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, группы 21209

Панас Матвея Алексеевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Мичуров М.А.

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:* $\varphi^0_{i,j,k} = 0$.

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);*

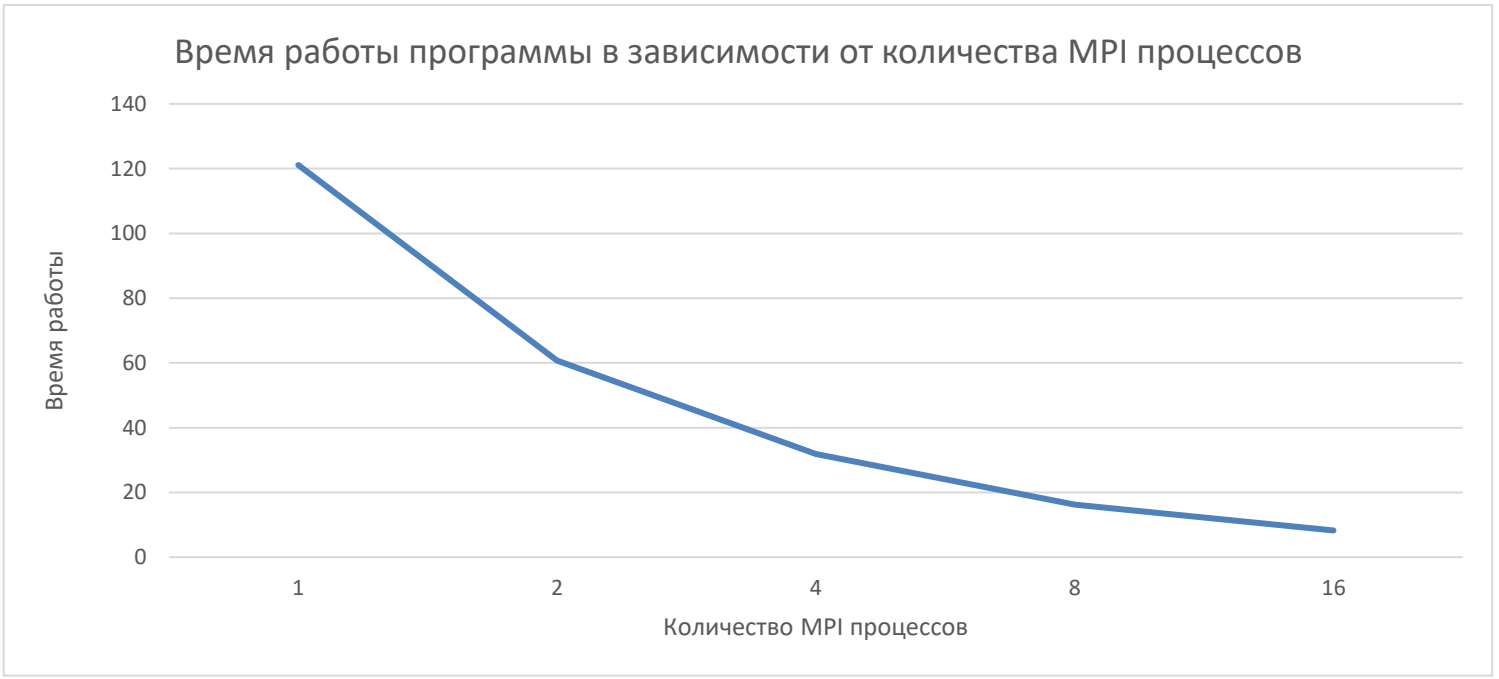
3. *Запустил программы при использовании 1, 2, 4, 8, 16 ядер со следующими параметрами:* $\varepsilon = 10^{-3}, N_x = 400, N_y = 400, N_z = 400$*;*

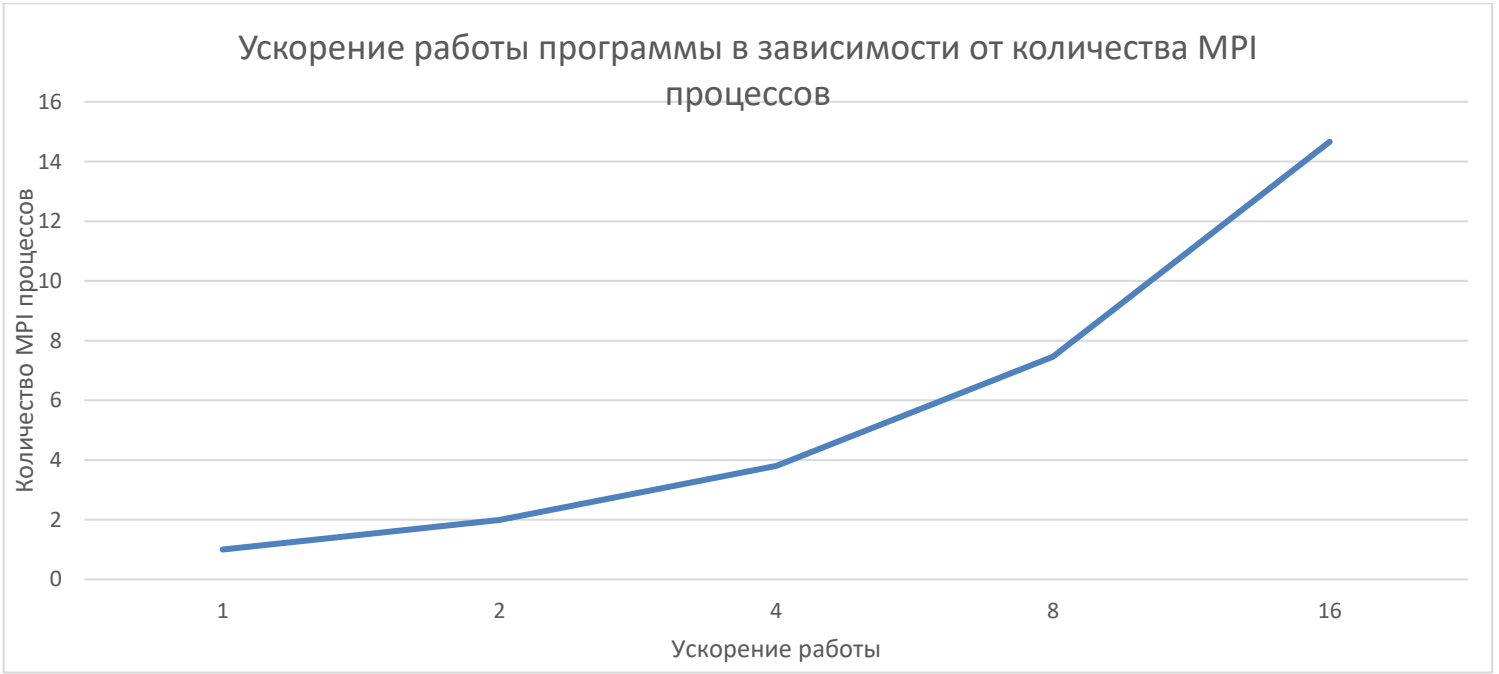
4. *Составил графики зависимости времени работы, ускорения работы и эффективности распараллеливания программы от количества MPI процессов. Также добавил график зависимости времени работы 2-ой версии программы без учета лишней итерации от количества MPI процессов;*

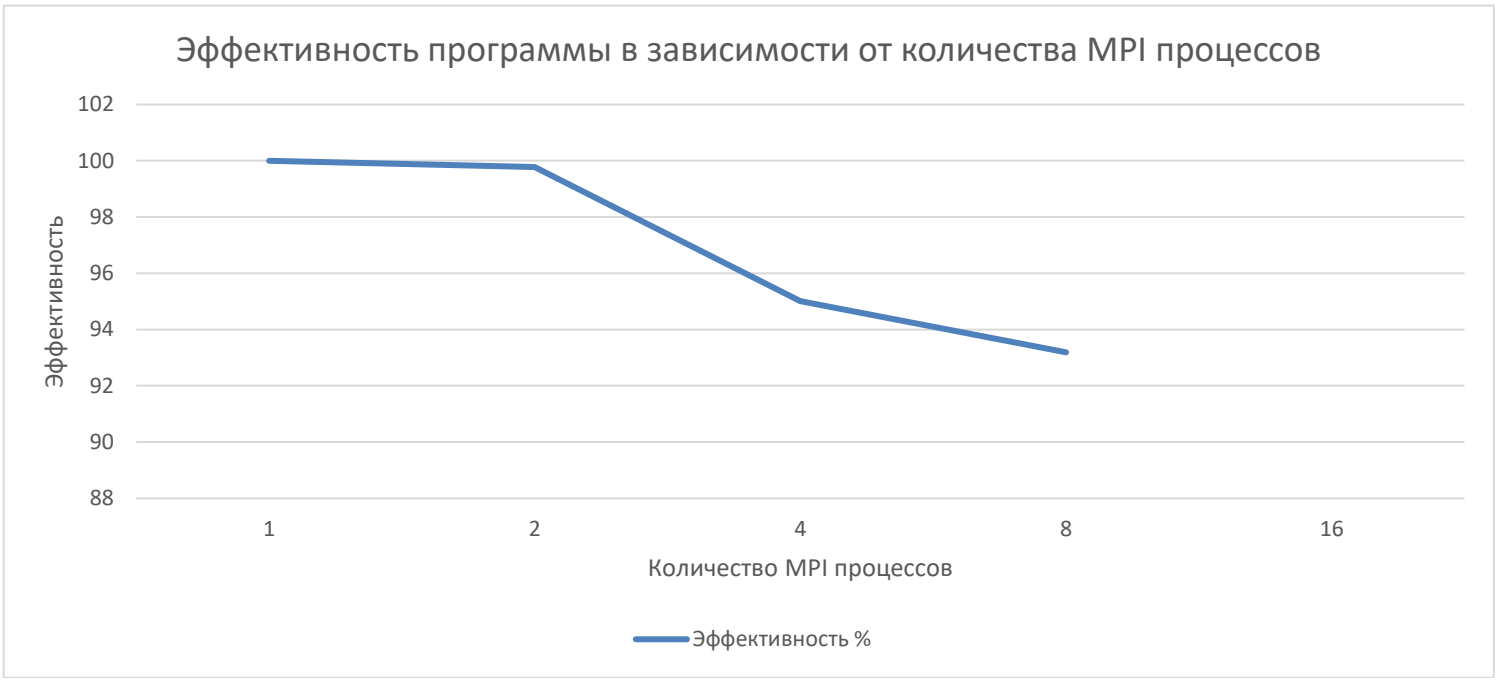
5. *Выполнил профилирование программ с помощью MPE при использовании 16 ядер;*

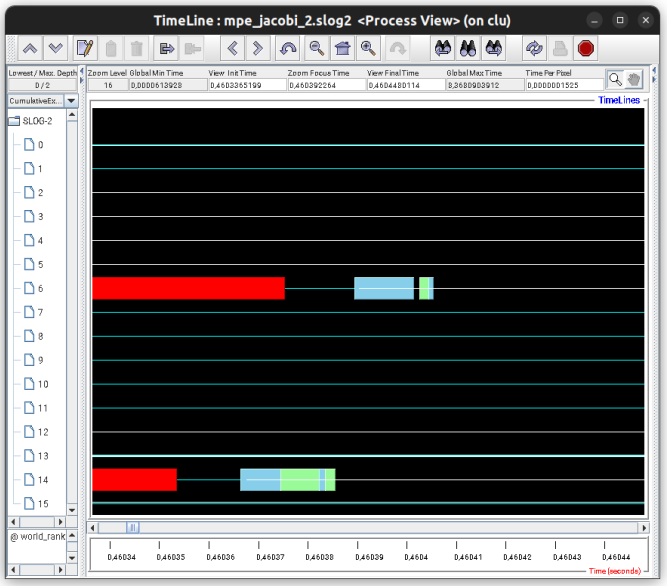
Результаты измерения

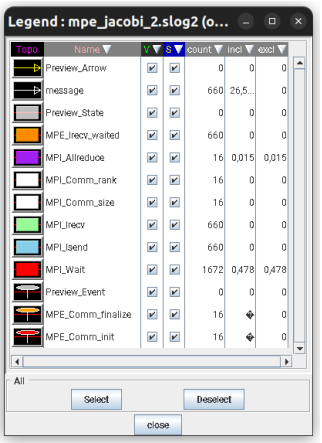
Количство MPI процессов	Время работы	Ускорение работы	Эффективность рапараллеливания
1	121.106	1	100
2	60.688	1.988	99.777
4	31.867	3.800	95.008
8	16.244	7.455	93.191
16	8.259	14.662	91.638

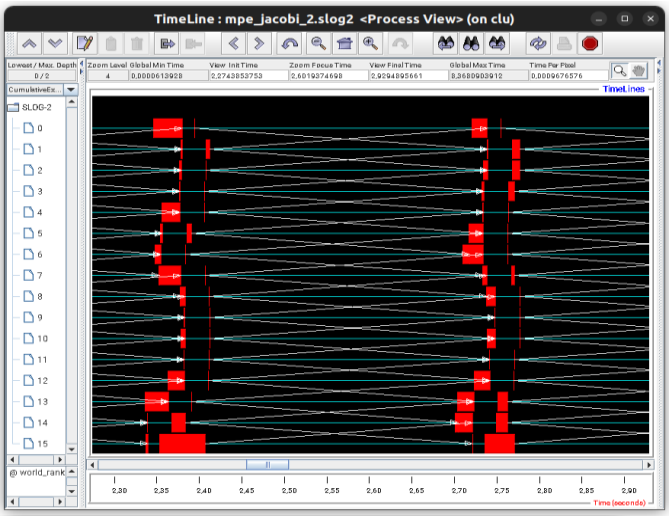


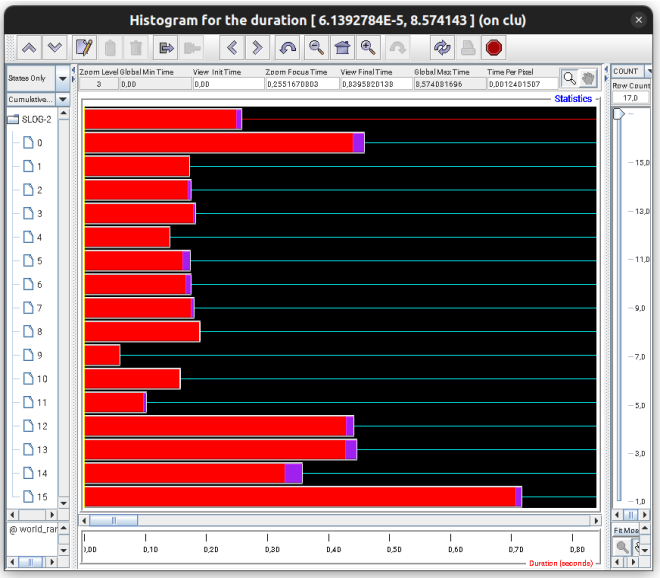


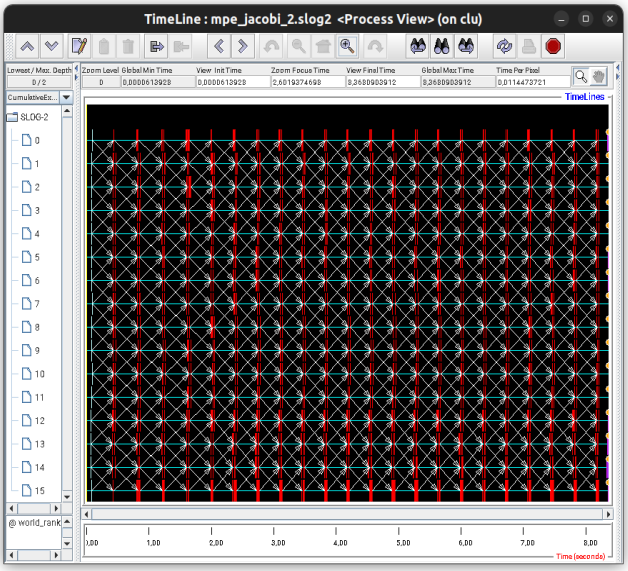

Эффективность программы в зависимости от количества MPI процессов
102
100
98
96
94
92
90
88
Эффективность
1
2
4
8
16
Количество MPI процессов
Эффективность %


TimeLine : mpe_jacobi_2.log2 <Process View> (on clu)
TimeLines
SLOG-2
@ world_rank
Time (seconds)


Legend : mpe_jacobi_2.log2 (o...
Topo
Name
count
incl
excl
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_2.log2 <Process View> (on clu)
TimeLines
SLOG-2
@ world_rank
Time (seconds)


Histogram for the duration [6.1392784E-5, 8.574143] (on clu)
Statistics
SLOG-2
Duration (seconds)


TimeLine : mpe_jacobi_2.log2 <Process View> (on clu)
TimeLines
SLOG-2
@ world_rank
Time (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:

Обмены граничными значениями подобластей выполняются на фоне вычислений внутренних значений функции;
Программа хорошо масштабируется

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

#define X_0 (double)-1.0
#define Y_0 (double)-1.0
#define Z_0 (double)-1.0

#define D_X (double)2.0
#define D_Y (double)2.0
#define D_Z (double)2.0

#define N_X 400
#define N_Y 400
#define N_Z 400

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

#define H_X_2 (H_X * H_X)
#define H_Y_2 (H_Y * H_Y)
#define H_Z_2 (H_Z * H_Z)

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

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

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

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

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

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

double 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;

    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++) {
        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(x: i, y: j, z: k)] = phi(get_x(offset + i), get_y(j), get_z(k));
          curr_func[get_index(x: i, y: j, z: k)] = phi(get_x(offset + i), get_y(j), get_z(k));
        } else {
          prev_func[get_index(x: i, y: j, z: k)] = 0;
          curr_func[get_index(x: i, y: j, z: k)] = 0;
        }
      }
  }

  void swap_func(double **prev_func, double **curr_func) {
    double *tmp = *prev_func;
    *prev_func = *curr_func;
    *curr_func = tmp;
  }

  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(x: i + 1, y: j, z: k)] + prev_func[get_index(x: i - 1, y: j, z: k)]) / H_X_2;
          f_j = (prev_func[get_index(x: i, y: j + 1, z: k)] + prev_func[get_index(x: i, y: j - 1, z: k)]) / H_Y_2;
          f_k = (prev_func[get_index(x: i, y: j, z: k + 1)] + prev_func[get_index(x: i, y: j, z: k - 1)]) / H_Z_2;

          curr_func[get_index(x: i, y: j, z: 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);

          tmp_max_diff = fabs(x: curr_func[get_index(x: i, y: j, z: k)] - prev_func[get_index(x: i, y: j, z: 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 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) {
        if (proc_rank != 0) {
          f_i = (prev_func[get_index(x: 1, y: j, z: k)] + up_border_layer[get_index(x: 0, y: j, z: k)]) / H_X_2;
          f_j = (prev_func[get_index(x: 0, y: j + 1, z: k)] + prev_func[get_index(x: 0, y: j - 1, z: k)]) / H_Y_2;
```

```
        f_j = (prev_func[get_index(x: 0, y: j + 1, z: k)] + prev_func[get_index(x: 0, y: j - 1, z: k)]) / H_Y_2;
        f_k = (prev_func[get_index(x: 0, y: j, z: k + 1)] + prev_func[get_index(x: 0, y: j, z: k - 1)]) / H_Z_2;

        curr_func[get_index(x: 0, y: j, z: k)] = (f_i + f_j + f_k - rho(get_x(i: offset), get_y(j), get_z(k))) / (2 / H_X_2 + 2 / H_Y_2 + 2 / H_Z_2 + A);
        tmp_max_diff = fabs(x: curr_func[get_index(x: 0, y: j, z: k)] - prev_func[get_index(x: 0, y: j, z: k)]);
        if (tmp_max_diff > max_diff) max_diff = tmp_max_diff;
      }

      if (proc_rank != proc_count - 1) {
        f_i = (prev_func[get_index(x: layer_height - 2, y: j, z: k)] + down_border_layer[get_index(x: 0, y: j, z: k)]) / H_X_2;
        f_j = (prev_func[get_index(x: layer_height - 1, y: j + 1, z: k)] + prev_func[get_index(x: layer_height - 1, y: j - 1, z: k)]) / H_Y_2;
        f_k = (prev_func[get_index(x: layer_height - 1, y: j, z: k + 1)] + prev_func[get_index(x: layer_height - 1, y: j, z: k - 1)]) / H_Z_2;

        curr_func[get_index(x: layer_height - 1, y: j, z: 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);

        tmp_max_diff = fabs(x: curr_func[get_index(x: layer_height - 1, y: j, z: k)] - prev_func[get_index(x: layer_height - 1, y: j, z: 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(x: curr_func[get_index(x: i, y: j, z: 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(sendbuf: &max_proc_delta, recvbuf: &max_delta, count: 1, datatype: MPI_DOUBLE, op: MPI_MAX, comm: MPI_COMM_WORLD);

  return max_delta;
}
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;

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

  MPI_Init(&argc, &argv);

  MPI_Comm_size(comm: MPI_COMM_WORLD, size: &proc_count);
  MPI_Comm_rank(comm: MPI_COMM_WORLD, &proc_rank);

  layer_heights = malloc(size: sizeof(int) * proc_count);
  offsets = malloc(size: sizeof(int) * proc_count);
  divide_area_into_layers(layer_heights, offsets, proc_count);

  prev_func = malloc(size: sizeof(double) * layer_heights[proc_rank] * N_Y * N_Z);
  curr_func = malloc(size: 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(size: sizeof(double) * N_Y * N_Z);
  down_border_layer = malloc(size: 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;
    MPI_Iallreduce(sendbuf: &prev_proc_max_diff, recvbuf: &max_diff, count: 1, datatype: MPI_DOUBLE, op: MPI_MAX, comm: MPI_COMM_WORLD, request: &reduce_max_diff_req);

    swap_func(&prev_func, &curr_func);

    if (proc_rank != 0) {
      double *prev_up_border = prev_func;
      MPI_Isend(buf: prev_up_border, count: N_Y * N_Z, datatype: MPI_DOUBLE, dest: proc_rank - 1, tag: proc_rank, comm: MPI_COMM_WORLD, request: &send_up_req);
      MPI_Irecv(buf: up_border_layer, count: N_Y * N_Z, datatype: MPI_DOUBLE, source: proc_rank - 1, tag: proc_rank - 1, comm: MPI_COMM_WORLD, request: &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(buf: prev_down_border, count: N_Y * N_Z, datatype: MPI_DOUBLE, dest: proc_rank + 1, tag: proc_rank, comm: MPI_COMM_WORLD, request: &send_down_req);
      MPI_Irecv(buf: down_border_layer, count: N_Y * N_Z, datatype: MPI_DOUBLE, source: proc_rank + 1, tag: proc_rank + 1, comm: MPI_COMM_WORLD, request: &recv_down_req);
    }

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

    if (proc_rank != 0) {
      MPI_Wait(request: &send_up_req, status: MPI_STATUS_IGNORE);
      MPI_Wait(request: &recv_up_req, status: MPI_STATUS_IGNORE);
    }

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

    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);

    MPI_Wait(request: &reduce_max_diff_req, status: MPI_STATUS_IGNORE);

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

  swap_func(&prev_func, &curr_func);

  max_diff = calc_max_diff(curr_func, layer_heights[proc_rank], offsets[proc_rank]);

  finish_time = MPI_Wtime();

  if (proc_rank == 0) {
    printf(format: "Time: %lf\n", finish_time - start_time);
    printf(format: "Max difference: %lf\n", max_diff);
  }

  free(ptr: offsets);
  free(ptr: layer_heights);
  free(ptr: prev_func);
  free(ptr: curr_func);
  free(ptr: up_border_layer);
  free(ptr: down_border_layer);

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```