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. *Написал 2-ую версию программы, использующую асинхронный сбор максимальной разности между значениями вычисленных функций на предыдущей и текущей итерации при помощи MPI_Iallreduce (при этом выполняется вычисление значений функции на 1 итерацию вперёд);*

3. *Запустил программы при использовании 1, 2, 4, 8, 16 ядер со следующими параметрами:* $\varepsilon = 10^{-3}$, $N_x = 400$, $N_y = 400$, $N_z = 400$;

4. *Составил графики зависимости времени работы, ускорения работы и эффективности распараллеливания программы от количества MPI процессов. Также добавил график зависимости времени работы 2-ой версии программы без учета лишней итерации от количества MPI процессов;*

5. *Выполнил профилирование программ с помощью MPE при использовании 16 ядер;*

Команды для компиляции и запуска

Команда для компиляции MPI программы

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

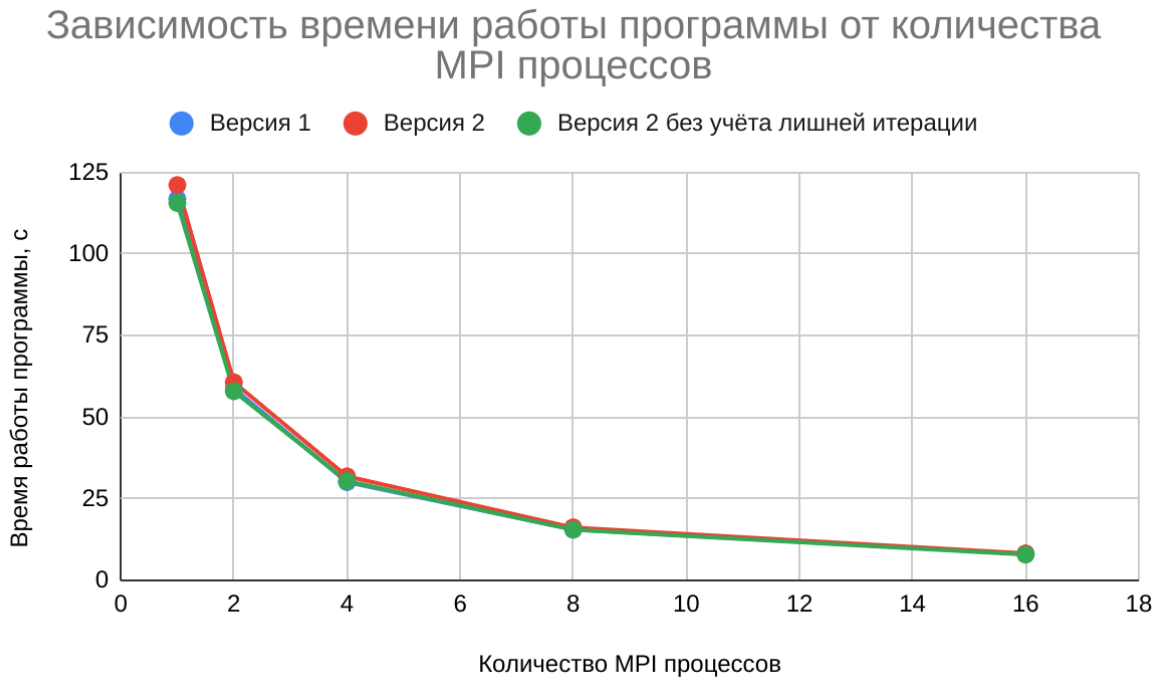
Команда для компиляции MPI программы с использованием MPE

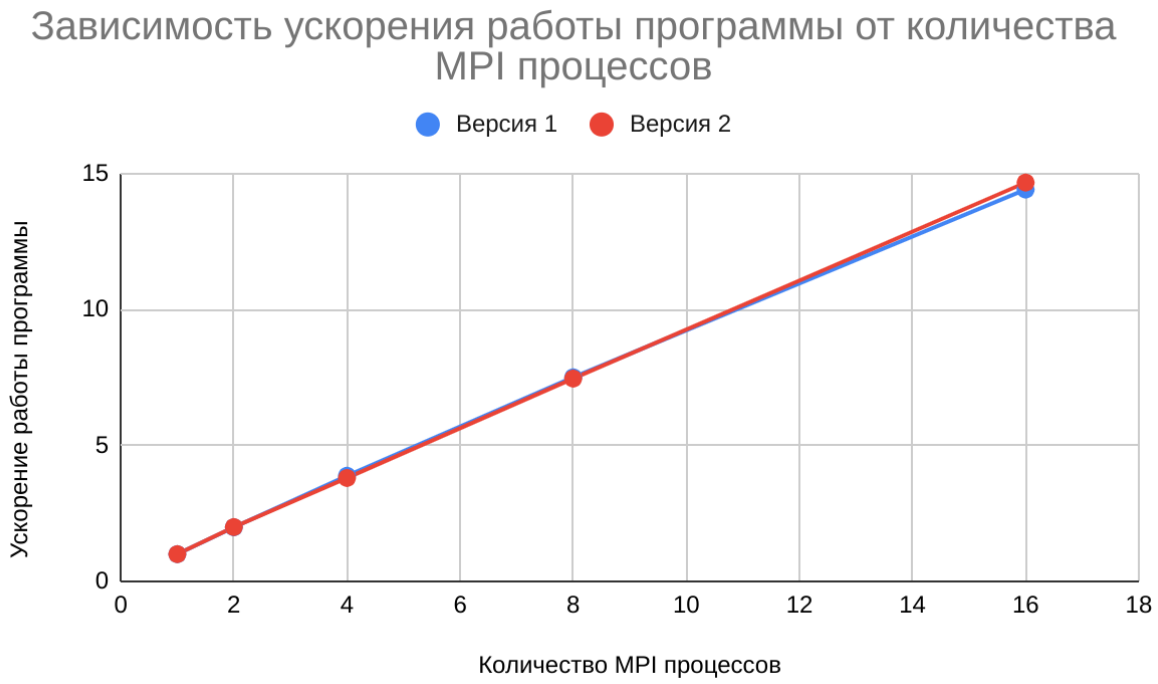
```
hpcuser265@clu:~> mpecc
-mpilog
--std=c99
-o mpe_multiply
mpi_multiply.c
-lm
```

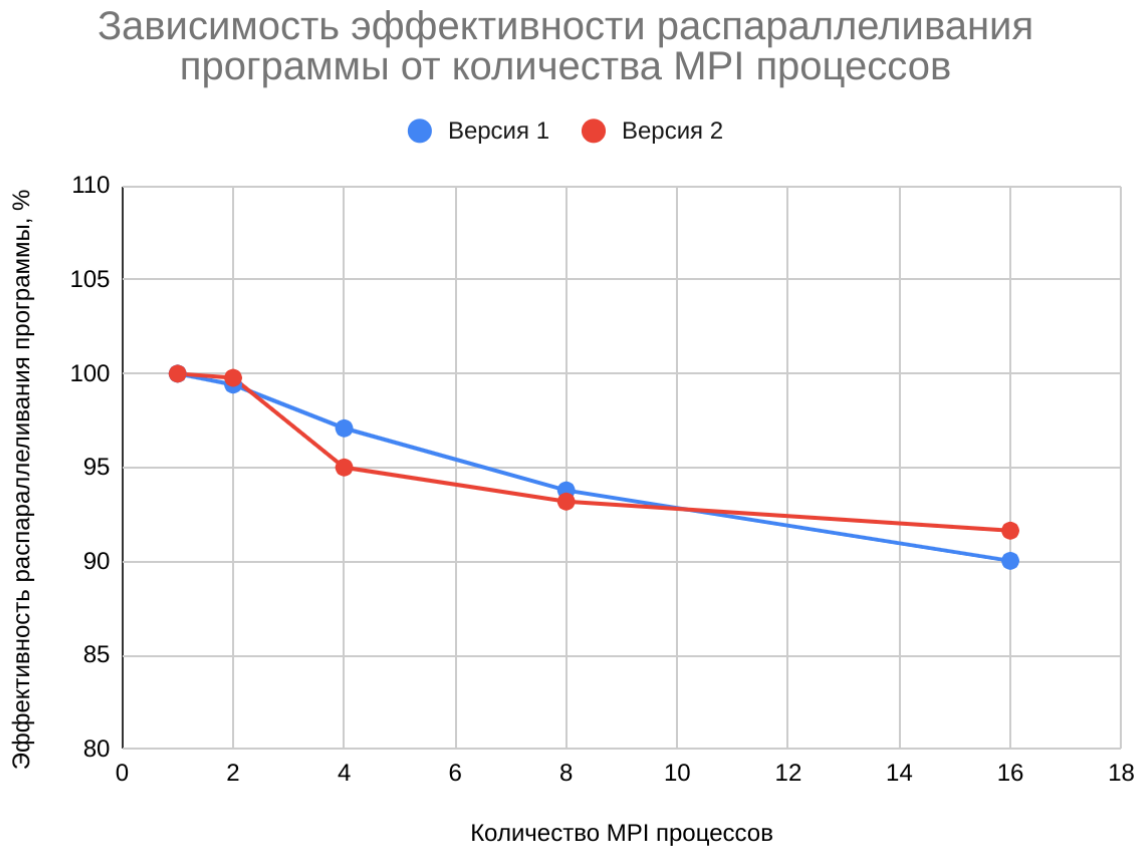
Результаты измерения

Таблица с результатами измерений

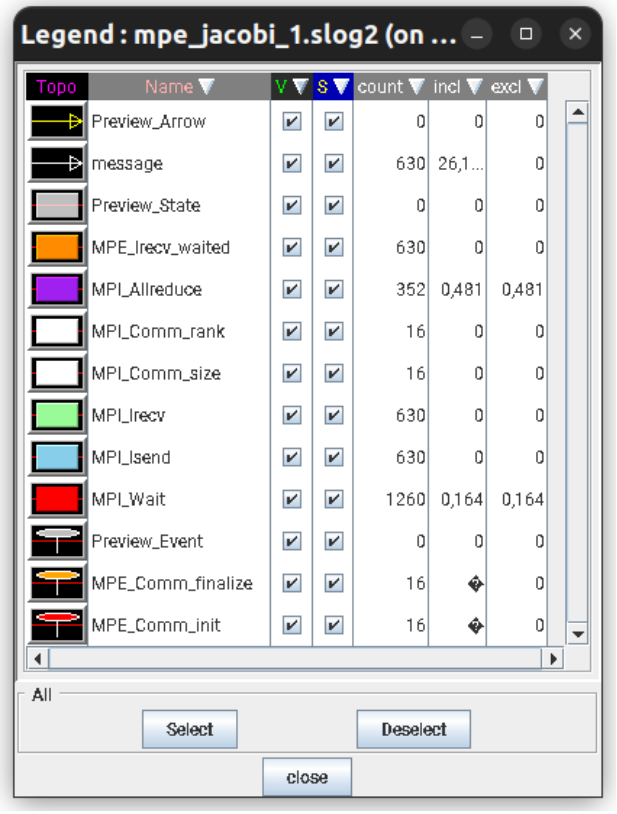
Количество MPI процессов	Время работы программы			Ускорение работы программы		Эффективность распараллеливания программы	
	Версия 1	Версия 2	Версия 2 без учёта лишней итерации	Версия 1	Версия 2	Версия 1	Версия 2
1	116,866122	121,106482	115,6016419	1	1	100	100
2	58,775767	60,688081	57,92953186	1,98833852	1,99555629	99,41692637	99,7778146
4	30,091582	31,867197	30,41868805	3,88368155	3,80034936	97,0920389	95,0087342
8	15,576002	16,244372	15,50599145	7,50296013	7,45528863	93,78700163	93,1911079
16	8,112851	8,259677	7,884237136	14,4050620	14,6623750	90,03163777	91,6398440

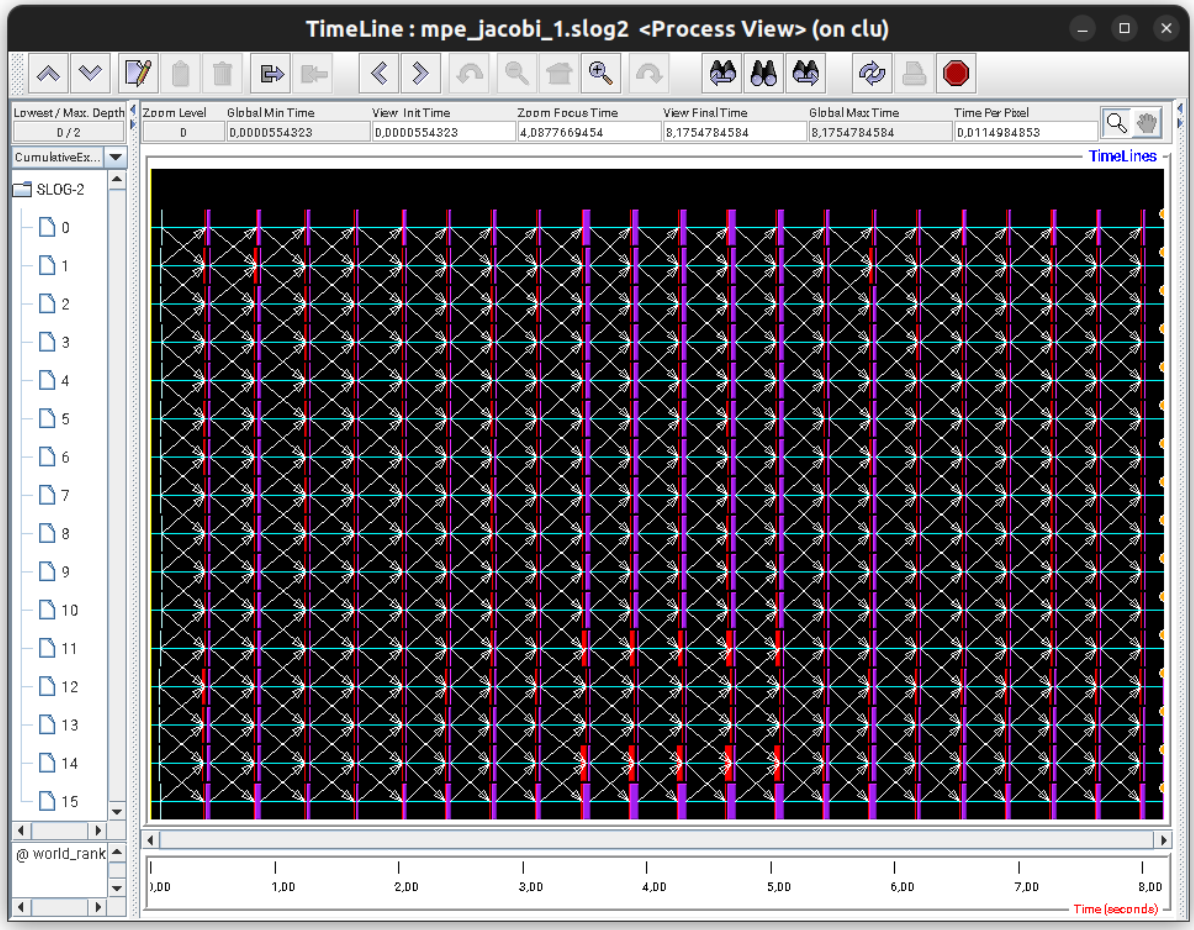


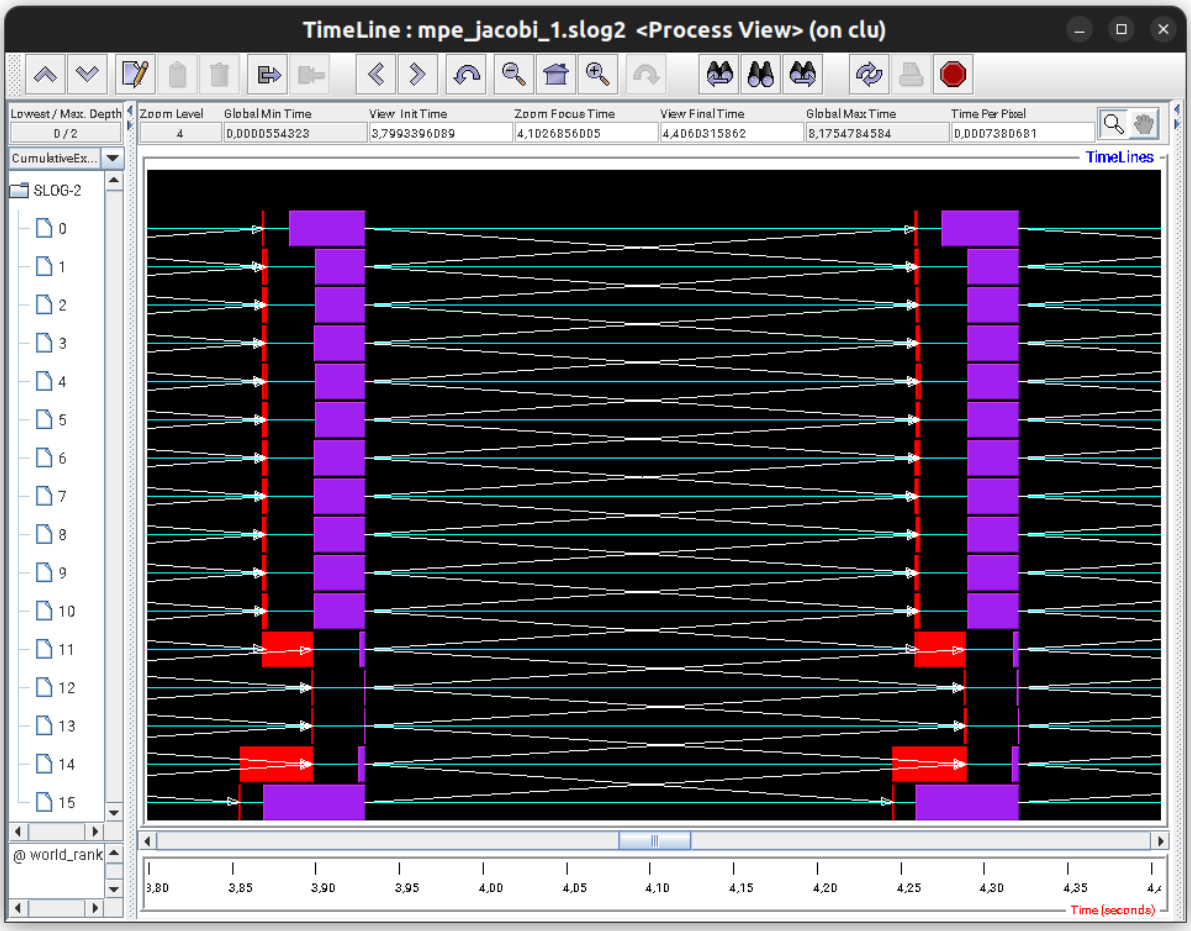

Зависимость ускорения работы программы от количества MPI процессов
Версия 1
Версия 2
Ускорение работы программы
Количество MPI процессов
0
5
10
15
0
2
4
6
8
10
12
14
16
18

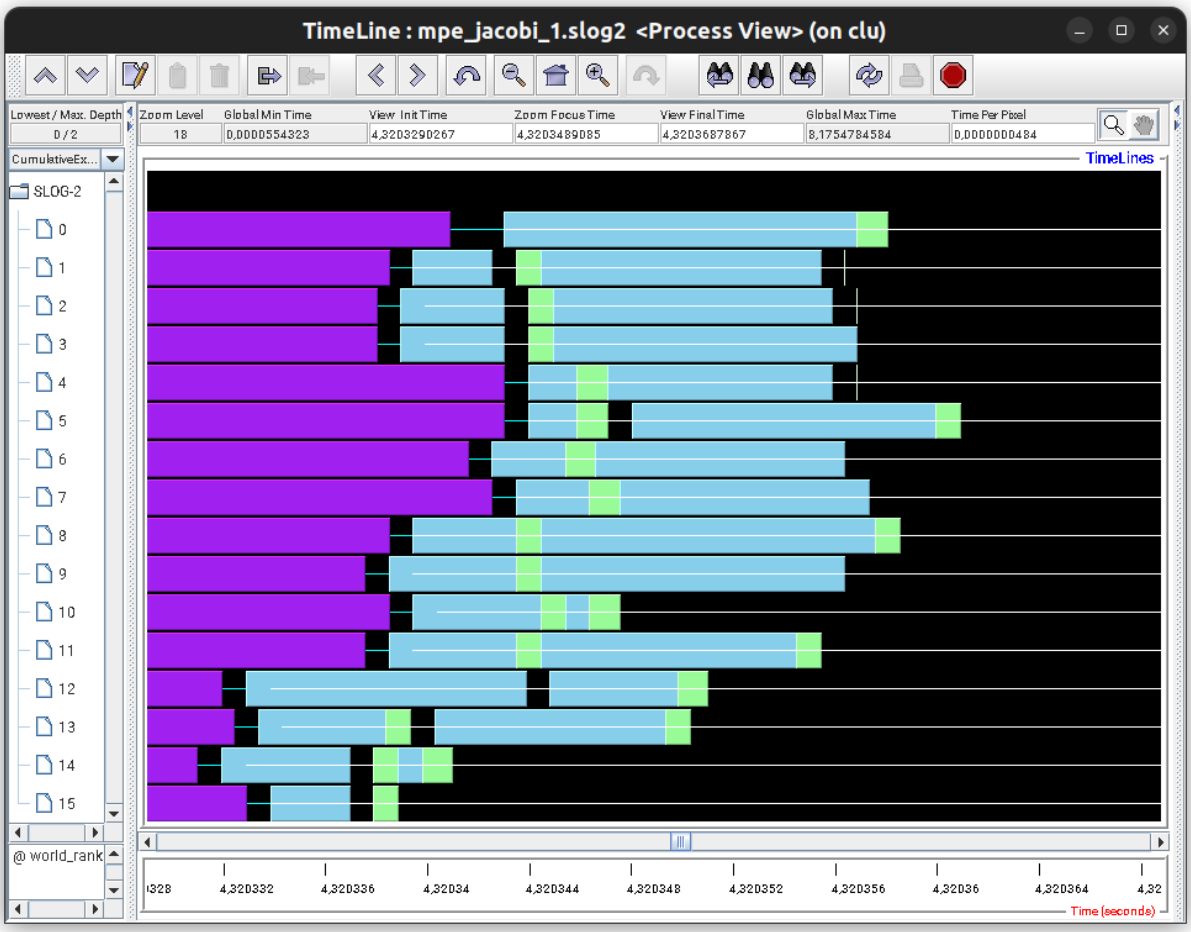

Зависимость эффективности распараллеливания программы от количества MPI процессов
Версия 1
Версия 2
Эффективность распараллеливания программы, %
Количество MPI процессов
80
85
90
95
100
105
110
0
2
4
6
8
10
12
14
16
18

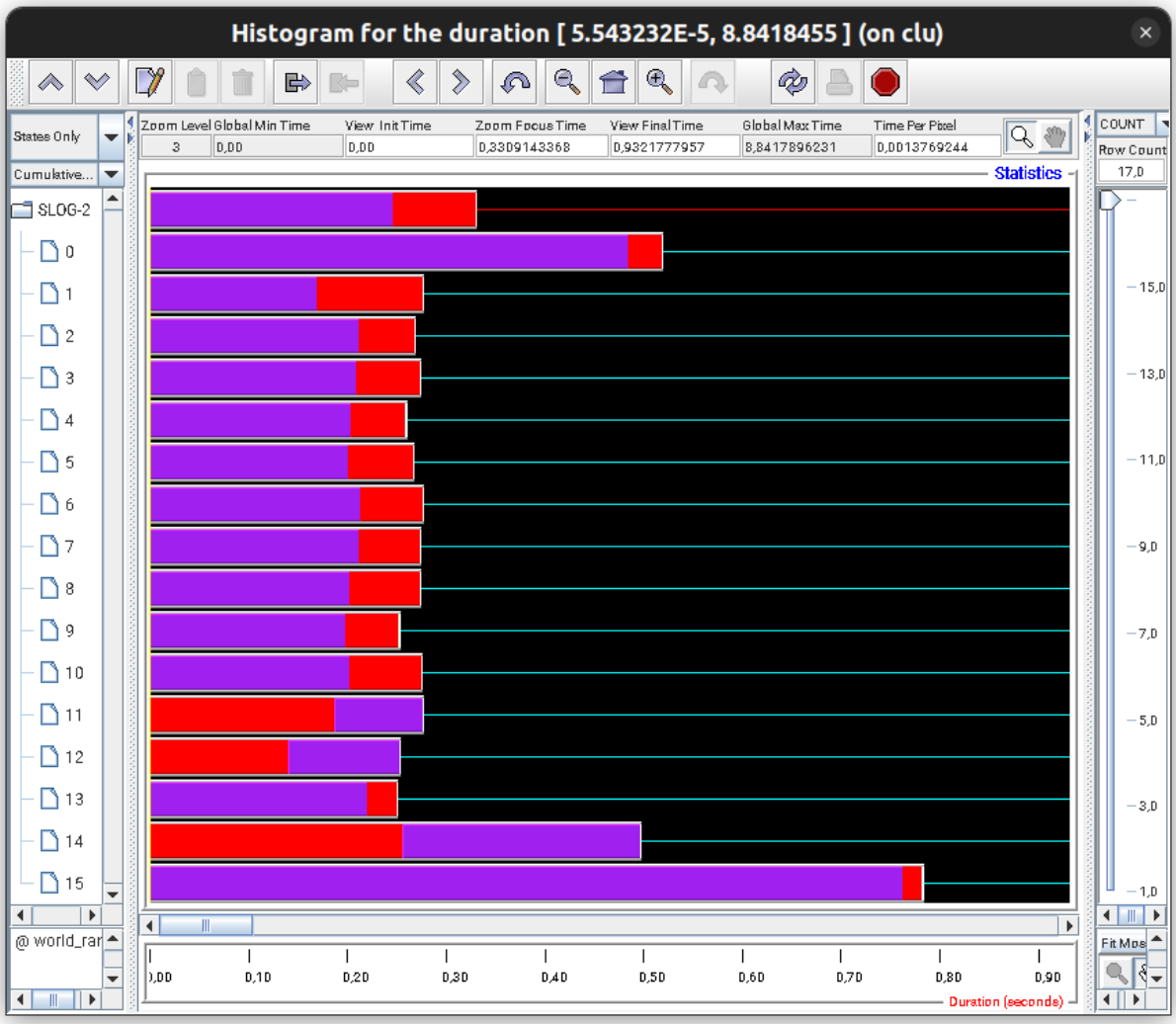
Профилирование 1-ой версии программы



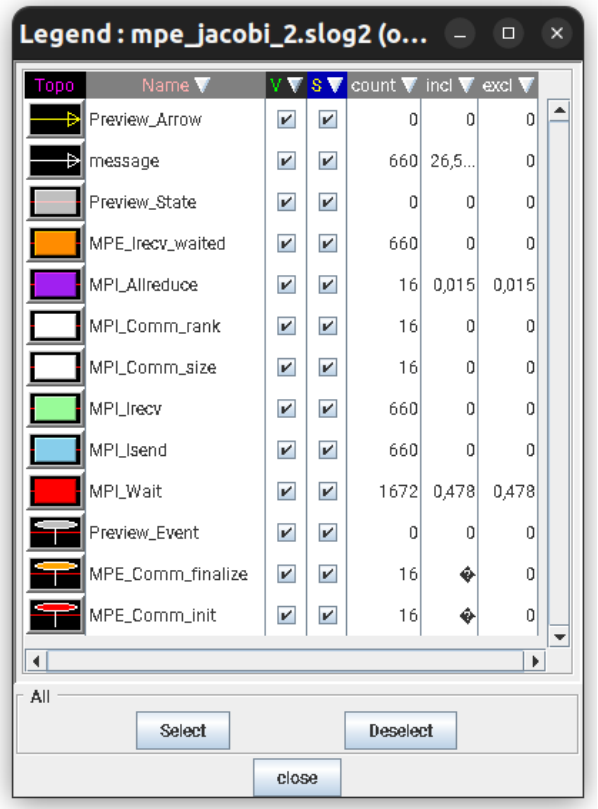


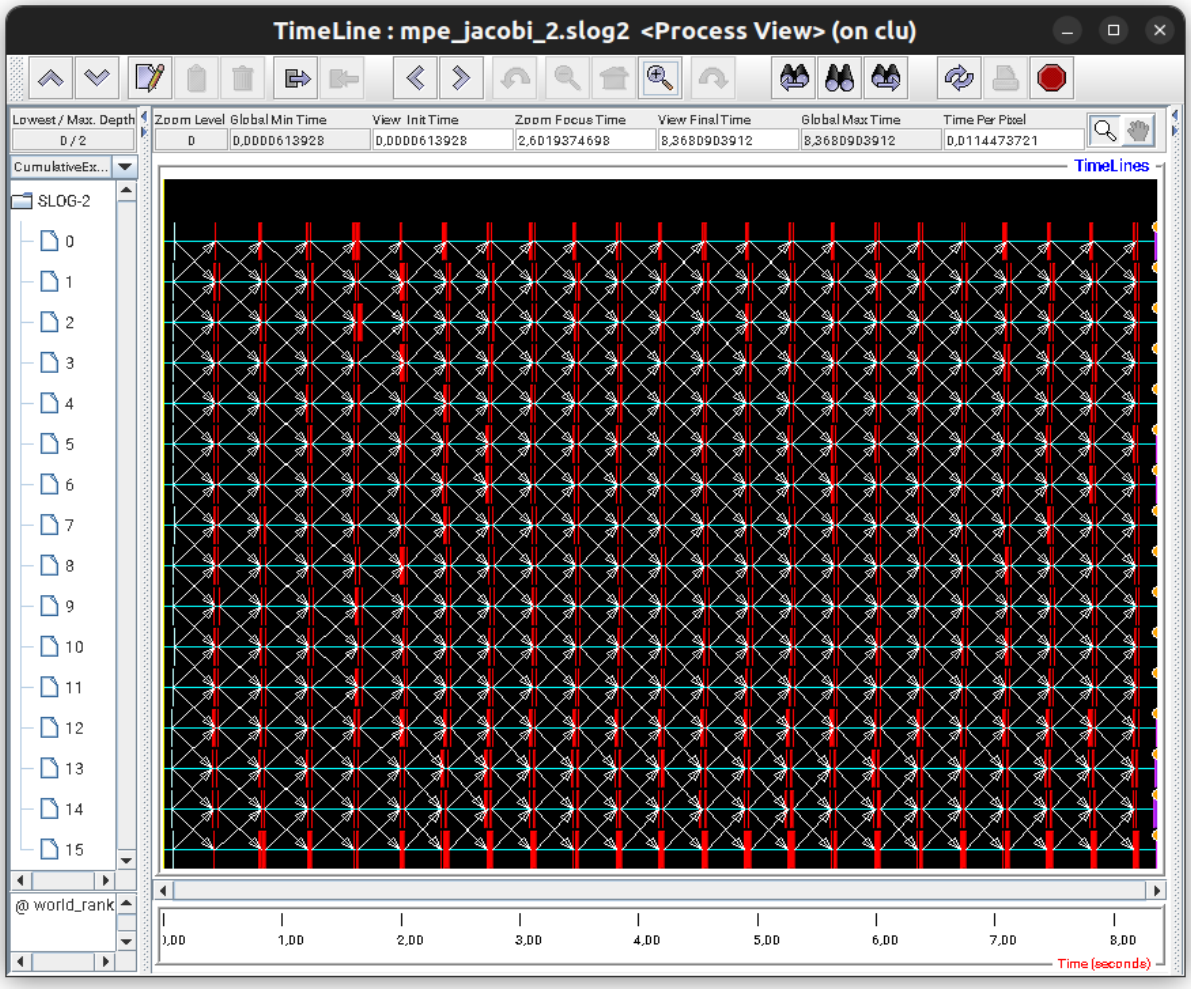

TimeLine : mpe_jacobi_1.slog2 <Process View> (on clu)
Lowest / Max. Depth
0 / 2
Zoom Level
4
Global Min Time
0,0000554323
View Init Time
3,7993396089
Zoom Focus Time
4,1026856005
View Final Time
4,4060315862
Global Max Time
8,1754784584
Time Per Pixel
0,0007380681
CumulativeEx...
SLOG-2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
@ world_rank
TimeLines
3,80
3,85
3,90
3,95
4,00
4,05
4,10
4,15
4,20
4,25
4,30
4,35
Time (seconds)

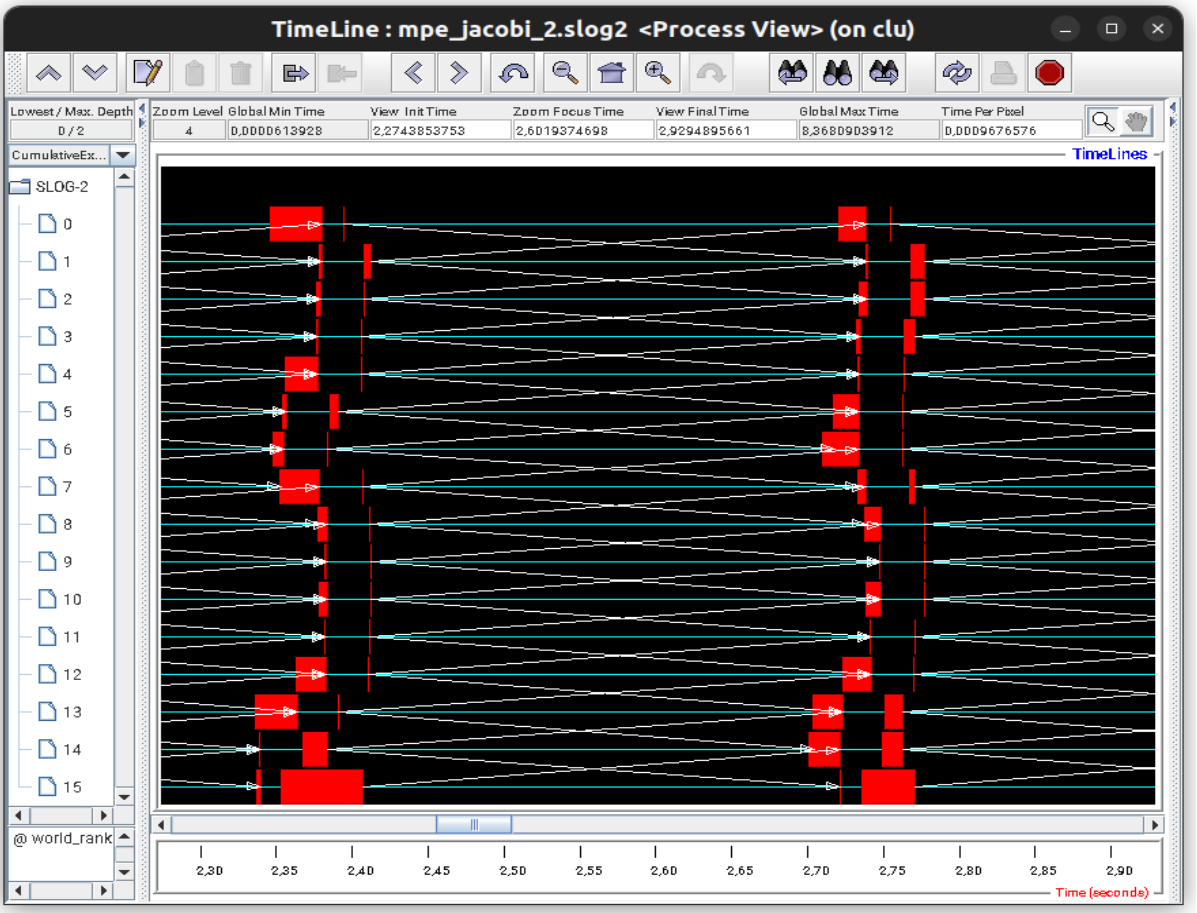

TimeLine : mpe_jacobi_1.slog2 <Process View> (on clu)
Lowest / Max. Depth
0 / 2
Zoom Level
18
Global Min Time
0,0000554323
View Init Time
4,3203290267
Zoom Focus Time
4,3203489085
View Final Time
4,3203687867
Global Max Time
8,1754784584
Time Per Pixel
0,0000000484
CumulativeEx...
SLOG-2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
@ world_rank
TimeLines
4,320328
4,3203332
4,3203336
4,320340
4,3203344
4,3203348
4,3203352
4,3203356
4,3203360
4,3203364
Time (seconds)

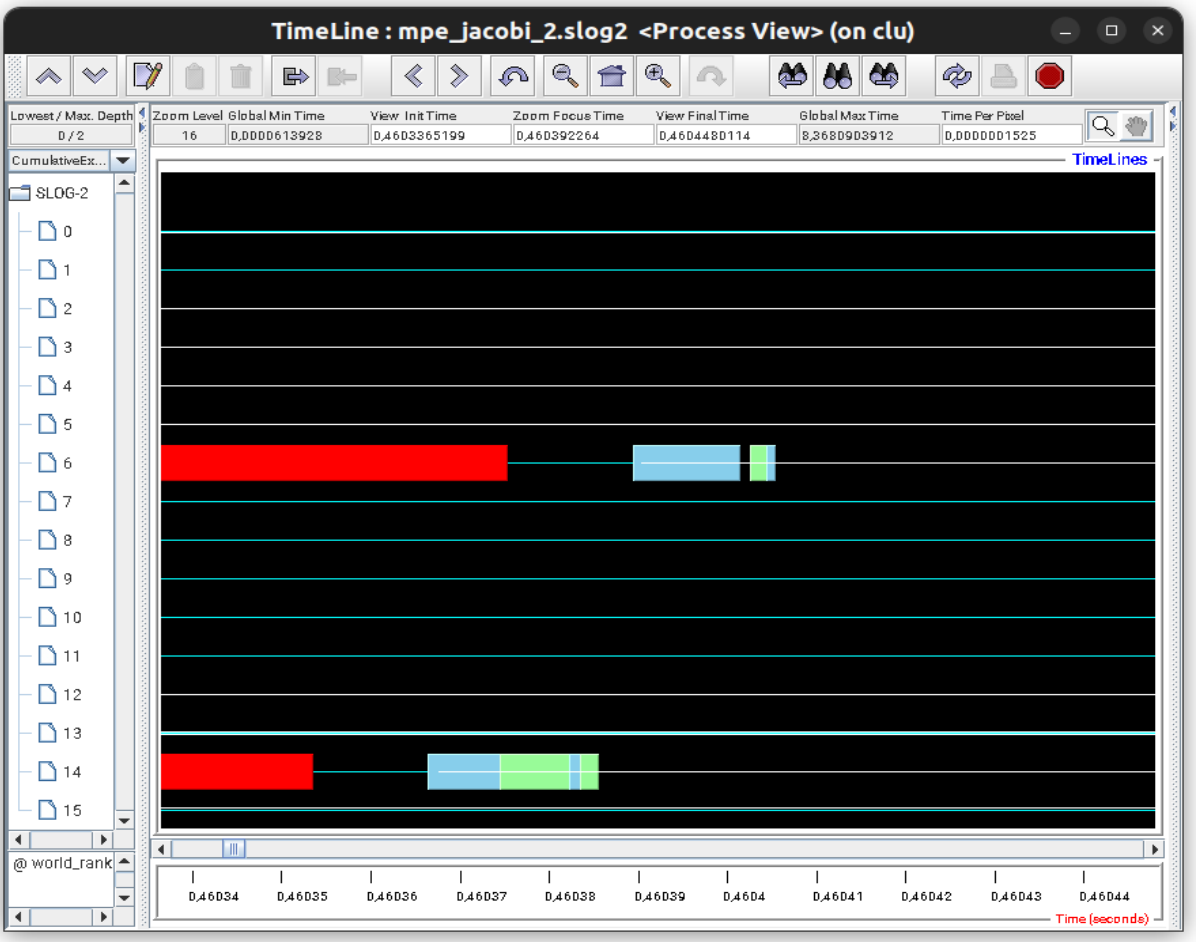

Histogram for the duration [5.543232E-5, 8.8418455] (on clu)
States Only
Cumulative...
SLOG-2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
@ world_rar
Zoom Level
Global Min Time
View Init Time
Zoom Focus Time
View Final Time
Global Max Time
Time Per Pixel
3
0,00
0,00
0,3309143368
0,9321777957
8,8417896231
0,0013769244
Statistics
COUNT
Row Count
17,0
15,0
13,0
11,0
9,0
7,0
5,0
3,0
1,0
Fit Mos
0,00
0,10
0,20
0,30
0,40
0,50
0,60
0,70
0,80
0,90
Duration (seconds)

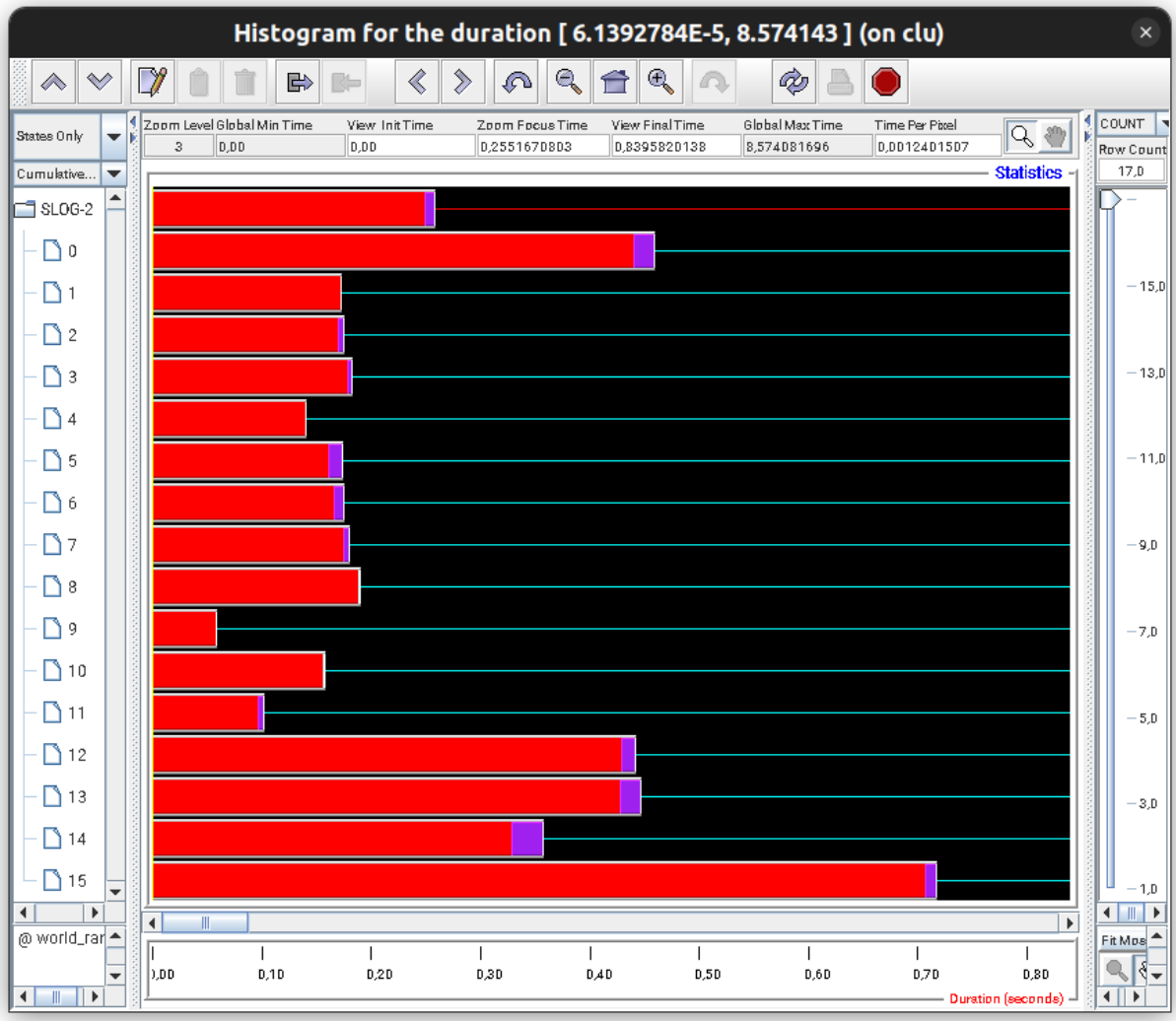
Профилирование 2-ой версии программы






TimeLine : mpe_jacobi_2.slog2 <Process View> (on clu)
Lowest / Max. Depth
0 / 2
Zoom Level
4
Global Min Time
0.0000613928
View Init Time
2.2743853753
Zoom Focus Time
2.6019374698
View Final Time
2.9294895661
Global Max Time
8.3680903912
Time Per Pixel
0.0009676576
CumulativeEx...
SLOG-2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
@ world_rank
TimeLines
2.30
2.35
2.40
2.45
2.50
2.55
2.60
2.65
2.70
2.75
2.80
2.85
2.90
Time (seconds)


TimeLine : mpe_jacobi_2.slog2 <Process View> (on clu)
Lowest / Max. Depth
0 / 2
Zoom Level
16
Global Min Time
0.0000613928
View Init Time
0.46D3365199
Zoom Focus Time
0.46D392264
View Final Time
0.46D448D114
Global Max Time
8.3680903912
Time Per Pixel
0.0000001525
CumulativeEx...
SLOG-2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
@ world_rank
TimeLines
0.46D34
0.46D35
0.46D36
0.46D37
0.46D38
0.46D39
0.46D4
0.46D41
0.46D42
0.46D43
0.46D44
Time (seconds)


Histogram for the duration [6.1392784E-5, 8.574143] (on clu)
States Only
Cumulative...
SLOG-2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
@ world_ran
Zoom Level
3
Global Min Time
0,00
View Init Time
0,00
Zoom Focus Time
0,2551670803
View Final Time
0,8395820138
Global Max Time
8,574081696
Time Per Pixel
0,0012401507
Statistics
COUNT
Row Count
17,0
15,0
13,0
11,0
9,0
7,0
5,0
3,0
1,0
Fit Mode
0,00
0,10
0,20
0,30
0,40
0,50
0,60
0,70
0,80
Duration (seconds)

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы:

- *Освоил методы распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области;*
- *Реализовал 2 версии программы и сравнил их;*

Анализируя результаты исследований, можно сделать следующие выводы:

- *Обмены граничными значениями подобластей выполняются на фоне вычислений внутренних значений функции;*
- *Программы хорошо масштабируется;*
- *Без учета лишней итерации 2-ая версия программы работает быстрее. Связано это с тем, что операция MPI_Allreduce блокирует все процессы и достаточно много времени. Выполняя сбор данных на фоне счёта, уменьшается влияние функции на время программы.*

ПРИЛОЖЕНИЕ (ЛИСТИНГ ПРОГРАММЫ)

mpi_jacobi_1.c

```c
/**
 * @file mpi_jacobi.c
 * @author ptrvsrg (s.petrov1@g.nsu.ru)
 * @brief The solution of equation by the Jacobi method in a 3D domain in the case
 *        of a 1D decomposition of the domain
 * @version 1.0
 */

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

// Initial coordinates
#define X_0 (double)-1.0
#define Y_0 (double)-1.0
#define Z_0 (double)-1.0

// Dimension size
#define D_X (double)2.0
#define D_Y (double)2.0
#define D_Z (double)2.0

// Grid size
#define N_X 400
#define N_Y 400
#define N_Z 400

// Step size
#define H_X (D_X / (N_X - 1))
#define H_Y (D_Y / (N_Y - 1))
#define H_Z (D_Z / (N_Z - 1))

// Square of step size
#define H_X_2 (H_X * H_X)
#define H_Y_2 (H_Y * H_Y)
#define H_Z_2 (H_Z * H_Z)

// Parameters
#define A (double)1.0E5
#define EPSILON (double)1.0E-3

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);
double get_x(int i);
double get_y(int j);
double 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 swap_func(double **prev_func, double **curr_func);
```

```c
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,
                   const double *up_border_layer, const 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;
    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 *prev_func = NULL;
    double *curr_func = NULL;
    MPI_Request send_up_req;
    MPI_Request send_down_req;
    MPI_Request recv_up_req;
    MPI_Request recv_down_req;

    // Check grid size
    if (N_X < 3 || N_Y < 3 || N_Z < 3) {
        fprintf(stderr, "Incorrect grid size\n");
        return EXIT_FAILURE;
    }

    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
    prev_func = malloc(sizeof(double) * layer_heights[proc_rank] * N_Y * N_Z);
    curr_func = malloc(sizeof(double) * layer_heights[proc_rank] * N_Y * N_Z);
    init_layers(prev_func, 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;

        // Swap functions
        swap_func(&prev_func, &curr_func);

        // Start sending and receiving border
        if (proc_rank != 0) {
```

```
            double *prev_up_border = prev_func;
            MPI_Isend(prev_up_border, N_Y * N_Z, MPI_DOUBLE, proc_rank - 1,
                      proc_rank, MPI_COMM_WORLD, &send_up_req);
            MPI_Irecv(up_border_layer, N_Y * N_Z, MPI_DOUBLE, proc_rank - 1,
                      proc_rank - 1, MPI_COMM_WORLD, &recv_up_req);
        }

        if (proc_rank != proc_count - 1) {
            double *prev_down_border =
                prev_func + (layer_heights[proc_rank] - 1) * N_Y * N_Z;
            MPI_Isend(prev_down_border, N_Y * N_Z, MPI_DOUBLE, proc_rank + 1,
                      proc_rank, MPI_COMM_WORLD, &send_down_req);
            MPI_Irecv(down_border_layer, N_Y * N_Z, MPI_DOUBLE, proc_rank + 1,
                      proc_rank + 1, MPI_COMM_WORLD, &recv_down_req);
        }

        // Calculate center
        tmp_max_diff = calc_center(prev_func, curr_func, layer_heights[proc_rank],
                                   offsets[proc_rank]);

        // Finish sending and receiving border
        if (proc_rank != 0) {
            MPI_Wait(&send_up_req, MPI_STATUS_IGNORE);
            MPI_Wait(&recv_up_req, MPI_STATUS_IGNORE);
        }

        if (proc_rank != proc_count - 1) {
            MPI_Wait(&send_down_req, MPI_STATUS_IGNORE);
            MPI_Wait(&recv_down_req, MPI_STATUS_IGNORE);
        }

        // Calculate border
        proc_max_diff = calc_border(prev_func, 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(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(prev_func);
    free(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;
}

/**
 * @brief Gets array index by coordinates of a point in a 3D area
 *
 * @param x X-axis coordinate
 * @param y Y-axis coordinate
 * @param z Z-axis coordinate
 * @return Array index
 */
int get_index(int x, int y, int z) {
    return x * N_Y * N_Z + y * N_Z + z;
}

/**
 * @brief Gets coordinate of a point by node index for X axis
 *
 * @param i Node index
 * @return X-axis coordinate
 */
double get_x(int i) {
    return X_0 + i * H_X;
}

/**
 * @brief Gets coordinate of a point by node index for Y axis
 *
 * @param j Node index
 * @return Y-axis coordinate
 */
double get_y(int j) {
    return Y_0 + j * H_Y;
}

/**
 * @brief Gets coordinate of a point by node index for Z axis
 *
 * @param k Node index
 * @return Z-axis coordinate
 */
double get_z(int k) {
    return Z_0 + k * H_Z;
}

/**
 * @brief Divides area into layers
```

```
 *
 * @param layer_heights Address of array of layer heights
 * @param offsets Address of array of layer offsets
 * @param proc_count Count of processes
 */
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];
    }
}

/**
 * @brief Initializes the layers
 *
 * @param prev_func Address of array of values of the previous calculated function
 * @param curr_func Address of array of values of the current calculated function
 * @param layer_height Height of the layer
 * @param offset Offset of the layer
 */
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++) {
                bool isBorder = (offset + i == 0) || (j == 0) || (k == 0) ||
                                (offset + i == N_X - 1) || (j == N_Y - 1) ||
                                (k == N_Z - 1);
                if (isBorder) {
                    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;
                }
            }
}

/**
 * @brief Swaps the address of the array of values of the previous calculated
 *        function and the address of the array of values of the current calculated
 *        function
 *
 * @param prev_func Address of address of array of values of the previous calculated
 *                  function
 * @param curr_func Address of address of array of values of the current calculated
 *                  function
 */
void swap_func(double **prev_func, double **curr_func) {
    double *tmp = *prev_func;
```

```
    *prev_func = *curr_func;
    *curr_func = tmp;
}

/**
 * @brief Calculate function values in internal nodes
 *
 * @param prev_func Address of array of values of the previous calculated function
 * @param curr_func Address of array of values of the current calculated function
 * @param layer_height Height of the layer
 * @param offset Offset of the layer
 * @return Maximum differences of the previous and current calculated functions
 */
double calc_center(const double *prev_func, double *curr_func, int layer_height,
                   int offset) {
    double f_i = 0.0;
    double f_j = 0.0;
    double f_k = 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) {
                f_i = (prev_func[get_index(i + 1, j, k)] +
                       prev_func[get_index(i - 1, j, k)]) / H_X_2;
                f_j = (prev_func[get_index(i, j + 1, k)] +
                       prev_func[get_index(i, j - 1, k)]) / H_Y_2;
                f_k = (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)] =
                    (f_i + f_j + f_k - 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;
}

/**
 * @brief
 *
 * @param prev_func Address of array of values of the previous calculated function
 * @param curr_func Address of array of values of the current calculated function
 * @param up_border_layer Address of array of values of the upper border
 * @param down_border_layer Address of array of values of the lower border
 * @param layer_height Height of the layer
 * @param offset Offset of the layer
 * @param proc_rank Rank of process
 * @param proc_count Count of processes
 * @return Maximum differences of the previous and current calculated functions
 */
double calc_border(const double *prev_func, double *curr_func,
```

```
                    const double *up_border_layer, const double *down_border_layer,
                    int layer_height, int offset, int proc_rank, int proc_count) {
    double f_i = 0.0;
    double f_j = 0.0;
    double f_k = 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) {
                f_i = (prev_func[get_index(1, j, k)] +
                       up_border_layer[get_index(0, j, k)]) / H_X_2;
                f_j = (prev_func[get_index(0, j + 1, k)] +
                       prev_func[get_index(0, j - 1, k)]) / H_Y_2;
                f_k = (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)] =
                    (f_i + f_j + f_k - 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) {
                f_i = (prev_func[get_index(layer_height - 2, j, k)] +
                       down_border_layer[get_index(0, j, k)]) / H_X_2;
                f_j = (prev_func[get_index(layer_height - 1, j + 1, k)] +
                       prev_func[get_index(layer_height - 1, j - 1, k)]) / H_Y_2;
                f_k = (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)] =
                    (f_i + f_j + f_k -
                    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;
}

/**
 * @brief Calculate the maximum differences of the calculated and theoretical
functions
 *
```

```
 * @param curr_func Address of array of values of the current calculated function
 * @param layer_height Height of the layer
 * @param offset Offset of the layer
 * @return Maximum differences of the calculated and theoretical functions
 */
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;
}
```

mpi_jacobi_2.c

```
/**
 * @file mpi_jacobi.c
 * @author ptrvsrg (s.petrov1@g.nsu.ru)
 * @brief The solution of equation by the Jacobi method in a 3D domain in the case
 *        of a 1D decomposition of the domain
 * @version 2.0
 */

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

// Initial coordinates
#define X_0 (double)-1.0
#define Y_0 (double)-1.0
#define Z_0 (double)-1.0

// Dimension size
#define D_X (double)2.0
#define D_Y (double)2.0
#define D_Z (double)2.0

// Grid size
#define N_X 400
#define N_Y 400
#define N_Z 400

// Step size
```

```
#define H_X (D_X / (N_X - 1))
#define H_Y (D_Y / (N_Y - 1))
#define H_Z (D_Z / (N_Z - 1))

// Square of step size
#define H_X_2 (H_X * H_X)
#define H_Y_2 (H_Y * H_Y)
#define H_Z_2 (H_Z * H_Z)

// Parameters
#define A (double)1.0E5
#define EPSILON (double)1.0E-3

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);
double get_x(int i);
double get_y(int j);
double 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 swap_func(double **prev_func, double **curr_func);
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;
    double start_time = 0.0;
    double finish_time = 0.0;
    double prev_proc_max_diff = EPSILON;
    double max_diff = 0.0;
    int *layer_heights = NULL;
    int *offsets = NULL;
    double *up_border_layer = NULL;
    double *down_border_layer = NULL;
    double *prev_func = NULL;
    double *curr_func = NULL;
    MPI_Request send_up_req;
    MPI_Request send_down_req;
    MPI_Request recv_up_req;
    MPI_Request recv_down_req;
    MPI_Request reduce_max_diff_req;

    // Check grid size
    if (N_X < 3 || N_Y < 3 || N_Z < 3) {
        fprintf(stderr, "Incorrect grid size\n");
        return EXIT_FAILURE;
    }

    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
    prev_func = malloc(sizeof(double) * layer_heights[proc_rank] * N_Y * N_Z);
    curr_func = malloc(sizeof(double) * layer_heights[proc_rank] * N_Y * N_Z);
    init_layers(prev_func, 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_1 = 0.0;
        double tmp_max_diff_2 = 0.0;

        // Start calculating the differences of the previous and current calculated
        // functions for previous iterations
        MPI_Iallreduce(&prev_proc_max_diff, &max_diff, 1, MPI_DOUBLE, MPI_MAX,
                       MPI_COMM_WORLD, &reduce_max_diff_req);

        // Swap functions
        swap_func(&prev_func, &curr_func);

        // Start sending and receiving border
        if (proc_rank != 0) {
            double *prev_up_border = prev_func;
            MPI_Isend(prev_up_border, N_Y * N_Z, MPI_DOUBLE, proc_rank - 1,
                      proc_rank, MPI_COMM_WORLD, &send_up_req);
            MPI_Irecv(up_border_layer, N_Y * N_Z, MPI_DOUBLE, proc_rank - 1,
                      proc_rank - 1, MPI_COMM_WORLD, &recv_up_req);
        }

        if (proc_rank != proc_count - 1) {
            double *prev_down_border =
                prev_func + (layer_heights[proc_rank] - 1) * N_Y * N_Z;
            MPI_Isend(prev_down_border, N_Y * N_Z, MPI_DOUBLE, proc_rank + 1,
                      proc_rank, MPI_COMM_WORLD, &send_down_req);
            MPI_Irecv(down_border_layer, N_Y * N_Z, MPI_DOUBLE, proc_rank + 1,
                      proc_rank + 1, MPI_COMM_WORLD, &recv_down_req);
        }

        // Calculate center
        tmp_max_diff_1 = calc_center(prev_func, curr_func, layer_heights[proc_rank],
                                     offsets[proc_rank]);

        // Finish sending and receiving border
        if (proc_rank != 0) {
            MPI_Wait(&send_up_req, MPI_STATUS_IGNORE);
            MPI_Wait(&recv_up_req, MPI_STATUS_IGNORE);
        }

        if (proc_rank != proc_count - 1) {
            MPI_Wait(&send_down_req, MPI_STATUS_IGNORE);
```

```
            MPI_Wait(&recv_down_req, MPI_STATUS_IGNORE);
        }

        // Calculate border
        tmp_max_diff_2 = calc_border(prev_func, curr_func, up_border_layer,
                                     down_border_layer, layer_heights[proc_rank],
                                     offsets[proc_rank], proc_rank, proc_count);

        // Start calculating the differences of the previous and current calculated
        // functions for previous iterations
        MPI_Wait(&reduce_max_diff_req, MPI_STATUS_IGNORE);

        prev_proc_max_diff = fmax(tmp_max_diff_1, tmp_max_diff_2);
    } while (max_diff >= EPSILON);

    // Swap functions as extra iteration is made
    swap_func(&prev_func, &curr_func);

    // Calculate the differences of the calculated and theoretical functions
    max_diff = calc_max_diff(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(prev_func);
    free(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;
}

/**
 * @brief Gets array index by coordinates of a point in a 3D area
 *
 * @param x X-axis coordinate
 * @param y Y-axis coordinate
 * @param z Z-axis coordinate
 * @return Array index
 */
int get_index(int x, int y, int z) {
    return x * N_Y * N_Z + y * N_Z + z;
```

```
}

/**
 * @brief Gets coordinate of a point by node index for X axis
 *
 * @param i Node index
 * @return X-axis coordinate
 */
double get_x(int i) {
    return X_0 + i * H_X;
}

/**
 * @brief Gets coordinate of a point by node index for Y axis
 *
 * @param j Node index
 * @return Y-axis coordinate
 */
double get_y(int j) {
    return Y_0 + j * H_Y;
}

/**
 * @brief Gets coordinate of a point by node index for Z axis
 *
 * @param k Node index
 * @return Z-axis coordinate
 */
double get_z(int k) {
    return Z_0 + k * H_Z;
}

/**
 * @brief Divides area into layers
 *
 * @param layer_heights Address of array of layer heights
 * @param offsets Address of array of layer offsets
 * @param proc_count Count of processes
 */
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];
    }
}

/**
 * @brief Initializes the layers
 *
 * @param prev_func Address of array of values of the previous calculated function
 * @param curr_func Address of array of values of the current calculated function
 * @param layer_height Height of the layer
```

```
 * @param offset Offset of the layer
 */
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++) {
                bool isBorder = (offset + i == 0) || (j == 0) || (k == 0) ||
                                (offset + i == N_X - 1) || (j == N_Y - 1) ||
                                (k == N_Z - 1);
                if (isBorder) {
                    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;
                }
            }
}

/**
 * @brief Swaps the address of the array of values of the previous calculated
 *        function and the address of the array of values of the current calculated
 *        function
 *
 * @param prev_func Address of address of array of values of the previous calculated
 *                  function
 * @param curr_func Address of address of array of values of the current calculated
 *                  function
 */
void swap_func(double **prev_func, double **curr_func) {
    double *tmp = *prev_func;
    *prev_func = *curr_func;
    *curr_func = tmp;
}

/**
 * @brief Calculate function values in internal nodes
 *
 * @param prev_func Address of array of values of the previous calculated function
 * @param curr_func Address of array of values of the current calculated function
 * @param layer_height Height of the layer
 * @param offset Offset of the layer
 * @return Maximum differences of the previous and current calculated functions
 */
double calc_center(const double *prev_func, double *curr_func, int layer_height,
                   int offset) {
    double f_i = 0.0;
    double f_j = 0.0;
    double f_k = 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) {
                f_i = (prev_func[get_index(i + 1, j, k)] +
```

```
                    prev_func[get_index(i - 1, j, k)]) / H_X_2;
                f_j = (prev_func[get_index(i, j + 1, k)] +
                    prev_func[get_index(i, j - 1, k)]) / H_Y_2;
                f_k = (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)] =
                    (f_i + f_j + f_k - 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;
}

/**
 * @brief
 *
 * @param prev_func Address of array of values of the previous calculated function
 * @param curr_func Address of array of values of the current calculated function
 * @param up_border_layer Address of array of values of the upper border
 * @param down_border_layer Address of array of values of the lower border
 * @param layer_height Height of the layer
 * @param offset Offset of the layer
 * @param proc_rank Rank of process
 * @param proc_count Count of processes
 * @return Maximum differences of the previous and current calculated functions
 */
double calc_border(const double *prev_func, double *curr_func,
                   const double *up_border_layer, const double *down_border_layer,
                   int layer_height, int offset, int proc_rank, int proc_count) {
    double f_i = 0.0;
    double f_j = 0.0;
    double f_k = 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) {
                f_i = (prev_func[get_index(1, j, k)] +
                    up_border_layer[get_index(0, j, k)]) / H_X_2;
                f_j = (prev_func[get_index(0, j + 1, k)] +
                    prev_func[get_index(0, j - 1, k)]) / H_Y_2;
                f_k = (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)] =
                    (f_i + f_j + f_k - 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) {
                f_i = (prev_func[get_index(layer_height - 2, j, k)] +
                       down_border_layer[get_index(0, j, k)]) / H_X_2;
                f_j = (prev_func[get_index(layer_height - 1, j + 1, k)] +
                       prev_func[get_index(layer_height - 1, j - 1, k)]) / H_Y_2;
                f_k = (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)] =
                    (f_i + f_j + f_k -
                    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;
}

/**
 * @brief Calculate the maximum differences of the calculated and theoretical
functions
 *
 * @param curr_func Address of array of values of the current calculated function
 * @param layer_height Height of the layer
 * @param offset Offset of the layer
 * @return Maximum differences of the calculated and theoretical functions
 */
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>
```