

**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 .....	3
ЗАДАНИЕ .....	3
ОПИСАНИЕ РАБОТЫ .....	4
ЗАКЛЮЧЕНИЕ .....	5
Приложение 1. Программа, реализующая параллельный алгоритм решения уравнения методом Якоби .....	6
Приложение 2. Скрипт для PBS .....	29
Приложение 3. График зависимости времени от числа процессов .....	30
Приложение 4. Результаты профилирования на 16 процессах .....	31

## 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

$$\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} + \frac{\partial^2 \varphi}{\partial z^2} - a\varphi = \rho, \quad a \geq 0,$$

MPI, реализующую решение уравнения  $\varphi = \varphi(x, y, z)$ ,  $\rho = \rho(x, y, z)$ , 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:  $\varphi_{i,j,k}^0 = 0$ .

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 – Vcast и Waitany, но если Vcast 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 Vcast, 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.

```
C main.c  X
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <mpi.h>
4  #include <string.h>
5  #include <stdbool.h>
6  #include <math.h>
7
8  #include "problem_data.h"
9  #include "matrix_3d.h"
10 #include "local_problem.h"
11 #include "problem_data.h"
12 #include "proc_meta.h"
13
14 typedef struct {
15     ProcMeta meta;
16     ProblemData task;
17     LocalProblem problem;
18 } ProcData;
19
20 int init_procdata(ProcData* this) {
21     pd_init(&this->task);
22     pm_init(&this->meta, &this->task);
23     lp_init(&this->problem, &this->task, &this->meta);
24
25     return EXIT_SUCCESS;
26 }
27
28 void free_procdata(ProcData* this) {
29     pm_free(&this->meta);
30     lp_free(&this->problem);
31 }
32
33 bool local_solved(const Matrix3D* old, const Matrix3D* new, double epsilon) {
34     double delta;
35
36     for (int x = 1; x < mt_x_len(old) - 1; ++x) {
37         for (int y = 1; y < mt_y_len(old) - 1; ++y) {
38             for (int z = 1; z < mt_z_len(old) - 1; ++z) {
39                 delta = fabs(mt_get(new, x, y, z) - mt_get(old, x, y, z));
40                 if (delta >= epsilon) return false;
41             }
42         }
43     }
44
45     return true;
46 }
```

```
48 bool global_solved(const ProcMeta* meta, const Matrix3D* old, const Matrix3D* new, double epsilon) {
49     const bool solved_here = local_solved(old, new, epsilon);
50     bool* buf = NULL;
51     if (pm_is_root(meta)) {
52         buf = malloc(pm_count(meta) * sizeof *buf);
53     }
54
55     MPI_Gather(&solved_here, 1, MPI_C_BOOL,
56             buf, 1, MPI_C_BOOL,
57             pm_root(meta), pm_comm(meta));
58
59     bool solved_all;
60     if (pm_is_root(meta)) {
61         solved_all = true;
62         for (int i = 0; i < pm_count(meta); ++i) {
63             if (!buf[i]) {
64                 solved_all = false;
65                 break;
66             }
67         }
68         free(buf);
69     }
70
71     MPI_Bcast(&solved_all, 1, MPI_C_BOOL, pm_root(meta), pm_comm(meta));
72     return solved_all;
73 }
74
75 void isend_plane(const ProcData* data, int plane_index, MPI_Request out_requests[NB_COUNT], int tag) {
76     const int neighbour = pm_neighbour(&data->meta, plane_index);
77     if (neighbour == NO_NEIGHBOUR) {
78         out_requests[plane_index] = MPI_REQUEST_NULL;
79         return;
80     }
81
82     const double* plane = lp_out_plane_c(&data->problem, plane_index);
83
84     MPI_Isend(plane, pm_disc_len_y(&data->meta) * pm_disc_len_z(&data->meta), MPI_DOUBLE,
85             neighbour, tag, pm_comm(&data->meta),
86             &out_requests[plane_index]);
87 }
88
89 void irecv_plane(ProcData* data, int plane_index, MPI_Request in_requests[NB_COUNT], int tag) {
90     const int neighbour = pm_neighbour(&data->meta, plane_index);
91     if (neighbour == NO_NEIGHBOUR) {
92         in_requests[plane_index] = MPI_REQUEST_NULL;
93         return;
94     }
95
96     double* plane = lp_in_plane(&data->problem, plane_index);
97
98     MPI_Irecv(plane, pm_disc_len_y(&data->meta) * pm_disc_len_z(&data->meta), MPI_DOUBLE,
99             neighbour, tag, pm_comm(&data->meta),
100             &in_requests[plane_index]);
101 }
```

C main.c ×

```
103 void isend_neighbour_planes(const ProcData* data, MPI_Request out_requests[NB_COUNT], int tag) {
104     for (int i = 0; i < NB_COUNT; ++i) {
105         isend_plane(data, i, out_requests, tag);
106     }
107 }
108
109 void irectv_neighbour_planes(ProcData* data, MPI_Request in_requests[NB_COUNT], int tag) {
110     for (int i = 0; i < NB_COUNT; ++i) {
111         irectv_plane(data, i, in_requests, tag);
112     }
113 }
```



```
115 void solve_equation(ProcData* data, double epsilon) {
116     const int tag = 123;
117     MPI_Request out_requests[NB_COUNT];
118     for (int i = 0; i < NB_COUNT; ++i) {
119         out_requests[i] = MPI_REQUEST_NULL;
120     }
121     MPI_Request in_requests[NB_COUNT];
122
123
124     int iterations = 0;
125     do {
126         ++iterations;
127
128         MPI_Waitall(NB_COUNT, out_requests, MPI_STATUSES_IGNORE);
129         isend_neighbour_planes(data, out_requests, tag);
130
131         lp_swap_cubes(&data->problem);
132
133         irecv_neighbour_planes(data, in_requests, tag);
134         lp_iterate(&data->problem);
135
136         for (int i = 0; i < NB_COUNT; ++i) {
137             if (in_requests[i] != MPI_REQUEST_NULL) continue;
138             lp_finish_plane(&data->problem, i);
139         }
140
141         for (;;) {
142             int plane_index;
143             MPI_Waitany(NB_COUNT, in_requests, &plane_index, MPI_STATUS_IGNORE);
144             if (plane_index == MPI_UNDEFINED) break;
145
146             in_requests[plane_index] = MPI_REQUEST_NULL;
147
148             lp_finish_plane(&data->problem, plane_index);
149         }
150
151     } while (!global_solved(
152         &data->meta,
153         lp_old_mat(&data->problem),
154         lp_new_mat(&data->problem), epsilon)
155     );
156
157     if (pm_is_root(&data->meta)) {
158         fprintf(stderr, "Iterations: %d\n", iterations);
159     }
160 }
```

```
162 double get_precision(const double* solution, const ProblemData* task) {
163     double max_delta = 0.0;
164
165     const int len[DIMS] = {pd_disc_x(task), pd_disc_y(task), pd_disc_z(task)};
166
167     for (int x = 0; x < len[X]; ++x) {
168         for (int y = 0; y < len[Y]; ++y) {
169             for (int z = 0; z < len[Z]; ++z) {
170                 const double phi = pd_phi(task, x, y, z);
171                 const double solved = solution[len[Y] * len[Z] * x + len[Z] * y + z];
172
173                 const double delta = fabs(solved - phi) / phi;
174                 if (delta > max_delta) {
175                     max_delta = delta;
176                 }
177             }
178         }
179     }
180
181     return max_delta;
182 }
```

```

184 double gather_solution(const ProcData* data) {
185     double* solution = NULL;
186     int* recvcunts = NULL;
187     int* displs = NULL;
188
189     const Matrix3D* local_solution = lp_new_mat_c(&data->problem);
190     const int plane_size = mt_y_len(local_solution) * mt_z_len(local_solution);
191
192     if (pm_is_root(&data->meta)) {
193         solution = malloc(pd_disc_x(&data->task) *
194             pd_disc_y(&data->task) * pd_disc_z(&data->task) * sizeof *solution);
195         recvcunts = malloc(pm_count(&data->meta) * sizeof *recvcunts);
196         displs = malloc(pm_count(&data->meta) * sizeof *displs);
197
198         for (int i = 0; i < pm_count(&data->meta) - 1; ++i) {
199             recvcunts[i] = data->meta.data_per_proc;
200         }
201         recvcunts[pm_count(&data->meta) - 1] = data->meta.last_data_count;
202
203         ++recvcunts[0];
204         ++recvcunts[pm_count(&data->meta) - 1];
205
206         for (int i = 0; i < pm_count(&data->meta); ++i) {
207             recvcunts[i] *= plane_size;
208         }
209
210         int curr_displ = 0;
211         for (int i = 0; i < pm_count(&data->meta); ++i) {
212             displs[i] = curr_displ;
213             curr_displ += recvcunts[i];
214         }
215     }
216
217     const bool left = pm_neighbour(&data->meta, NB_LEFT) == NO_NEIGHBOUR;
218     const bool right = pm_neighbour(&data->meta, NB_RIGHT) == NO_NEIGHBOUR;
219
220     int plane_count = pm_disc_len_x(&data->meta);
221     if (left) ++plane_count;
222     if (right) ++plane_count;
223
224     MPI_Gatherv(mt_get_ref(local_solution, (left) ? 0 : 1, 0, 0),
225         plane_count * plane_size, MPI_DOUBLE,
226         solution, recvcunts, displs,
227         MPI_DOUBLE, pm_root(&data->meta), pm_comm(&data->meta));
228
229     if (pm_is_root(&data->meta)) {
230         const double precision = get_precision(solution, &data->task);
231
232         free(solution);
233         free(displs);
234         free(recvcunts);
235
236         return precision;
237     }
238     return 0;

```

C main.c ×

```
241 void test_solution() {
242     ProcData data;
243     init_procddata(&data);
244
245     const double start = MPI_Wtime();
246     solve_equation(&data, EPSILON);
247     const double precision = gather_solution(&data);
248     const double end = MPI_Wtime();
249
250     if (pm_is_root(&data.meta)) {
251         printf("Cores %d: %f; Precision: %.10f\n",
252             pm_count(&data.meta), end - start, precision);
253     }
254
255     free_procddata(&data);
256 }
257
258 int main(int argc, char* argv[]) {
259     MPI_Init(&argc, &argv);
260     test_solution();
261     MPI_Finalize();
262
263     return EXIT_SUCCESS;
264 }
```

C types.h ×

```
1  #ifndef LAB4_TYPES_H
2  #define LAB4_TYPES_H
3
4  #include <stddef.h>
5  #include <stdbool.h>
6  #include <mpi.h>
7
8  #define NB_LEFT 0
9  #define NB_RIGHT 1
10
11 #define NB_COUNT 2
12
13 #define DIMS 3
14 #define X 0
15 #define Y 1
16 #define Z 2
17
18 typedef struct {
19     int proc_count;
20     int proc_rank;
21     int proc_coord;
22
23     int data_per_proc;
24     int last_data_count;
25
26     MPI_Comm global_comm;
27
28     int neighbours[NB_COUNT];
29
30     int task_dims[DIMS];
31     int task_coords[DIMS];
32 } ProcMeta;
33
34 typedef struct {
35     int dimensions[3];
36     double* data;
37 } Matrix3D;
38
39 typedef struct {
40     double factor;
41     double h_2[DIMS];
42     Matrix3D rho;
43 } Constants;
44
```

```
45 typedef struct {
46     Matrix3D slice[2]; //old and new
47     int slice_dims[DIMS];
48     int inner_dims[DIMS];
49     int inner_offset[DIMS];
50
51     int plane_x_coords[NB_COUNT];
52     int plane_neighbour_x_coords[NB_COUNT];
53
54     int old;
55     int new;
56
57     Constants constants;
58 } LocalProblem;
59
60 typedef double (*func_r3) (double x, double y, double z);
61
62 typedef struct {
63     double area_offset[DIMS];
64     double global_area[DIMS];
65
66     double a;
67     func_r3 phi;
68     func_r3 rho;
69
70     int discrete_dimensions[DIMS];
71     double local_area[DIMS];
72 } ProblemData;
73
74 #endif // !LAB4_TYPES_H
```

C proc\_meta.h X

```
1  #ifndef LAB4_PROC_META_H
2  #define LAB4_PROC_META_H
3
4  #include "types.h"
5  #include "local_problem.h"
6
7  #define NO_NEIGHBOUR (-1)
8
9  void pm_init(ProcMeta* this, const ProblemData* data);
10 void pm_free(ProcMeta* this);
11
12 int pm_count(const ProcMeta* this);
13 int pm_coord(const ProcMeta* this);
14 int pm_rank(const ProcMeta* this);
15 int pm_neighbour(const ProcMeta* this, int index);
16
17 MPI_Comm pm_comm(const ProcMeta* this);
18
19 int pm_disc(const ProcMeta* this, int coord, int local_coord);
20 int pm_disc_x(const ProcMeta* this, int local_x);
21 int pm_disc_y(const ProcMeta* this, int local_y);
22 int pm_disc_z(const ProcMeta* this, int local_z);
23
24 int pm_disc_len(const ProcMeta* this, int coord);
25 int pm_disc_len_x(const ProcMeta* this);
26 int pm_disc_len_y(const ProcMeta* this);
27 int pm_disc_len_z(const ProcMeta* this);
28
29 int pm_root(const ProcMeta* this);
30 bool pm_is_root(const ProcMeta* this);
31
32 #endif // !LAB4_PROC_META_H
```

C proc\_meta.c X

```
1  #include "proc_meta.h"
2
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  int get_data_per_proc(int global_size, int proc_count) {
8      return global_size / proc_count + (global_size % proc_count ? 1 : 0);
9  }
10
11 int get_last_cut_size(int global_size, int proc_count) {
12     int data_per_proc = get_data_per_proc(global_size, proc_count);
13     return global_size - data_per_proc * (proc_count - 1);
14 }
15
16 int get_cut_size(int global_size, int proc_count, int rank) {
17     return (rank < proc_count - 1)
18         ? get_data_per_proc(global_size, proc_count)
19         : get_last_cut_size(global_size, proc_count);
20 }
21
22 int get_cart_rank(const ProcMeta* this, int coord) {
23     int rank;
24     MPI_Cart_rank(pm_comm(this), &coord, &rank);
25     return rank;
26 }
27
28 void determine_neighbours(ProcMeta* this) {
29     const int coords[NB_COUNT] = {pm_coord(this) - 1, pm_coord(this) + 1};
30
31     for (int i = 0; i < NB_COUNT; ++i) {
32         if (coords[i] < 0 || coords[i] >= pm_count(this)) {
33             this->neighbours[i] = NO_NEIGHBOUR;
34         } else {
35             this->neighbours[i] = get_cart_rank(this, coords[i]);
36         }
37     }
38 }
39
40 void cube_without_edges(const ProblemData* data, int dimensions[DIMS]) {
41     for (int coord = 0; coord < DIMS; ++coord) {
42         dimensions[coord] = pd_disc(data, coord) - 2;
43     }
44 }
45
46 void calculate_cuts(ProcMeta* this, const int dimensions[DIMS]) {
47     memcpy(this->task_dims, dimensions, DIMS * sizeof *this->task_dims);
48     this->task_dims[X] = get_cut_size(dimensions[X], pm_count(this), pm_coord(this));
49
50     memset(this->task_coords, 0, DIMS * sizeof *this->task_dims);
51     this->task_coords[X] = pm_coord(this) * get_data_per_proc(dimensions[X], pm_count(this));
52 }
```



```
54 void pm_init(ProcMeta* this, const ProblemData* data) {
55     MPI_Comm_size(MPI_COMM_WORLD, &this->proc_count);
56     MPI_Comm_rank(MPI_COMM_WORLD, &this->proc_rank);
57
58     const int periodic = 0;
59     MPI_Cart_create(MPI_COMM_WORLD, 1, &this->proc_count, &periodic, 1, &this->global_comm);
60     MPI_Cart_coords(this->global_comm, this->proc_rank, 1, &this->proc_coord);
61     MPI_Cart_rank(this->global_comm, &this->proc_coord, &this->proc_rank);
62
63     determine_neighbours(this);
64
65     int dims_without_edges[DIMS];
66     cube_without_edges(data, dims_without_edges);
67     calculate_cuts(this, dims_without_edges);
68
69     this->data_per_proc = get_data_per_proc(dims_without_edges[X], pm_count(this));
70     this->last_data_count = get_last_cut_size(dims_without_edges[X], pm_count(this));
71 }
72
73 void pm_free(ProcMeta* this) {
74     MPI_Comm_free(&this->global_comm);
75 }
76
77 int pm_neighbour(const ProcMeta* this, int index) {
78     return this->neighbours[index];
79 }
80
81 MPI_Comm pm_comm(const ProcMeta* this) {
82     return this->global_comm;
83 }
84
85 int pm_coord(const ProcMeta* this) {
86     return this->proc_coord;
87 }
88
89 int pm_root(const ProcMeta* this) {
90     return get_cart_rank(this, 0);
91 }
92
93 int pm_rank(const ProcMeta* this) {
94     return this->proc_rank;
95 }
96
97 bool pm_is_root(const ProcMeta* this) {
98     return (pm_rank(this) == pm_root(this));
99 }
100
101 int pm_count(const ProcMeta* this) {
102     return this->proc_count;
103 }
104
105 int pm_disc(const ProcMeta* this, int coord, int local_coord) {
106     return this->task_coords[coord] + 1 + local_coord;
107 }
```

C proc\_meta.c X

```
108
109 int pm_disc_x(const ProcMeta* this, int local_x) {
110     return pm_disc(this, X, local_x);
111 }
112
113 int pm_disc_y(const ProcMeta* this, int local_y) {
114     return pm_disc(this, Y, local_y);
115 }
116
117 int pm_disc_z(const ProcMeta* this, int local_z) {
118     return pm_disc(this, Z, local_z);
119 }
120
121 int pm_disc_len(const ProcMeta* this, int coord) {
122     return this->task_dims[coord];
123 }
124
125 int pm_disc_len_x(const ProcMeta* this) {
126     return pm_disc_len(this, X);
127 }
128
129 int pm_disc_len_y(const ProcMeta* this) {
130     return pm_disc_len(this, Y);
131 }
132
133 int pm_disc_len_z(const ProcMeta* this) {
134     return pm_disc_len(this, Z);
135 }
```

C problem\_data.h ×

```
1  #ifndef LAB4_PROBLEM_DATA_H
2  #define LAB4_PROBLEM_DATA_H
3
4  #include "types.h"
5  #include "local_problem.h"
6
7  #define X_0 (-1)
8  #define Y_0 (-1)
9  #define Z_0 (-1)
10
11 #define D_X 2
12 #define D_Y 2
13 #define D_Z 2
14
15 #define A 100000
16
17 #define EPSILON 0.00000001
18
19 #define N_X 300
20 #define N_Y 300
21 #define N_Z 300
22
23 double phi(double x, double y, double z);
24 double rho(double x, double y, double z);
25
26 double pd_h(const ProblemData* this, int coord);
27 double pd_h_x(const ProblemData* this);
28 double pd_h_y(const ProblemData* this);
29 double pd_h_z(const ProblemData* this);
30
31 double pd_area(const ProblemData* this, int coord, int index);
32 double pd_area_x(const ProblemData* this, int x);
33 double pd_area_y(const ProblemData* this, int y);
34 double pd_area_z(const ProblemData* this, int z);
35
36 int pd_disc(const ProblemData* this, int coord);
37 int pd_disc_x(const ProblemData* this);
38 int pd_disc_y(const ProblemData* this);
39 int pd_disc_z(const ProblemData* this);
40
41 double pd_a(const ProblemData* this);
42 double pd_factor(const ProblemData* this);
43
44 double pd_func(const ProblemData* this, const int discrete_point[DIMS], func_r3 func);
45 double pd_rho_ar(const ProblemData* this, const int discrete_point[DIMS]);
46 double pd_rho(const ProblemData* this, int x, int y, int z);
47 double pd_phi_ar(const ProblemData* this, const int discrete_point[DIMS]);
48 double pd_phi(const ProblemData* this, int x, int y, int z);
49
50 void pd_init(ProblemData* this);
51
52 #endif // !LAB4_PROBLEM_DATA_H
```

C problem\_data.c ×

```
1  #include "problem_data.h"
2
3  double pd_h(const ProblemData* this, int coord) {
4      return this->local_area[coord];
5  }
6
7  double pd_h_x(const ProblemData* this) {
8      return pd_h(this, X);
9  }
10
11 double pd_h_y(const ProblemData* this) {
12     return pd_h(this, Y);
13 }
14
15 double pd_h_z(const ProblemData* this) {
16     return pd_h(this, Z);
17 }
18
19 double pd_a(const ProblemData* this) {
20     return this->a;
21 }
22
23 double pd_factor(const ProblemData* this) {
24     return 1 / (
25         2 / (pd_h_x(this) * pd_h_x(this)) +
26         2 / (pd_h_y(this) * pd_h_y(this)) +
27         2 / (pd_h_z(this) * pd_h_z(this)) +
28         pd_a(this)
29     );
30 }
31
32 double pd_area(const ProblemData* this, int coord, int index) {
33     return this->area_offset[coord] + index * pd_h(this, coord);
34 }
35
36 double pd_area_x(const ProblemData* this, int x) {
37     return pd_area(this, X, x);
38 }
39
40 double pd_area_y(const ProblemData* this, int y) {
41     return pd_area(this, Y, y);
42 }
43
44 double pd_area_z(const ProblemData* this, int z) {
45     return pd_area(this, Z, z);
46 }
47
48 double pd_func(const ProblemData* this, const int discrete_point[DIMS], func_r3 func) {
49     const double x = pd_area_x(this, discrete_point[X]);
50     const double y = pd_area_y(this, discrete_point[Y]);
51     const double z = pd_area_z(this, discrete_point[Z]);
52
53     return func(x, y, z);
54 }
```

C problem\_data.c X

```
55
56 double pd_rho_ar(const ProblemData* this, const int discrete_point[DIMS]) {
57     return pd_func(this, discrete_point, this->rho);
58 }
59
60 double pd_rho(const ProblemData* this, int x, int y, int z) {
61     const int discrete_point[DIMS] = {x, y, z};
62     return pd_func(this, discrete_point, this->rho);
63 }
64
65 double pd_phi_ar(const ProblemData* this, const int discrete_point[DIMS]) {
66     return pd_func(this, discrete_point, this->phi);
67 }
68
69 double pd_phi(const ProblemData* this, int x, int y, int z) {
70     const int discrete_point[DIMS] = {x, y, z};
71     return pd_func(this, discrete_point, this->phi);
72 }
73
74 void count_local_area(double local_area[DIMS], double global_area[DIMS], int dimensions[DIMS]) {
75     for (int i = 0; i < DIMS; ++i) {
76         if (dimensions[i] == 0)
77             local_area[i] = 0;
78         local_area[i] = global_area[i] /
79             (dimensions[i] - 1);
80     }
81 }
82
83 void pd_init(ProblemData* this) {
84     this->area_offset[X] = X_0;
85     this->area_offset[Y] = Y_0;
86     this->area_offset[Z] = Z_0;
87
88     this->global_area[X] = D_X;
89     this->global_area[Y] = D_Y;
90     this->global_area[Z] = D_Z;
91
92     this->a = A;
93
94     this->phi = phi;
95     this->rho = rho;
96
97     this->discrete_dimensions[X] = N_X;
98     this->discrete_dimensions[Y] = N_Y;
99     this->discrete_dimensions[Z] = N_Z;
100
101     count_local_area(this->local_area, this->global_area, this->discrete_dimensions);
102 }
103
104 double phi(double x, double y, double z) {
105     return x * x + y * y + z * z;
106 }
107
108 double rho(double x, double y, double z) {
109     return 6 - A * phi(x, y, z);
110 }
```

C problem\_data.c ×

```
111
112 int pd_disc(const ProblemData* this, int coord) {
113     return this->discrete_dimensions[coord];
114 }
115
116 int pd_disc_x(const ProblemData* this) {
117     return pd_disc(this, X);
118 }
119
120 int pd_disc_y(const ProblemData* this) {
121     return pd_disc(this, Y);
122 }
123
124 int pd_disc_z(const ProblemData* this) {
125     return pd_disc(this, Z);
126 }
```

C matrix\_3d.h ×

```
1  #ifndef LAB4_TENSOR_H
2  #define LAB4_TENSOR_H
3
4  #include "types.h"
5
6  void mt_init(Matrix3D* this, const int dimensions[DIMS]);
7
8  int mt_len(const Matrix3D* this, int coord);
9  int mt_x_len(const Matrix3D* this);
10 int mt_y_len(const Matrix3D* this);
11 int mt_z_len(const Matrix3D* this);
12
13 double mt_get(const Matrix3D* this, int x, int y, int z);
14 const double* mt_get_ref(const Matrix3D* this, int x, int y, int z);
15 double* mt_set(Matrix3D* this, int x, int y, int z);
16
17 double mt_get_ar(const Matrix3D* this, const int dimensions[DIMS]);
18 const double* mt_get_ref_ar(const Matrix3D* this, const int dimensions[DIMS]);
19 double* mt_set_ar(Matrix3D* this, const int dimensions[DIMS]);
20
21 double mt_get_by_i(const Matrix3D* this, int index);
22 const double* mt_get_ref_by_i(const Matrix3D* this, int index);
23 double* mt_set_by_i(Matrix3D* this, int index);
24
25 void mt_free(Matrix3D* this);
26
27 #endif // !LAB4_TENSOR_H
```

C matrix\_3d.c X

```
1  #include "matrix_3d.h"
2
3  #include <stdlib.h>
4  #include <string.h>
5
6  int get_index(const Matrix3D* this, int x, int y, int z) {
7      return mt_y_len(this) * mt_z_len(this) * x
8          + mt_z_len(this) * y
9          + z;
10 }
11
12 double mt_get(const Matrix3D* this, int x, int y, int z) {
13     return mt_get_by_i(this, get_index(this, x, y, z));
14 }
15
16 double mt_get_ar(const Matrix3D* this, const int dimensions[DIMS]) {
17     return mt_get(this, dimensions[X], dimensions[Y], dimensions[Z]);
18 }
19
20 const double* mt_get_ref(const Matrix3D* this, int x, int y, int z) {
21     return mt_get_ref_by_i(this, get_index(this, x, y, z));
22 }
23
24 const double* mt_get_ref_ar(const Matrix3D* this, const int dimensions[DIMS]) {
25     return mt_get_ref(this, dimensions[X], dimensions[Y], dimensions[Z]);
26 }
27
28 double* mt_set(Matrix3D* this, int x, int y, int z) {
29     return mt_set_by_i(this, get_index(this, x, y, z));
30 }
31
32 double* mt_set_ar(Matrix3D* this, const int dimensions[DIMS]) {
33     return mt_set(this, dimensions[X], dimensions[Y], dimensions[Z]);
34 }
35
36 double mt_get_by_i(const Matrix3D* this, int index) {
37     return this->data[index];
38 }
39
40 const double* mt_get_ref_by_i(const Matrix3D* this, int index) {
41     return &this->data[index];
42 }
43
44 double* mt_set_by_i(Matrix3D* this, int index) {
45     return &this->data[index];
46 }
47
48 void mt_init(Matrix3D* this, const int dimensions[DIMS]) {
49     memcpy(this->dimensions, dimensions, DIMS * sizeof *this->dimensions);
50     this->data = calloc(mt_x_len(this) * mt_y_len(this) * mt_z_len(this), sizeof *this->data);
51 }
52
53 int mt_len(const Matrix3D* this, int coord) {
54     return this->dimensions[coord];
55 }
```

C matrix\_3d.c X

```
56
57 int mt_x_len(const Matrix3D* this) {
58     return mt_len(this, X);
59 }
60
61 int mt_y_len(const Matrix3D* this) {
62     return mt_len(this, Y);
63 }
64
65 int mt_z_len(const Matrix3D* this) {
66     return mt_len(this, Z);
67 }
68
69 void mt_free(Matrix3D* this) {
70     free(this->data);
71 }
```

C local\_problem.h X

```
1  #ifndef LAB4_LOCAL_PROBLEM_H
2  #define LAB4_LOCAL_PROBLEM_H
3
4  #include "types.h"
5
6  #include "matrix_3d.h"
7  #include "problem_data.h"
8
9  void lp_init(LocalProblem* this, const ProblemData* data, const ProcMeta* meta);
10 void lp_free(LocalProblem* this);
11
12 void lp_iterate(LocalProblem* this);
13 void lp_swap_cubes(LocalProblem* this);
14 void lp_finish_plane(LocalProblem* this, int plane_index);
15
16 Matrix3D* lp_old_mat(LocalProblem* this);
17 Matrix3D* lp_new_mat(LocalProblem* this);
18
19 const Matrix3D* lp_old_mat_c(const LocalProblem* this);
20 const Matrix3D* lp_new_mat_c(const LocalProblem* this);
21
22 double* lp_in_plane(LocalProblem* this, int index);
23
24 double* lp_out_plane(LocalProblem* this, int index);
25 const double* lp_out_plane_c(const LocalProblem* this, int index);
26
27 #endif // !LAB4_LOCAL_PROBLEM_H
```



C local\_problem.c x

```
51 void lp_finish_plane(LocalProblem* this, int plane_index) {
52     const int x = this->plane_neighbour_x_coords[plane_index];
53     for (int y = this->inner_offset[Y]; y < this->inner_dims[Y] + this->inner_offset[Y]; ++y) {
54         for (int z = this->inner_offset[Z]; z < this->inner_dims[Z] + this->inner_offset[Z]; ++z) {
55             update_solution(this, x, y, z);
56         }
57     }
58 }
59
60 void lp_iterate(LocalProblem* this) {
61     for (int x = this->plane_neighbour_x_coords[NB_LEFT] + 1; x < this->plane_neighbour_x_coords[NB_RIGHT]; ++x) {
62         for (int y = this->inner_offset[Y]; y < this->inner_dims[Y] + this->inner_offset[Y]; ++y) {
63             for (int z = this->inner_offset[Z]; z < this->inner_dims[Z] + this->inner_offset[Z]; ++z) {
64                 update_solution(this, x, y, z);
65             }
66         }
67     }
68 }
69
70 void init_constants(Constants* this, const ProblemData* data, const ProcMeta* meta, const int inner_dims[DIMS]) {
71     this->factor = pd_factor(data);
72
73     this->h_2[X] = pd_h_x(data) * pd_h_x(data);
74     this->h_2[Y] = pd_h_y(data) * pd_h_y(data);
75     this->h_2[Z] = pd_h_z(data) * pd_h_z(data);
76
77     mt_init(&this->rho, inner_dims);
78
79     for (int x = 0; x < inner_dims[X]; ++x) {
80         for (int y = 0; y < inner_dims[Y]; ++y) {
81             for (int z = 0; z < inner_dims[Z]; ++z) {
82                 *mt_set(&this->rho, x, y, z) = pd_rho(data,
83                     pm_disc_x(meta, x), pm_disc_y(meta, y), pm_disc_z(meta, z));
84             }
85         }
86     }
87 }
88 }
```

```
89 void init_edge_values(LocalProblem* this, const ProblemData* data, const ProcMeta* meta) {
90     int plane_coords[DIMS][NB_COUNT];
91     int other_coords[DIMS][2] = {{Y, Z}, {X, Z}, {Y, X}};
92
93     for (int coord = 0; coord < DIMS; ++coord) {
94         plane_coords[coord][NB_LEFT] = 0;
95         plane_coords[coord][NB_RIGHT] = this->slice_dims[coord] - 1;
96     }
97
98     int vec[DIMS] = {};
99     for (int coord = 0; coord < DIMS; ++coord) {
100         for (int pos = 0; pos < NB_COUNT; ++pos) {
101             vec[coord] = plane_coords[coord][pos];
102             const int x = other_coords[coord][0];
103             const int y = other_coords[coord][1];
104
105             for (vec[x] = 0; vec[x] < this->slice_dims[x]; ++vec[x]) {
106                 for (vec[y] = 0; vec[y] < this->slice_dims[y]; ++vec[y]) {
107                     *mt_set_ar(lp_old_mat(this), vec) = pd_phi(data,
108                         pm_disc_x(meta, vec[X] - this->inner_offset[X]),
109                         pm_disc_y(meta, vec[Y] - this->inner_offset[Y]),
110                         pm_disc_z(meta, vec[Z] - this->inner_offset[Z]));
111                     *mt_set_ar(lp_new_mat(this), vec) = pd_phi(data,
112                         pm_disc_x(meta, vec[X] - this->inner_offset[X]),
113                         pm_disc_y(meta, vec[Y] - this->inner_offset[Y]),
114                         pm_disc_z(meta, vec[Z] - this->inner_offset[Z]));
115                 }
116             }
117         }
118     }
119 }
```

C local\_problem.c X

```
1  #include "local_problem.h"
2
3  #include "proc_meta.h"
4
5  #include <string.h>
6
7  double get_h_2(const LocalProblem* this, int coord) {
8      return this->constants.h_2[coord];
9  }
10
11 double get_h_x_2(const LocalProblem* this) {
12     return get_h_2(this, X);
13 }
14
15 double get_h_y_2(const LocalProblem* this) {
16     return get_h_2(this, Y);
17 }
18
19 double get_h_z_2(const LocalProblem* this) {
20     return get_h_2(this, Z);
21 }
22
23 const Matrix3D* get_rho(const LocalProblem* this) {
24     return &this->constants.rho;
25 }
26
27 double get_factor(const LocalProblem* this) {
28     return this->constants.factor;
29 }
30
31 void lp_swap_cubes(LocalProblem* this) {
32     int tmp = this->old;
33     this->old = this->new;
34     this->new = tmp;
35 }
36
37 void update_solution(LocalProblem* this, int x, int y, int z) {
38     const Matrix3D* old = lp_old_mat(this);
39     Matrix3D* new = lp_new_mat(this);
40
41     *mt_set(new, x, y, z) = get_factor(this) * (
42         (mt_get(old, x + 1, y, z) + mt_get(old, x - 1, y, z)) / get_h_x_2(this) +
43         (mt_get(old, x, y + 1, z) + mt_get(old, x, y - 1, z)) / get_h_y_2(this) +
44         (mt_get(old, x, y, z + 1) + mt_get(old, x, y, z - 1)) / get_h_z_2(this) -
45         mt_get(get_rho(this),
46             x - this->inner_offset[X],
47             y - this->inner_offset[Y],
48             z - this->inner_offset[Z]));
49 }
50
```

C local\_problem.c X

```
120
121 void lp_init(LocalProblem* this, const ProblemData* data, const ProcMeta* meta) {
122     memcpy(this->inner_dims, meta->task_dims, DIMS * sizeof *this->inner_dims);
123     memcpy(this->slice_dims, meta->task_dims, DIMS * sizeof *this->slice_dims);
124
125     for (int i = 0; i < DIMS; ++i) {
126         this->slice_dims[i] += 2;
127         this->inner_offset[i] = 1;
128     }
129
130     init_constants(&this->constants, data, meta, this->inner_dims);
131
132     for (int i = 0; i < 2; ++i) {
133         mt_init(&this->slice[i], this->slice_dims);
134     }
135
136     this->old = 0;
137     this->new = 1;
138
139     init_edge_values(this, data, meta);
140
141     this->plane_x_coords[NB_LEFT] = 0;
142     this->plane_x_coords[NB_RIGHT] = this->slice_dims[X] - 1;
143
144     this->plane_neighbour_x_coords[NB_LEFT] = this->inner_offset[X];
145     this->plane_neighbour_x_coords[NB_RIGHT] = this->slice_dims[X] - 2;
146 }
147
148 void lp_free(LocalProblem* this) {
149     mt_free(&this->constants.rho);
150     for (int i = 0; i < 2; ++i) {
151         mt_free(&this->slice[i]);
152     }
153 }
154
155 Matrix3D* lp_old_mat(LocalProblem* this) {
156     return &this->slice[this->old];
157 }
158
159 Matrix3D* lp_new_mat(LocalProblem* this) {
160     return &this->slice[this->new];
161 }
162
163 const Matrix3D* lp_old_mat_c(const LocalProblem* this) {
164     return &this->slice[this->old];
165 }
166
167 const Matrix3D* lp_new_mat_c(const LocalProblem* this) {
168     return &this->slice[this->new];
169 }
170
171 double* lp_in_plane(LocalProblem* this, int index) {
172     return mt_set(lp_old_mat(this), this->plane_x_coords[index], this->inner_offset[Y], this->inner_offset[Z]);
173 }
```

C local\_problem.c X

```
174
175 double* lp_out_plane(LocalProblem* this, int index) {
176     return mt_set(lp_new_mat(this), this->plane_neighbour_x_coords[index], this->inner_offset[Y], this->inner_offset[Z]);
177 }
178
179 const double* lp_out_plane_c(const LocalProblem* this, int index) {
180     return mt_get_ref(lp_new_mat_c(this), this->plane_neighbour_x_coords[index], this->inner_offset[Y], this->inner_offset[Z]);
181 }
182
```

## Приложение 2. Скрипт для PBS.

```
#!/bin/bash

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

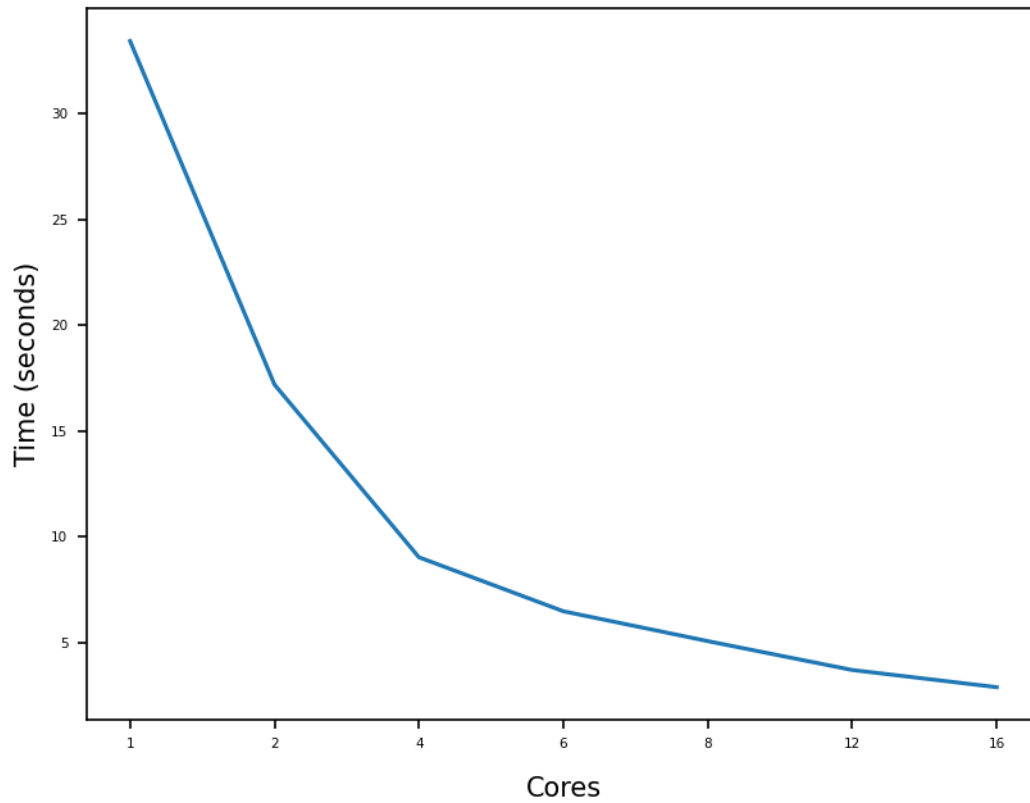
cd $PBS_O_WORKDIR

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

P0	Application	MPI_Isend	MPI_Irecv	Application
P1	MPI_Bcast	MPI_Waitall	MPI_Isend	MPI_Irecv
P2	MPI_Waitall	MPI_Isend	MPI_Isend	MPI_Irecv
P3	MPI_Bcast	MPI_Waitall	MPI_Isend	MPI_Irecv
P4	MPI_Waitall	MPI_Isend	MPI_Isend	MPI_Irecv
P5	MPI_Bcast	MPI_Waitall	MPI_Isend	MPI_Irecv
P6	MPI_Waitall	MPI_Isend	MPI_Isend	MPI_Irecv
P7	MPI_Bcast	MPI_Waitall	MPI_Isend	MPI_Irecv
P8	MPI_Waitall	MPI_Isend	MPI_Isend	MPI_Irecv
P9	MPI_Bcast	MPI_Waitall	MPI_Isend	MPI_Irecv
P10	MPI_Waitall	MPI_Isend	MPI_Isend	MPI_Irecv
P11	MPI_Bcast	MPI_Waitall	MPI_Isend	MPI_Irecv
P12	MPI_Waitall	MPI_Isend	MPI_Isend	MPI_Irecv
P13	MPI_Bcast	MPI_Waitall	MPI_Isend	MPI_Irecv
P14	MPI_Waitall	MPI_Isend	MPI_Isend	MPI_Irecv
P15	MPI_Bcast	MPI_Waitall	MPI_Isend	MPI_Irecv

P0	Application	MPI_Waitany	Application	MPI_Gather
P1	Application	Application	MPI_Bcast	Application
P2	Application	Application	Application	MPI_Bcast
P3	Application	MPI_Waitany	Application	MPI_Bcast
P4	Application	MPI_Waitany	MPI_Gather	MPI_Bcast
P5	Application	MPI_Waitany	MPI_Bcast	Application
P6	Application	Application	Application	MPI_Bcast
P7	Application	Application	MPI_Bcast	Application
P8	Application	MPI_Waitany	Application	MPI_Gather
P9	Application	MPI_Waitany	Application	MPI_Bcast
P10	Application	Application	MPI_Gather	Application
P11	Application	MPI_Waitany	Application	MPI_Bcast
P12	Application	MPI_Waitany	MPI_Gather	MPI_Bcast
P13	Application	Application	Application	MPI_Bcast
P14	Application	MPI_Waitany	MPI_Gather	MPI_Bcast
P15	Application	MPI_Waitany	Application	MPI_Bcast