МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

«Умножение матриц в MPI при помощи 2D решетки»

Студента 2 курса, 21211 группы

Петрова Сергея Евгеньевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Антон Юрьевич Кудинов

СОДЕРЖАНИЕ

ЦЕЛЬ

Ознакомиться с написанием параллельных MPI-программ с использованием построения 2D решетки;

ЗАДАНИЕ

1. *Реализовать параллельный алгоритм умножения матрицы при помощи 2D решетки;*
2. *Исследовать производительность параллельной программы в зависимости от размера матрицы и размера решетки.*
3. *Выполнить профилирование программы с помощью MPE при использовании 16-и ядер.*

ВАРИАНТ ЗАДАНИЯ

Вычисляется произведение $C = A \times B$, *где A – матрица размера* $n_1 \times n_2$, *B – матрица* $n_2 \times n_3$ *и C – матрица результатов* $n_1 \times n_3$.

Исходные матрицы первоначально доступны на нулевом процессе, и матрица результатов возвращена в нулевой процесс.

Параллельное выполнение алгоритма осуществляется на двумерной (2D) решетке компьютеров размером $p_1 \times p_2$. *Матрица A разрезана на* p_1 *горизонтальных полос, матрица B разрезана на* p_2 *вертикальных полос, и матрица результата C разрезана на* $p_1 \times p_2$ *подматрицы.*

Каждый компьютер (i, j) *вычисляет произведение i-й горизонтальной полосы матрицы A и j-й вертикальной полосы матрицы B, произведение получено в подматрице* (i, j) *матрицы C.*

Последовательные стадии вычисления

1. Матрица A распределяется по горизонтальным полосам вдоль координаты $(x, 0)$.

2. Матрица B распределяется по вертикальным полосам вдоль координаты $(0, y)$.

3. Полосы A распространяются в измерении y.

4. Полосы B распространяются в измерении x.

5. Каждый процесс вычисляет одну подматрицу произведения.

6. Матрица C собирается из (x, y) *плоскости.*

Исходные данные

Матрица A содержит строки, состоящие из чисел, соответствующих номеру строки.

Матрица B содержит столбцы, состоящие из чисел, соответствующих номеру столбца.

Матрица C содержит на (i, j) *позиции число* $i \times j \times n_2$

ОПИСАНИЕ РАБОТЫ

Описание выполненной работы

1. *Написал параллельную программу, реализующую умножение матрица, использую построение 2D решетки;*
2. *Запустил программу при использовании от 1 до 16 ядер со следующими размерами матриц:* $n_1 = 2000,\ n_2 = 1500,\ n_3 = 2500$*;*
3. *Запустил программу при использовании 16 ядер при различных размерах сетки со следующими размерами матриц:* $n_1 = 4000,\ n_2 = 3000,\ n_3 = 5000$*;*
4. *Выполнил профилирование программы с помощью MPE при использовании 16 ядер со следующими размерами матриц:* $n_1 = 4000,\ n_2 = 3000,\ n_3 = 5000$*;*

Команды для компиляции и запуска

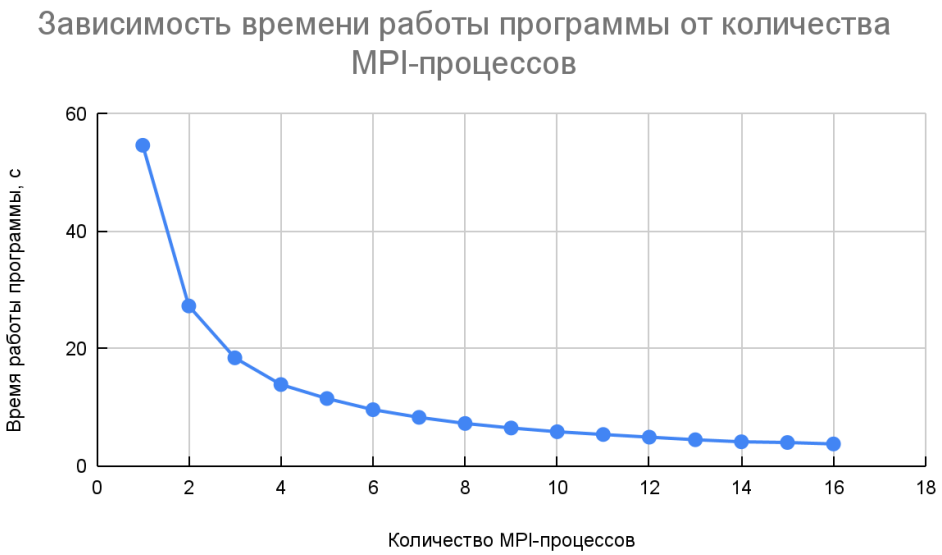
Команда для компиляции MPI программы

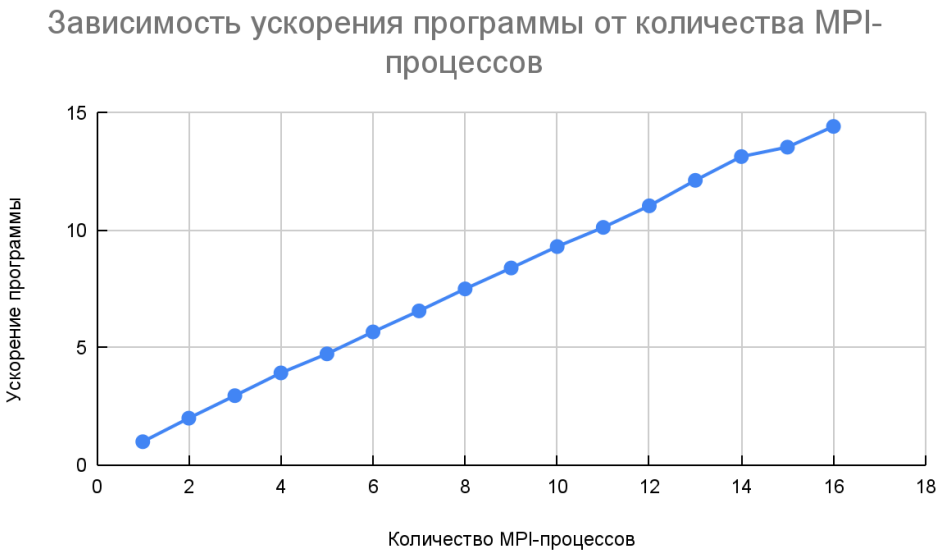
```
hpcuser265@clu:~> mpicc
--std=c99
-o mpi_multiply
mpi_multiply.c
-lm
```

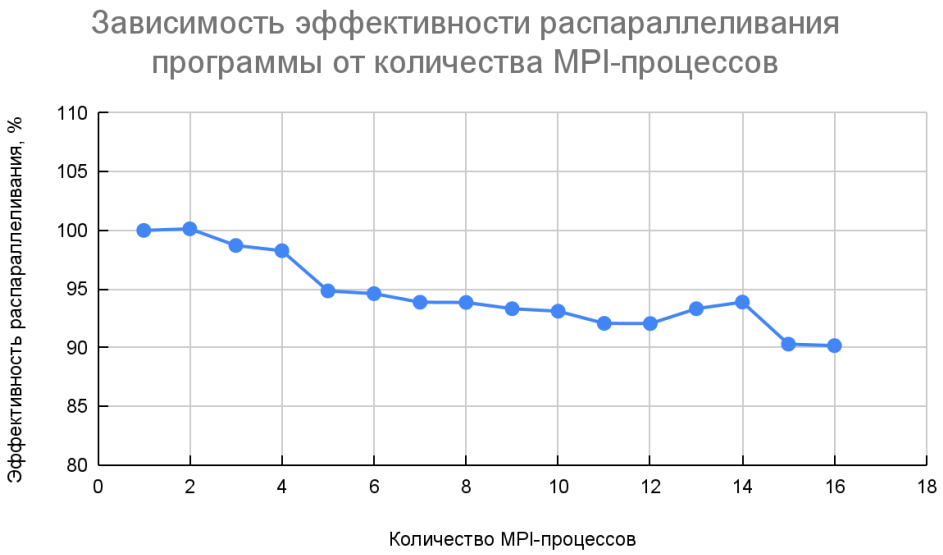
Команда для компиляции MPI программы с использованием MPE

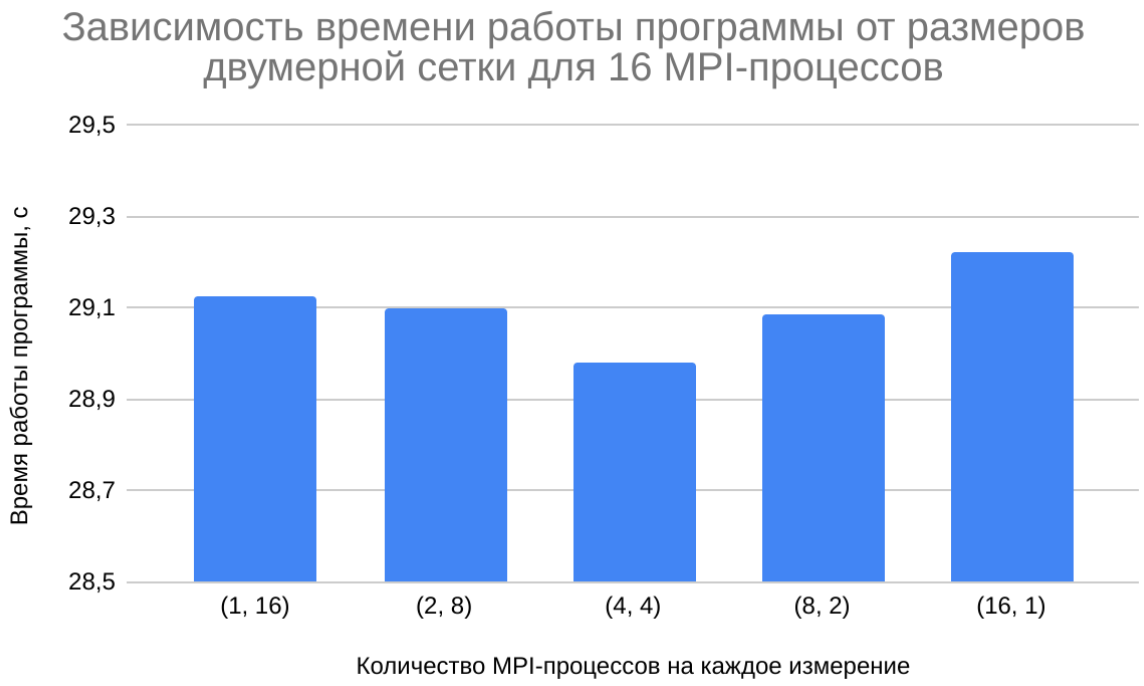
```
hpcuser265@clu:~> mpecc
-mpilog
--std=c99
-o mpe_multiply
mpi_multiply.c
-lm
```

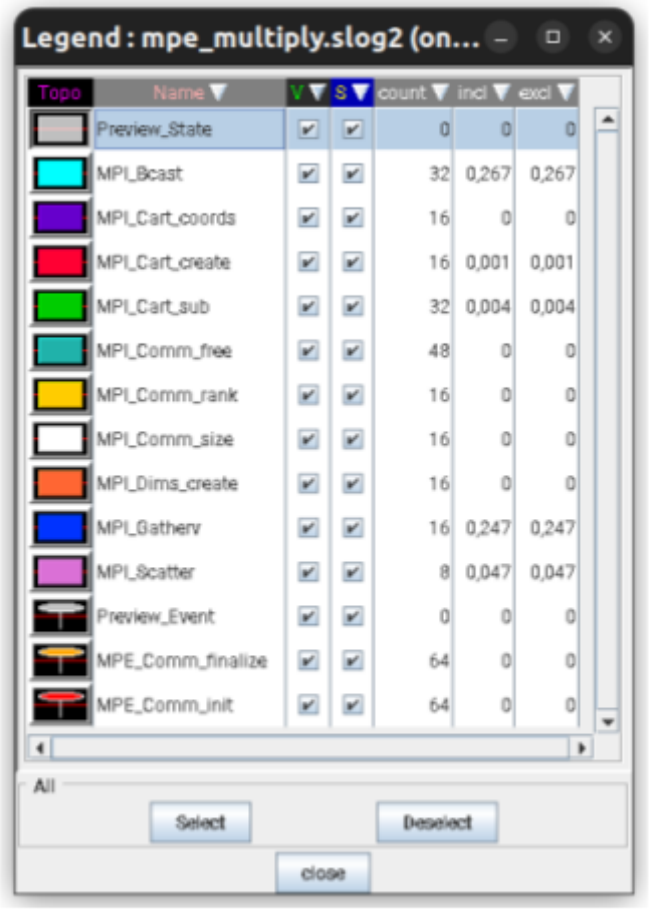
Результаты измерения



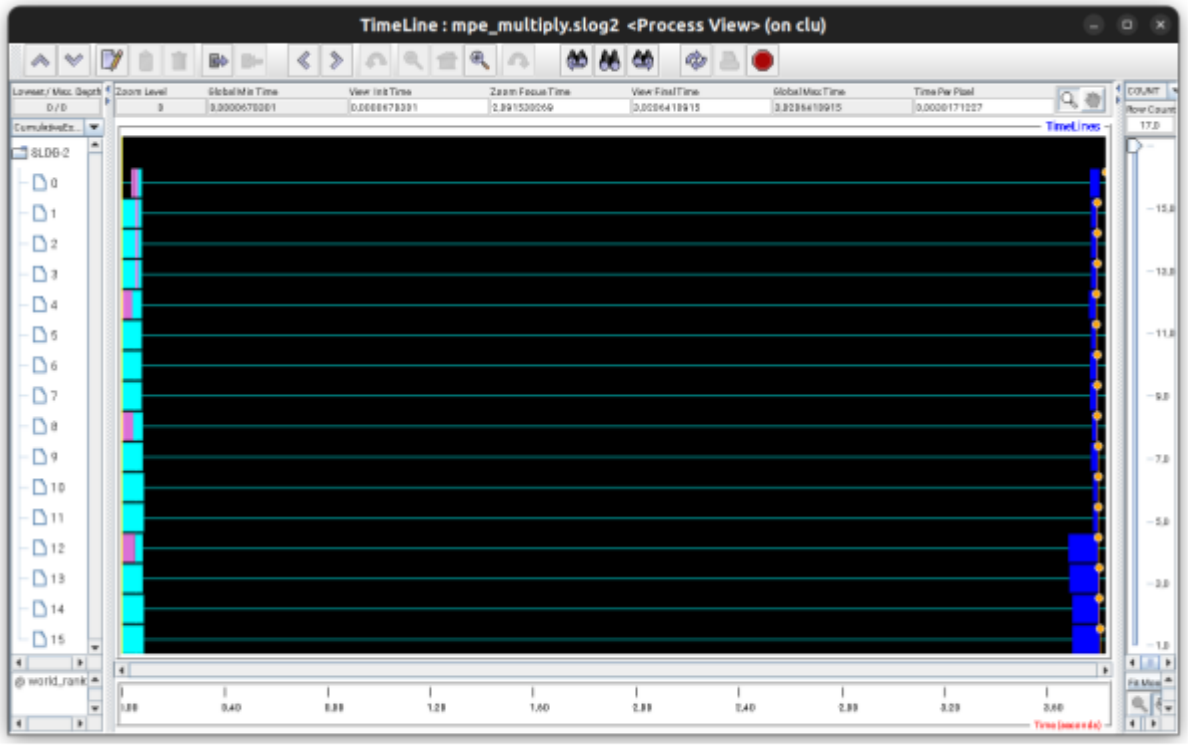



Зависимость эффективности распараллеливания программы от количества MPI-процессов
Эффективность распараллеливания, %
110
105
100
95
90
85
80
0
2
4
6
8
10
12
14
16
18
Количество MPI-процессов

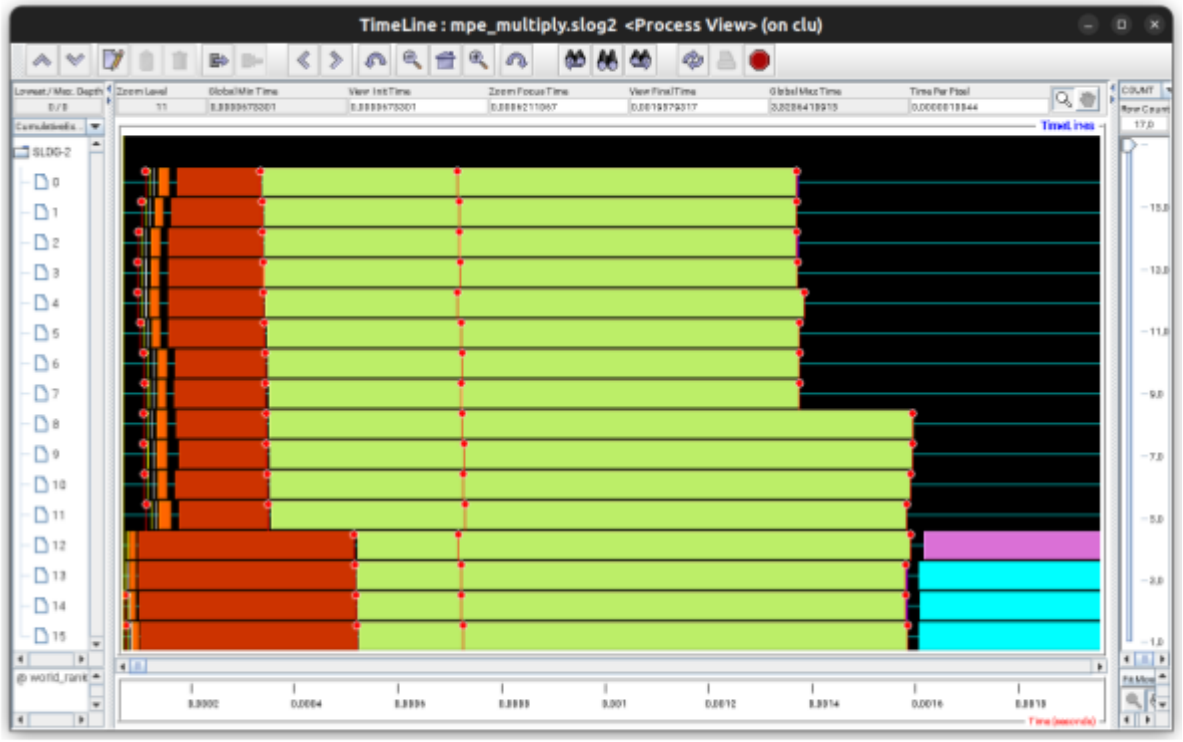

Зависимость времени работы программы от размеров двумерной сетки для 16 MPI-процессов
Время работы программы, с
29,5
29,3
29,1
28,9
28,7
28,5
(1, 16)
(2, 8)
(4, 4)
(8, 2)
(16, 1)
Количество MPI-процессов на каждое измерение



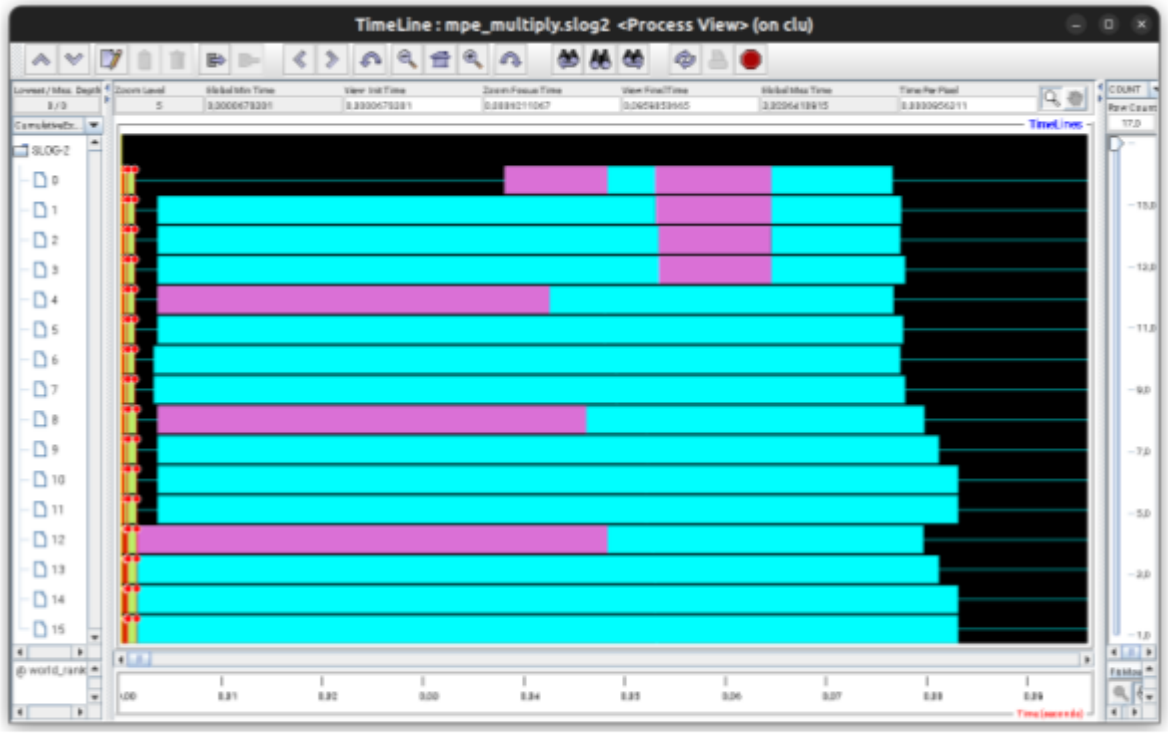
Легенда временной шкалы



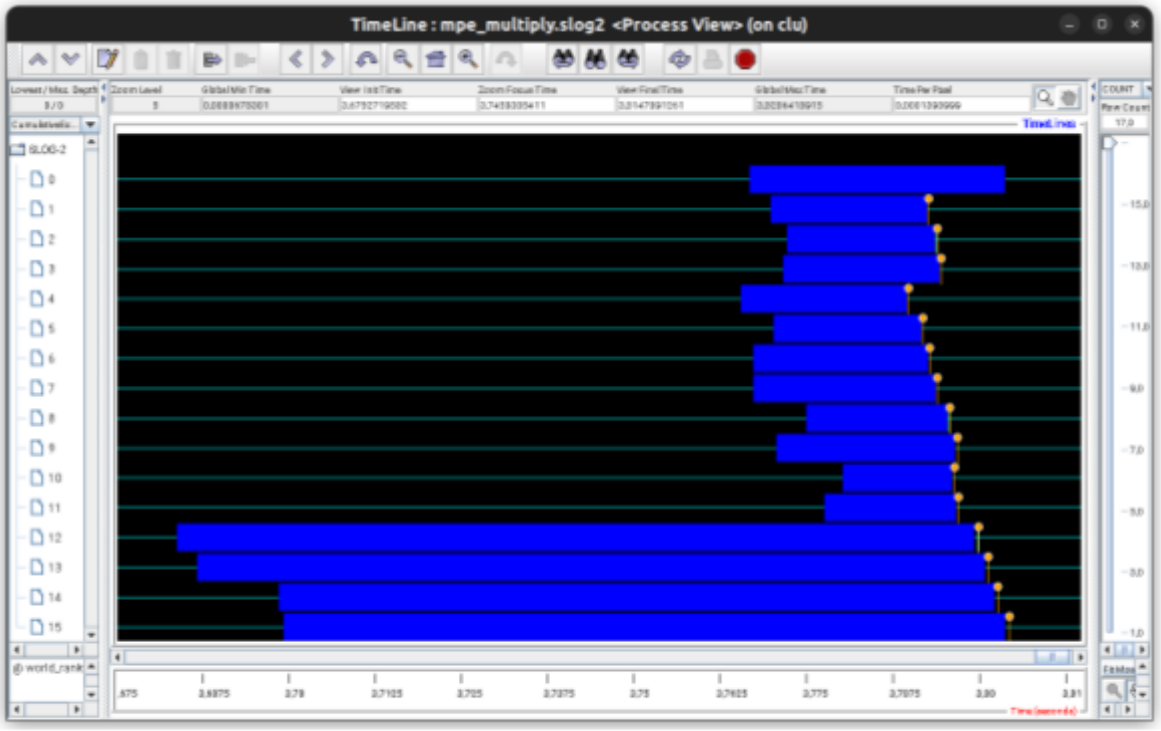
Полная временная шкала



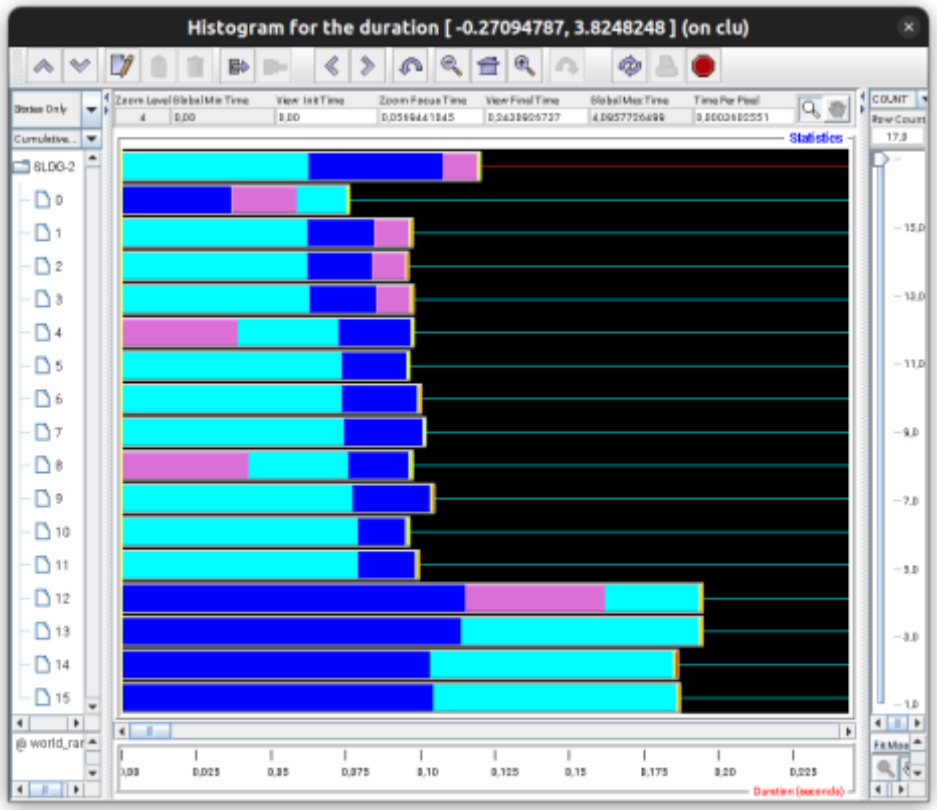
Участок временной шкалы от 0 до 0,0015 секунд



Участок временной шкалы с 0 до 0,09 секунд



Участок временной шкалы от 3,66 до 3,83 секунд



Гистограмма продолжительности MPI команд

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы:

- *Ознакомился с написанием параллельных MPI-программ с использованием построения 2D решетки;*
- *Реализовал параллельный алгоритм умножения матрицы при помощи 2D решетки;*
- *Исследовал производительность параллельной программы в зависимости от количества узлов при автоматическом выборе размеров решетки;*
- *Исследовал производительность параллельной программы в зависимости от размеров решетки при фиксированном количестве узлов;*
- *Выполнил профилирование программы с помощью MPE при использовании 16 узлов;*

Анализируя результаты исследований, можно сделать следующие выводы:

- *Эффективность распараллеливания не опускается ниже 90%, поэтому программа хорошо масштабируема;*
- *Время работы программы для 16 узлов минимально при размере решетки (4, 4). Это связано с тем, что при таком размере решетки количество вызовов функции MPI_Scatter минимально;*
- *Большую часть времени в работе программы занимает умножение блоков матриц;*
- *Из-за того, что нулевой процесс принадлежит первой строке и первому столбцу, в нём дважды вызывается MPI_Scatter, остальные процессы тем временем простаивают, вызвав функции MPI_Scatter или MPI_Bcast;*

ПРИЛОЖЕНИЕ (ЛИСТИНГ ПРОГРАММЫ)

mpi_multiply.c

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

#define DIMS_COUNT 2
#define X 0
#define Y 1

void init_dims(int dims[DIMS_COUNT], int proc_count, int argc,
               char **argv);
void init_communicators(const int dims[DIMS_COUNT],
                        MPI_Comm *comm_grid, MPI_Comm *comm_rows,
                        MPI_Comm *comm_columns);
void generate_matrix(double *matrix, int column, int leading_row,
                     int leading_column, bool onRows);
void split_A(const double *A, double *A_block, int A_block_size,
             int n_2, int coords_y, MPI_Comm comm_rows,
             MPI_Comm comm_columns);
void split_B(const double *B, double *B_block, int B_block_size,
             int n_2, int aligned_n3, int coords_x,
             MPI_Comm comm_rows, MPI_Comm comm_columns);
void multiply(const double *A_block, const double *B_block,
              double *C_block, int A_block_size, int B_block_size,
              int n_2);
void gather_C(const double *C_block, double *C, int A_block_size,
              int B_block_size, int aligned_n1, int aligned_n3,
              int proc_count, MPI_Comm comm_grid);
bool check_C(const double *C, int column, int leading_row,
             int leading_column, int n_2);

int main(int argc, char **argv)
{
    int n_1 = 2000;
    int n_2 = 1500;
    int n_3 = 2500;
    int proc_rank;
    int proc_count;
    int aligned_n1;
    int aligned_n3;
    int A_block_size;
    int B_block_size;
    int dims[DIMS_COUNT] = {};
    int coords[DIMS_COUNT] = {};
    double start_time;
    double finish_time;
    double *A = NULL;
    double *B = NULL;
    double *C = NULL;
    double *A_block = NULL;
```

```
double *B_block = NULL;
double *C_block = NULL;
MPI_Comm comm_grid;
MPI_Comm comm_rows;
MPI_Comm comm_columns;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
MPI_Comm_size(MPI_COMM_WORLD, &proc_count);

init_dims(dims, proc_count, argc, argv);

init_communicators(dims, &comm_grid, &comm_rows, &comm_columns);

MPI_Cart_coords(comm_grid, proc_rank, DIMS_COUNT, coords);

A_block_size = ceil((double)n_1 / dims[X]);
B_block_size = ceil((double)n_3 / dims[Y]);
aligned_n1 = A_block_size * dims[X];
aligned_n3 = B_block_size * dims[Y];

if (coords[X] == 0 && coords[Y] == 0)
{
    A = malloc(sizeof(double) * aligned_n1 * n_2);
    B = malloc(sizeof(double) * n_2 * aligned_n3);
    C = malloc(sizeof(double) * aligned_n1 * aligned_n3);

    generate_matrix(A, n_2, n_1, n_2, true);
    generate_matrix(B, aligned_n3, n_2, n_3, false);
}

start_time = MPI_Wtime();

A_block = malloc(sizeof(double) * A_block_size * n_2);
B_block = malloc(sizeof(double) * B_block_size * n_2);
C_block = malloc(sizeof(double) * A_block_size * B_block_size);

split_A(A, A_block, A_block_size, n_2, coords[Y],
        comm_rows, comm_columns);
split_B(B, B_block, B_block_size, n_2, aligned_n3, coords[X],
        comm_rows, comm_columns);

multiply(A_block, B_block, C_block, A_block_size,
         B_block_size, n_2);

gather_C(C_block, C, A_block_size, B_block_size, aligned_n1,
         aligned_n3, proc_count, comm_grid);

finish_time = MPI_Wtime();

if (coords[Y] == 0 && coords[X] == 0)
{
    printf("Is matrix C correct? - %s\n",
    check_C(C, aligned_n3, n_1, n_3, n_2) ? "yes" : "no");
```

```
        printf("Time: %lf\n", finish_time - start_time);

        free(A);
        free(B);
        free(C);
    }

    free(A_block);
    free(B_block);
    free(C_block);
    MPI_Comm_free(&comm_grid);
    MPI_Comm_free(&comm_rows);
    MPI_Comm_free(&comm_columns);

    MPI_Finalize();

    return EXIT_SUCCESS;
}

void init_dims(int dims[DIMS_COUNT], int proc_count, int argc,
               char **argv)
{
    if (argc < 3)
        MPI_Dims_create(proc_count, DIMS_COUNT, dims);
    else
    {
        dims[X] = atoi(argv[1]);
        dims[Y] = atoi(argv[2]);
    }
}

void init_communicators(const int dims[DIMS_COUNT],
                        MPI_Comm *comm_grid, MPI_Comm *comm_rows,
                        MPI_Comm *comm_columns)
{
    int reorder = 1;
    int periods[DIMS_COUNT] = {};
    int sub_dims[DIMS_COUNT] = {};

    MPI_Cart_create(MPI_COMM_WORLD, DIMS_COUNT, dims, periods,
                    reorder, comm_grid);

    sub_dims[X] = false;
    sub_dims[Y] = true;
    MPI_Cart_sub(*comm_grid, sub_dims, comm_rows);

    sub_dims[X] = true;
    sub_dims[Y] = false;
    MPI_Cart_sub(*comm_grid, sub_dims, comm_columns);
}

void generate_matrix(double *matrix, int column, int leading_row,
                     int leading_column, bool onRows)
{
    for (int i = 0; i < leading_row; ++i)
```

```
        for (int j = 0; j < leading_column; ++j)
            matrix[i * column + j] = onRows ? i : j;
}

void split_A(const double *A, double *A_block, int A_block_size,
             int n_2, int coords_y, MPI_Comm comm_rows,
             MPI_Comm comm_columns)
{
    if (coords_y == 0)
    {
        MPI_Scatter(A, A_block_size * n_2, MPI_DOUBLE, A_block,
                    A_block_size * n_2, MPI_DOUBLE, 0,
                    comm_columns);
    }

    MPI_Bcast(A_block, A_block_size * n_2, MPI_DOUBLE, 0,
              comm_rows);
}

void split_B(const double *B, double *B_block, int B_block_size,
             int n_2, int aligned_n3, int coords_x,
             MPI_Comm comm_rows, MPI_Comm comm_columns)
{
    if (coords_x == 0)
    {
        MPI_Datatype column_not_resized_t;
        MPI_Datatype column_resized_t;

        MPI_Type_vector(n_2, B_block_size, aligned_n3, MPI_DOUBLE,
                        &column_not_resized_t);
        MPI_Type_commit(&column_not_resized_t);

        MPI_Type_create_resized(column_not_resized_t, 0,
                                B_block_size * sizeof(double),
                                &column_resized_t);
        MPI_Type_commit(&column_resized_t);

        MPI_Scatter(B, 1, column_resized_t, B_block,
                    B_block_size * n_2, MPI_DOUBLE, 0, comm_rows);

        MPI_Type_free(&column_not_resized_t);
        MPI_Type_free(&column_resized_t);
    }

    MPI_Bcast(B_block, B_block_size * n_2, MPI_DOUBLE, 0,
              comm_columns);
}

void multiply(const double *A_block, const double *B_block,
              double *C_block, int A_block_size,
              int B_block_size, int n_2)
{
    for (int i = 0; i < A_block_size; ++i)
        for (int j = 0; j < B_block_size; ++j)
            C_block[i * B_block_size + j] = 0;
```

```
    for (int i = 0; i < A_block_size; ++i)
        for (int j = 0; j < n_2; ++j)
            for (int k = 0; k < B_block_size; ++k)
                C_block[i*B_block_size + k] +=
                A_block[i*n_2 + j] * B_block[j*B_block_size + k];
}

void gather_C(const double *C_block, double *C, int A_block_size,
              int B_block_size, int aligned_n1, int aligned_n3,
              int proc_count, MPI_Comm comm_grid)
{
    MPI_Datatype not_resized_recv_t;
    MPI_Datatype resized_recv_t;

    int dims_x = aligned_n1 / A_block_size;
    int dims_y = aligned_n3 / B_block_size;
    int *recv_counts = malloc(sizeof(int) * proc_count);
    int *displs = malloc(sizeof(int) * proc_count);

    MPI_Type_vector(A_block_size, B_block_size, aligned_n3,
                    MPI_DOUBLE, &not_resized_recv_t);
    MPI_Type_commit(&not_resized_recv_t);

    MPI_Type_create_resized(not_resized_recv_t, 0,
                            B_block_size * sizeof(double),
                            &resized_recv_t);
    MPI_Type_commit(&resized_recv_t);

    for (int i = 0; i < dims_x; ++i)
        for (int j = 0; j < dims_y; ++j)
        {
            recv_counts[i * dims_y + j] = 1;
            displs[i * dims_y + j] = j + i * dims_y * A_block_size;
        }

    MPI_Gatherv(C_block, A_block_size * B_block_size, MPI_DOUBLE, C,
                recv_counts, displs, resized_recv_t, 0, comm_grid);

    MPI_Type_free(&not_resized_recv_t);
    MPI_Type_free(&resized_recv_t);
    free(recv_counts);
    free(displs);
}

bool check_C(const double *C, int column, int leading_row,
             int leading_column, int n_2)
{
    for (int i = 0; i < leading_row; ++i)
        for (int j = 0; j < leading_column; ++j)
            if (C[i * column + j] != (double)i * j * n_2)
                return false;

    return true;
}
```

run.sh

```
#!/bin/bash

#PBS -l walltime=00:05:00
#PBS -l select=<nodes>:ncpus=<cpus>:mpiprocs=<processes>
#PBS -m n

cd $PBS_O_WORKDIR

MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')
echo "Number of MPI process: $MPI_NP"

echo 'File $PBS_NODEFILE:'
cat  $PBS_NODEFILE
echo

mpirun -hostfile $PBS_NODEFILE -np $MPI_NP <program> <args>
```