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 второго курса, группы 19201

(Колюжнов Егор Дмитриевич)

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А.Ю.Власенко

Новосибирск 2021

СОДЕРЖАНИЕ

ЦЕЛЬ

Практическое освоение методов 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 $\frac{\partial^2\varphi}{\partial x^2}+\frac{\partial^2\varphi}{\partial y^2}+\frac{\partial^2\varphi}{\partial z^2}-a\varphi=\rho,\ a\geq 0,$ $\varphi=\varphi(x,y,z),\ \rho=\rho(x,y,z),$ 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.

Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.

3. Выполнить профилирование программы с помощью MPE при использовании 16-и ядер. По профилю убедиться, что коммуникации происходят на фоне счета.

ОПИСАНИЕ РАБОТЫ

1)Написана программа на C с использованием MPI, реализующая параллельный алгоритм решения уравнения (1) методом Якоби в случае одномерной декомпозиции области в соответствии с заданием (находится в приложении 1).

В качестве размеров были взяты N_x = N_y = N_z = 300 и

область моделирования: $[-1;1] \times [-1;1] \times [-1;1]$,

искомая функция: $\varphi(x,y,z) = x^2 + y^2 + z^2$,

правая часть уравнения: $\rho(x,y,z) = 6 - a \cdot \varphi(x,y,z)$,

параметр уравнения: $a = 10^5$,

порог сходимости: $\varepsilon = 10^{-8}$,

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

Также программа считает максимальную погрешность среди всех узлов и количество итераций.

2)Написан скрипт для Altair PBS Pro, выделяющий 2 узла по 8 ядер, компилирующий программу на сервере и исполняющий ее для 1,2,4,6,8,12,16 процессов, а для 16 ядер еще и профилирующий их (Части результатов профилирования находятся в приложении 4). (Находится в приложении 2)

3)Скрипт был добавлен в очередь и выполнен:

qsub –e err –o out du_jobs.h

```
Cores 1: 33.416726; Precision: 0.0000001977
Cores 2: 17.188338; Precision: 0.0000001977
Cores 4: 9.031985; Precision: 0.0000001977
Cores 6: 6.480895; Precision: 0.0000001977
Cores 8: 5.070964; Precision: 0.0000001977
Cores 12: 3.706213; Precision: 0.0000001977
Cores 16: 2.895929; Precision: 0.0000001977
```

4)По полученным данным был построен график зависимости времени от числа процессов и размера решетки (Находится в приложении 3).

ЗАКЛЮЧЕНИЕ

Был реализован параллельный алгоритм реализующий решение дифференциального уравнения методом Якоби в трехмерной области в случае одномерной декомпозиции области. Также были произведены измерения времени в зависимости от количества процессов. Наибольшая эффективность была при запуске на 2-х процессах и на 12-ти процессах, что может быть объяснено тем, что процессоры были 8-ми ядерные, а программа только начиная с 12-ти процессов запускалась на обоих узлах, так как при 6-ти и 8-ми процессах эффективность была близка к нулю. Из результатов профилирования видно, что на MPI-функции ушло 12% от времени исполнения, а функции, на которые ушло больше всего времени – Bcast и Waitany, но если Bcast еще выполнял блокирующую операцию приема небольшого числа данных о сходимости, то Waitany ждала завершения фоновой операции над большим числом данных, и стоит отметить, что времени на нее ушло столько же, сколько и на Bcast, несмотря на значительное различие в количестве передаваемых данных, что показывает целесообразность использования здесь неблокирующих операций.

Приложение 1. Программа, реализующая параллельный алгоритм решения уравнения методом Якоби.

main.c

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

#include "problem_data.h"
#include "matrix_3d.h"
#include "local_problem.h"
#include "problem_data.h"
#include "proc_meta.h"

typedef struct {
    ProcMeta meta;
    ProblemData task;
    LocalProblem problem;
} ProcData;

int init_procdata(ProcData* this) {
    pd_init(&this->task);
    pm_init(&this->meta, &this->task);
    lp_init(&this->problem, &this->task, &this->meta);

    return EXIT_SUCCESS;
}

void free_procdata(ProcData* this) {
    pm_free(&this->meta);
    lp_free(&this->problem);
}

bool local_solved(const Matrix3D* old, const Matrix3D* new, double epsilon) {
    double delta;

    for (int x = 1; x < mt_x_len(old) - 1; ++x) {
        for (int y = 1; y < mt_y_len(old) - 1; ++y) {
            for (int z = 1; z < mt_z_len(old) - 1; ++z) {
                delta = fabs(mt_get(new, x, y, z) - mt_get(old, x, y, z));
                if (delta >= epsilon) return false;
            }
        }
    }

    return true;
}
```

main.c

```c
bool global_solved(const ProcMeta* meta, const Matrix3D* old, const Matrix3D* new, double epsilon) {
    const bool solved_here = local_solved(old, new, epsilon);
    bool* buf = NULL;
    if (pm_is_root(meta)) {
        buf = malloc(pm_count(meta) * sizeof *buf);
    }

    MPI_Gather(&solved_here, 1, MPI_C_BOOL,
               buf, 1, MPI_C_BOOL,
               pm_root(meta), pm_comm(meta));

    bool solved_all;
    if (pm_is_root(meta)) {
        solved_all = true;
        for (int i = 0; i < pm_count(meta); ++i) {
            if (!buf[i]) {
                solved_all = false;
                break;
            }
        }
        free(buf);
    }

    MPI_Bcast(&solved_all, 1, MPI_C_BOOL, pm_root(meta), pm_comm(meta));
    return solved_all;
}

void isend_plane(const ProcData* data, int plane_index, MPI_Request out_requests[NB_COUNT], int tag) {
    const int neighbour = pm_neighbour(&data->meta, plane_index);
    if (neighbour == NO_NEIGHBOUR) {
        out_requests[plane_index] = MPI_REQUEST_NULL;
        return;
    }

    const double* plane = lp_out_plane_c(&data->problem, plane_index);

    MPI_Isend(plane, pm_disc_len_y(&data->meta) * pm_disc_len_z(&data->meta), MPI_DOUBLE,
        neighbour, tag, pm_comm(&data->meta),
        &out_requests[plane_index]);
}

void irecv_plane(ProcData* data, int plane_index, MPI_Request in_requests[NB_COUNT], int tag) {
    const int neighbour = pm_neighbour(&data->meta, plane_index);
    if (neighbour == NO_NEIGHBOUR) {
        in_requests[plane_index] = MPI_REQUEST_NULL;
        return;
    }

    double* plane = lp_in_plane(&data->problem, plane_index);

    MPI_Irecv(plane, pm_disc_len_y(&data->meta) * pm_disc_len_z(&data->meta), MPI_DOUBLE,
        neighbour, tag, pm_comm(&data->meta),
        &in_requests[plane_index]);
}
```

main.c

```c
void isend_neighbour_planes(const ProcData* data, MPI_Request out_requests[NB_COUNT], int tag) {
    for (int i = 0; i < NB_COUNT; ++i) {
        isend_plane(data, i, out_requests, tag);
    }
}

void irecv_neighbour_planes(ProcData* data, MPI_Request in_requests[NB_COUNT], int tag) {
    for (int i = 0; i < NB_COUNT; ++i) {
        irecv_plane(data, i, in_requests, tag);
    }
}
```

main.c

```c
void solve_equation(ProcData* data, double epsilon) {
    const int tag = 123;
    MPI_Request out_requests[NB_COUNT];
    for (int i = 0; i < NB_COUNT; ++i) {
        out_requests[i] = MPI_REQUEST_NULL;
    }
    MPI_Request in_requests[NB_COUNT];


    int iterations = 0;
    do {
        ++iterations;

        MPI_Waitall(NB_COUNT, out_requests, MPI_STATUSES_IGNORE);
        isend_neighbour_planes(data, out_requests, tag);

        lp_swap_cubes(&data->problem);

        irecv_neighbour_planes(data, in_requests, tag);
        lp_iterate(&data->problem);

        for (int i = 0; i < NB_COUNT; ++i) {
            if (in_requests[i] != MPI_REQUEST_NULL) continue;
            lp_finish_plane(&data->problem, i);
        }

        for (;;) {
            int plane_index;
            MPI_Waitany(NB_COUNT, in_requests, &plane_index, MPI_STATUS_IGNORE);
            if (plane_index == MPI_UNDEFINED) break;

            in_requests[plane_index] = MPI_REQUEST_NULL;

            lp_finish_plane(&data->problem, plane_index);
        }

    } while (!global_solved(
        &data->meta,
        lp_old_mat(&data->problem),
        lp_new_mat(&data->problem), epsilon)
    );

    if (pm_is_root(&data->meta)) {
        fprintf(stderr, "Iterations: %d\n", iterations);
    }
}
```

main.c

```c
double get_precision(const double* solution, const ProblemData* task) {
    double max_delta = 0.0;

    const int len[DIMS] = {pd_disc_x(task), pd_disc_y(task), pd_disc_z(task)};

    for (int x = 0; x < len[X]; ++x) {
        for (int y = 0; y < len[Y]; ++y) {
            for (int z = 0; z < len[Z]; ++z) {
                const double phi = pd_phi(task, x, y, z);
                const double solved = solution[len[Y] * len[Z] * x + len[Z] * y + z];

                const double delta = fabs(solved - phi) / phi;
                if (delta > max_delta) {
                    max_delta = delta;
                }
            }
        }
    }

    return max_delta;
}
```

main.c

```c
double gather_solution(const ProcData* data) {
    double* solution = NULL;
    int* recvcounts = NULL;
    int* displs = NULL;

    const Matrix3D* local_solution = lp_new_mat_c(&data->problem);
    const int plane_size = mt_y_len(local_solution) * mt_z_len(local_solution);

    if (pm_is_root(&data->meta)) {
        solution = malloc(pd_disc_x(&data->task) *
            pd_disc_y(&data->task) * pd_disc_z(&data->task) * sizeof *solution);
        recvcounts = malloc(pm_count(&data->meta) * sizeof *recvcounts);
        displs = malloc(pm_count(&data->meta) * sizeof *displs);

        for (int i = 0; i < pm_count(&data->meta) - 1; ++i) {
            recvcounts[i] = data->meta.data_per_proc;
        }
        recvcounts[pm_count(&data->meta) - 1] = data->meta.last_data_count;

        ++recvcounts[0];
        ++recvcounts[pm_count(&data->meta) - 1];

        for (int i = 0; i < pm_count(&data->meta); ++i) {
            recvcounts[i] *= plane_size;
        }

        int curr_displ = 0;
        for (int i = 0; i < pm_count(&data->meta); ++i) {
            displs[i] = curr_displ;
            curr_displ += recvcounts[i];
        }
    }

    const bool left = pm_neighbour(&data->meta, NB_LEFT) == NO_NEIGHBOUR;
    const bool right = pm_neighbour(&data->meta, NB_RIGHT) == NO_NEIGHBOUR;

    int plane_count = pm_disc_len_x(&data->meta);
    if (left) ++plane_count;
    if (right) ++plane_count;

    MPI_Gatherv(mt_get_ref(local_solution, (left) ? 0 : 1, 0, 0),
        plane_count * plane_size, MPI_DOUBLE,
        solution, recvcounts, displs,
        MPI_DOUBLE, pm_root(&data->meta), pm_comm(&data->meta));

    if (pm_is_root(&data->meta)) {
        const double precision = get_precision(solution, &data->task);

        free(solution);
        free(displs);
        free(recvcounts);

        return precision;
    }
    return 0;
```

main.c

```c
void test_solution() {
    ProcData data;
    init_procdata(&data);

    const double start = MPI_Wtime();
    solve_equation(&data, EPSILON);
    const double precision = gather_solution(&data);
    const double end = MPI_Wtime();

    if (pm_is_root(&data.meta)) {
        printf("Cores %d: %f; Precision: %.10f\n",
            pm_count(&data.meta), end - start, precision);
    }

    free_procdata(&data);
}

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    test_solution();
    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

types.h

```c
#ifndef LAB4_TYPES_H
#define LAB4_TYPES_H

#include <stddef.h>
#include <stdbool.h>
#include <mpi.h>

#define NB_LEFT 0
#define NB_RIGHT 1

#define NB_COUNT 2

#define DIMS 3
#define X 0
#define Y 1
#define Z 2

typedef struct {
    int proc_count;
    int proc_rank;
    int proc_coord;

    int data_per_proc;
    int last_data_count;

    MPI_Comm global_comm;

    int neighbours[NB_COUNT];

    int task_dims[DIMS];
    int task_coords[DIMS];
} ProcMeta;

typedef struct {
    int dimensions[3];
    double* data;
} Matrix3D;

typedef struct {
    double factor;
    double h_2[DIMS];
    Matrix3D rho;
} Constants;
```

types.h

```c
typedef struct {
    Matrix3D slice[2]; //old and new
    int slice_dims[DIMS];
    int inner_dims[DIMS];
    int inner_offset[DIMS];

    int plane_x_coords[NB_COUNT];
    int plane_neighbour_x_coords[NB_COUNT];

    int old;
    int new;

    Constants constants;
} LocalProblem;

typedef double (*func_r3) (double x, double y, double z);

typedef struct {
    double area_offset[DIMS];
    double global_area[DIMS];

    double a;
    func_r3 phi;
    func_r3 rho;

    int discrete_dimensions[DIMS];
    double local_area[DIMS];
} ProblemData;

#endif // !LAB4_TYPES_H
```

proc_meta.h

```c
#ifndef LAB4_PROC_META_H
#define LAB4_PROC_META_H

#include "types.h"
#include "local_problem.h"

#define NO_NEIGHBOUR (-1)

void pm_init(ProcMeta* this, const ProblemData* data);
void pm_free(ProcMeta* this);

int pm_count(const ProcMeta* this);
int pm_coord(const ProcMeta* this);
int pm_rank(const ProcMeta* this);
int pm_neighbour(const ProcMeta* this, int index);

MPI_Comm pm_comm(const ProcMeta* this);

int pm_disc(const ProcMeta* this, int coord, int local_coord);
int pm_disc_x(const ProcMeta* this, int local_x);
int pm_disc_y(const ProcMeta* this, int local_y);
int pm_disc_z(const ProcMeta* this, int local_z);

int pm_disc_len(const ProcMeta* this, int coord);
int pm_disc_len_x(const ProcMeta* this);
int pm_disc_len_y(const ProcMeta* this);
int pm_disc_len_z(const ProcMeta* this);

int pm_root(const ProcMeta* this);
bool pm_is_root(const ProcMeta* this);

#endif // !LAB4_PROC_META_H
```

proc_meta.c

```c
#include "proc_meta.h"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int get_data_per_proc(int global_size, int proc_count) {
    return global_size / proc_count + (global_size % proc_count ? 1 : 0);
}

int get_last_cut_size(int global_size, int proc_count) {
     int data_per_proc = get_data_per_proc(global_size, proc_count);
    return global_size - data_per_proc * (proc_count - 1);
}

int get_cut_size(int global_size, int proc_count, int rank) {
    return (rank < proc_count - 1)
            ? get_data_per_proc(global_size, proc_count)
            : get_last_cut_size(global_size, proc_count);
}

int get_cart_rank(const ProcMeta* this, int coord) {
    int rank;
    MPI_Cart_rank(pm_comm(this), &coord, &rank);
    return rank;
}

void determine_neighbours(ProcMeta* this) {
    const int coords[NB_COUNT] = {pm_coord(this) - 1, pm_coord(this) + 1};

    for (int i = 0; i < NB_COUNT; ++i) {
        if (coords[i] < 0 || coords[i] >= pm_count(this)) {
            this->neighbours[i] = NO_NEIGHBOUR;
        } else {
            this->neighbours[i] = get_cart_rank(this, coords[i]);
        }
    }
}

void cube_without_edges(const ProblemData* data, int dimensions[DIMS]) {
    for (int coord = 0; coord < DIMS; ++coord) {
        dimensions[coord] = pd_disc(data, coord) - 2;
    }
}

void calculate_cuts(ProcMeta* this, const int dimensions[DIMS]) {
    memcpy(this->task_dims, dimensions, DIMS * sizeof *this->task_dims);
    this->task_dims[X] = get_cut_size(dimensions[X], pm_count(this), pm_coord(this));

    memset(this->task_coords, 0, DIMS * sizeof *this->task_dims);
    this->task_coords[X] = pm_coord(this) * get_data_per_proc(dimensions[X], pm_count(this));
}
```

proc_meta.c

```c
void pm_init(ProcMeta* this, const ProblemData* data) {
    MPI_Comm_size(MPI_COMM_WORLD, &this->proc_count);
    MPI_Comm_rank(MPI_COMM_WORLD, &this->proc_rank);

    const int periodic = 0;
    MPI_Cart_create(MPI_COMM_WORLD, 1, &this->proc_count, &periodic, 1, &this->global_comm);
    MPI_Cart_coords(this->global_comm, this->proc_rank, 1, &this->proc_coord);
    MPI_Cart_rank(this->global_comm, &this->proc_coord, &this->proc_rank);

    determine_neighbours(this);

    int dims_without_edges[DIMS];
    cube_without_edges(data, dims_without_edges);
    calculate_cuts(this, dims_without_edges);

    this->data_per_proc = get_data_per_proc(dims_without_edges[X], pm_count(this));
    this->last_data_count = get_last_cut_size(dims_without_edges[X], pm_count(this));
}

void pm_free(ProcMeta* this) {
    MPI_Comm_free(&this->global_comm);
}

int pm_neighbour(const ProcMeta* this, int index) {
    return this->neighbours[index];
}

MPI_Comm pm_comm(const ProcMeta* this) {
    return this->global_comm;
}

int pm_coord(const ProcMeta* this) {
    return this->proc_coord;
}

int pm_root(const ProcMeta* this) {
    return get_cart_rank(this, 0);
}

int pm_rank(const ProcMeta* this) {
    return this->proc_rank;
}

bool pm_is_root(const ProcMeta* this) {
    return (pm_rank(this) == pm_root(this));
}

int pm_count(const ProcMeta* this) {
    return this->proc_count;
}

int pm_disc(const ProcMeta* this, int coord, int local_coord) {
    return this->task_coords[coord] + 1 + local_coord;
}
```

proc_meta.c

```c
int pm_disc_x(const ProcMeta* this, int local_x) {
    return pm_disc(this, X, local_x);
}

int pm_disc_y(const ProcMeta* this, int local_y) {
    return pm_disc(this, Y, local_y);
}

int pm_disc_z(const ProcMeta* this, int local_z) {
    return pm_disc(this, Z, local_z);
}

int pm_disc_len(const ProcMeta* this, int coord) {
    return this->task_dims[coord];
}

int pm_disc_len_x(const ProcMeta* this) {
    return pm_disc_len(this, X);
}

int pm_disc_len_y(const ProcMeta* this) {
    return pm_disc_len(this, Y);
}

int pm_disc_len_z(const ProcMeta* this) {
    return pm_disc_len(this, Z);
}
```

problem_data.h

```c
#ifndef LAB4_PROBLEM_DATA_H
#define LAB4_PROBLEM_DATA_H

#include "types.h"
#include "local_problem.h"

#define X_0 (-1)
#define Y_0 (-1)
#define Z_0 (-1)

#define D_X 2
#define D_Y 2
#define D_Z 2

#define A 100000

#define EPSILON 0.00000001

#define N_X 300
#define N_Y 300
#define N_Z 300

double phi(double x, double y, double z);
double rho(double x, double y, double z);

double pd_h(const ProblemData* this, int coord);
double pd_h_x(const ProblemData* this);
double pd_h_y(const ProblemData* this);
double pd_h_z(const ProblemData* this);

double pd_area(const ProblemData* this, int coord, int index);
double pd_area_x(const ProblemData* this, int x);
double pd_area_y(const ProblemData* this, int y);
double pd_area_z(const ProblemData* this, int z);

int pd_disc(const ProblemData* this, int coord);
int pd_disc_x(const ProblemData* this);
int pd_disc_y(const ProblemData* this);
int pd_disc_z(const ProblemData* this);

double pd_a(const ProblemData* this);
double pd_factor(const ProblemData* this);

double pd_func(const ProblemData* this, const int discrete_point[DIMS], func_r3 func);
double pd_rho_ar(const ProblemData* this, const int discrete_point[DIMS]);
double pd_rho(const ProblemData* this, int x, int y, int z);
double pd_phi_ar(const ProblemData* this, const int discrete_point[DIMS]);
double pd_phi(const ProblemData* this, int x, int y, int z);

void pd_init(ProblemData* this);

#endif // !LAB4_PROBLEM_DATA_H
```

problem_data.c

```c
#include "problem_data.h"

double pd_h(const ProblemData* this, int coord) {
    return this->local_area[coord];
}

double pd_h_x(const ProblemData* this) {
    return pd_h(this, X);
}

double pd_h_y(const ProblemData* this) {
    return pd_h(this, Y);
}

double pd_h_z(const ProblemData* this) {
    return pd_h(this, Z);
}

double pd_a(const ProblemData* this) {
    return this->a;
}

double pd_factor(const ProblemData* this) {
    return 1 / (
        2 / (pd_h_x(this) * pd_h_x(this)) +
        2 / (pd_h_y(this) * pd_h_y(this)) +
        2 / (pd_h_z(this) * pd_h_z(this)) +
        pd_a(this)
    );
}

double pd_area(const ProblemData* this, int coord, int index) {
    return this->area_offset[coord] + index * pd_h(this, coord);
}

double pd_area_x(const ProblemData* this, int x) {
    return pd_area(this, X, x);
}

double pd_area_y(const ProblemData* this, int y) {
    return pd_area(this, Y, y);
}

double pd_area_z(const ProblemData* this, int z) {
    return pd_area(this, Z, z);
}

double pd_func(const ProblemData* this, const int discrete_point[DIMS], func_r3 func) {
    const double x = pd_area_x(this, discrete_point[X]);
    const double y = pd_area_y(this, discrete_point[Y]);
    const double z = pd_area_z(this, discrete_point[Z]);

    return func(x, y, z);
}
```

problem_data.c

```c
double pd_rho_ar(const ProblemData* this, const int discrete_point[DIMS]) {
    return pd_func(this, discrete_point, this->rho);
}

double pd_rho(const ProblemData* this, int x, int y, int z) {
    const int discrete_point[DIMS] = {x, y, z};
    return pd_func(this, discrete_point, this->rho);
}

double pd_phi_ar(const ProblemData* this, const int discrete_point[DIMS]) {
    return pd_func(this, discrete_point, this->phi);
}

double pd_phi(const ProblemData* this, int x, int y, int z) {
    const int discrete_point[DIMS] = {x, y, z};
    return pd_func(this, discrete_point, this->phi);
}

void count_local_area(double local_area[DIMS], double global_area[DIMS], int dimensions[DIMS]) {
    for (int i = 0; i < DIMS; ++i) {
        if (dimensions[i] == 0)
            local_area[i] = 0;
        local_area[i] = global_area[i] /
                        (dimensions[i] - 1);
    }
}

void pd_init(ProblemData* this) {
    this->area_offset[X] = X_0;
    this->area_offset[Y] = Y_0;
    this->area_offset[Z] = Z_0;

    this->global_area[X] = D_X;
    this->global_area[Y] = D_Y;
    this->global_area[Z] = D_Z;

    this->a = A;

    this->phi = phi;
    this->rho = rho;

    this->discrete_dimensions[X] = N_X;
    this->discrete_dimensions[Y] = N_Y;
    this->discrete_dimensions[Z] = N_Z;

    count_local_area(this->local_area, this->global_area, this->discrete_dimensions);
}

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);
```

problem_data.c

```c
int pd_disc(const ProblemData* this, int coord) {
    return this->discrete_dimensions[coord];
}

int pd_disc_x(const ProblemData* this) {
    return pd_disc(this, X);
}

int pd_disc_y(const ProblemData* this) {
    return pd_disc(this, Y);
}

int pd_disc_z(const ProblemData* this) {
    return pd_disc(this, Z);
}
```

matrix_3d.h

```c
#ifndef LAB4_TENSOR_H
#define LAB4_TENSOR_H

#include "types.h"

void mt_init(Matrix3D* this, const int dimensions[DIMS]);

int mt_len(const Matrix3D* this, int coord);
int mt_x_len(const Matrix3D* this);
int mt_y_len(const Matrix3D* this);
int mt_z_len(const Matrix3D* this);

double mt_get(const Matrix3D* this, int x, int y, int z);
const double* mt_get_ref(const Matrix3D* this, int x, int y, int z);
double* mt_set(Matrix3D* this, int x, int y, int z);

double mt_get_ar(const Matrix3D* this, const int dimensions[DIMS]);
const double* mt_get_ref_ar(const Matrix3D* this, const int dimensions[DIMS]);
double* mt_set_ar(Matrix3D* this, const int dimensions[DIMS]);

double mt_get_by_i(const Matrix3D* this, int index);
const double* mt_get_ref_by_i(const Matrix3D* this, int index);
double* mt_set_by_i(Matrix3D* this, int index);

void mt_free(Matrix3D* this);

#endif // !LAB4_TENSOR_H
```

matrix_3d.c

```c
#include "matrix_3d.h"

#include <stdlib.h>
#include <string.h>

int get_index(const Matrix3D* this, i const Matrix3D *this
    return mt_y_len(this) * mt_z_len(this) * x
         + mt_z_len(this) * y
         + z;
}

double mt_get(const Matrix3D* this, int x, int y, int z) {
    return mt_get_by_i(this, get_index(this, x, y, z));
}

double mt_get_ar(const Matrix3D* this, const int dimensions[DIMS]) {
    return mt_get(this, dimensions[X], dimensions[Y], dimensions[Z]);
}

const double* mt_get_ref(const Matrix3D* this, int x, int y, int z) {
    return mt_get_ref_by_i(this, get_index(this, x, y, z));
}

const double* mt_get_ref_ar(const Matrix3D* this, const int dimensions[DIMS]) {
    return mt_get_ref(this, dimensions[X], dimensions[Y], dimensions[Z]);
}

double* mt_set(Matrix3D* this, int x, int y, int z) {
    return mt_set_by_i(this, get_index(this, x, y, z));
}

double* mt_set_ar(Matrix3D* this, const int dimensions[DIMS]) {
    return mt_set(this, dimensions[X], dimensions[Y], dimensions[Z]);
}

double mt_get_by_i(const Matrix3D* this, int index) {
    return this->data[index];
}

const double* mt_get_ref_by_i(const Matrix3D* this, int index) {
    return &this->data[index];
}

double* mt_set_by_i(Matrix3D* this, int index) {
    return &this->data[index];
}

void mt_init(Matrix3D* this, const int dimensions[DIMS]) {
    memcpy(this->dimensions, dimensions, DIMS * sizeof *this->dimensions);
    this->data = calloc(mt_x_len(this) * mt_y_len(this) * mt_z_len(this),  sizeof *this->data);
}

int mt_len(const Matrix3D* this, int coord) {
    return this->dimensions[coord];
}
```

matrix_3d.c

```c
int mt_x_len(const Matrix3D* this) {
    return mt_len(this, X);
}

int mt_y_len(const Matrix3D* this) {
    return mt_len(this, Y);
}

int mt_z_len(const Matrix3D* this) {
    return mt_len(this, Z);
}

void mt_free(Matrix3D* this) {
    free(this->data);
}
```

local_problem.h

```c
#ifndef LAB4_LOCAL_PROBLEM_H
#define LAB4_LOCAL_PROBLEM_H

#include "types.h"

#include "matrix_3d.h"
#include "problem_data.h"

void lp_init(LocalProblem* this, const ProblemData* data, const ProcMeta* meta);
void lp_free(LocalProblem* this);

void lp_iterate(LocalProblem* this);
void lp_swap_cubes(LocalProblem* this);
void lp_finish_plane(LocalProblem* this, int plane_index);

Matrix3D* lp_old_mat(LocalProblem* this);
Matrix3D* lp_new_mat(LocalProblem* this);

const Matrix3D* lp_old_mat_c(const LocalProblem* this);
const Matrix3D* lp_new_mat_c(const LocalProblem* this);

double* lp_in_plane(LocalProblem* this, int index);

double* lp_out_plane(LocalProblem* this, int index);
const double* lp_out_plane_c(const LocalProblem* this, int index);

#endif // !LAB4_LOCAL_PROBLEM_H
```

local_problem.c

```c
void lp_finish_plane(LocalProblem* this, int plane_index) {
    const int x = this->plane_neighbour_x_coords[plane_index];
    for (int y = this->inner_offset[Y]; y < this->inner_dims[Y] + this->inner_offset[Y]; ++y) {
        for (int z = this->inner_offset[Z]; z < this->inner_dims[Z] + this->inner_offset[Z]; ++z) {
            update_solution(this, x, y, z);
        }
    }
}

void lp_iterate(LocalProblem* this) {
    for (int x = this->plane_neighbour_x_coords[NB_LEFT] + 1; x < this->plane_neighbour_x_coords[NB_RIGHT]; ++x) {
        for (int y = this->inner_offset[Y]; y < this->inner_dims[Y] + this->inner_offset[Y]; ++y) {
            for (int z = this->inner_offset[Z]; z < this->inner_dims[Z] + this->inner_offset[Z]; ++z) {
                update_solution(this, x, y, z);
            }
        }
    }
}

void init_constants(Constants* this, const ProblemData* data, const ProcMeta* meta, const int inner_dims[DIMS]) {
    this->factor = pd_factor(data);

    this->h_2[X] = pd_h_x(data) * pd_h_x(data);
    this->h_2[Y] = pd_h_y(data) * pd_h_y(data);
    this->h_2[Z] = pd_h_z(data) * pd_h_z(data);

    mt_init(&this->rho, inner_dims);

    for (int x = 0; x < inner_dims[X]; ++x) {
        for (int y = 0; y < inner_dims[Y]; ++y) {
            for (int z = 0; z < inner_dims[Z]; ++z) {
                *mt_set(&this->rho, x, y, z) = pd_rho(data,
                    pm_disc_x(meta, x), pm_disc_y(meta, y), pm_disc_z(meta, z));
            }
        }
    }
}
```

local_problem.c

```c
void init_edge_values(LocalProblem* this, const ProblemData* data, const ProcMeta* meta) {
    int plane_coords[DIMS][NB_COUNT];
    int other_coords[DIMS][2] = {{Y, Z}, {X, Z}, {Y, X}};

    for (int coord = 0; coord < DIMS; ++coord) {
        plane_coords[coord][NB_LEFT] = 0;
        plane_coords[coord][NB_RIGHT] = this->slice_dims[coord] - 1;
    }

    int vec[DIMS] = {};
    for (int coord = 0; coord < DIMS; ++coord) {
        for (int pos = 0; pos < NB_COUNT; ++pos) {
            vec[coord] = plane_coords[coord][pos];
            const int x = other_coords[coord][0];
            const int y = other_coords[coord][1];

            for (vec[x] = 0; vec[x] < this->slice_dims[x]; ++vec[x]) {
                for (vec[y] = 0; vec[y] < this->slice_dims[y]; ++vec[y]) {
                    *mt_set_ar(lp_old_mat(this), vec) = pd_phi(data,
                        pm_disc_x(meta, vec[X] - this->inner_offset[X]),
                        pm_disc_y(meta, vec[Y] - this->inner_offset[Y]),
                        pm_disc_z(meta, vec[Z] - this->inner_offset[Z]));
                    *mt_set_ar(lp_new_mat(this), vec) = pd_phi(data,
                        pm_disc_x(meta, vec[X] - this->inner_offset[X]),
                        pm_disc_y(meta, vec[Y] - this->inner_offset[Y]),
                        pm_disc_z(meta, vec[Z] - this->inner_offset[Z]));
                }
            }
        }
    }
}
```

local_problem.c

```c
#include "local_problem.h"

#include "proc_meta.h"

#include <string.h>

double get_h_2(const LocalProblem* this, int coord) {
    return this->constants.h_2[coord];
}

double get_h_x_2(const LocalProblem* this) {
    return get_h_2(this, X);
}

double get_h_y_2(const LocalProblem* this) {
    return get_h_2(this, Y);
}

double get_h_z_2(const LocalProblem* this) {
    return get_h_2(this, Z);
}

const Matrix3D* get_rho(const LocalProblem* this) {
    return &this->constants.rho;
}

double get_factor(const LocalProblem* this) {
    return this->constants.factor;
}

void lp_swap_cubes(LocalProblem* this) {
    int tmp = this->old;
    this->old = this->new;
    this->new = tmp;
}

void update_solution(LocalProblem* this, int x, int y, int z) {
    const Matrix3D* old = lp_old_mat(this);
    Matrix3D* new = lp_new_mat(this);

    *mt_set(new, x, y, z) = get_factor(this) * (
        (mt_get(old, x + 1, y, z) + mt_get(old, x - 1, y, z)) / get_h_x_2(this) +
        (mt_get(old, x, y + 1, z) + mt_get(old, x, y - 1, z)) / get_h_y_2(this) +
        (mt_get(old, x, y, z + 1) + mt_get(old, x, y, z - 1)) / get_h_z_2(this) -
            mt_get(get_rho(this),
                x - this->inner_offset[X],
                y - this->inner_offset[Y],
                z - this->inner_offset[Z]));
}
```

local_problem.c

```c
void lp_init(LocalProblem* this, const ProblemData* data, const ProcMeta* meta) {
    memcpy(this->inner_dims, meta->task_dims, DIMS * sizeof *this->inner_dims);
    memcpy(this->slice_dims, meta->task_dims, DIMS * sizeof *this->slice_dims);

    for (int i = 0; i < DIMS; ++i) {
        this->slice_dims[i] += 2;
        this->inner_offset[i] = 1;
    }

    init_constants(&this->constants, data, meta, this->inner_dims);

    for (int i = 0; i < 2; ++i) {
        mt_init(&this->slice[i], this->slice_dims);
    }

    this->old = 0;
    this->new = 1;

    init_edge_values(this, data, meta);

    this->plane_x_coords[NB_LEFT] = 0;
    this->plane_x_coords[NB_RIGHT] = this->slice_dims[X] - 1;

    this->plane_neighbour_x_coords[NB_LEFT] = this->inner_offset[X];
    this->plane_neighbour_x_coords[NB_RIGHT] = this->slice_dims[X] - 2;
}

void lp_free(LocalProblem* this) {
    mt_free(&this->constants.rho);
    for (int i = 0; i < 2; ++i) {
        mt_free(&this->slice[i]);
    }
}

Matrix3D* lp_old_mat(LocalProblem* this) {
    return &this->slice[this->old];
}

Matrix3D* lp_new_mat(LocalProblem* this) {
    return &this->slice[this->new];
}

const Matrix3D* lp_old_mat_c(const LocalProblem* this) {
    return &this->slice[this->old];
}

const Matrix3D* lp_new_mat_c(const LocalProblem* this) {
    return &this->slice[this->new];
}

double* lp_in_plane(LocalProblem* this, int index) {
    return mt_set(lp_old_mat(this), this->plane_x_coords[index], this->inner_offset[Y], this->inner_offset[Z]);
}
```

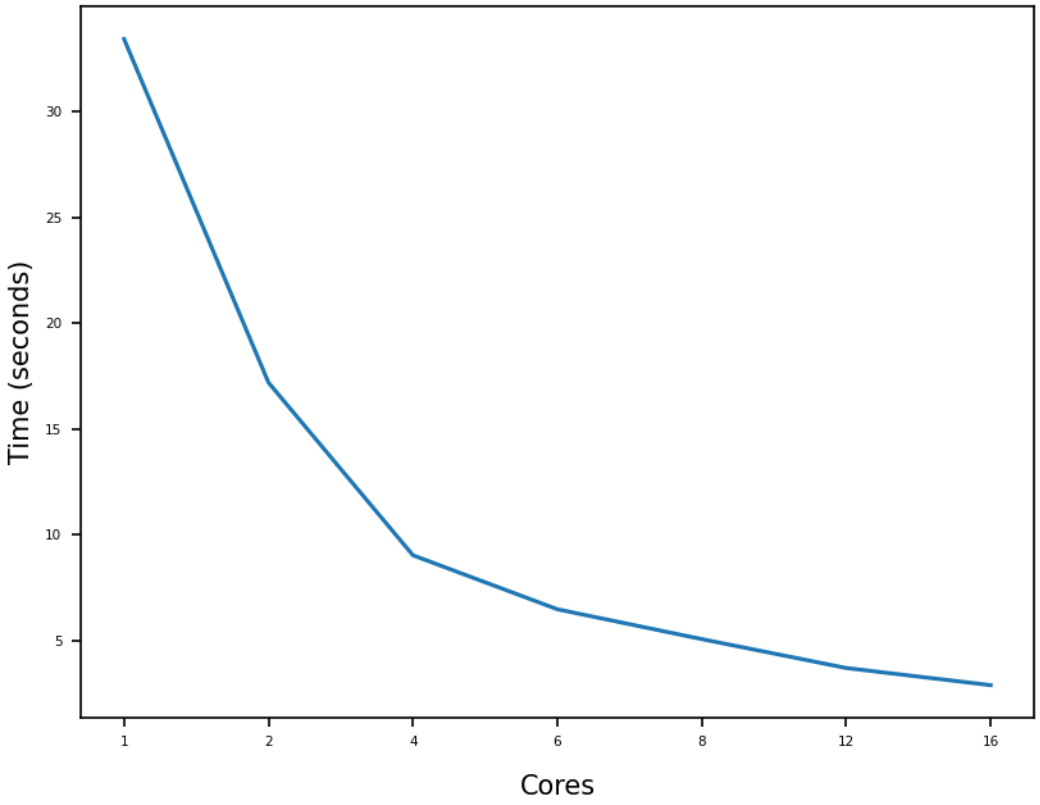
local_problem.c

```c
double* lp_out_plane(LocalProblem* this, int index) {
    return mt_set(lp_new_mat(this), this->plane_neighbour_x_coords[index], this->inner_offset[Y], this->inner_offset[Z]);
}

const double* lp_out_plane_c(const LocalProblem* this, int index) {
    return mt_get_ref(lp_new_mat_c(this), this->plane_neighbour_x_coords[index], this->inner_offset[Y], this->inner_offset[Z]);
}
```

Приложение 2.Скрипт для PBS.

```
#!/bin/bash

#PBS -l select=2:ncpus=8:mpiprocs=8,place=free:exclhost
#PBS -l walltime=00:15:00

cd $PBS_O_WORKDIR

mpiicc -O3 -std=gnu99 *.c

for i in {1,2,4,6,8,12}
do
mpirun -machinefile $PBS_NODEFILE -np $i ./a.out
done

mpirun -trace -machinefile $PBS_NODEFILE -np 16 ./a.out
```

Приложение 3. График зависимости времени от числа процессов и размера решетки.



Приложение 4. Результаты профилирования на 16 процессах.

Group All_Processes					
Group Application	38.1386 s	43.4436 s	16	2.38366 s	
MPI_Comm_size	10e-6 s	10e-6 s	16	625e-9 s	
MPI_Comm_free	37e-6 s	37e-6 s	16	2.3125e-6 s	
MPI_Comm_rank	16e-6 s	16e-6 s	16	1e-6 s	
MPI_Cart_rank	1.629e-3 s	1.629e-3 s	2366	688.503e-9 s	
MPI_Finalize	4.894e-3 s	4.894e-3 s	16	305.875e-6 s	
MPI_Bcast	1.76685 s	1.76685 s	560	3.15508e-3 s	
MPI_Cart_create	2.28099e-3 s	2.28099e-3 s	16	142.562e-6 s	
MPI_Isend	7.54999e-3 s	7.54999e-3 s	1050	7.19047e-6 s	
MPI_Irecv	1.415e-3 s	1.415e-3 s	1050	1.34762e-6 s	
MPI_Cart_coords	117e-6 s	117e-6 s	16	7.3125e-6 s	
MPI_Wtime	99e-6 s	99e-6 s	32	3.09375e-6 s	
MPI_Waitall	2.03199e-3 s	2.03199e-3 s	560	3.62856e-6 s	
MPI_Waitany	1.88178 s	1.88178 s	1610	1.16881e-3 s	
MPI_Gather	792.248e-3 s	792.248e-3 s	560	1.41473e-3 s	
MPI_Gatherv	844.081e-3 s	844.081e-3 s	16	52.7551e-3 s