_Comm &grid_comm,
MPI_Comm &row_comm, MPI_Comm &col_comm)
    {
        int size, rank;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        dim = static_cast<int>(std::sqrt(size));
        int dims[2] = {dim, dim};
        int periods[2] = {0, 0};
        int reorder = 1;

         MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
&grid_comm);
        MPI_Comm_rank(grid_comm, &rank);
        int coords[2];
        MPI_Cart_coords(grid_comm, rank, 2, coords);
        row = coords[0];
        col = coords[1];

        int remain_dims_row[2] = {0, 1};
        int remain_dims_col[2] = {1, 0};

        MPI_Cart_sub(grid_comm, remain_dims_row, &row_comm);
        MPI_Cart_sub(grid_comm, remain_dims_col, &col_comm);
    }

    void local(const std::vector<int> &a, const std::vector<int>
&b, std::vector<int> &c, int block_size)
    {
        for (int i = 0; i < block_size; ++i)
        {
            for (int j = 0; j < block_size; ++j)
            {
                int temp = 0;
                for (int k = 0; k < block_size; ++k)
                {
                            temp += a[i * block_size + k] * b[k *
block_size + j];
                }
                c[i * block_size + j] += temp;
```

```
            }
        }
    }

    void fox(int n, std::vector<int> &a, std::vector<int> &b,
std::vector<int> &c, int dim, int row, int col, MPI_Comm row_comm,
MPI_Comm col_comm)
    {
        int block_size = n / dim;
        std::vector<int> temp_a(block_size * block_size, 0);

        int src = (row + 1) % dim;
        int dst = (row + dim - 1) % dim;

        for (int stage = 0; stage < dim; ++stage)
        {
            int root = (row + stage) % dim;

            if (root == col)
            {
                      MPI_Bcast(a.data(), block_size * block_size,
MPI_INT, root, row_comm);
                local(a, b, c, block_size);
            }
            else
            {
                   MPI_Bcast(temp_a.data(), block_size * block_size,
MPI_INT, root, row_comm);
                local(temp_a, b, c, block_size);
            }

             MPI_Sendrecv_replace(b.data(), block_size * block_size,
MPI_INT, dst, 0, src, 0, col_comm, MPI_STATUS_IGNORE);
        }
    }

    std::vector<int> generate_random_matrix(int size)
    {
        std::vector<int> matrix(size * size);
        for (int i = 0; i < size * size; ++i)
        {
            matrix[i] = rand() % 4 + 1;
        }
        return matrix;
    }

    void print_matrix(const std::vector<int> &matrix, int size)
    {
        for (int i = 0; i < size; ++i)
        {
            for (int j = 0; j < size; ++j)
            {
                std::cout << matrix[i * size + j] << " ";
            }
            std::cout << std::endl;
        }
```

```
    }

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

        int rank, size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        if (argc < 3)
        {
            if (rank == 0)
            {
                        std::cerr << "Usage: " << argv[0] << "
--matrix-size <size> --output-file <file>" << std::endl;
            }
            MPI_Finalize();
            return 1;
        }

        int matrix_size = std::stoi(argv[2]);
        std::string output_file = argv[4];

        if (matrix_size % static_cast<int>(std::sqrt(size)) != 0)
        {
            if (rank == 0)
            {
                 std::cerr << "Error: Matrix size must be divisible
by the square root of the number of processes." << std::endl;
            }
            MPI_Finalize();
            return 1;
        }

        int dim, row, col;
        MPI_Comm grid_comm, row_comm, col_comm;
        grid(dim, row, col, grid_comm, row_comm, col_comm);

        int block_size = matrix_size / dim;

        std::vector<int> local_a(block_size * block_size, 0);
        std::vector<int> local_b(block_size * block_size, 0);
        std::vector<int> local_c(block_size * block_size, 0);

        std::vector<int> a, b;
        if (rank == 0)
        {
            a = generate_random_matrix(matrix_size);
            b = generate_random_matrix(matrix_size);
        }

        MPI_Barrier(MPI_COMM_WORLD);
          MPI_Scatter(a.data(), block_size * block_size, MPI_INT,
local_a.data(), block_size * block_size, MPI_INT, 0, MPI_COMM_WORLD);
```

```
            MPI_Scatter(b.data(), block_size * block_size, MPI_INT,
local_b.data(), block_size * block_size, MPI_INT, 0, MPI_COMM_WORLD);

          double start_time = MPI_Wtime();
           fox(matrix_size, local_a, local_b, local_c, dim, row, col,
row_comm, col_comm);
          double end_time = MPI_Wtime();

          std::vector<int> result;
          if (rank == 0)
          {
              result.resize(matrix_size * matrix_size, 0);
          }

               MPI_Gather(local_c.data(), block_size * block_size,
MPI_INT,  result.data(),  block_size  *  block_size,  MPI_INT,  0,
MPI_COMM_WORLD);

          if (rank == 0)
          {
              double execution_time = end_time - start_time;
               // std::cout << "Execution time: " << execution_time <<
" seconds" << std::endl;

              // Save to file
              std::ofstream f(output_file, std::ios::app);
              f << "Matrix size: " << matrix_size << " ";
              f << "Time: " << execution_time << " ";
              f << "Processes: " << size << std::endl;
          }

          MPI_Finalize();
          return 0;
      }
```

Название файла: time.py

```
import re
import matplotlib.pyplot as plt

def parse_line(line):
    match = re.match(r'Matrix size: (\d+) Time: ([0-9.e-]+) Processes: (\d+)', line)
    if match:
        matrix_size = int(match.group(1))
        execution_time = float(match.group(2))
        num_processes = int(match.group(3))
            return {'matrix_size': matrix_size, 'execution_time': execution_time, 'num_processes':
num_processes}
    return None

data = []
with open('data/data.txt', 'r') as file:
    for line in file:
        item = parse_line(line)
```

```
            if item:
                data.append(item)

        matrix_groups = {
            '10': [8, 10, 12, 14],
            '50': [48, 49, 50, 52],
            '100': [100, 102, 104, 105],
            '200': [200, 201, 203, 198],
            '500': [498, 500, 501, 504],
            '1000': [1000, 1001, 1002]
        }

        plt.figure(figsize=(12, 8))

        for group_name, sizes in matrix_groups.items():
            num_processes = []
            execution_times = []
            for item in data:
                if item['matrix_size'] in sizes:
                    num_processes.append(item['num_processes'])
                    execution_times.append(item['execution_time'])

            sorted_data = sorted(zip(num_processes, execution_times))
            sorted_num_processes, sorted_execution_times = zip(*sorted_data)

                plt.plot(sorted_num_processes, sorted_execution_times, marker='o', label=f'{group_name}
Размер матриц')

        plt.xlabel('Число процессов')
        plt.ylabel('Время')
        # plt.title('Execution Time vs Number of Processes for Different Matrix Size Groups')
        plt.legend()
        plt.grid(True)

        plt.yscale('log')

        plt.savefig('graphics/execution_time_vs_processes_grouped.png')
```

Название файла: speedup.py

```
import re
import matplotlib.pyplot as plt

def get_key(my_dict, val):
    for key, value in my_dict.items():
        if val in value:
            return int(key)

    return False


def parse_line(line):
      match = re.match(r'Matrix size: (\d+) Time: ([0-9.e-]+)
Processes: (\d+)', line)
```

```
        if match:
            matrix_size = int(match.group(1))
            execution_time = float(match.group(2))
            num_processes = int(match.group(3))
              return {'matrix_size': matrix_size, 'execution_time':
execution_time, 'num_processes': num_processes}
        return None


    seq_data = {}
    with open('data/seq.txt', 'r') as file:
        for line in file:
            item = parse_line(line)
            if item and item['num_processes'] == 1:
                                    seq_data[item['matrix_size']]   =
item['execution_time']

    parallel_data = []
    with open('data/data.txt', 'r') as file:
        for line in file:
            item = parse_line(line)
            if item:
                parallel_data.append(item)

    matrix_groups = {
        '10': [8, 10, 12, 14],
        '50': [48, 49, 50, 52],
        '100': [100, 102, 104, 105],
        '200': [200, 201, 203, 198],
        '500': [498, 500, 501, 504],
        '1000': [1000, 1001, 1002]
    }

    plt.figure(figsize=(12, 8))

    for group_name, sizes in matrix_groups.items():
        num_processes = []
        speedup = []
      for item in parallel_data:
            if item['matrix_size'] in sizes:
                key = get_key(matrix_groups, item['matrix_size'])
                seq_time = seq_data.get(key)
                if seq_time is not None:
                                        speedup_value  =  seq_time  /
item['execution_time']
                    num_processes.append(item['num_processes'])
                    speedup.append(speedup_value)

        sorted_data = sorted(zip(num_processes, speedup))
       sorted_num_processes, sorted_speedup = zip(*sorted_data)

         plt.plot(sorted_num_processes, sorted_speedup, marker='o',
label=f'{group_name} Разамер матриц')

    plt.xlabel('Число процессов')
```

```
plt.ylabel('Ускорение')
# plt.title('Speedup vs Number of Processes for Different
Matrix Size Groups')
plt.legend()
plt.grid(True)

plt.savefig('graphics/speedup_vs_processes_grouped.png')
```

Название файла: run_par.sh

```
#!/bin/bash

output_file="data/data.txt"

> $output_file

mpic++ -o bin/time src/time.cpp -std=c++11

for size in 10 50 100 200 500 1000
do
    for n in 4 25
    do
        echo "Running with matrix size $size and $n processes"
          mpiexec -n $n --oversubscribe bin/time --matrix-size
$size --output-file $output_file
    done
done

for size in 12 51 102 201 501 1002
do
    for n in 9
    do
        echo "Running with matrix size $size and $n processes"
          mpiexec -n $n --oversubscribe bin/time --matrix-size
$size --output-file $output_file
    done
done

for size in 12 52 100 200 500 1000
do
    echo "Running with matrix size $size and 16 processes"
     mpiexec -n 16 --oversubscribe bin/time --matrix-size $size
--output-file $output_file
done

for size in 12 48 102 198 498 1002
do
    echo "Running with matrix size $size and 36 processes"
     mpiexec -n 36 --oversubscribe bin/time --matrix-size $size
--output-file $output_file
done

for size in 14 49 105 203 504 1001
do
    echo "Running with matrix size $size and 49 processes"
```

```
          mpiexec -n 49 --oversubscribe bin/time --matrix-size $size
--output-file $output_file
     done

     for size in 8 48 104 200 504 1000
     do
         echo "Running with matrix size $size and 64 processes"
          mpiexec -n 64 --oversubscribe bin/time --matrix-size $size
--output-file $output_file
     done
```

Название файла: run.sh

```
#!/bin/bash

./run_script.sh
./run_script_seq.sh

python3 src/time.py
python3 src/speedup.py
```

Приложение Б

Matrix size: 10 Time: 5.7157e-05 Processes: 4

Matrix size: 10 Time: 0.000404332 Processes: 25

Matrix size: 50 Time: 0.000364708 Processes: 4

Matrix size: 50 Time: 0.000464574 Processes: 25

Matrix size: 100 Time: 0.00179974 Processes: 4

Matrix size: 100 Time: 0.00146315 Processes: 25

Matrix size: 200 Time: 0.00744215 Processes: 4

Matrix size: 200 Time: 0.00730264 Processes: 25

Matrix size: 500 Time: 0.116733 Processes: 4

Matrix size: 500 Time: 0.105715 Processes: 25

Matrix size: 1000 Time: 1.2927 Processes: 4

Matrix size: 1000 Time: 0.817457 Processes: 25

Matrix size: 12 Time: 6.5822e-05 Processes: 9

Matrix size: 51 Time: 0.000174976 Processes: 9

Matrix size: 102 Time: 0.000935409 Processes: 9

Matrix size: 201 Time: 0.00636919 Processes: 9

Matrix size: 501 Time: 0.101757 Processes: 9

Matrix size: 1002 Time: 0.842516 Processes: 9

Matrix size: 12 Time: 0.000102821 Processes: 16

Matrix size: 52 Time: 0.000225379 Processes: 16

Matrix size: 100 Time: 0.0012961 Processes: 16

Matrix size: 200 Time: 0.00876709 Processes: 16

Matrix size: 500 Time: 0.112339 Processes: 16

Matrix size: 1000 Time: 0.892516 Processes: 16

Matrix size: 12 Time: 0.000537379 Processes: 36

Matrix size: 48 Time: 0.000526419 Processes: 36

Matrix size: 102 Time: 0.00153417 Processes: 36

Matrix size: 198 Time: 0.011114 Processes: 36

Matrix size: 498 Time: 0.154374 Processes: 36

Matrix size: 1002 Time: 0.909034 Processes: 36

Matrix size: 14 Time: 0.00167996 Processes: 49

Matrix size: 49 Time: 0.000826046 Processes: 49

Matrix size: 105 Time: 0.00425079 Processes: 49

Matrix size: 203 Time: 0.0143906 Processes: 49

Matrix size: 504 Time: 0.117804 Processes: 49

Matrix size: 1001 Time: 0.997096 Processes: 49

Matrix size: 8 Time: 0.00178457 Processes: 64

Matrix size: 48 Time: 0.00156755 Processes: 64

Matrix size: 104 Time: 0.00560197 Processes: 64

Matrix size: 200 Time: 0.0110302 Processes: 64

Matrix size: 504 Time: 0.131882 Processes: 64

Matrix size: 1000 Time: 0.968691 Processes: 64