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. 3384 _____ Рудаков А.Л.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2025

Цель работы.

Создание параллельной программы, вычисляющей произведение матриц, при помощи MPI с использованием виртуальной топологии и функций передачи информации между процессами.

Задание.

Вариант 1.

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)

Задание №1: Ленточный алгоритм 1 (горизонтальные полосы).

Выполнение работы.

Для выполнения задания написан код на языке программирования C++, при помощи средств MPI разделяющий программу на параллельную между функциям *MPI_Init(&argc, &argv)* и *MPI_Finalize(),* а также реализующий для каждого процесса определение его ранга, посредством функции *MPI_Comm_rank(MPI_COMM_WORLD, &proc_rang)* и общее количество процессов при помощи функции *MPI_Comm_size(MPI_COMM_WORLD, &num_proc)*.

Написаны функции: умножение квадратных матриц, умножение строк (прямоугольную матрицу) на квадратную матрицу, инициализация матриц

рандомными значениями, инициализация матрицы ручным вводом, вывод матрицы, вычисления строк, которые будут отправлены/получены относительно номера процесса, парсинг cli аргументов для ввода (флаг *-s* задает размер матриц, флаг *-i* символизирует о ручном вводе, если его нет, матрицы инициализируются рандомно, флаг *-p* задает то, нужно ли выводить матрицы или нет).

При помощи *MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, 0, &grid_comm)* создается новая циклическая декартова топология размерностью 1xN.

Далее для каждого процесса вычисляются строки, первой матрицы, которые должны быть ему отправлены (строки делятся поровну, если количество процессов кратно количеству строк в матрице, если нет, то первые процессы берут на себя на 1 строку больше, чем последние).

Если ранг процесса равен 0, то в нем создаются матрицы (либо случайно, либо ручным вводом), после чего происходит их умножение внутри 0 процесса (для замера времени на одном процессе).

Далее всем матрицам рассылается вторая матрица через *MPI_Bcast(..)*.

Далее каждому процессу через *MPI_Send(..)* отправляются их строки первой матрицы.

Внутри процессов с рангами не равными 0 происходит получение второй матрицы через *MPI_Bcast(..)*, после чего получение через *MPI_Recv(..)* своей части первой матрицы, после чего каждый процесс умножает их и отправляет обратно 0 процессу через MPI_Send(..).

Нулевой процесс так же выполняет умножение своей части матрицы, после чего через *MPI_Recv(..)* получает от каждого процесса свою часть матрицы, после чего сравнивает матрицы, полученные путем умножения на одном процессе и параллельно на нескольких.

Схема Петри представлена на рис.1.

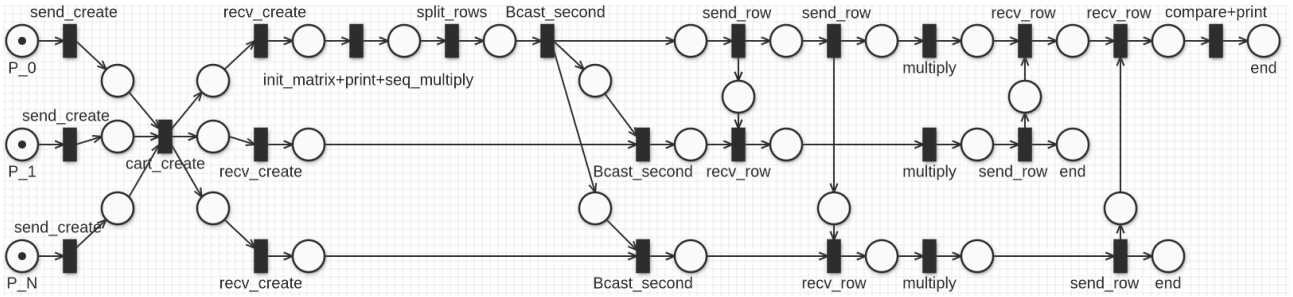

Рисунок 1 — Схема Петри.

Код, написанный на языке программирования C++ см. Приложение А.

Результаты работы программы см. Приложение В.

Зависимость времени работы программы от количества процессов можно увидеть в табл.1 и на рис.2.

Табл.1 см. Приложение Б.

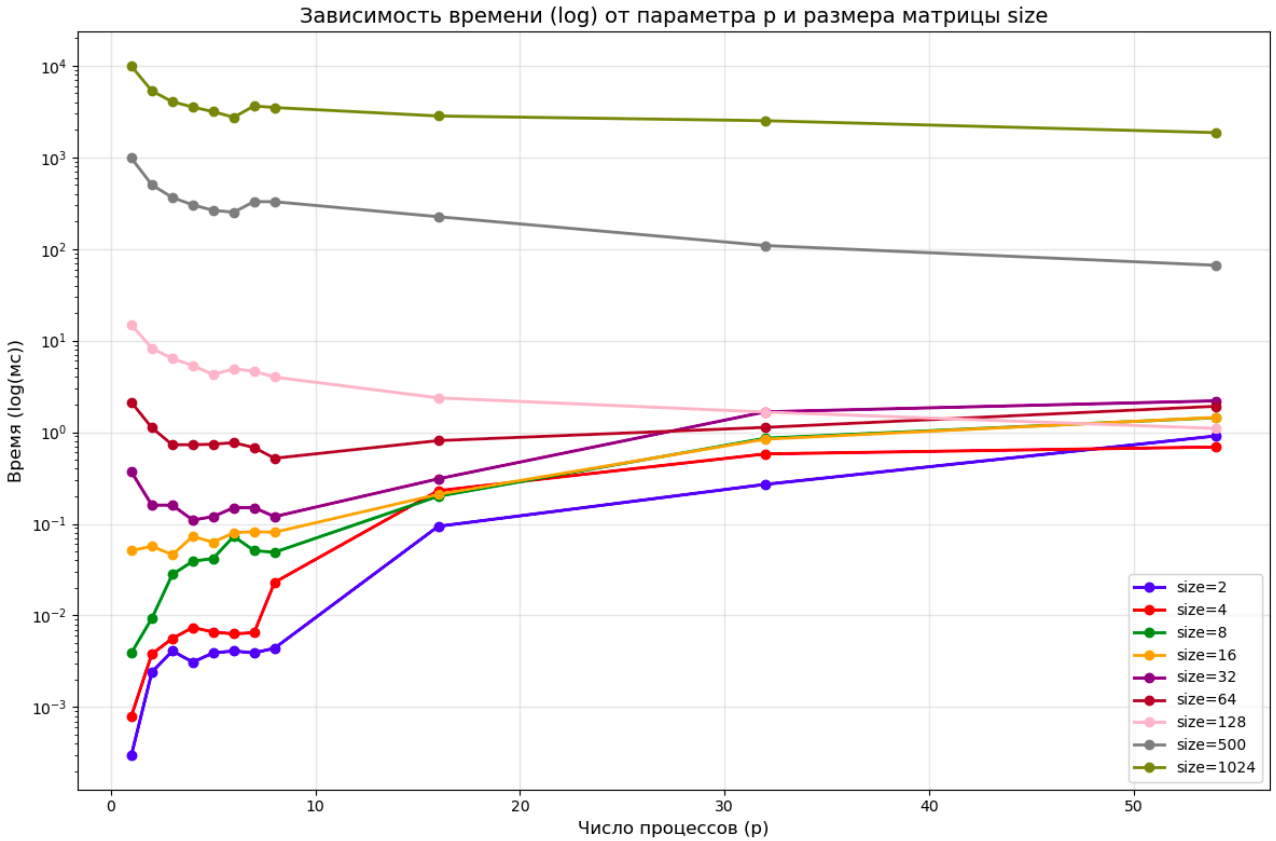

Рисунок 2 - График зависимости времени (log) от числа процессов и размеров матрицы

На рис. 3 изображен график зависимости ускорения (log) от числа процессов.

На рис. 4 изображен график зависимости ускорения от числа процессов.

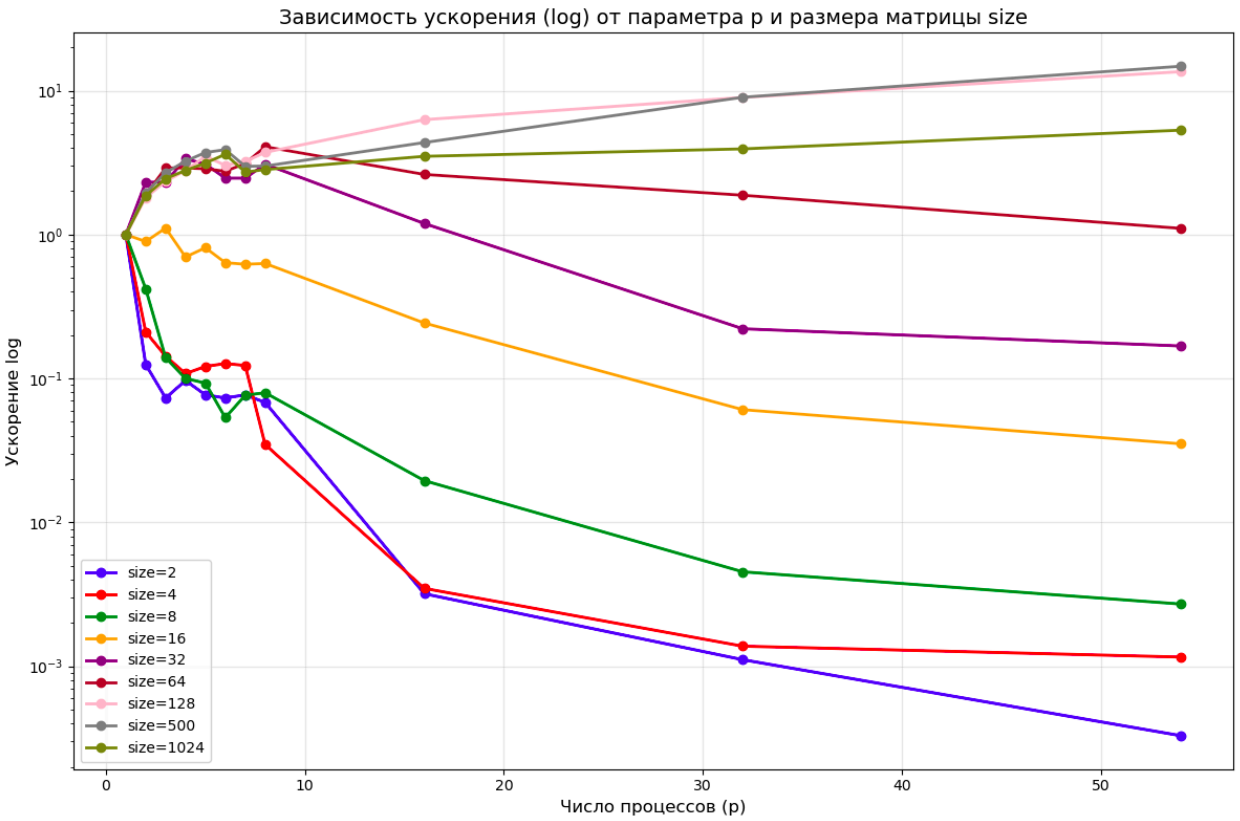

Рисунок 3 - График зависимости ускорения (log) от числа процессов и размеров матрицы

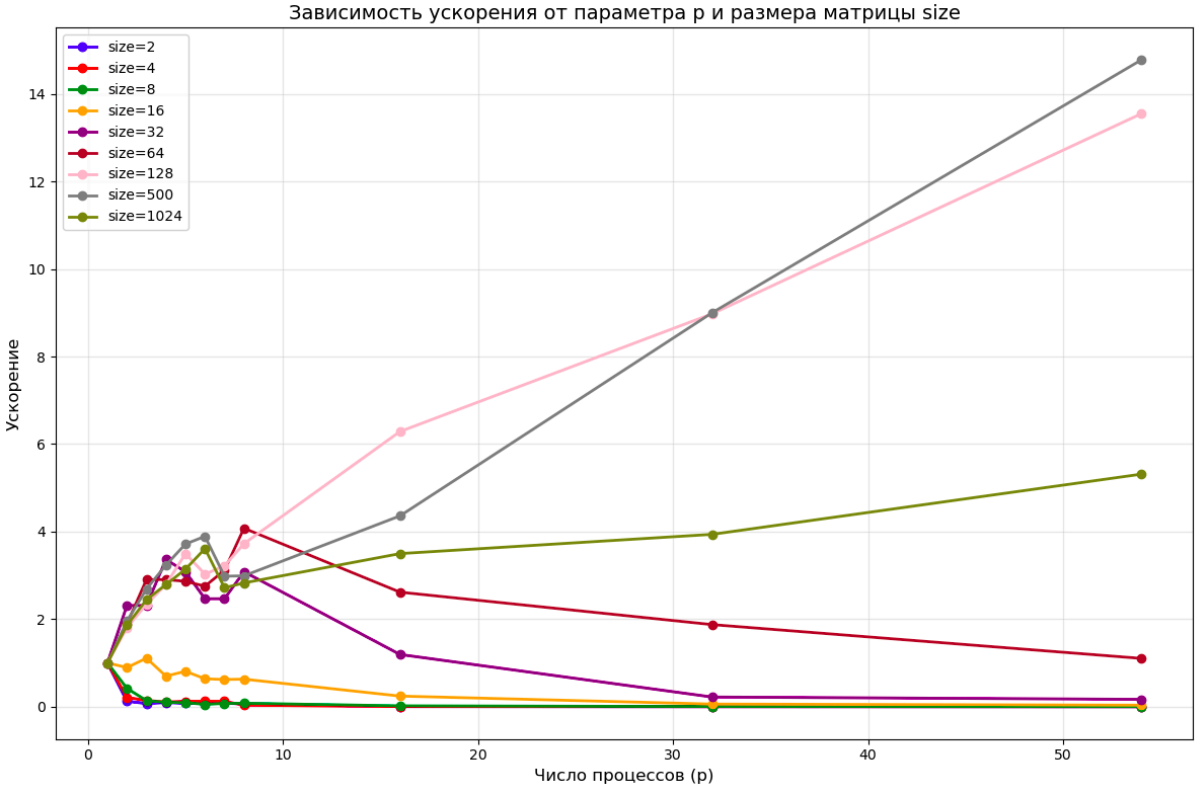

Рисунок 4 - График зависимости ускорения от числа процессов и размеров матрицы

Из полученных результатов можно заметить, что начиная с размеров матриц 32x32 время выполнения умножений на количестве процессов, меньших, чем размер матрицы становится меньше, чем на одном процессе и ускорение соответственно возрастает. Когда число процессов больше либо равно размеру матрицы, то время работы больше, чем время работы на одном процессе, что объясняется тем, что затраты времени на передачу данных между процессами больше, чем затраты времени на последовательное вычисление, так как в таком случае им передается либо 1, либо ни одной строки.

На матрицах размеров, меньших чем 32 ускорения не происходит, так как точно так же затраты времени на передачу данных между сообщениями перевешивают затраты времени на линейное вычисление.

Наибольшее значение ускорения получилось для матриц размером 128x128 и 500x500, запущенных на 64 процессах. Это объясняется тем, что при меньших размерах вычисления слишком быстрые, время коммуникации становится сравнимым или превышает время вычислений, при бОльших размерах вычисления доминируют, но коммуникационные расходы времени становятся слишком большими.

Выводы.

Создана параллельная программа, вычисляющая произведение матриц, при помощи MPI с использованием виртуальной декартовой топологии 1xN и функций передачи информации между процессами. Наибольшее ускорение получено при умножении матриц, размерами 128x128 и 500x500, запущенных на 64 процессах.

ПРИЛОЖЕНИЕ
ИСХОДНЫЙ КОД

Файл mpi_lb6.cpp:

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


void matrix_multiply(const std::vector<std::vector<int>>& first,
                const std::vector<std::vector<int>>& second,
                std::vector<std::vector<int>>& result, int m_size) {
  for (int i = 0; i < m_size; i++) {
     for (int j = 0; j < m_size; j++) {
        result[i][j] = 0;
        for (int k = 0; k < m_size; k++) {
           result[i][j] += first[i][k] * second[k][j];
        }
     }
  }
}

void row_matrix_multiply(const std::vector<std::vector<int>>& first_local,
                  const std::vector<std::vector<int>>& second,
                  std::vector<std::vector<int>>& result_local, int m_size, int local_rows) {
  for (int i = 0; i < local_rows; i++) {
        for (int j = 0; j < m_size; j++) {
           result_local[i][j] = 0;
           for (int k = 0; k < m_size; k++) {
              result_local[i][j] += first_local[i][k] * second[k][j];
           }
        }
     }
}


void initialize_matrix(std::vector<std::vector<int>>& matrix, int m_size) {
  for (int i = 0; i < m_size; i++) {
     for (int j = 0; j < m_size; j++) {
        matrix[i][j] = rand() % 10;
     }
```

```
    }
}

void input_matrix(std::vector<std::vector<int>>& matrix, int m_size, int matrix_num) {
    std::cout << "Enter matrix " << matrix_num << " (" << m_size << "x" << m_size << ") row by row:" <<
std::endl;
    for (int i = 0; i < m_size; i++) {
        std::cout << "Row " << i + 1 << ": ";
        for (int j = 0; j < m_size; j++) {
            std::cin >> matrix[i][j];
        }
    }
}

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

bool check_equal(const std::vector<std::vector<int>>& first,
                 const std::vector<std::vector<int>>& second,
                 int m_size){
    for (int i = 0; i < m_size; i++) {
        for (int j = 0; j < m_size; j++) {
            if (first[i][j] != second[i][j]) {
                return false;
            }
        }
    }
    return true;
}

void compute_rows(int& start_row, int& end_row, int& local_rows,
                  int current_number, int rows_per_process, int remainder) {
    if (current_number < remainder) {
```

```
        local_rows = rows_per_process + 1;
        start_row = current_number * local_rows;
    } else {
        local_rows = rows_per_process;
        start_row = remainder * (rows_per_process + 1) + (current_number - remainder) * rows_per_process;
    }
    end_row = start_row + local_rows;
}


void parse_arguments(int argc, char* argv[], int& m_size, bool& is_print, bool& is_input) {
    m_size = 6;
    is_print = false;
    is_input = false;

    for (int i = 1; i < argc; i++) {
        std::string arg = argv[i];
        if (arg == "-s" && i + 1 < argc) {
            m_size = std::atoi(argv[i + 1]);
            i++;
        } else if (arg == "-p") {
            is_print = true;
        } else if (arg == "-i") {
            is_input = true;
        }
    }
}


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);

    int m_size = 6;
    bool is_print, is_input;
    parse_arguments(argc, argv, m_size, is_print, is_input);

    int dims[1] = {size};
    int periods[1] = {0};
```

```
MPI_Comm grid_comm;
MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, 0, &grid_comm);

int grid_rank;
MPI_Comm_rank(grid_comm, &grid_rank);

int rows_per_process = m_size / size;
int remainder = m_size % size;

int start_row, end_row, local_rows;
compute_rows(start_row, end_row, local_rows, grid_rank, rows_per_process, remainder);

if (rank == 0) {
    std::cout << "Matrix size: " << m_size << "x" << m_size << std::endl;
    std::cout << "Number of processes: " << size << std::endl;
}

if (rank == 0) {
    std::vector<std::vector<int>> first(m_size, std::vector<int>(m_size));
    std::vector<std::vector<int>> second(m_size, std::vector<int>(m_size));
    std::vector<std::vector<int>> result_seq(m_size, std::vector<int>(m_size));

    if (is_input) {
        input_matrix(first, m_size, 1);
        input_matrix(second, m_size, 2);
    } else {
        initialize_matrix(first, m_size);
        initialize_matrix(second, m_size);
    }

    if (is_print) {
        std::cout << "Matrix A:" << std::endl;
        print_matrix(first, m_size);
        std::cout << "Matrix B:" << std::endl;
        print_matrix(second, m_size);
    }

    double start_time = MPI_Wtime();
    matrix_multiply(first, second, result_seq, m_size);
    double end_time = MPI_Wtime();

    std::cout << "Sequential time: " << (end_time - start_time) * 1000 << " ms" << std::endl;
```

```
// std::ofstream out1("time_1.txt", std::ios::app);
// out1 << (end_time - start_time) * 1000<< "\n";
// out1.close();

for (int i = 0; i < m_size; i++) {
    MPI_Bcast(second[i].data(), m_size, MPI_INT, 0, grid_comm);
}

for (int dest = 1; dest < size; dest++) {
    int dest_start, dest_end, dest_rows;
    compute_rows(dest_start, dest_end, dest_rows, dest, rows_per_process, remainder);

    for (int i = dest_start; i < dest_end; i++) {
        MPI_Send(first[i].data(), m_size, MPI_INT, dest, 0, grid_comm);
    }
}

std::vector<std::vector<int>> first_local(local_rows, std::vector<int>(m_size));
std::vector<std::vector<int>> result_local(local_rows, std::vector<int>(m_size));

for (int i = 0; i < local_rows; i++) {
    first_local[i] = first[start_row + i];
}

start_time = MPI_Wtime();
row_matrix_multiply(first_local, second, result_local, m_size, local_rows);

std::vector<std::vector<int>> result_par(m_size, std::vector<int>(m_size));

for (int i = 0; i < local_rows; i++) {
    result_par[start_row + i] = result_local[i];
}

for (int src = 1; src < size; src++) {
    int src_start, src_end, src_rows;
    compute_rows(src_start, src_end, src_rows, src, rows_per_process, remainder);

    for (int i = src_start; i < src_end; i++) {
        MPI_Recv(result_par[i].data(), m_size, MPI_INT, src, 0, grid_comm, MPI_STATUS_IGNORE);
    }
}
```

```
        end_time = MPI_Wtime();

        std::cout << "Parallel time: " << (end_time - start_time) * 1000 << " ms" << std::endl;

        // std::ofstream out2("time_n.txt", std::ios::app);
        // out2 << (end_time - start_time) * 1000 << "\n";
        // out2.close();

        if (is_print) {
            std::cout << "Matrix seq:" << std::endl;
            print_matrix(result_seq, m_size);
            std::cout << "Matrix par:" << std::endl;
            print_matrix(result_par, m_size);
        }

        bool correct = check_equal(result_seq, result_par, m_size);
        std::cout << "Result is " << (correct ? "CORRECT" : "INCORRECT") << std::endl;

    } else {
        std::vector<std::vector<int>> second(m_size, std::vector<int>(m_size));
        for (int i = 0; i < m_size; i++) {
            MPI_Bcast(second[i].data(), m_size, MPI_INT, 0, grid_comm);
        }

        std::vector<std::vector<int>> first_local(local_rows, std::vector<int>(m_size));
        std::vector<std::vector<int>> result_local(local_rows, std::vector<int>(m_size));

        for (int i = 0; i < local_rows; i++) {
            MPI_Recv(first_local[i].data(), m_size, MPI_INT, 0, 0, grid_comm, MPI_STATUS_IGNORE);
        }

        row_matrix_multiply(first_local, second, result_local, m_size, local_rows);

        for (int i = 0; i < local_rows; i++) {
            MPI_Send(result_local[i].data(), m_size, MPI_INT, 0, 0, grid_comm);
        }
    }

    MPI_Comm_free(&grid_comm);
    MPI_Finalize();
    return 0;
}
```

ПРИЛОЖЕНИЕ Б
ТАБЛИЦЫ

Таблица 1 - Зависимость времени работы программы от числа процессов и размеров матриц. Время указано в мс.

Размер матрицы	Количество процессов										
	1	2	3	4	5	6	7	8	16	32	64
2	0.0003	0.0024	0.0041	0.0031	0.0039	0.0041	0.0039	0.0044	0.094	0.27	0.91
4	0.0008	0.0038	0.0056	0.0074	0.0066	0.0063	0.0065	0.023	0.23	0.58	0.69
8	0.0039	0.0093	0.028	0.039	0.042	0.073	0.051	0.049	0.20	0.86	1.44
16	0.051	0.057	0.046	0.073	0.063	0.080	0.082	0.081	0.21	0.84	1.45
32	0.37	0.16	0.16	0.11	0.12	0.15	0.15	0.12	0.31	1.67	2.20
64	2.12	1.12	0.73	0.73	0.74	0.77	0.68	0.52	0.81	1.13	1.92
128	14.9	8.25	6.36	5.34	4.26	4.92	4.63	4.00	2.37	1.66	1.10
500	981	501	365	303	264	252	329	328	225	109	66.4
1024	9892	5313	4046	3544	3150	2738	3641	3497	2828	2512	1862

ПРИЛОЖЕНИЕ В
ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ

При нескольких запусках программы получены подобные результаты:

sun@aleksundr:~/Документы/PA/lb6$ mpiexec --oversubscribe -n 8 mpi_lb6 -s 32
Matrix size: 32x32
Number of processes: 8
Sequential time: 0.43572 ms
Parallel time: 0.129135 ms
Result is CORRECT

sun@aleksundr:~/Документы/PA/lb6$ mpiexec --oversubscribe -n 2 mpi_lb6 -s 2 -p
Matrix size: 2x2
Number of processes: 2
Matrix A:
3 6
7 5
Matrix B:
3 5
6 2
Sequential time: 0.000333 ms
Parallel time: 0.002033 ms
Matrix seq:
45 27
51 45
Matrix par:
45 27
51 45
Result is CORRECT

sun@aleksundr:~/Документы/PA/lb6$ mpiexec --oversubscribe -n 2 mpi_lb6 -s 2 -i
Matrix size: 2x2
Number of processes: 2
Enter matrix 1 (2x2) row by row:
Row 1: 1 1
Row 2: 1 1
Enter matrix 2 (2x2) row by row:
Row 1: 1 0
Row 2: 0 1
Sequential time: 0.000816 ms
Parallel time: 0.015809 ms
Result is CORRECT