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 решения системы линейных алгебраических уравнений с помощью MPI»

студента 2 курса, 19202 группы

Ивакина Александра Олеговича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
к.т.н., доцент
А.Ю. Власенко

Новосибирск 2021

СОДЕРЖАНИЕ

ЦЕЛЬ

Изучить методы распараллеливания операций над матрицами и векторами используя технологию MPI.

ЗАДАНИЕ

Составить три версии программы, решающие систему линейных уравнений с разной степенью распараллеливания.

ОПИСАНИЕ РАБОТЫ

График зависимости времени от числа ядер для первой версии:

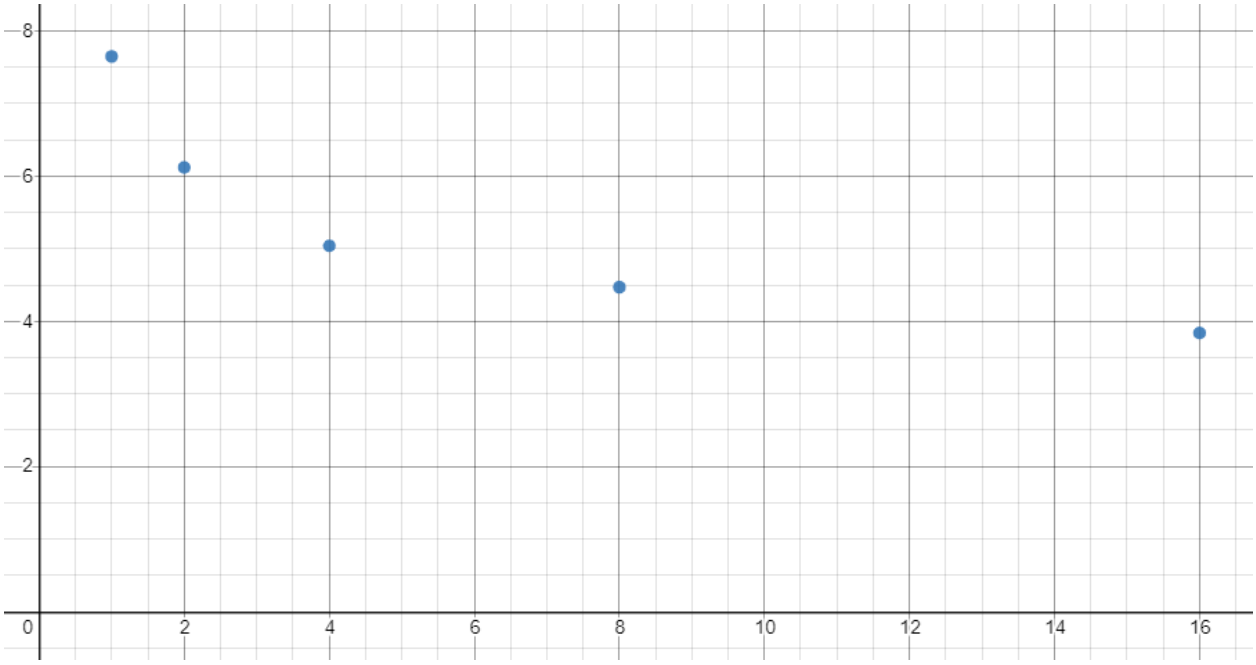


График зависимости времени от числа ядер для второй версии:

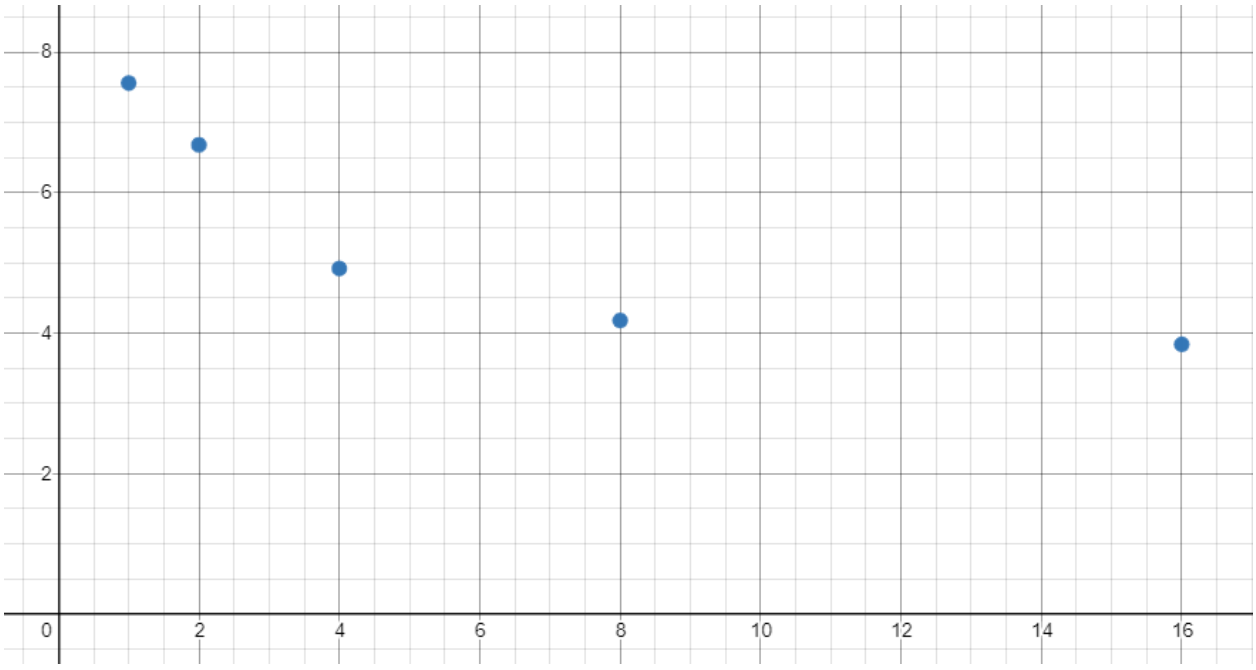


График зависимости ускорения от числа ядер для первой версии:

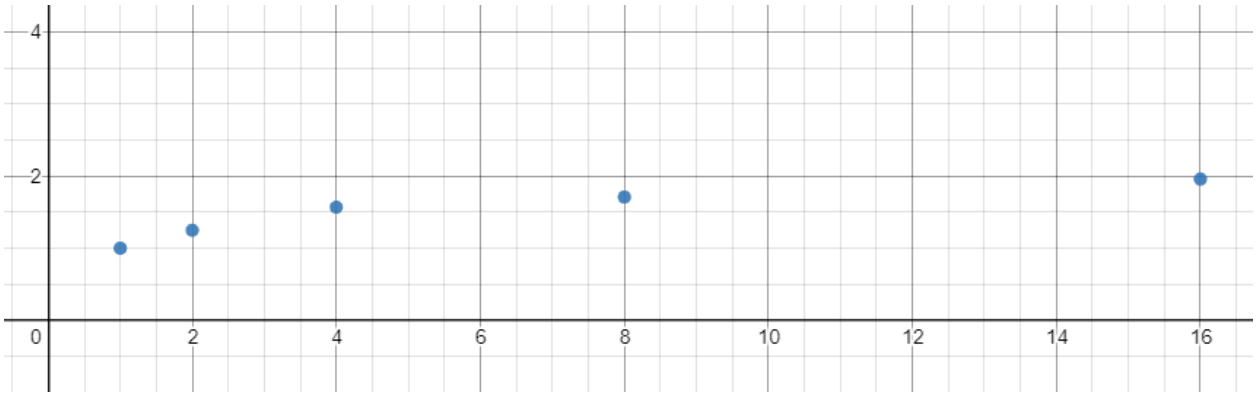


График зависимости ускорения от числа ядер для второй версии:

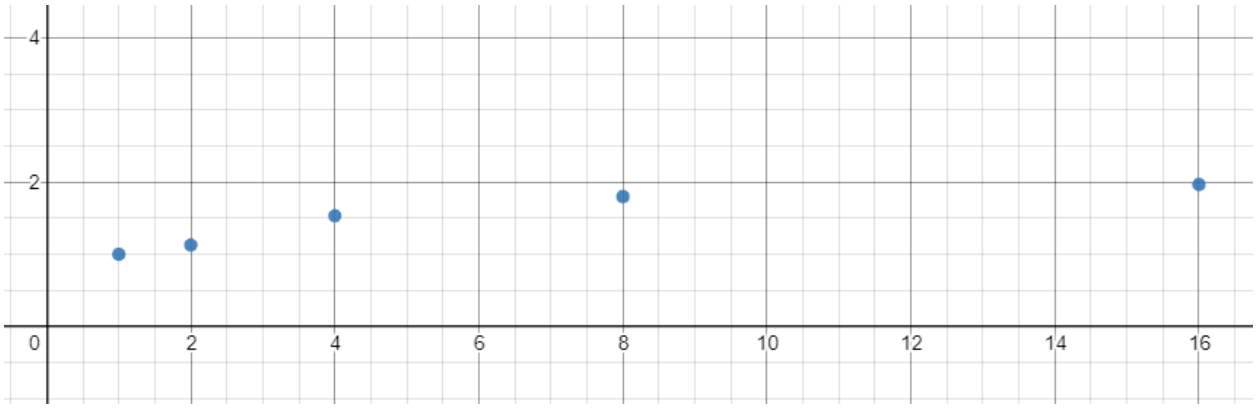


График зависимости эффективности от числа ядер для первой версии:

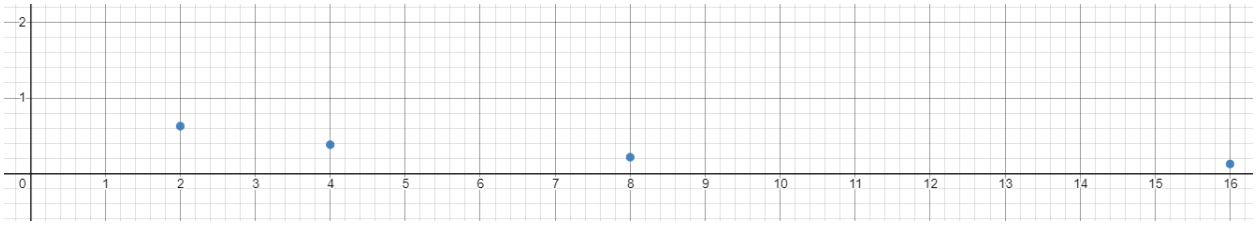
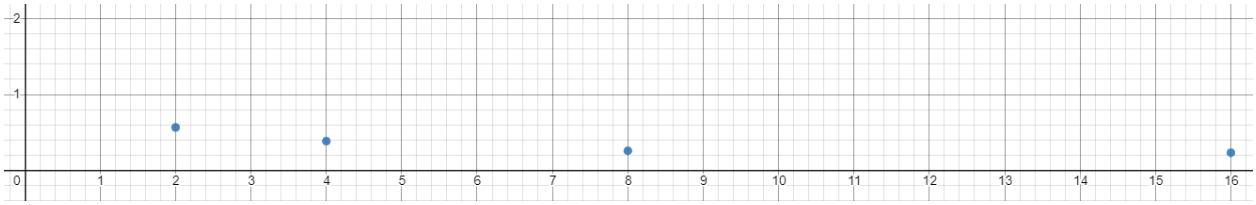
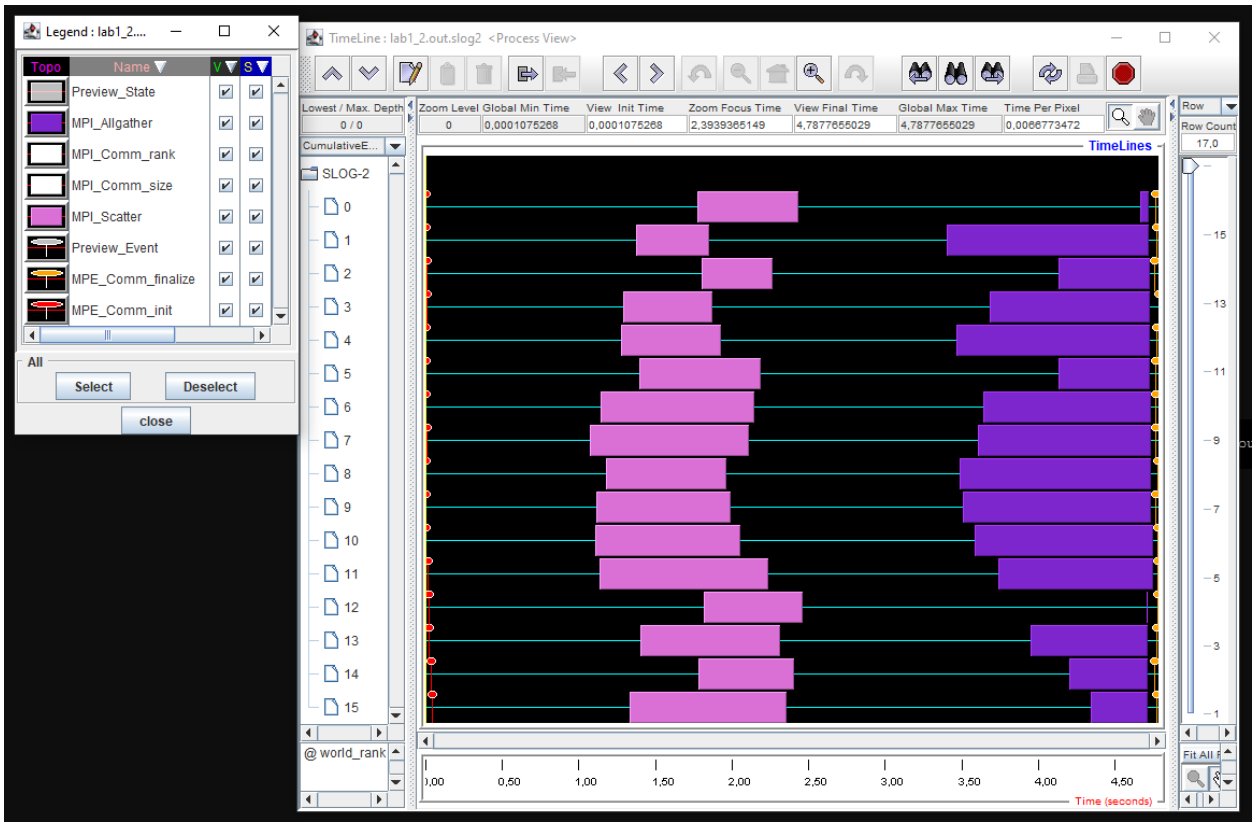


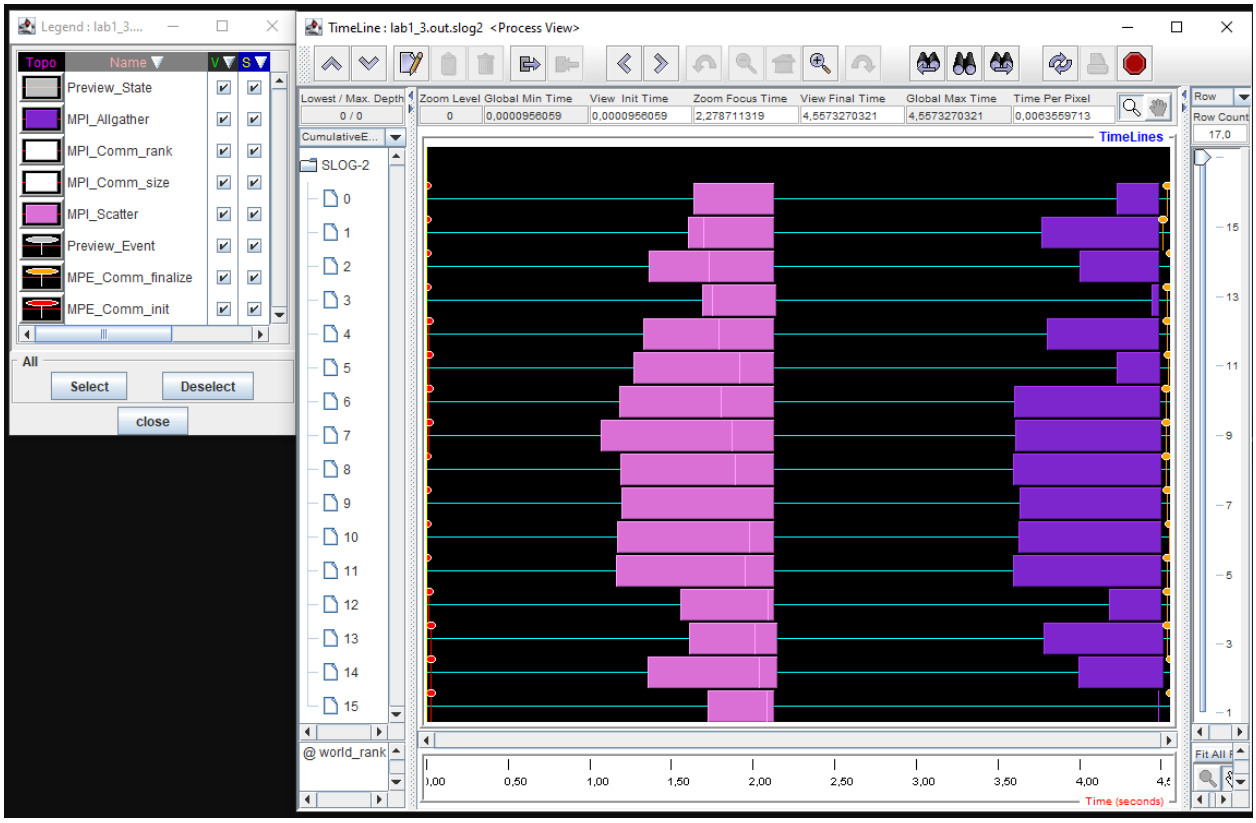
График зависимости эффективности от числа ядер для второй версии:



Профилирование для первой версии:



Профилирование для второй версии:



ЗАКЛЮЧЕНИЕ

Была изучена технология MPI, рассмотрены методы распараллеливания работы численных программ.

Приложение 1. Код программы

Первая версия

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

void print_matrix(double* matrix, int n) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
printf("%f ", matrix[i*n + j]);
}
printf("\n");
}
}

void print_vector(double* vector, int n) {
for (int i = 0; i < n; i++) {
printf("%f ", vector[i]);
}
printf("\n");
}

double rand_double(double min, double max) {
return min + (rand() / (RAND_MAX / (max - min)));
}

void make_zero_matrix(double* matrix, int n) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
matrix[i*n + j] = 0;
}
}
}

void matrix_copy(double* matrix1, double* matrix2, int n) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
matrix1[i*n + j] = matrix2[i*n + j];
}
}
}

void vector_copy(double* vector1, double* vector2, int n) {
for (int i = 0; i < n; i++) {
vector1[i] = vector2[i];
}
}

void make_one_two_matrix(double* matrix, int n) {
```

```
48.for (int i = 0; i < n; i++) {
49.for (int j = 0; j < n; j++) {
50.if (i == j) {
51.matrix[i*n + j] = 2;
52.}
53.else {
54.matrix[i*n + j] = 1;
55.}
56.}
57.}
58.}
59.
60.void make_random_matrix(double* matrix, int n) {
61.for (int i = 0; i < n; i++) {
62.for (int j = i; j < n; j++) {
63.matrix[i*n + j] = rand_double(0, 1);
64.if (i == j) {
65.matrix[i*n + j] += 1000;
66.}
67.else {
68.matrix[j*n + i] = matrix[i*n + j];
69.}
70.}
71.}
72.}
73.
74.double vec_len(double* vector, int n) {
75.double len = 0;
76.for (int i = 0; i < n; i++) {
77.len += vector[i] * vector[i];
78.}
79.
80.return sqrt(len);
81.}
82.
83.double scalar_mul(double* vector1, double* vector2, int n) {
84.double result = 0;
85.for (int i = 0; i < n; i++) {
86.result += vector1[i] * vector2[i];
87.}
88.
89.return result;
90.}
91.
92.void matrix_mul_vector(double* matrix, double* vector, double*
result, int n) {
93.double vec[n];
94.for (int i = 0; i < n; i++) {
95.for (int j = 0; j < n; j++) {
96.vec[j] = matrix[i * n + j];
97.}
98.result[i] = scalar_mul(vec, vector, n);
99.}
```

```
100.}
101.
102.void matrix_mul_double(double* matrix, double dub, int n) {
103.for (int i = 0; i < n; i++) {
104.for (int j = 0; j < n; j++) {
105.matrix[i*n + j] *= dub;
106.}
107.}
108.}
109.
110.void vector_mul_double(double* vector, double dub, int n) {
111.for (int i = 0; i < n; i++) {
112.vector[i] *= dub;
113.}
114.}
115.
116.void vector_minus_vector(double* vector1, double* vector2, int
n) {
117.for (int i = 0; i < n; i++) {
118.vector1[i] -= vector2[i];
119.}
120.}
121.
122.int is_finished(double* matrix, double* vector, double* b,
double epsilon, int n) {
123.
124.double* A = (double*)malloc(n * n * sizeof(double));
125.make_zero_matrix(A, n);
126.
127.//write(0, "He\n", 3);
128.matrix_copy(A, matrix, n);
129.double* Ax = (double*)malloc(n * sizeof(double));
130.
131.matrix_mul_vector(A, vector, Ax, n);
132.vector_minus_vector(Ax, b, n);
133.
134.if ((sqrt(scalar_mul(Ax, Ax, n)) / sqrt(scalar_mul(b, b, n))) <
epsilon) {
135.free(A);
136.free(Ax);
137.return 1;
138.}
139.
140.free(A);
141.free(Ax);
142.return 0;
143.}
144.
145.void count_y(double* y, double* A, double* x_n, double* b, int
n){
146.matrix_mul_vector(A, x_n, y, n);
147.
148.vector_minus_vector(y, b, n);
```

```
149.}
150.
151.double count_tau(double* A, double* y, int n) {
152.double* Ay = (double*)malloc(n * sizeof(double));
153.matrix_mul_vector(A, y, Ay, n);
154.
155.double tau = (scalar_mul(y, Ay, n) / scalar_mul(Ay, Ay, n));
156.
157.free(Ay);
158.return tau;
159.}
160.
161.double* solve_eq(double* matrix, double* values, int n) {
162.double epsilon = 0.001;
163.
164.double* y = (double*)malloc(n * sizeof(double));
165.double* x_i = (double*)malloc(n * sizeof(double));
166.vector_copy(x_i, values, n);
167.
168.while (!is_finished(matrix, x_i, values, epsilon, n)) {
169.//print_vector(x_i, n);
170.count_y(y, matrix, x_i, values, n);
171.
172.double tau = count_tau(matrix, y, n);
173.
174.vector_mul_double(y, tau, n);
175.
176.vector_minus_vector(x_i, y, n);
177./*for (int i = 0; i < n; i++) {
178.
179. }*/
180.}
181.
182.//printf(")\n");
183.free(y);
184.
185.return x_i;
186.}
187.
188.int main() {
189.// SETTING UP
190.srand(time(NULL));
191.int n;
192.printf("Enter size of matrix: ");
193.scanf("%d", &n);
194.
195.double* matrix = (double*)malloc(n * n * sizeof(double));
196.make_zero_matrix(matrix, n);
197.
198.//make_one_two_matrix(matrix, n);
199.make_random_matrix(matrix, n);
200.//print_matrix(matrix, n);
201.
```

```
202.double* values = (double*)malloc(n * sizeof(double));
203.for (int i = 0; i < n; i++) {
204.values[i] = n + 1;
205.}
206.
207.// ACTION
208.double* x_n = solve_eq(matrix, values, n);
209.
210.for (int i = 0; i < n; i++) {
211.printf("%f ", x_n[i]);
212.}
213.printf("\n");
214.
215.// FREEING
216.free(values);
217.free(x_n);
218.free(matrix);
219.return 0;
220.}
221.
```

Вторая версия

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

void print_matrix(double* matrix, int n) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
printf("%f ", matrix[i*n + j]);
}
printf("\n");
}
}

void print_vector(double* vector, int n) {
for (int i = 0; i < n; i++) {
printf("%f ", vector[i]);
}
printf("\n");
}

double rand_double(double min, double max) {
return min + (rand() / (RAND_MAX / (max - min)));
}

void make_zero_matrix(double* matrix, int n) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
matrix[i*n + j] = 0;
}
}
}

void matrix_copy(double* matrix1, double* matrix2, int n) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
matrix1[i*n + j] = matrix2[i*n + j];
}
}
}

void vector_copy(double* vector1, double* vector2, int n) {
for (int i = 0; i < n; i++) {
vector1[i] = vector2[i];
}
}
```

```
48.
49.void make_one_two_matrix(double* matrix, int n) {
50.for (int i = 0; i < n; i++) {
51.for (int j = 0; j < n; j++) {
52.if (i == j) {
53.matrix[i*n + j] = 2;
54.}
55.else {
56.matrix[i*n + j] = 1;
57.}
58.}
59.}
60.}
61.
62.void make_one_matrix(double* matrix, int n) {
63.for (int i = 0; i < n; i++) {
64.for (int j = 0; j < n; j++) {
65.if (i == j) {
66.matrix[i*n + j] = 1;
67.}
68.else {
69.matrix[i*n + j] = 0;
70.}
71.}
72.}
73.}
74.
75.void make_random_matrix(double* matrix, int n) {
76.for (int i = 0; i < n; i++) {
77.for (int j = i; j < n; j++) {
78.matrix[i*n + j] = rand_double(0, 1);
79.if (i == j) {
80.matrix[i*n + j] += 1000;
81.}
82.else {
83.matrix[j*n + i] = matrix[i*n + j];
84.}
85.}
86.}
87.}
88.
89.double vec_len(double* vector, int n) {
90.double len = 0;
91.for (int i = 0; i < n; i++) {
92.len += vector[i] * vector[i];
93.}
94.
95.return sqrt(len);
96.}
97.
98.double scalar_mul_part(double* vector1, double* vector2, int n,
size_t proc_num, size_t rank) {
99.double result = 0;
```

```
100.double result_tmp = 0;
101.size_t lines = n / proc_num;
102.size_t offset = lines * rank;
103.
104.for (int i = 0; i < lines; i++) {
105.result_tmp += vector1[i + offset] * vector2[i + offset];
106.}
107.
108.MPI_Allreduce(&result_tmp, &result, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
109.
110.return result;
111.}
112.
113.double scalar_mul(double* vector1, double* vector2, int n,
size_t proc_num, size_t rank) {
114.double result = 0;
115.
116.for (int i = 0; i < n; i++) {
117.result += vector1[i] * vector2[i];
118.}
119.
120.return result;
121.}
122.
123.
124.void matrix_mul_vector(double* matrix, double* vector, double*
result, int n, size_t proc_num, size_t rank) {
125.size_t lines = n / proc_num;
126.size_t offset = lines * rank;
127.
128.double result_tmp[n];
129.
130.for (int i = 0; i < lines; i++) {
131.result_tmp[i] = 0;
132.for (int j = 0; j < n; j++) {
133.result_tmp[i] += matrix[i * n + j] * vector[j];
134.}
135.}
136.
137.MPI_Allgather(result_tmp, lines, MPI_DOUBLE, result, lines,
MPI_DOUBLE, MPI_COMM_WORLD);
138.}
139.
140.void vector_mul_double(double* vector, double dub, int n) {
141.for (int i = 0; i < n; i++) {
142.vector[i] *= dub;
143.}
144.}
145.
146.void vector_minus_vector(double* vector1, double* vector2, int
n) {
147.for (int i = 0; i < n; i++) {
```

```
148.vector1[i] -= vector2[i];
149.}
150.}
151.
152.int is_finished(double* matrix, double* vector, double* b,
double epsilon, int n, size_t proc_num, size_t rank) {
153.
154.double* A = (double*)malloc(n * n * sizeof(double));
155.make_zero_matrix(A, n);
156.
157.matrix_copy(A, matrix, n);
158.double* Ax = (double*)malloc(n * sizeof(double));
159.
160.matrix_mul_vector(A, vector, Ax, n, proc_num, rank);
161.vector_minus_vector(Ax, b, n);
162.
163.if ((vec_len(Ax, n) / vec_len(b, n)) < epsilon) {
164.free(A);
165.free(Ax);
166.return 1;
167.}
168.
169.free(A);
170.free(Ax);
171.return 0;
172.}
173.
174.void count_y(double* y, double* A, double* x_n, double* b, int
n, size_t proc_num, size_t rank){
175.matrix_mul_vector(A, x_n, y, n, proc_num, rank);
176.
177.vector_minus_vector(y, b, n);
178.}
179.
180.double count_tau(double* A, double* y, int n, size_t proc_num,
size_t rank) {
181.double* Ay = (double*)malloc(n * sizeof(double));
182.matrix_mul_vector(A, y, Ay, n, proc_num, rank);
183.
184.double tau = (sqrt(scalar_mul(y, Ay, n, proc_num, rank)) /
sqrt(scalar_mul(Ay, Ay, n, proc_num, rank)));
185.
186.free(Ay);
187.return tau;
188.}
189.
190.double* solve_eq(double* matrix, double* values, int n, size_t
proc_num, size_t rank) {
191.double epsilon = 0.001;
192.
193.double* y = (double*)malloc(n * sizeof(double));
194.double* x_i = (double*)malloc(n * sizeof(double));
195.vector_copy(x_i, values, n);
```

```
196.
197.while (!is_finished(matrix, x_i, values, epsilon, n, proc_num,
rank)) {
198.count_y(y, matrix, x_i, values, n, proc_num, rank);
199.
200.double tau = count_tau(matrix, y, n, proc_num, rank);
201.
202.vector_mul_double(y, tau, n);
203.
204.vector_minus_vector(x_i, y, n);
205.
206.
207.}
208.
209.free(y);
210.
211.return x_i;
212.}
213.
214.int main(int argc, char** argv) {
215.// SETTING UP
216.MPI_Init(&argc, &argv);
217.int proc_num;
218.MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
219.int rank;
220.MPI_Comm_rank(MPI_COMM_WORLD, &rank);
221.
222.srand(time(NULL));
223.int n = 16;
224.
225.double* matrix = (double*)malloc(n * n * sizeof(double));
226.make_zero_matrix(matrix, n);
227.
228.//make_one_two_matrix(matrix, n);
229.//make_random_matrix(matrix, n);
230.make_one_matrix(matrix, n);
231.
232.double* values = (double*)malloc(n * sizeof(double));
233.for (int i = 0; i < n; i++) {
234.values[i] = n + 1;
235.}
236.
237.// MPI staff
238.size_t lines = n / proc_num;
239.double* matrix_part = (double*)malloc(lines * n *
sizeof(double));
240.MPI_Scatter(matrix, lines * n, MPI_DOUBLE, matrix_part, lines *
n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
241.
242.// ACTION
243.double* x_n = solve_eq(matrix_part, values, n, proc_num, rank);
244.
245.for (int i = 0; i < n; i++) {
```

```
246.printf("%f ", x_n[i]);
247.}
248.printf("\n");
249.
250.// FREEING
251.MPI_Finalize();
252.free(values);
253.free(x_n);
254.free(matrix);
255.return 0;
256.}
257.
```

Третья версия

```
#include <stdio.h>
1.#include <stdlib.h>
2.#include <math.h>
3.#include <time.h>
4.#include <string.h>
5.#include <mpi.h>
6.
7.void print_matrix(double* matrix, int n) {
8.for (int i = 0; i < n; i++) {
9.for (int j = 0; j < n; j++) {
10.printf("%f ", matrix[i*n + j]);
11.}
12.printf("\n");
13.}
14.}
15.
16.void print_vector(double* vector, int n) {
17.for (int i = 0; i < n; i++) {
18.printf("%f ", vector[i]);
19.}
20.printf("\n");
21.}
22.
23.double rand_double(double min, double max) {
24.return min + (rand() / (RAND_MAX / (max - min)));
25.}
26.
27.void make_zero_matrix(double* matrix, int n) {
28.for (int i = 0; i < n; i++) {
29.for (int j = 0; j < n; j++) {
30.matrix[i*n + j] = 0;
31.}
32.}
33.}
34.
35.void matrix_copy(double* matrix1, double* matrix2, int n) {
36.for (int i = 0; i < n; i++) {
37.for (int j = 0; j < n; j++) {
38.matrix1[i*n + j] = matrix2[i*n + j];
39.}
40.}
41.}
42.
43.void vector_copy(double* vector1, double* vector2, int n) {
44.for (int i = 0; i < n; i++) {
45.vector1[i] = vector2[i];
46.}
47.}
48.
49.void make_one_two_matrix(double* matrix, int n) {
```

```
50.for (int i = 0; i < n; i++) {
51.for (int j = 0; j < n; j++) {
52.if (i == j) {
53.matrix[i*n + j] = 2;
54.}
55.else {
56.matrix[i*n + j] = 1;
57.}
58.}
59.}
60.}
61.
62.void make_one_matrix(double* matrix, int n) {
63.for (int i = 0; i < n; i++) {
64.for (int j = 0; j < n; j++) {
65.if (i == j) {
66.matrix[i*n + j] = 1;
67.}
68.else {
69.matrix[i*n + j] = 0;
70.}
71.}
72.}
73.}
74.
75.void make_random_matrix(double* matrix, int n) {
76.for (int i = 0; i < n; i++) {
77.for (int j = i; j < n; j++) {
78.matrix[i*n + j] = rand_double(0, 1);
79.if (i == j) {
80.matrix[i*n + j] += 1000;
81.}
82.else {
83.matrix[j*n + i] = matrix[i*n + j];
84.}
85.}
86.}
87.}
88.
89.double vec_len(double* vector, int n) {
90.double len = 0;
91.for (int i = 0; i < n; i++) {
92.len += vector[i] * vector[i];
93.}
94.
95.return sqrt(len);
96.}
97.
98.double scalar_mul_part(double* vector1, double* vector2, int n,
size_t proc_num, size_t rank) {
99.double result = 0;
100.double result_tmp = 0;
101.size_t lines = n / proc_num;
```

```
size_t offset = lines * rank;

for (int i = 0; i < lines; i++) {
result_tmp += vector1[i + offset] * vector2[i + offset];
}

MPI_Allreduce(&result_tmp, &result, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

return result;
}

double scalar_mul(double* vector1, double* vector2, int n,
size_t proc_num, size_t rank) {
double result = 0;

for (int i = 0; i < n; i++) {
result += vector1[i] * vector2[i];
}

return result;
}


void matrix_mul_vector(double* matrix, double* vector, double*
result, int n, size_t proc_num, size_t rank) {
size_t lines = n / proc_num;
size_t offset = lines * rank;

double result_tmp[n];

for (int i = 0; i < lines; i++) {
result_tmp[i] = 0;
for (int j = 0; j < n; j++) {
result_tmp[i] += matrix[i * n + j] * vector[j];
}
}

MPI_Allgather(result_tmp, lines, MPI_DOUBLE, result, lines,
MPI_DOUBLE, MPI_COMM_WORLD);
}

void vector_mul_double(double* vector, double dub, int n) {
for (int i = 0; i < n; i++) {
vector[i] *= dub;
}
}

void vector_minus_vector_part_2(double* vector1, double*
vector2_part, int n, size_t proc_num, size_t rank) {
double result_tmp[n];
size_t lines = n / proc_num;
size_t offset = lines * rank;
```

```
150.
151.for (int i = 0; i < lines; i++) {
152.result_tmp[i] = vector1[offset + i] - vector2_part[i];
153.}
154.
155.MPI_Allgather(result_tmp, lines, MPI_DOUBLE, vector1, lines,
MPI_DOUBLE, MPI_COMM_WORLD);
156.}
157.
158.void vector_minus_vector(double* vector1, double* vector2, int
n) {
159.for (int i = 0; i < n; i++) {
160.vector1[i] -= vector2[i];
161.}
162.}
163.
164.int is_finished(double* matrix, double* vector, double* b,
double epsilon, int n, size_t proc_num, size_t rank) {
165.
166.double* A = (double*)malloc(n * n * sizeof(double));
167.make_zero_matrix(A, n);
168.
169.matrix_copy(A, matrix, n);
170.double* Ax = (double*)malloc(n * sizeof(double));
171.
172.matrix_mul_vector(A, vector, Ax, n, proc_num, rank);
173.vector_minus_vector(Ax, b, n);
174.
175.if ((vec_len(Ax, n) / vec_len(b, n)) < epsilon) {
176.free(A);
177.free(Ax);
178.return 1;
179.}
180.
181.free(A);
182.free(Ax);
183.return 0;
184.}
185.
186.void count_y_part(double* y, double* A, double* x_n, double*
b_part, int n, size_t proc_num, size_t rank){
187.matrix_mul_vector(A, x_n, y, n, proc_num, rank);
188.
189.vector_minus_vector_part_2(y, b_part, n, proc_num, rank);
190.}
191.
192.void count_y(double* y, double* A, double* x_n, double* b, int
n, size_t proc_num, size_t rank){
193.matrix_mul_vector(A, x_n, y, n, proc_num, rank);
194.
195.vector_minus_vector(y, b, n);
196.}
197.
```

```
198.double count_tau(double* A, double* y, int n, size_t proc_num,
size_t rank) {
199.double* Ay = (double*)malloc(n * sizeof(double));
200.matrix_mul_vector(A, y, Ay, n, proc_num, rank);
201.
202.double tau = (sqrt(scalar_mul(y, Ay, n, proc_num, rank)) /
sqrt(scalar_mul(Ay, Ay, n, proc_num, rank)));
203.
204.free(Ay);
205.return tau;
206.}
207.
208.double* solve_eq(double* matrix, double* values, int n, size_t
proc_num, size_t rank) {
209.double epsilon = 0.001;
210.
211.double* y = (double*)malloc(n * sizeof(double));
212.double* x_i = (double*)malloc(n * sizeof(double));
213.vector_copy(x_i, values, n);
214.
215.int vec_part = (n / proc_num);
216.double* x_i_part = (double*)malloc(vec_part * sizeof(double));
217.double* values_part = (double*)malloc(vec_part *
sizeof(double));
218.MPI_Scatter(values, vec_part, MPI_DOUBLE, values_part, vec_part,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
219.
220.while (!is_finished(matrix, x_i, values, epsilon, n, proc_num,
rank)) {
221./*for (int i = 0; i < n; i++) {
222. printf("%lf ", x_i[i]);
223. }
224. printf("\n");*/
225.count_y_part(y, matrix, x_i, values, n, proc_num, rank);
226.
227.double tau = count_tau(matrix, y, n, proc_num, rank);
228.
229.vector_mul_double(y, tau, n);
230.
231.vector_minus_vector(x_i, y, n);
232.
233.}
234.
235.free(y);
236.
237.return x_i;
238.}
239.
240.int main(int argc, char** argv) {
241.// SETTING UP
242.MPI_Init(&argc, &argv);
243.int proc_num;
```

```
244.MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
245.int rank;
246.MPI_Comm_rank(MPI_COMM_WORLD, &rank);
247.
248.srand(time(NULL));
249.int n = 16;
250.
251.double* matrix = (double*)malloc(n * n * sizeof(double));
252.make_zero_matrix(matrix, n);
253.
254.//make_one_two_matrix(matrix, n);
255.//make_random_matrix(matrix, n);
256.make_one_matrix(matrix, n);
257.
258.double* values = (double*)malloc(n * sizeof(double));
259.for (int i = 0; i < n; i++) {
260.values[i] = n + 1;
261.}
262.
263.// MPI staff
264.size_t lines = n / proc_num;
265.double* matrix_part = (double*)malloc(lines * n *
sizeof(double));
266.MPI_Scatter(matrix, lines * n, MPI_DOUBLE, matrix_part, lines *
n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
267.
268.// ACTION
269.double* x_n = solve_eq(matrix_part, values, n, proc_num, rank);
270.
271.for (int i = 0; i < n; i++) {
272.printf("%f ", x_n[i]);
273.}
274.printf("\n");
275.
276.// FREEING
277.MPI_Finalize();
278.free(values);
279.free(x_n);
280.free(matrix);
281.return 0;
282.}
283.
```