**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

«Параллельная реализация метода Якоби в трехмерной области»

Студента 2 курса, 21211 группы

**Петрова Сергея Евгеньевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Антон Юрьевич Кудинов

**СОДЕРЖАНИЕ**

## ЦЕЛЬ

*Освоить методы распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области;*

## ЗАДАНИЕ

1. *Написать параллельную программу на языке C/C++ с использованием MPI, реализующую решение уравнения* $\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} + \frac{\partial^2 \varphi}{\partial z^2} - a\varphi = \rho$ *методом Якоби в трехмерной области в случае одномерной декомпозиции области. Уделить внимание тому, чтобы обмены граничными значениями подобластей выполнялись на фоне счета.*
2. *Измерить время работы программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 16. Размеры сетки и порог сходимости подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.*
3. *Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.*
4. *Выполнить профилирование программы с помощью MPE при использовании 16 ядер. По профилю убедиться, что коммуникации происходят на фоне счета.*

## ИСХОДНЫЕ ДАННЫЕ

*Исходные данные для тестирования реализаций представленного метода и выполнения лабораторной работы взяты следующие:*

- *Область моделирования: [-1;1] × [-1;1] × [-1;1];*
- *Искомая функция:* $\varphi(x,\ y,\ z) = x^2 + y^2 + z^2$*;*
- *Правая часть уравнения:* $\rho(x,\ y,\ z) = 6 - \varphi(x,\ y,\ z)$*;*
- *Параметр уравнения:* $a = 10^5$*;*
- *Порог сходимости:* $\varepsilon = 10^{-8}$*;*
- *Начальное приближение:* $\varphi_{i,j,k}^0 = 0$.

# ОПИСАНИЕ РАБОТЫ

## Описание выполненной работы

1. *Написал параллельную программу, реализующую решение уравнения* $\frac{\partial^2\varphi}{\partial x^2} + \frac{\partial^2\varphi}{\partial y^2} + \frac{\partial^2\varphi}{\partial z^2} - a\varphi = \rho$ *методом Якоби в трехмерной области в случае одномерной декомпозиции области;*

2. *Запустил программу при использовании 1, 2, 4, 8, 16 ядер со следующими размерами сетки:* $N_x = 300,\ N_y = 300,\ N_z = 300$*;*

3. *Выполнил профилирование программы с помощью MPE при использовании 16 ядер со следующими размерами матриц:* $N_x = 300,\ N_y = 300,\ N_z = 300$*;*

## Команды для компиляции и запуска

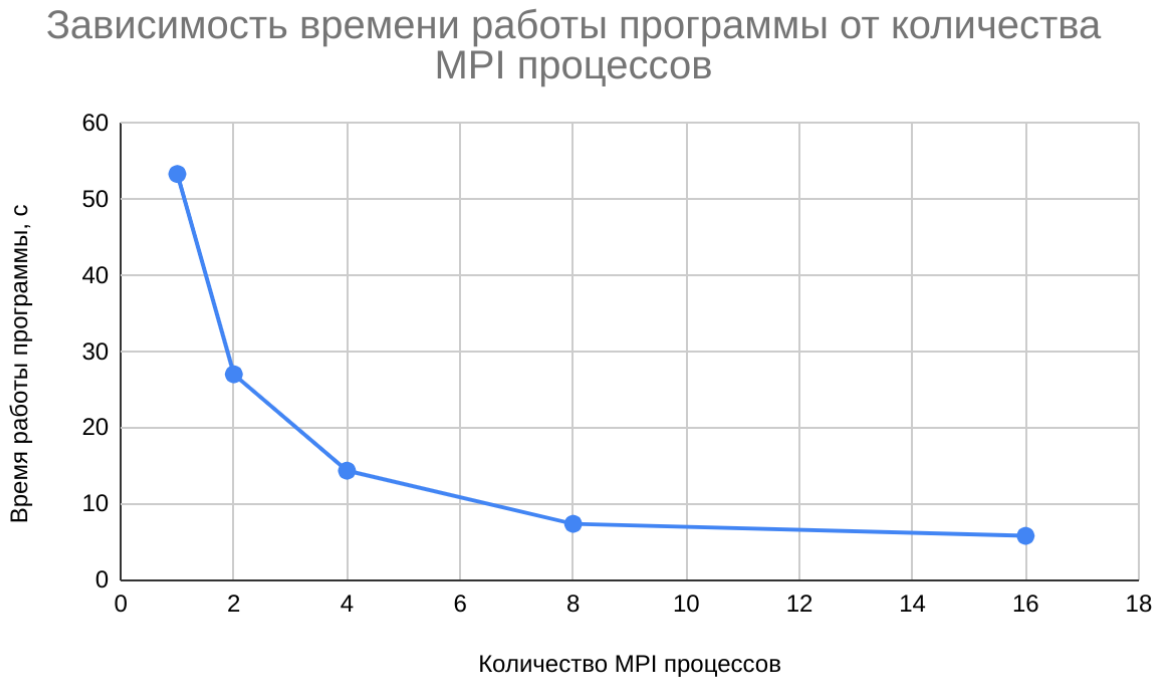
*Команда для компиляции MPI программы*

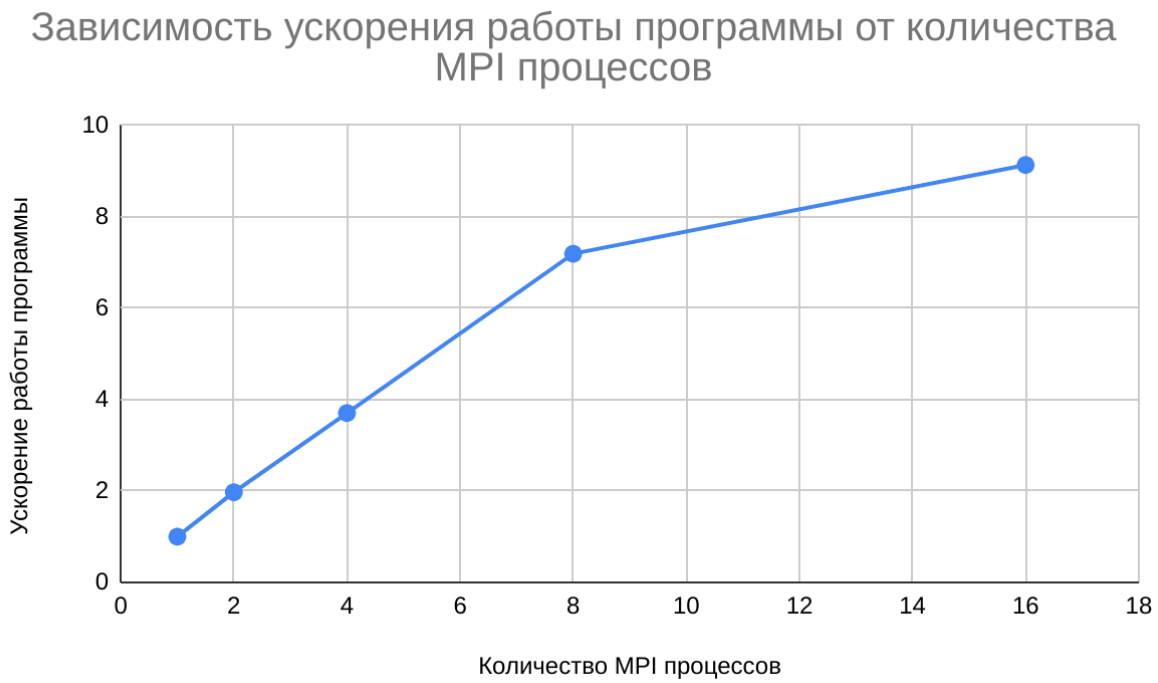
```
hpcuser265@clu:~> mpicc
--std=c99
-o mpi_multiply
mpi_multiply.c
-lm
```

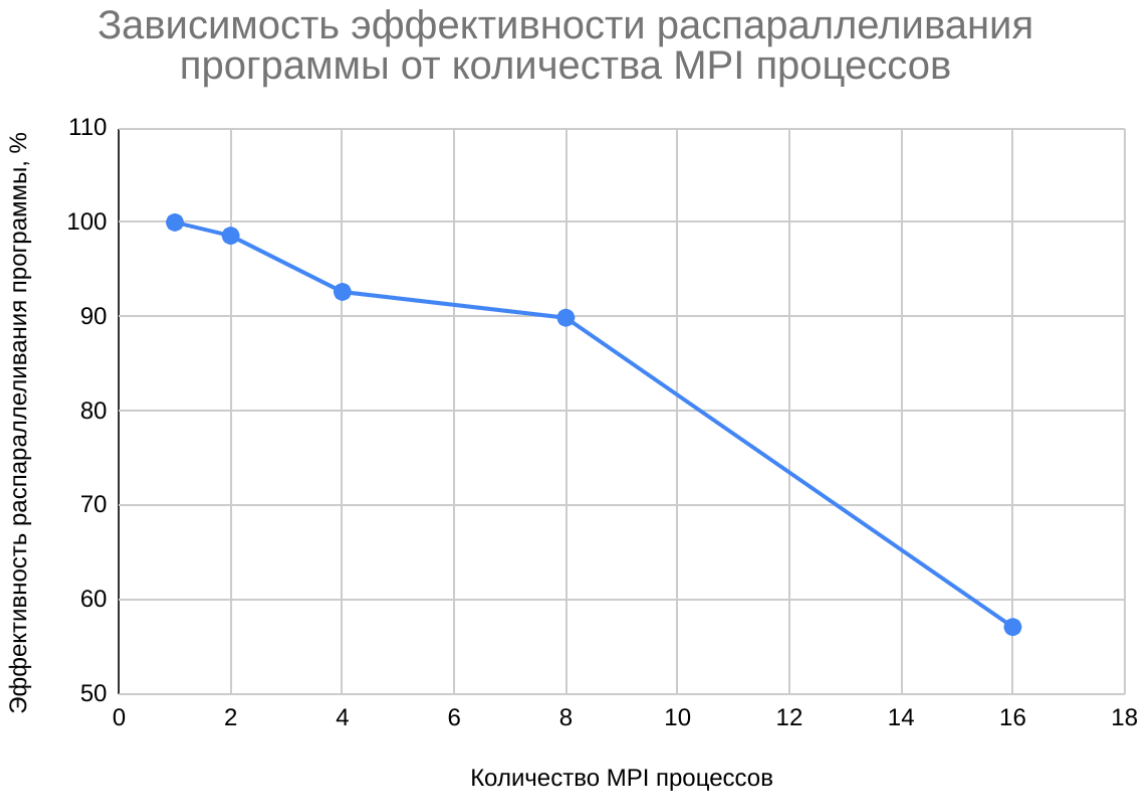
*Команда для компиляции MPI программы с использованием MPE*

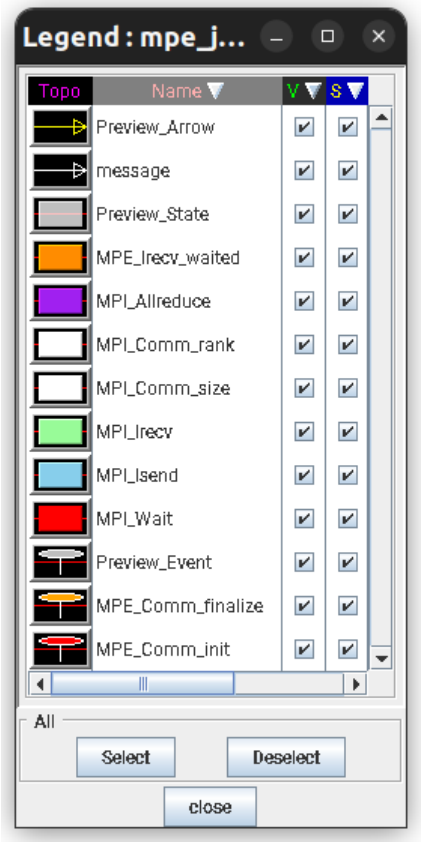
```
hpcuser265@clu:~> mpecc
-mpilog
--std=c99
-o mpe_multiply
mpi_multiply.c
-lm
```

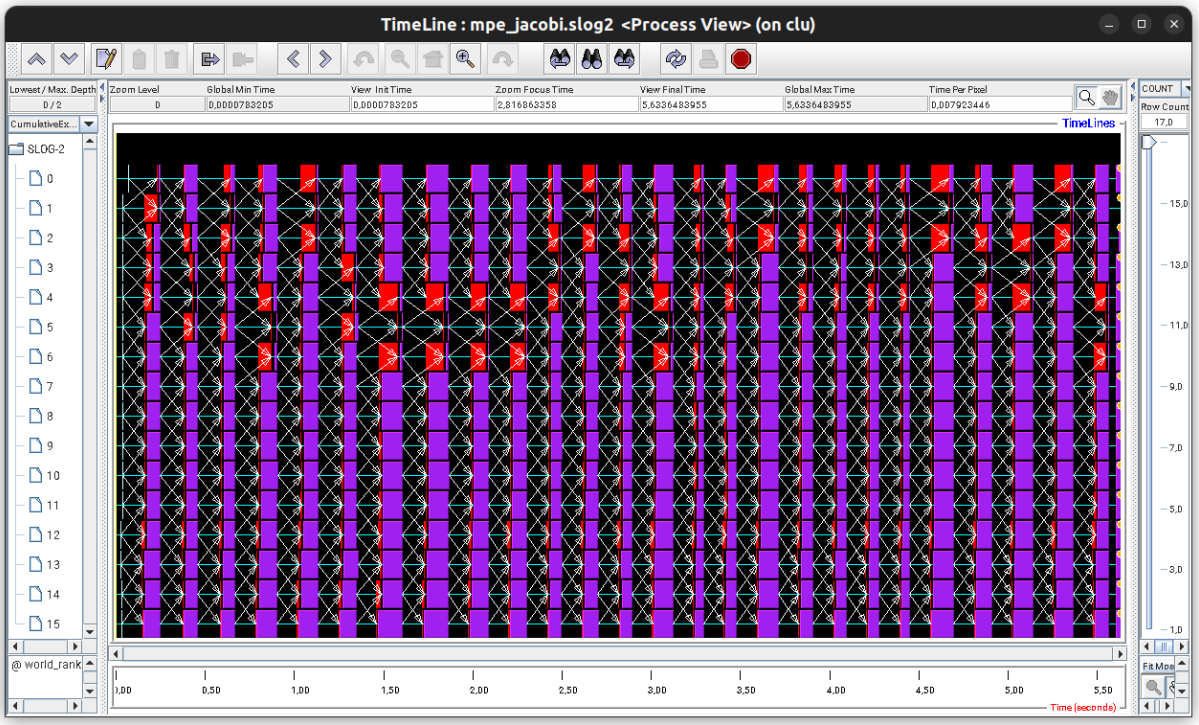
## Результаты измерения

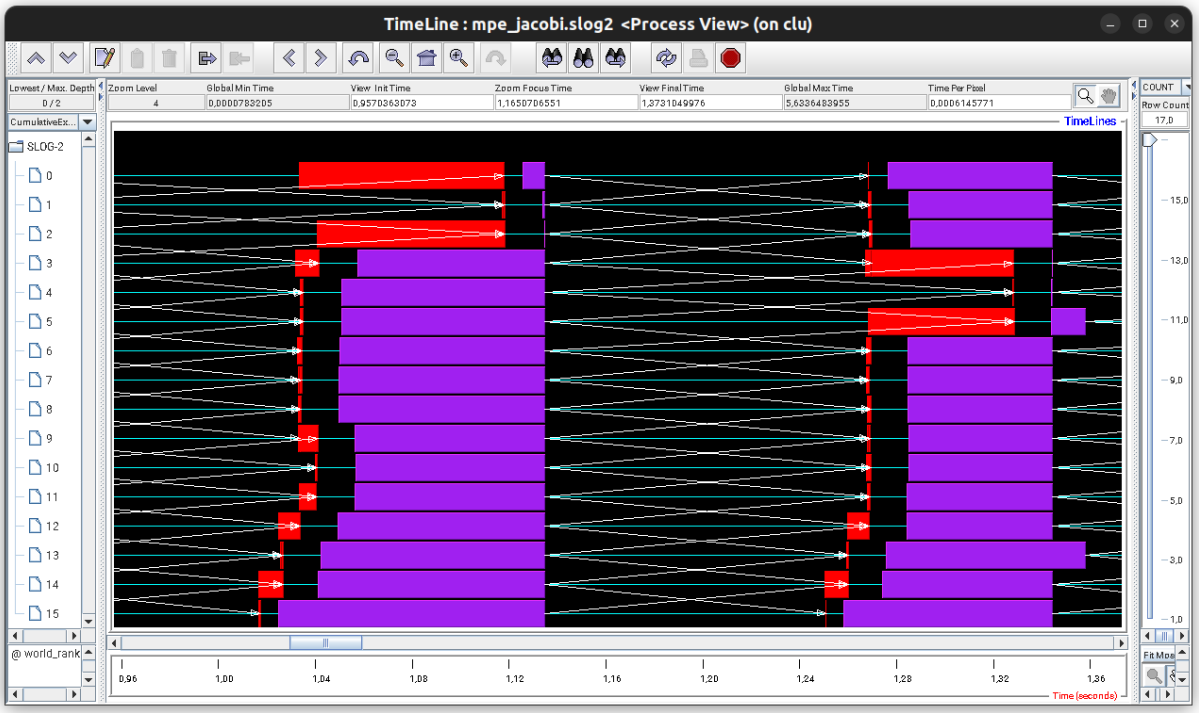


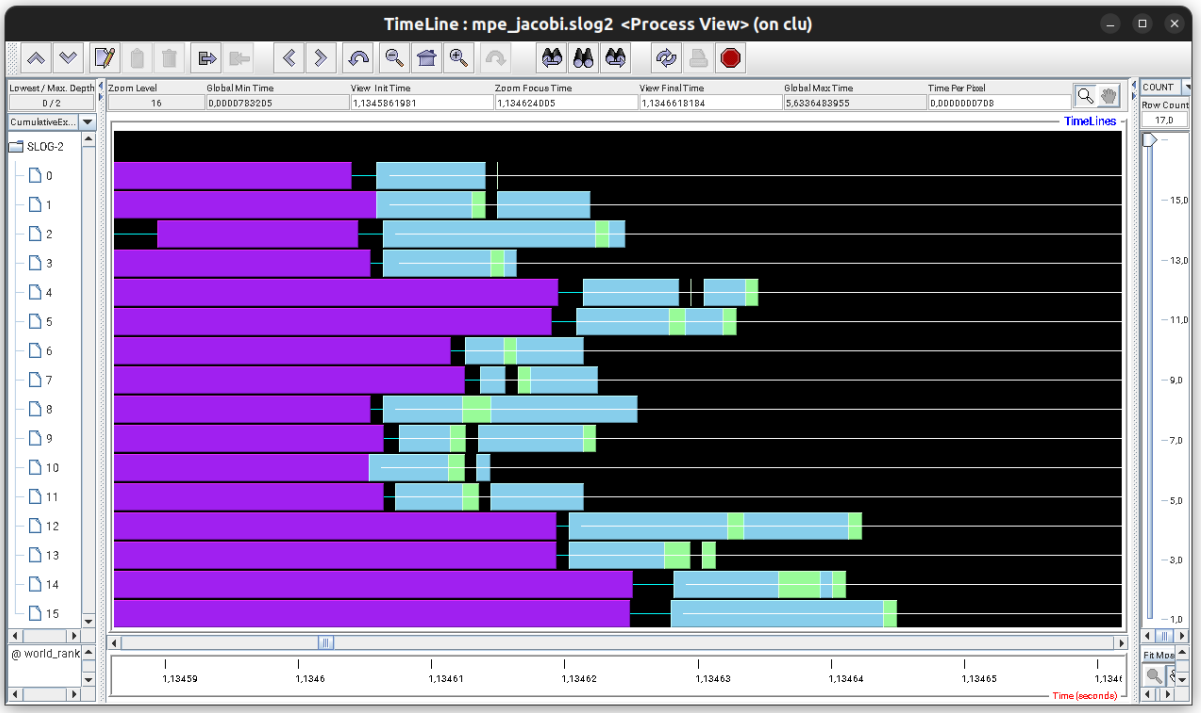



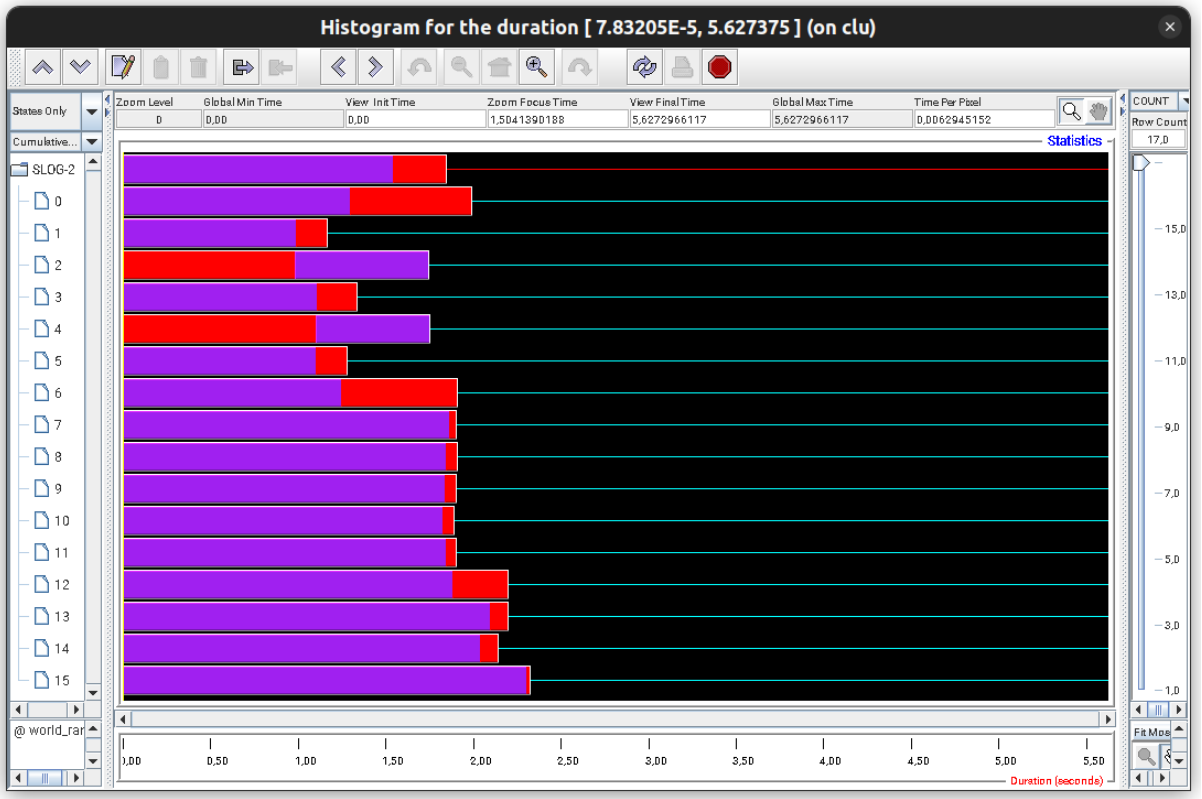

Зависимость эффективности распараллеливания программы от количества MPI процессов
Эффективность распараллеливания программы, %
110
100
90
80
70
60
50
0
2
4
6
8
10
12
14
16
18
Количество MPI процессов


Legend : mpe_j...
Topo
Name
V
S
Preview_Arrow
message
Preview_State
MPE_Irecv_waited
MPI_Allreduce
MPI_Comm_rank
MPI_Comm_size
MPI_Irecv
MPI_Isend
MPI_Wait
Preview_Event
MPE_Comm_finalize
MPE_Comm_init
All
Select
Deselect
close


TimeLine : mpe_jacobi.slog2 <Process View> (on clu)
TimeLines
SLOG-2
Time (seconds)


TimeLine : mpe_jacobi.slog2 <Process View> (on clu)
TimeLines
SLOG-2
Time (seconds)


TimeLine : mpe_jacobi.slog2 <Process View> (on clu)
TimeLines
SLOG-2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
@ world_rank
1,13459
1,1346
1,13461
1,13462
1,13463
1,13464
1,13465
Time (seconds)


Histogram for the duration [ 7.83205E-5, 5.627375 ] (on clu)
Statistics
SLOG-2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0,00
0,50
1,00
1,50
2,00
2,50
3,00
3,50
4,00
4,50
5,00
5,50
Duration (seconds)

## ЗАКЛЮЧЕНИЕ

*В ходе выполнения лабораторной работы:*

- *Освоил методы распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области;*

*Анализируя результаты исследований, можно сделать следующие выводы:*

- *При 16 процессах уменьшается эффективность распараллеливания уменьшается. Связать это можно с блокирующей операцией MPI_Allreduce;*
- *По временной шкале, полученной при профилировании, видно, что после начала асинхронной передачи границ вычисляются значения функции во внутренних точках до вызова MPI_Wait. После завершения приёма границ вычисляются значения функции на границе области.*

# ПРИЛОЖЕНИЕ (ЛИСТИНГ ПРОГРАММЫ)

*mpi_jacobi.c*

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

// Initial coordinates
const double X_0 = -1.0;
const double Y_0 = -1.0;
const double Z_0 = -1.0;

// Dimension size
const double D_X = 2.0;
const double D_Y = 2.0;
const double D_Z = 2.0;

// Grid size
const int N_X = 300;
const int N_Y = 300;
const int N_Z = 300;

// Step size
const double H_X = D_X / (N_X - 1);
const double H_Y = D_Y / (N_Y - 1);
const double H_Z = D_Z / (N_Z - 1);

// Square of step size
const double H_X_2 = H_X * H_X;
const double H_Y_2 = H_Y * H_Y;
const double H_Z_2 = H_Z * H_Z;

// Parameters
const double a = 1.0E5;
const double epsilon = 1.0E-8;

double phi(double x, double y, double z);
double rho(double x, double y, double z);
int get_index(int x, int y, int z);
int get_x(int i);
int get_y(int j);
int get_z(int k);
void divide_area_into_layers(int *layer_heights, int *offsets,
                             int proc_count);
void init_layers(double *prev_func, double *curr_func,
                 int layer_height, int offset);
void send_up_layer(const double *send_layer, double *recv_layer,
                   int proc_rank, MPI_Request *send_up_req,
                   MPI_Request *recv_up_req);
void send_down_layer(const double *send_layer, double *recv_layer,
                     int proc_rank, MPI_Request *send_down_req,
                     MPI_Request *recv_down_req);
```

```
void receive_layer(MPI_Request *send_req, MPI_Request *recv_req);
double calc_center(const double *prev_func, double *curr_func,
                   int layer_height, int offset);
double calc_border(const double *prev_func, double *curr_func,
                   double *up_border_layer,
                   double *down_border_layer, int layer_height,
                   int offset, int proc_rank, int proc_count);
double calc_max_diff(const double *func, int layer_height,
                     int offset);

int main(int argc, char **argv)
{
    int proc_rank = 0;
    int proc_count = 0;
    int is_prev_func = 0;
    int is_curr_func = 1;
    double start_time = 0.0;
    double finish_time = 0.0;
    double max_diff = 0.0;
    int *layer_heights = NULL;
    int *offsets = NULL;
    double *up_border_layer = NULL;
    double *down_border_layer = NULL;
    double *(func[2]) = { NULL, NULL };
    MPI_Request send_up_req;
    MPI_Request send_down_req;
    MPI_Request recv_up_req;
    MPI_Request recv_down_req;

    MPI_Init(&argc, &argv);

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

    // Divide area
    layer_heights = malloc(sizeof(int) * proc_count);
    offsets = malloc(sizeof(int) * proc_count);
    divide_area_into_layers(layer_heights, offsets, proc_count);

    // Init layers
    func[is_prev_func] = malloc(sizeof(double) *
                         layer_heights[proc_rank] * N_Y * N_Z);
    func[is_curr_func] = malloc(sizeof(double) *
                         layer_heights[proc_rank] * N_Y * N_Z);
    init_layers(func[is_prev_func], func[is_curr_func],
                layer_heights[proc_rank], offsets[proc_rank]);

    up_border_layer = malloc(sizeof(double) * N_Y * N_Z);
    down_border_layer = malloc(sizeof(double) * N_Y * N_Z);

    start_time = MPI_Wtime();

    do
    {
        double tmp_max_diff = 0.0;
```

```
        double proc_max_diff = 0.0;

        // Layer swap
        is_prev_func = 1 - is_prev_func;
        is_curr_func = 1 - is_curr_func;

        // Send border
        if (proc_rank != 0)
            send_up_layer(func[is_prev_func], up_border_layer,
                          proc_rank, &send_up_req, &recv_up_req);

        if (proc_rank != proc_count - 1)
            send_down_layer(func[is_prev_func] +
                            (layer_heights[proc_rank] - 1)*N_Y*N_Z,
                            down_border_layer, proc_rank,
                            &send_down_req, &recv_down_req);

        // Calculate center
        proc_max_diff = calc_center(func[is_prev_func],
                                    func[is_curr_func],
                                    layer_heights[proc_rank],
                                    offsets[proc_rank]);

        // Receive border
        if (proc_rank != 0)
            receive_layer(&send_up_req, &recv_up_req);

        if (proc_rank != proc_count - 1)
            receive_layer(&send_down_req, &recv_down_req);

        // Calculate border
        tmp_max_diff = calc_border(func[is_prev_func],
                                   func[is_curr_func],
                                   up_border_layer,
                                   down_border_layer,
                                   layer_heights[proc_rank],
                                   offsets[proc_rank], proc_rank,
                                   proc_count);
        proc_max_diff = fmax(tmp_max_diff, proc_max_diff);

        // Calculate the differences of the previous and current
           calculated functions
        MPI_Allreduce(&proc_max_diff, &max_diff, 1, MPI_DOUBLE,
                      MPI_MAX, MPI_COMM_WORLD);
    } while (max_diff >= epsilon);

    // Calculate the differences of the calculated and
       theoretical functions
    max_diff = calc_max_diff(func[is_curr_func],
                             layer_heights[proc_rank],
                             offsets[proc_rank]);

    finish_time = MPI_Wtime();

    if (proc_rank == 0)
    {
```

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



    free(offsets);
    free(layer_heights);
    free(func[is_prev_func]);
    free(func[is_curr_func]);
    free(up_border_layer);
    free(down_border_layer);

    MPI_Finalize();

    return EXIT_SUCCESS;
}

double rho(double x, double y, double z)
{
    return 6 - a * phi(x, y, z);
}

double phi(double x, double y, double z)
{
    return x * x + y * y + z * z;
}

int get_index(int x, int y, int z)
{
    return x * N_Y * N_Z + y * N_Z + z;
}

int get_x(int i)
{
    return X_0 + i * H_X;
}

int get_y(int j)
{
    return Y_0 + j * H_Y;
}

int get_z(int k)
{
    return Z_0 + k * H_Z;
}

void divide_area_into_layers(int *layer_heights, int *offsets,
                             int proc_count)
{
    int offset = 0;
    for (int i = 0; i < proc_count; ++i)
    {
        layer_heights[i] = N_X / proc_count;
```

```
        // Distribute the remainder of the processes
        if (i < N_X % proc_count)
            layer_heights[i]++;

        offsets[i] = offset;
        offset += layer_heights[i];
    }
}

void init_layers(double *prev_func, double *curr_func,
                 int layer_height, int offset)
{
    for (int i = 0; i < layer_height; ++i)
        for (int j = 0; j < N_Y; j++)
            for (int k = 0; k < N_Z; k++)
            {
                if ((offset + i == 0) || (j == 0) || (k == 0) ||
                   (offset + i == N_X - 1) || (j == N_Y - 1) ||
                   (k == N_Z - 1))
                {
                    prev_func[get_index(i, j, k)] =
                        phi(get_x(offset + i), get_y(j), get_z(k));
                    curr_func[get_index(i, j, k)] =
                        phi(get_x(offset + i), get_y(j), get_z(k));
                }
                else
                {
                    prev_func[get_index(i, j, k)] = 0;
                    curr_func[get_index(i, j, k)] = 0;
                }
            }
}

void send_up_layer(const double *send_layer, double *recv_layer,
                   int proc_rank, MPI_Request *send_up_req,
                   MPI_Request *recv_up_req)
{
    MPI_Isend(send_layer, N_Y * N_Z, MPI_DOUBLE, proc_rank - 1,
              proc_rank, MPI_COMM_WORLD, send_up_req);
    MPI_Irecv(recv_layer, N_Y * N_Z, MPI_DOUBLE, proc_rank - 1,
              proc_rank - 1, MPI_COMM_WORLD, recv_up_req);
}

void send_down_layer(const double *send_layer, double *recv_layer,
                     int proc_rank, MPI_Request *send_down_req,
                     MPI_Request *recv_down_req)
{
    MPI_Isend(send_layer, N_Y * N_Z, MPI_DOUBLE, proc_rank + 1,
              proc_rank, MPI_COMM_WORLD, send_down_req);
    MPI_Irecv(recv_layer, N_Y * N_Z, MPI_DOUBLE, proc_rank + 1,
              proc_rank + 1, MPI_COMM_WORLD, recv_down_req);
}
```

```
void receive_layer(MPI_Request *send_req, MPI_Request *recv_req)
{
    MPI_Wait(send_req, MPI_STATUS_IGNORE);
    MPI_Wait(recv_req, MPI_STATUS_IGNORE);
}
double calc_center(const double *prev_func, double *curr_func,
                   int layer_height, int offset)
{
    double Fi = 0.0;
    double Fj = 0.0;
    double Fk = 0.0;
    double tmp_max_diff = 0.0;
    double max_diff = 0.0;
    for (int i = 1; i < layer_height - 1; ++i)
        for (int j = 1; j < N_Y - 1; ++j)
            for (int k = 1; k < N_Z - 1; ++k)
            {
                Fi = (prev_func[get_index(i + 1, j, k)] +
                      prev_func[get_index(i - 1, j, k)]) / H_X_2;
                Fj = (prev_func[get_index(i, j + 1, k)] +
                      prev_func[get_index(i, j - 1, k)]) / H_Y_2;
                Fk = (prev_func[get_index(i, j, k + 1)] +
                      prev_func[get_index(i, j, k - 1)]) / H_Z_2;


                curr_func[get_index(i, j, k)] = (Fi + Fj + Fk -
                    rho(get_x(offset + i), get_y(j), get_z(k))) /
                    (2 / H_X_2 + 2 / H_Y_2 + 2 / H_Z_2 + a);

                // Update max difference
                tmp_max_diff = fabs(curr_func[get_index(i, j, k)] -
                                    prev_func[get_index(i, j, k)]);
                if (tmp_max_diff > max_diff)
                    max_diff = tmp_max_diff;
            }

    return max_diff;
}

double calc_border(const double *prev_func, double *curr_func,
                   double *up_border_layer,
                   double *down_border_layer, int layer_height,
                   int offset, int proc_rank, int proc_count)
{
    double Fi = 0.0;
    double Fj = 0.0;
    double Fk = 0.0;
    double tmp_max_diff = 0.0;
    double max_diff = 0.0;

    for (int j = 1; j < N_Y - 1; ++j)
        for (int k = 1; k < N_Z - 1; ++k)
        {
            // Calculate the upper border
            if (proc_rank != 0)
            {
```

```
                Fi = (prev_func[get_index(1, j, k)] +
                     up_border_layer[get_index(0, j, k)]) / H_X_2;
                Fj = (prev_func[get_index(0, j + 1, k)] +
                     prev_func[get_index(0, j - 1, k)]) / H_Y_2;
                Fk = (prev_func[get_index(0, j, k + 1)] +
                     prev_func[get_index(0, j, k - 1)]) / H_Z_2;

                curr_func[get_index(0, j, k)] = (Fi + Fj + Fk -
                    rho(get_x(offset), get_y(j), get_z(k))) /
                    (2 / H_X_2 + 2 / H_Y_2 + 2 / H_Z_2 + a);

                // Update max difference
                tmp_max_diff = fabs(curr_func[get_index(0, j, k)] -
                                    prev_func[get_index(0, j, k)]);
                if (tmp_max_diff > max_diff)
                    max_diff = tmp_max_diff;
            }

            // Calculate the lower border
            if (proc_rank != proc_count - 1)
            {
                Fi = (prev_func[get_index(layer_height - 2, j, k)] +
                     down_border_layer[get_index(0, j, k)]) / H_X_2;
                Fj = (prev_func[get_index(layer_height - 1,
                     j + 1, k)] + prev_func[get_index(
                     layer_height - 1, j - 1, k)]) / H_Y_2;
                Fk = (prev_func[get_index(layer_height - 1, j,
                k + 1)] + prev_func[get_index(layer_height - 1, j,
                k - 1)]) / H_Z_2;

                curr_func[get_index(layer_height - 1, j, k)] =
                    (Fi + Fj + Fk - rho(get_x(offset +
                    layer_height - 1), get_y(j), get_z(k))) /
                    (2 / H_X_2 + 2 / H_Y_2 + 2 / H_Z_2 + a);

                // Check for calculation end
                tmp_max_diff = fabs(curr_func[get_index(
                    layer_height - 1, j, k)] -
                    prev_func[get_index(layer_height - 1, j, k)]);
                if (tmp_max_diff > max_diff)
                    max_diff = tmp_max_diff;
            }
        }

    return max_diff;
}

double calc_max_diff(const double *curr_func, int layer_height, int
offset)
{
    double tmp_max_delta = 0.0;
    double max_proc_delta = 0.0;
    double max_delta = 0.0;

    for (int i = 0; i < layer_height; ++i)
```

```
        for (int j = 0; j < N_Y; ++j)
            for (int k = 0; k < N_Z; ++k)
            {
                tmp_max_delta = fabs(curr_func[get_index(i, j, k)] -
                    phi(get_x(offset + i), get_y(j), get_z(k)));
                if (tmp_max_delta > max_proc_delta)
                    max_proc_delta = tmp_max_delta;
            }

    MPI_Allreduce(&max_proc_delta, &max_delta, 1, MPI_DOUBLE,
                  MPI_MAX, MPI_COMM_WORLD);

    return max_delta;
}
```

*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>
```