###### 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

**Карпова Матвея Владимировича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

к. т. н., доцент

А. Ю. Власенко

Новосибирск 2021

**Содержание**

[ЗАДАНИЕ 3](#__RefHeading___Toc2441_419973115)

[ОПИСАНИЕ РАБОТЫ 4](#__RefHeading___Toc2443_419973115)

[ПРИЛОЖЕНИЕ 1 6](#__RefHeading___Toc2445_419973115)

[ПРИЛОЖЕНИЕ 2 16](#__RefHeading___Toc2447_419973115)

[ПРИЛОЖЕНИЕ 3 16](#__RefHeading___Toc2449_419973115)

[ПРИЛОЖЕНИЕ 4 27](#__RefHeading___Toc2451_419973115)

[ПРИЛОЖЕНИЕ 5 27](#__RefHeading___Toc2453_419973115)

[ПРИЛОЖЕНИЕ 6 41](#__RefHeading___Toc2455_419973115)

[ПРИЛОЖЕНИЕ 7 42](#__RefHeading___Toc2457_419973115)

[ПРИЛОЖЕНИЕ 8 42](#__RefHeading___Toc2459_419973115)

[ПРИЛОЖЕНИЕ 9 43](#__RefHeading___Toc2461_419973115)

[ПРИЛОЖЕНИЕ 10 44](#__RefHeading___Toc2463_419973115)

[ПРИЛОЖЕНИЕ 11 45](#__RefHeading___Toc2465_419973115)

[ПРИЛОЖЕНИЕ 12 46](#__RefHeading___Toc2467_419973115)

# ЗАДАНИЕ

1. Написать программу на языке C или C++, которая реализует алгоритм простых итераций решения системы линейных алгебраических уравнений вида Ax=b.
2. Программу распараллелить с помощью MPI с разрезанием матрицы A по строкам или столбцам на близкие по размеру, возможно, не одинаковые, части. Соседние строки матрицы должны располагаться в одном или в соседних MPI-процессах. Реализовать два варианта программы:

\* Вариант 1: векторы x и b дублируются в каждом MPI-процессе,

\* Вариант 2: векторы x и b разрезаются между MPI-процессами аналогично матрице A.

1. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 2, 4, 8, 16. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные, параметры N и ε подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.
2. Выполнить профилирование двух вариантов программы с помощью MPE при использовании 16-и ядер. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программы.

# ОПИСАНИЕ РАБОТЫ

1. Напишем программу на языке C, которая реализует алгоритм простых итераций для решения СЛАУ вида Ax = b (см. Приложение 1). Убедимся, что программа работает корректно (см. Приложение 2).
2. Программу из п.1 распараллелим с помощью MPI сначала с разрезанием матрицы А по строкам и дублированием векторов x и b в каждом процессе (см. Приложение 3), а потом с разрезанием матрицы А по столбцам и разрезанием векторов x и b между MPI процессами (см. Приложение 5). Убедимся, что написанные программы работают корректно (см Приложение 4 и Приложение 6).
3. На кластере НГУ скомпилируем и запустим программы с intel\_mpi-4.1.3.049. Замерим время выполнения последовательной программы на одном процессорном ядре и двух вариантов параллельных программ на 2, 4, 8 и 16 процессорных ядрах для размера матрицы A 144x144. Результаты измерений (см. Приложение 7) изобразим на графике (см. Приложение 8). Построим графики ускорения и эффективности распараллеливания в зависимости от числа используемых ядер (см. Приложение 9 и Приложение 10).
4. Выполним профилирование двух вариантов программы с помощью MPE при использовании 16-и ядер. Посмотрим результат при помощи traceanalyzer (см. Приложение 11 и Приложение 12). Одна итерация первого варианта программы состоит из выполнения пользовательского кода, MPI\_Allreduce, пользовательского кода, MPI\_Allgather, пользовательского кода. Можно заметить, что MPI\_Allgather в данном случае требует больше времени на выполнение. Одна итерация второго варианта программы состоит из пользовательского кода, MPI\_Allreduce, пользовательского кода, MPI\_Allreduce, пользовательского кода. Тем не менее видно, что второй вариант программы больше времени тратит на выполнение MPI функций. Этим может объяснятся его меньшая эффективность при увеличении числа ядер по сравнению с первым вариантом программы.

ЗАКЛЮЧЕНИЕ

Из результатов измерений видно, что первый вариант программы является более эффективным по сравнению со вторым. После выполнения профилирования, можно сделать вывод, что причиной этому является большее время выполнений MPI функций во втором варианте программы. В первом случае в каждом процессе дублируются векторы x и b, из-за чего меньше времени тратится на обмен информацией и больше на сами вычисления.

# ПРИЛОЖЕНИЕ 1

Код программы, которая реализует алгоритм простых итераций для решения СЛАУ вида Ax = b:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

typedef struct Matrix Matrix;

struct Matrix {

double \*elements;

size\_t row\_size;

size\_t col\_size;

};

const int max\_element\_value = 20;

const int overload\_value = 50;

const double thao = 0.001;

const double epsilon = 0.00001;

double get\_element(const Matrix \*A, size\_t i, size\_t j) {

if (i >= A->col\_size || j >= A->row\_size) {

printf("Out of range");

return -1;

}

return A->elements[i \* A->row\_size + j];

}

void set\_element(const Matrix \*A, size\_t i, size\_t j, double value) {

A->elements[i \* A->row\_size + j] = value;

}

void free\_matrix(Matrix \*A) {

free(A->elements);

free(A);

}

Matrix\* create\_uninit\_matrix(size\_t col\_size, size\_t row\_size) {

Matrix \*A = malloc(sizeof(Matrix));

if (A == NULL) {

printf("Memory error\n");

return NULL;

}

A->elements = malloc(row\_size \* col\_size \* sizeof(double));

if (A->elements == NULL) {

printf("Memory error\n");

return NULL;

}

A->col\_size = col\_size;

A->row\_size = row\_size;

return A;

}

size\_t power(size\_t a, size\_t b) {

if (b == 0) {

return 1;

}

else {

return a \* power(a, b - 1);

}

}

double hash(size\_t i, size\_t j, int max\_element\_value) {

return (power(i + j, i + j) % max\_element\_value) \* ((double) (i + 1) / (i + j + 2));

}

void fill\_random\_matrix(Matrix \*A) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

double rand\_value = hash(i, j, max\_element\_value);

set\_element(A, i, j, rand\_value);

}

}

}

Matrix\* create\_random\_matrix(size\_t col\_size, size\_t row\_size) {

Matrix \*A = create\_uninit\_matrix(col\_size, row\_size);

if (A == NULL) {

return NULL;

}

fill\_random\_matrix(A);

return A;

}

void fill\_random\_sym\_matrix(Matrix \*A) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = i; j < A->row\_size; j++) {

double rand\_value = hash(i, j, max\_element\_value);

if (i == j) {

rand\_value += overload\_value;

}

set\_element(A, i, j, rand\_value);

}

}

for (size\_t i = 1; i < A->col\_size; i++) {

for (size\_t j = 0; j < i; j++) {

set\_element(A, i, j, get\_element(A, j, i));

}

}

}

Matrix\* create\_zero\_matrix(size\_t col\_size, size\_t row\_size) {

Matrix \*A = create\_uninit\_matrix(col\_size, row\_size);

if (A == NULL) {

return NULL;

}

for (size\_t i = 0; i < col\_size; i++) {

for (size\_t j = 0; j < row\_size; j++) {

set\_element(A, i, j, 0);

}

}

return A;

}

void print\_matrix(Matrix \*A) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

printf("%f ", get\_element(A, i, j));

}

printf("\n");

}

printf("\n");

}

Matrix\* mul\_mm(Matrix \*A, Matrix \*B) {

if (A->row\_size != B->col\_size) {

printf("Wrong matrix sizes\n");

return NULL;

}

Matrix \*C = create\_zero\_matrix(A->col\_size, B->row\_size);

if (C == NULL) {

return NULL;

}

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t k = 0; k < A->row\_size; k++) {

for (size\_t j = 0; j < B->row\_size; j++) {

double new\_value = get\_element(C, i, j) + get\_element(A, i, k) \* get\_element(B, k, j);

set\_element(C, i, j, new\_value);

}

}

}

return C;

}

void subtract\_mm(Matrix \*A, Matrix \*B) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

set\_element(A, i, j, get\_element(A, i, j) - get\_element(B, i, j));

}

}

}

void mul\_m\_number(Matrix \*A, double n) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

set\_element(A, i, j, get\_element(A, i, j) \* n);

}

}

}

double modulus(Matrix \*A) {

if (A->row\_size != 1) {

printf("Wrong argument\n");

return -1;

}

double result = 0;

for (size\_t i = 0; i < A->col\_size; i++) {

result += get\_element(A, i, 0) \* get\_element(A, i, 0);

}

return sqrt(result);

}

int solve\_eq(Matrix \*A, Matrix \*x, Matrix \*b) {

size\_t hit\_counter = 0;

size\_t divergence\_counter = 0;

double prev\_hit = 0;

int thao\_sign = 1;

do {

Matrix \*TMP = mul\_mm(A, x);

if (TMP == NULL) {

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

return 1;

}

subtract\_mm(TMP, b);

double hit = modulus(TMP) / modulus(b);

mul\_m\_number(TMP, thao\_sign \* thao);

subtract\_mm(x, TMP);

//printf("hit: %f\n", hit);

if (hit < epsilon) {

hit\_counter++;

}

else {

hit\_counter = 0;

}

if (hit > prev\_hit) {

divergence\_counter++;

}

else {

divergence\_counter = 0;

}

if (divergence\_counter > 5) {

if (hit == INFINITY || abs(hit) == NAN) {

printf("Can't solve this\n");

free\_matrix(TMP);

return 1;

}

thao\_sign = -thao\_sign;

divergence\_counter = 0;

}

prev\_hit = hit;

free\_matrix(TMP);

} while (hit\_counter < 5);

return 0;

}

int main(int argc, char \*\*argv) {

if (argc != 2) {

printf("Wrong number of args\n");

return 1;

}

struct timespec start, end;

size\_t N = atol(argv[1]);

Matrix \*x = create\_random\_matrix(N, 1);

if (x == NULL) {

return 1;

}

Matrix \*b = create\_random\_matrix(N, 1);

if (b == NULL) {

free\_matrix(x);

return 1;

}

Matrix \*A = create\_uninit\_matrix(N, N);

if (A == NULL) {

free\_matrix(x);

free\_matrix(b);

return 1;

}

fill\_random\_sym\_matrix(A);

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

if (solve\_eq(A, x, b) < 0) {

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

return 1;

}

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);

// Matrix\* Ax = mul\_mm(A, x);

// printf("Ax:\n");

// print\_matrix(Ax);

// printf("b:\n");

// print\_matrix(b);

printf("1st programm\n");

printf("Time taken: %f sec\n\n", end.tv\_sec - start.tv\_sec + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec));

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

//free\_matrix(Ax);

return 0;

}

# ПРИЛОЖЕНИЕ 2

Вывод программы из Приложения 1 при аргументе 5:

Ax:

0.499938

0.666622

2.999962

5.600035

13.333392

b:

0.500000

0.666667

3.000000

5.600000

13.333333

1st programm

Time taken: 0.000229 sec

# ПРИЛОЖЕНИЕ 3

Код программы с разрезанием матрицы А по строкам и дублированием b и x в каждом процессе:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include <mpi.h>

#include <unistd.h>

typedef struct Matrix Matrix;

struct Matrix {

double \*elements;

size\_t row\_size;

size\_t col\_size;

size\_t index;

};

const int max\_element\_value = 20;

const int overload\_value = 50;

const double thao = 0.001;

const double epsilon = 0.00001;

double get\_element(const Matrix \*A, size\_t i, size\_t j) {

if (i >= A->col\_size || j >= A->row\_size) {

printf("MATRIX SIZE: %lu %lu\n", A->col\_size, A->row\_size);

printf("I: %lu, J: %lu\n", i, j);

printf("Out of range\n");

return -1;

}

return A->elements[i \* A->row\_size + j];

}

void set\_element(const Matrix \*A, size\_t i, size\_t j, double value) {

A->elements[i \* A->row\_size + j] = value;

}

void free\_matrix(Matrix \*A) {

free(A->elements);

free(A);

}

Matrix\* create\_uninit\_matrix(size\_t col\_size, size\_t row\_size, size\_t index) {

Matrix \*A = malloc(sizeof(Matrix));

if (A == NULL) {

printf("Memory error\n");

return NULL;

}

A->elements = malloc(row\_size \* col\_size \* sizeof(double));

if (A->elements == NULL) {

printf("Memory error: %lu, %lu\n", row\_size, col\_size);

return NULL;

}

A->col\_size = col\_size;

A->row\_size = row\_size;

A->index = index;

return A;

}

size\_t power(size\_t a, size\_t b) {

if (b == 0) {

return 1;

}

else {

return a \* power(a, b - 1);

}

}

double hash(size\_t i, size\_t j, int max\_element\_value) {

return (power(i + j, i + j) % max\_element\_value) \* ((double) (i + 1) / (i + j + 2));

}

void fill\_random\_matrix(Matrix \*A) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

double rand\_value = hash(i, j, max\_element\_value);

set\_element(A, i, j, rand\_value);

}

}

}

Matrix\* create\_random\_matrix(size\_t col\_size, size\_t row\_size, size\_t index) {

Matrix \*A = create\_uninit\_matrix(col\_size, row\_size, index);

if (A == NULL) {

return NULL;

}

fill\_random\_matrix(A);

return A;

}

void fill\_random\_sym\_matrix(Matrix \*A) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = i; j < A->row\_size; j++) {

double rand\_value = hash(i, j, max\_element\_value);

if (i == j) {

rand\_value += overload\_value;

}

set\_element(A, i, j, rand\_value);

}

}

for (size\_t i = 1; i < A->col\_size; i++) {

for (size\_t j = 0; j < i; j++) {

set\_element(A, i, j, get\_element(A, j, i));

}

}

}

Matrix\* create\_zero\_matrix(size\_t col\_size, size\_t row\_size, size\_t index) {

Matrix \*A = create\_uninit\_matrix(col\_size, row\_size, index);

if (A == NULL) {

return NULL;

}

for (size\_t i = 0; i < col\_size; i++) {

for (size\_t j = 0; j < row\_size; j++) {

set\_element(A, i, j, 0);

}

}

return A;

}

void print\_matrix(Matrix \*A) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

printf("%f ", get\_element(A, i, j));

}

printf("\n");

}

printf("\n");

}

Matrix\* mul\_mm(Matrix \*A, Matrix \*B) {

if (A->row\_size != B->col\_size) {

printf("Wrong matrix sizes\n");

return NULL;

}

Matrix \*C = create\_zero\_matrix(A->col\_size, B->row\_size, A->index);

if (C == NULL) {

return NULL;

}

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t k = 0; k < A->row\_size; k++) {

for (size\_t j = 0; j < B->row\_size; j++) {

double new\_value = get\_element(C, i, j) + get\_element(A, i, k) \* get\_element(B, k, j);

set\_element(C, i, j, new\_value);

}

}

}

return C;

}

void subtract\_mm(Matrix \*A, Matrix \*B) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

set\_element(A, i, j, get\_element(A, i, j) - get\_element(B, A->index + i, j));

}

}

}

void mul\_m\_number(Matrix \*A, double n) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

set\_element(A, i, j, get\_element(A, i, j) \* n);

}

}

}

int get\_portion\_size(int n\_of\_elements, int n\_of\_blocks, int block\_number) {

int portion\_size = n\_of\_elements / n\_of\_blocks;

if (block\_number < n\_of\_elements % n\_of\_blocks) {

portion\_size++;

}

return portion\_size;

}

double distributed\_modulus(Matrix \*A) {

if (A->row\_size != 1) {

printf("Wrong argument\n");

return -1;

}

double sum = 0;

double result;

for (size\_t i = 0; i < A->col\_size; i++) {

sum += get\_element(A, i, 0) \* get\_element(A, i, 0);

}

MPI\_Allreduce(&sum, &result, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

return sqrt(result);

}

double modulus(Matrix \*A) {

if (A->row\_size != 1) {

printf("Wrong argument\n");

return -1;

}

double result = 0;

for (size\_t i = 0; i < A->col\_size; i++) {

result += get\_element(A, i, 0) \* get\_element(A, i, 0);

}

return sqrt(result);

}

int build\_x(Matrix\* x, Matrix \*A, int world\_size, int rank) {

Matrix \*new\_x = create\_uninit\_matrix(A->col\_size, 1, 0);

if (new\_x == NULL) {

return -1;

}

for (size\_t i = 0; i < A->col\_size; i++) {

set\_element(new\_x, i, 0, get\_element(x, A->index + i, 0) - get\_element(A, i, 0));

}

int \*recv\_counts = malloc(sizeof(int) \* world\_size);

if (recv\_counts == NULL) {

printf("Memory error\n");

free\_matrix(new\_x);

return -1;

}

for (size\_t i = 0; i < world\_size; i++) {

recv\_counts[i] = get\_portion\_size(x->col\_size, world\_size, i);

}

int \*displs = malloc(world\_size \* sizeof(int));

if (displs == NULL) {

printf("Memory error\n");

free\_matrix(new\_x);

free(recv\_counts);

return -1;

}

displs[0] = 0;

for (size\_t i = 1; i < world\_size; i++) {

displs[i] = displs[i-1] + recv\_counts[i-1];

}

if (MPI\_Allgatherv(new\_x->elements, recv\_counts[rank], MPI\_DOUBLE, x->elements, recv\_counts, displs, MPI\_DOUBLE, MPI\_COMM\_WORLD) != MPI\_SUCCESS) {

printf("ERROR BUILDING X\n");

free\_matrix(new\_x);

free(recv\_counts);

free(displs);

return -1;

}

free(new\_x);

free(recv\_counts);

free(displs);

return 0;

}

int solve\_eq(Matrix\* A\_part, Matrix\* x, Matrix \* b, int rank, int world\_size) {

size\_t hit\_counter = 0;

size\_t divergence\_counter = 0;

double prev\_hit = 0;

int thao\_sign = 1;

do {

Matrix \*TMP = mul\_mm(A\_part, x);

if (TMP == NULL) {

return -1;

}

subtract\_mm(TMP, b);

double hit = distributed\_modulus(TMP) / modulus(b);

mul\_m\_number(TMP, thao\_sign \* thao);

if (build\_x(x, TMP, world\_size, rank) < 0) {

free\_matrix(TMP);

return -1;

}

// if (rank == 0) {

// printf("hit: %f\n", hit);

// }

if (hit < epsilon) {

hit\_counter++;

}

else {

hit\_counter = 0;

}

if (hit > prev\_hit) {

divergence\_counter++;

}

else {

divergence\_counter = 0;

}

if (divergence\_counter > 5) {

thao\_sign = -thao\_sign;

divergence\_counter = 0;

}

if (hit == INFINITY || abs(hit) == NAN) {

printf("Can't solve this\n");

free\_matrix(TMP);

return -1;

}

prev\_hit = hit;

free\_matrix(TMP);

} while (hit\_counter < 5);

return 0;

}

int distribute\_matrix\_by\_rows(Matrix\* A, Matrix\* A\_part, int rank, int world\_size) {

int \*sendcounts = malloc(world\_size \* sizeof(int));

if (sendcounts == NULL) {

printf("Memory error\n");

return -1;

}

for (size\_t i = 0; i < world\_size; i++) {

sendcounts[i] = get\_portion\_size(A->row\_size, world\_size, i) \* A->col\_size;

}

int \*displs = malloc(world\_size \* sizeof(int));

displs[0] = 0;

if (displs == NULL) {

printf("Memory error\n");

free(sendcounts);

return -1;

}

for (size\_t i = 1; i < world\_size; i++) {

displs[i] = displs[i-1] + sendcounts[i-1];

}

A\_part->index = displs[rank] / A->row\_size;

if (MPI\_Scatterv(A->elements, sendcounts, displs, MPI\_DOUBLE, A\_part->elements, sendcounts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD) != MPI\_SUCCESS) {

free(displs);

free(sendcounts);

printf("Data distribution error\n");

return -1;

}

free(displs);

free(sendcounts);

return 0;

}

int main(int argc, char \*\*argv) {

if (argc != 2) {

printf("Wrong number of args\n");

return 1;

}

struct timespec start, end;

size\_t N = atol(argv[1]);

Matrix \*A = create\_uninit\_matrix(N, N, 0);

if (A == NULL) {

return 1;

}

Matrix \*x = create\_uninit\_matrix(N, 1, 0);

if (x == NULL) {

free\_matrix(A);

return 1;

}

Matrix \*b = create\_uninit\_matrix(N, 1, 0);

if (b == NULL) {

free\_matrix(A);

free\_matrix(x);

return 1;

}

if (MPI\_Init(&argc, &argv) != MPI\_SUCCESS) {

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

printf("Init error\n");

return 1;

}

int world\_size;

if (MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size) != MPI\_SUCCESS) {

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

printf("Comm\_size error\n");

return 1;

}

int rank;

if (MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank) != MPI\_SUCCESS) {

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

printf("Comm\_rank error\n");

return 1;

}

Matrix \*A\_part = create\_uninit\_matrix(get\_portion\_size(N, world\_size, rank), N, 0);

if (A\_part == NULL) {

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

if (rank == 0) {

fill\_random\_sym\_matrix(A);

fill\_random\_matrix(x);

fill\_random\_matrix(b);

}

if (MPI\_Bcast(x->elements, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD) != MPI\_SUCCESS) {

printf("Bcast error\n");

free\_matrix(A\_part);

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

if (MPI\_Bcast(b->elements, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD) != MPI\_SUCCESS) {

printf("Bcast error\n");

free\_matrix(A\_part);

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

if (distribute\_matrix\_by\_rows(A, A\_part, rank, world\_size) < 0) {

free\_matrix(A\_part);

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

if (rank == 0) {

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

}

if (solve\_eq(A\_part, x, b, rank, world\_size) < 0) {

free\_matrix(A);

free\_matrix(A\_part);

free\_matrix(x);

free\_matrix(b);

MPI\_Finalize();

return 1;

}

if (rank == 0) {

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);

}

// if (rank == 0) {

// Matrix\* Ax = mul\_mm(A, x);

// printf("Ax:\n");

// print\_matrix(Ax);

// printf("b:\n");

// print\_matrix(b);

// free\_matrix(Ax);

// }

if (rank == 0) {

printf("2nd programm %d processes\n", world\_size);

printf("Time taken: %f sec\n\n", end.tv\_sec - start.tv\_sec + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec));

}

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

free\_matrix(A\_part);

MPI\_Finalize();

return 0;

}

# ПРИЛОЖЕНИЕ 4

Вывод программы из Приложения 3 для 2 процессов и аргумента 5:

Ax:

0.499938

0.666622

2.999962

5.600035

13.333392

b:

0.500000

0.666667

3.000000

5.600000

13.333333

2nd programm 2 processes

Time taken: 0.001534 sec

# ПРИЛОЖЕНИЕ 5

Код программы с разрезанием матрицы А по стролбцам разрезанием x и b по столбцам:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include <mpi.h>

#include <unistd.h>

typedef struct Matrix Matrix;

struct Matrix {

double \*elements;

size\_t row\_size;

size\_t col\_size;

size\_t index;

};

const int max\_element\_value = 20;

const int overload\_value = 50;

const double thao = 0.001;

const double epsilon = 0.00001;

double get\_element(const Matrix \*A, size\_t i, size\_t j) {

if (i >= A->col\_size || j >= A->row\_size) {

printf("MATRIX SIZE: %lu %lu\n", A->col\_size, A->row\_size);

printf("I: %lu, J: %lu\n", i, j);

printf("Out of range\n");

return -1;

}

return A->elements[i \* A->row\_size + j];

}

void set\_element(const Matrix \*A, size\_t i, size\_t j, double value) {

A->elements[i \* A->row\_size + j] = value;

}

void free\_matrix(Matrix \*A) {

free(A->elements);

free(A);

}

Matrix\* create\_uninit\_matrix(size\_t col\_size, size\_t row\_size, size\_t index) {

Matrix \*A = malloc(sizeof(Matrix));

if (A == NULL) {

printf("Memory error\n");

return NULL;

}

A->elements = malloc(row\_size \* col\_size \* sizeof(double));

if (A->elements == NULL) {

printf("Memory error: %lu, %lu\n", row\_size, col\_size);

return NULL;

}

A->col\_size = col\_size;

A->row\_size = row\_size;

A->index = index;

return A;

}

size\_t power(size\_t a, size\_t b) {

if (b == 0) {

return 1;

}

else {

return a \* power(a, b - 1);

}

}

double hash(size\_t i, size\_t j, int max\_element\_value) {

return (power(i + j, i + j) % max\_element\_value) \* ((double) (i + 1) / (i + j + 2));

}

void fill\_random\_matrix(Matrix \*A) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

double rand\_value = hash(i, j, max\_element\_value);

set\_element(A, i, j, rand\_value);

}

}

}

Matrix\* create\_random\_matrix(size\_t col\_size, size\_t row\_size, size\_t index) {

Matrix \*A = create\_uninit\_matrix(col\_size, row\_size, index);

if (A == NULL) {

return NULL;

}

fill\_random\_matrix(A);

return A;

}

void fill\_random\_sym\_matrix(Matrix \*A) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = i; j < A->row\_size; j++) {

double rand\_value = hash(i, j, max\_element\_value);

if (i == j) {

rand\_value += overload\_value;

}

set\_element(A, i, j, rand\_value);

}

}

for (size\_t i = 1; i < A->col\_size; i++) {

for (size\_t j = 0; j < i; j++) {

set\_element(A, i, j, get\_element(A, j, i));

}

}

}

Matrix\* create\_zero\_matrix(size\_t col\_size, size\_t row\_size, size\_t index) {

Matrix \*A = create\_uninit\_matrix(col\_size, row\_size, index);

if (A == NULL) {

return NULL;

}

for (size\_t i = 0; i < col\_size; i++) {

for (size\_t j = 0; j < row\_size; j++) {

set\_element(A, i, j, 0);

}

}

return A;

}

void print\_matrix(Matrix \*A) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

printf("%f ", get\_element(A, i, j));

}

printf("\n");

}

printf("\n");

}

Matrix\* mul\_mm(Matrix \*A, Matrix \*B) {

if (A->row\_size != B->col\_size) {

printf("Wrong matrix sizes\n");

return NULL;

}

Matrix \*C = create\_zero\_matrix(A->col\_size, B->row\_size, A->index);

if (C == NULL) {

return NULL;

}

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t k = 0; k < A->row\_size; k++) {

for (size\_t j = 0; j < B->row\_size; j++) {

double new\_value = get\_element(C, i, j) + get\_element(A, i, k) \* get\_element(B, k, j);

set\_element(C, i, j, new\_value);

}

}

}

return C;

}

void subtract\_mm(Matrix \*A, Matrix \*B) {

for (size\_t i = 0; i < B->col\_size; i++) {

for (size\_t j = 0; j < B->row\_size; j++) {

set\_element(A, B->index + i, j, get\_element(A, B->index + i, j) - get\_element(B, i, j));

}

}

}

void mul\_m\_number(Matrix \*A, double n) {

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

set\_element(A, i, j, get\_element(A, i, j) \* n);

}

}

}

int get\_portion\_size(int n\_of\_elements, int n\_of\_blocks, int block\_number) {

int portion\_size = n\_of\_elements / n\_of\_blocks;

if (block\_number < n\_of\_elements % n\_of\_blocks) {

portion\_size++;

}

return portion\_size;

}

double distributed\_modulus(Matrix \*A) {

if (A->row\_size != 1) {

printf("Wrong argument\n");

return -1;

}

double sum = 0;

double result;

for (size\_t i = 0; i < A->col\_size; i++) {

sum += get\_element(A, i, 0) \* get\_element(A, i, 0);

}

MPI\_Allreduce(&sum, &result, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

return sqrt(result);

}

double modulus(Matrix \*A) {

if (A->row\_size != 1) {

printf("Wrong argument\n");

return -1;

}

double result = 0;

for (size\_t i = 0; i < A->col\_size; i++) {

result += get\_element(A, i, 0) \* get\_element(A, i, 0);

}

return sqrt(result);

}

int collect\_Ax\_min\_b(Matrix \*A) {

if (MPI\_Allreduce(MPI\_IN\_PLACE, A->elements, A->col\_size, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD) != MPI\_SUCCESS) {

printf("ERROR BUILDING X\n");

return -1;

}

return 0;

}

int build\_x(Matrix \*x\_part, Matrix \*A, int world\_size, int rank) {

for (size\_t i = 0; i < x\_part->col\_size; i++) {

set\_element(x\_part, i, 0, get\_element(x\_part, i, 0) - get\_element(A, x\_part->index + i, 0));

}

return 0;

}

int solve\_eq(Matrix \*A\_part, Matrix \*x\_part, Matrix \*b\_part, int rank, int world\_size) {

size\_t hit\_counter = 0;

size\_t divergence\_counter = 0;

double prev\_hit = 0;

int thao\_sign = 1;

do {

Matrix \*TMP = mul\_mm(A\_part, x\_part);

if (TMP == NULL) {

return -1;

}

subtract\_mm(TMP, b\_part);

collect\_Ax\_min\_b(TMP);

double hit = modulus(TMP) / distributed\_modulus(b\_part);

mul\_m\_number(TMP, thao\_sign \* thao);

if (build\_x(x\_part, TMP, world\_size, rank) < 0) {

free\_matrix(TMP);

return -1;

}

// if (rank == 0) {

// printf("hit: %f\n", hit);

// }

if (hit < epsilon) {

hit\_counter++;

}

else {

hit\_counter = 0;

}

if (hit > prev\_hit) {

divergence\_counter++;

}

else {

divergence\_counter = 0;

}

if (divergence\_counter > 5) {

thao\_sign = -thao\_sign;

divergence\_counter = 0;

}

if (hit == INFINITY || abs(hit) == NAN) {

printf("Can't solve this\n");

free\_matrix(TMP);

return -1;

}

prev\_hit = hit;

free\_matrix(TMP);

} while (hit\_counter < 5);

return 0;

}

Matrix \*copy\_m(Matrix \*A) {

Matrix \*A\_c = create\_uninit\_matrix(A->col\_size, A->row\_size, A->index);

if (A\_c == NULL) {

return NULL;

}

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

set\_element(A\_c, i, j, get\_element(A, i, j));

}

}

return A\_c;

}

int transpose\_m(Matrix \*A) {

Matrix \*A\_c = copy\_m(A);

if (A\_c == NULL) {

return -1;

}

double tmp = A->col\_size;

A->col\_size = A->row\_size;

A->row\_size = tmp;

for (size\_t i = 0; i < A->col\_size; i++) {

for (size\_t j = 0; j < A->row\_size; j++) {

set\_element(A, i, j, get\_element(A\_c, j, i));

}

}

free\_matrix(A\_c);

return 0;

}

int distribute\_matrix\_by\_cols(Matrix \*A, Matrix \*A\_part, int rank, int world\_size) {

int \*sendcounts = malloc(world\_size \* sizeof(int));

if (sendcounts == NULL) {

printf("Memory error\n");

return -1;

}

for (size\_t i = 0; i < world\_size; i++) {

sendcounts[i] = get\_portion\_size(A->row\_size, world\_size, i) \* A->col\_size;

}

int \*displs = malloc(world\_size \* sizeof(int));

if (displs == NULL) {

printf("Memory error\n");

free(sendcounts);

return -1;

}

displs[0] = 0;

for (size\_t i = 1; i < world\_size; i++) {

displs[i] = displs[i-1] + sendcounts[i-1];

}

A\_part->index = displs[rank] / A->col\_size;

if (MPI\_Scatterv(A->elements, sendcounts, displs, MPI\_DOUBLE, A\_part->elements, sendcounts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD) != MPI\_SUCCESS) {

free(displs);

free(sendcounts);

printf("Data distribution error\n");

return -1;

}

if (transpose\_m(A\_part) < 0) {

free(displs);

free(sendcounts);

return -1;

}

free(displs);

free(sendcounts);

return 0;

}

int distribute\_matrix\_by\_rows(Matrix \*A, Matrix \*A\_part, int rank, int world\_size) {

int \*sendcounts = malloc(world\_size \* sizeof(int));

if (sendcounts == NULL) {

printf("Memory error\n");

return -1;

}

for (size\_t i = 0; i < world\_size; i++) {

sendcounts[i] = get\_portion\_size(A->col\_size, world\_size, i) \* A->row\_size;

}

int \*displs = malloc(world\_size \* sizeof(int));

if (displs == NULL) {

printf("Memory error\n");

free(sendcounts);

return -1;

}

displs[0] = 0;

for (size\_t i = 1; i < world\_size; i++) {

displs[i] = displs[i-1] + sendcounts[i-1];

}

A\_part->index = displs[rank] / A->row\_size;

if (MPI\_Scatterv(A->elements, sendcounts, displs, MPI\_DOUBLE, A\_part->elements, sendcounts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD) != MPI\_SUCCESS) {

free(displs);

free(sendcounts);

printf("Data distribution error\n");

return -1;

}

free(displs);

free(sendcounts);

return 0;

}

int collect\_x(Matrix \*x\_part, Matrix \*x, int rank, int world\_size) {

int \*recv\_counts = malloc(sizeof(int) \* world\_size);

if (recv\_counts == NULL) {

printf("Memory error\n");

return -1;

}

for (size\_t i = 0; i < world\_size; i++) {

recv\_counts[i] = get\_portion\_size(x->col\_size, world\_size, i);

}

int \*displs = malloc(world\_size \* sizeof(int));

if (displs == NULL) {

printf("Memory error\n");

free(recv\_counts);

return -1;

}

displs[0] = 0;

for (size\_t i = 1; i < world\_size; i++) {

displs[i] = displs[i-1] + recv\_counts[i-1];

}

if (MPI\_Gatherv(x\_part->elements, x\_part->col\_size, MPI\_DOUBLE, x->elements, recv\_counts, displs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD) != MPI\_SUCCESS) {

printf("ERROR BUILDING X\n");

free(recv\_counts);

free(displs);

return -1;

}

free(recv\_counts);

free(displs);

return 0;

}

int main(int argc, char \*\*argv) {

if (argc != 2) {

printf("Wrong number of args\n");

return 1;

}

struct timespec start, end;

size\_t N = atol(argv[1]);

Matrix \*A = create\_uninit\_matrix(N, N, 0);

if (A == NULL) {

return 1;

}

Matrix \*x = create\_uninit\_matrix(N, 1, 0);

if (x == NULL) {

free\_matrix(A);

return 1;

}

Matrix \*b = create\_uninit\_matrix(N, 1, 0);

if (b == NULL) {

free\_matrix(A);

free\_matrix(x);

return 1;

}

if (MPI\_Init(&argc, &argv) != MPI\_SUCCESS) {

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

printf("Init error\n");

return 1;

}

int world\_size;

if (MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size) != MPI\_SUCCESS) {

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

printf("Comm\_size error\n");

return 1;

}

int rank;

if (MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank) != MPI\_SUCCESS) {

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

printf("Comm\_rank error\n");

return 1;

}

Matrix \*A\_part = create\_uninit\_matrix(get\_portion\_size(N, world\_size, rank), N, 0);

if (A\_part == NULL) {

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

Matrix \*x\_part = create\_uninit\_matrix(get\_portion\_size(N, world\_size, rank), 1, 0);

if (x\_part == NULL) {

free\_matrix(A\_part);

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

Matrix \*b\_part = create\_uninit\_matrix(get\_portion\_size(N, world\_size, rank), 1, 0);

if (b\_part == NULL) {

free\_matrix(A\_part);

free\_matrix(b\_part);

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

if (rank == 0) {

fill\_random\_sym\_matrix(A);

fill\_random\_matrix(x);

fill\_random\_matrix(b);

}

if (rank == 0) {

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

}

if (distribute\_matrix\_by\_rows(x, x\_part, rank, world\_size) < 0) {

free\_matrix(A\_part);

free\_matrix(b\_part);

free\_matrix(x\_part);

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

if (distribute\_matrix\_by\_rows(b, b\_part, rank, world\_size) < 0) {

free\_matrix(A\_part);

free\_matrix(b\_part);

free\_matrix(x\_part);

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

if (distribute\_matrix\_by\_cols(A, A\_part, rank, world\_size) < 0) {

free\_matrix(A\_part);

free\_matrix(b\_part);

free\_matrix(x\_part);

free\_matrix(x);

free\_matrix(b);

free\_matrix(A);

MPI\_Finalize();

return 1;

}

if (solve\_eq(A\_part, x\_part, b\_part, rank, world\_size) < 0) {

free\_matrix(A);

free\_matrix(A\_part);

free\_matrix(b\_part);

free\_matrix(x\_part);

free\_matrix(x);

free\_matrix(b);

MPI\_Finalize();

return 1;

}

if (collect\_x(x\_part, x, rank, world\_size) < 0) {

free\_matrix(A);

free\_matrix(A\_part);

free\_matrix(b\_part);

free\_matrix(x\_part);

free\_matrix(x);

free\_matrix(b);

MPI\_Finalize();

return 1;

}

if (rank == 0) {

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);

}

// if (rank == 0) {

// Matrix\* Ax = mul\_mm(A, x);

// printf("Ax:\n");

// print\_matrix(Ax);

// printf("b:\n");

// print\_matrix(b);

// free\_matrix(Ax);

// }

if (rank == 0) {

printf("3rd programm %d processes\n", world\_size);

printf("Time taken: %f sec\n", end.tv\_sec - start.tv\_sec + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec));

}

free\_matrix(b\_part);

free\_matrix(x\_part);

free\_matrix(A\_part);

free\_matrix(A);

free\_matrix(x);

free\_matrix(b);

MPI\_Finalize();

return 0;

}

# ПРИЛОЖЕНИЕ 6

Вывод программы из Приложения 3 для 2 процессов и аргумента 5:

Ax:

0.499938

0.666622

2.999962

5.600035

13.333392

b:

0.500000

0.666667

3.000000

5.600000

13.333333

3rd programm 2 processes

Time taken: 0.001423 sec

# ПРИЛОЖЕНИЕ 7

Результаты измерений работы последовательной программы на одном ядре и двух вариантов параллельной программы на 2, 4, 8 и 16 ядрах:

1st programm

Time taken: 57.015816 sec

2nd programm 2 processes

Time taken: 37.164817 sec

2nd programm 4 processes

Time taken: 24.792947 sec

2nd programm 8 processes

Time taken: 11.436346 sec

2nd programm 16 processes

Time taken: 10.775361 sec

3rd programm 2 processes

Time taken: 37.233487 sec

3rd programm 4 processes

Time taken: 20.486149 sec

3rd programm 8 processes

Time taken: 12.440955 sec

3rd programm 16 processes

Time taken: 11.839491 sec

# ПРИЛОЖЕНИЕ 8

График, построенный на основании результатов из Приложения 7:

# ПРИЛОЖЕНИЕ 9

График ускорения распараллеливания в зависимости от числа используемых ядер:

Вычислим ускорение для первого варианта параллельной программы по формуле (Sp = T1/Tp).

S1 = 1;

S2 = 57.015816 / 37.164817 = 1.534134179

S4 = 57.015816 / 24.792947 = 2.299678856

S8 = 57.015816 / 11.436346 = 4.985492394

S16 = 57.015816 / 10.775361 = 5.291313767

И для второго варианта:

S1 = 1

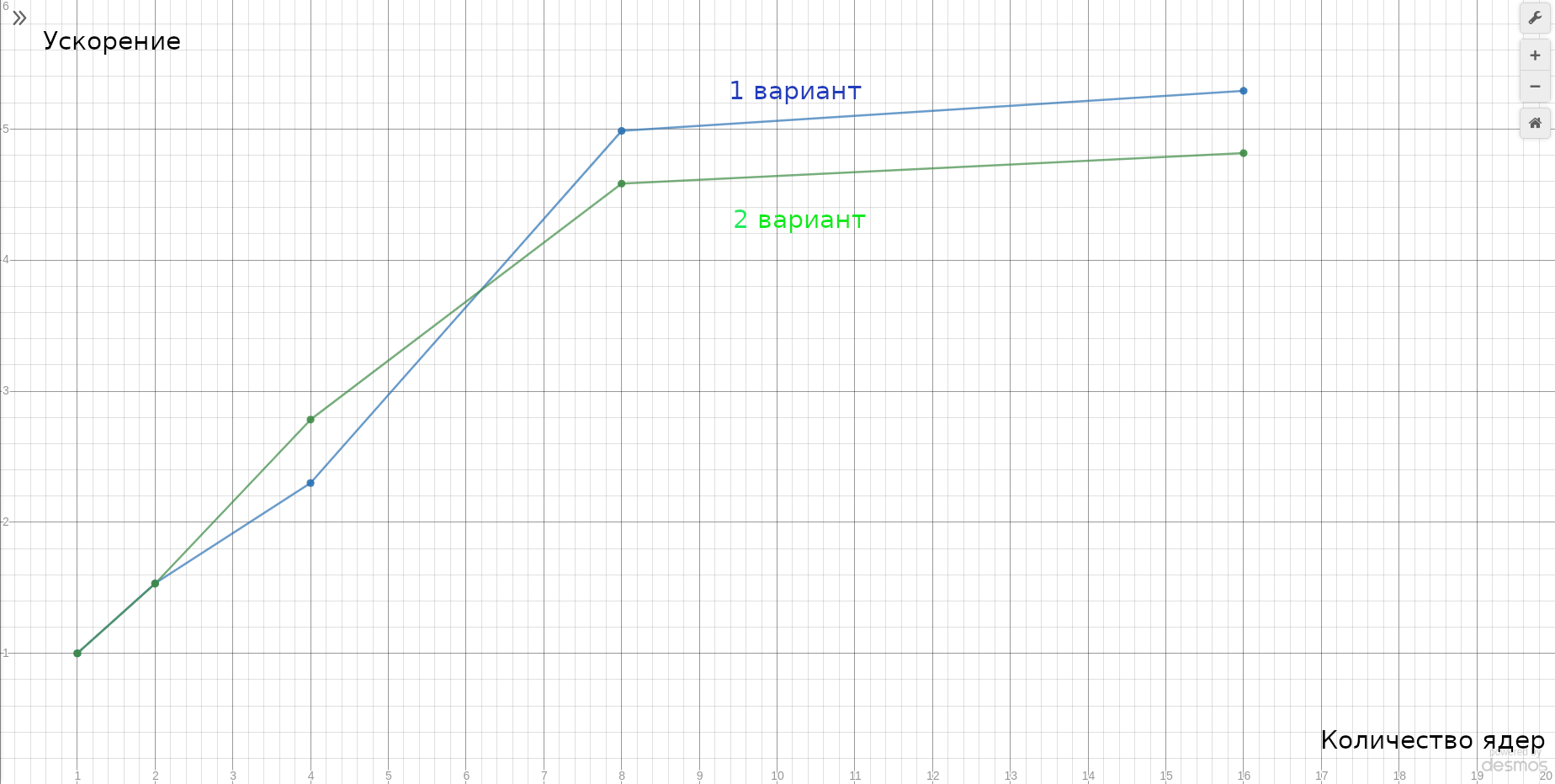
S2 = 57.015816 / 37.233487 = 1.531304763

S4 = 57.015816 / 20.486149 = 2.783139769

S8 = 57.015816 / 12.440955 = 4.582913128

S16 = 57.015816 / 11.839491 = 4.815732028

Изобразим результаты на графике:



# ПРИЛОЖЕНИЕ 10

График эффективности распараллеливания в зависимости от числа используемых ядер:

Вычислим эффективность для первого варианта параллельной программы по формуле Ep = (Sp / p) \* 100%:

E1 = (1 / 1) \* 100% = 100%

E2 = (1.534134179 / 2) \* 100% = 76.70670895%

E4 = (2.299678856 / 4) \* 100% = 57.4919714%

E8 = (4.985492394 / 8) \* 100% = 62.318654925%

E16 = (5.291313767 / 16) \* 100% = 33.070711044%

И для второго варианта параллельной программы:

E1 = (1 / 1) \* 100% = 100%

E2 = (1.531304763 / 2) \* 100% = 76.56523815%

E4 = (2.783139769 / 4) \* 100% = 69.578494225%

E8 = (4.582913128 / 8) \* 100% = 57.2864141%

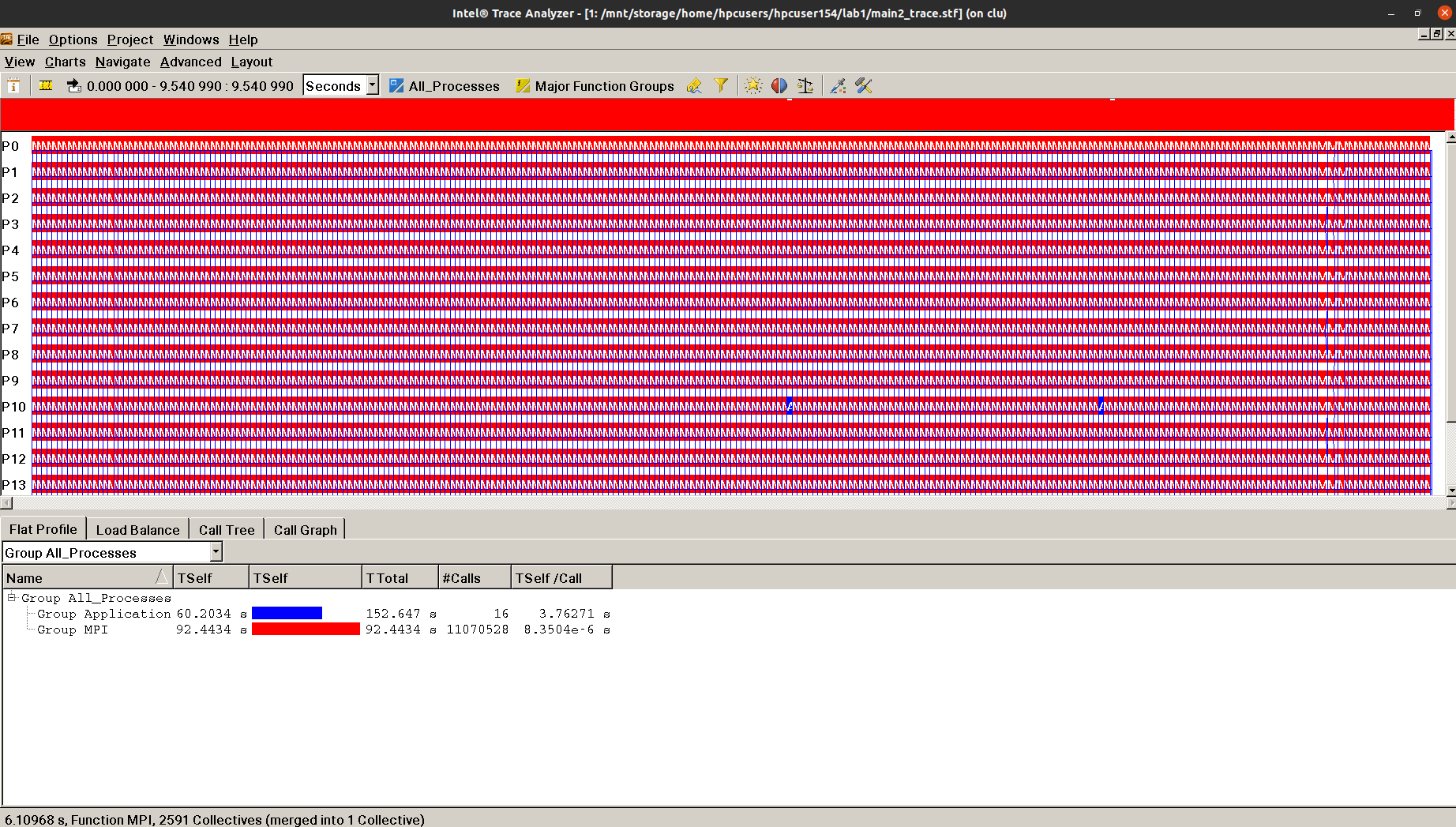
E16 = (4.815732028 / 16) \* 100% = 30.098325175%

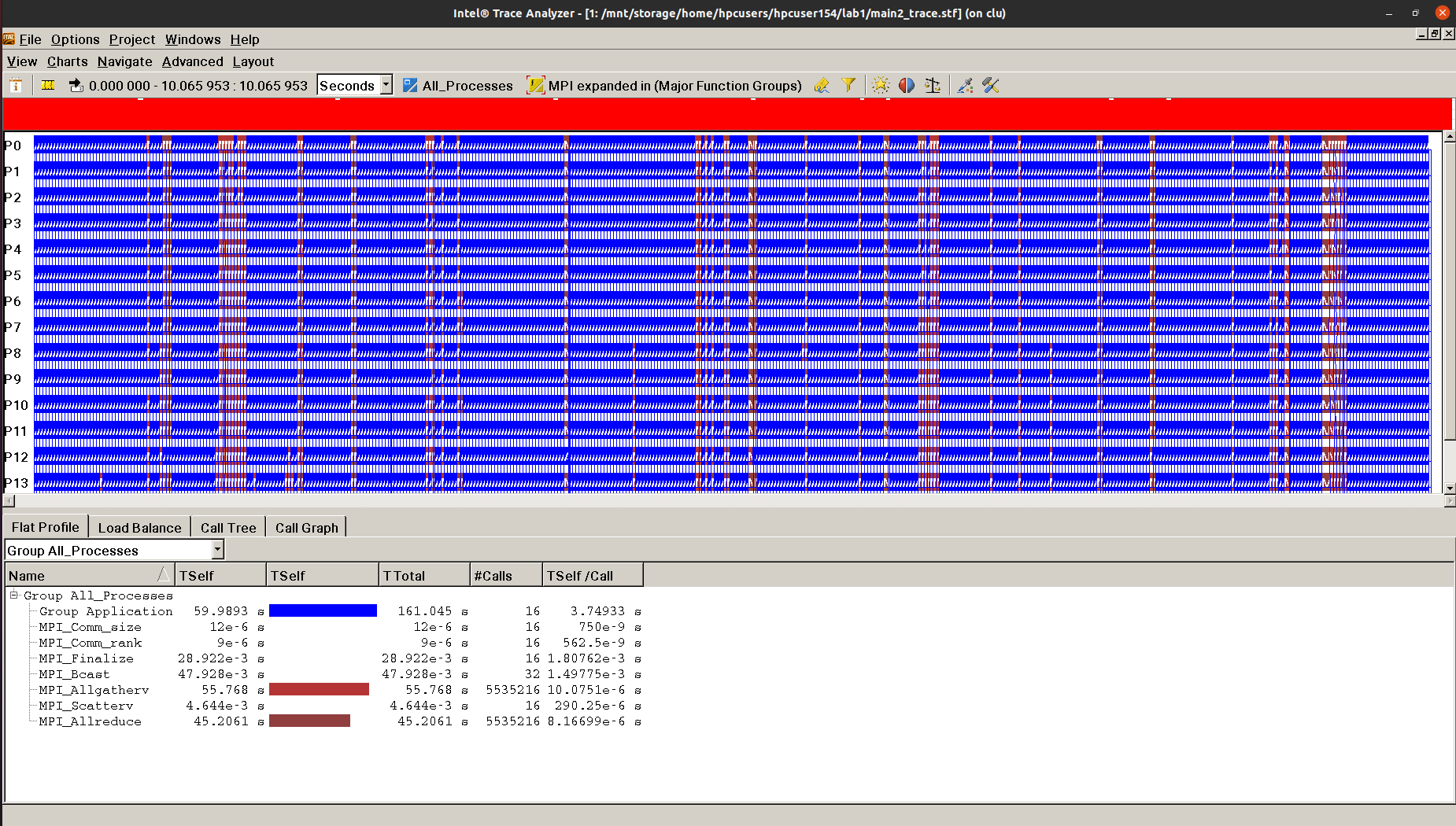
Изобразим результаты на графике:

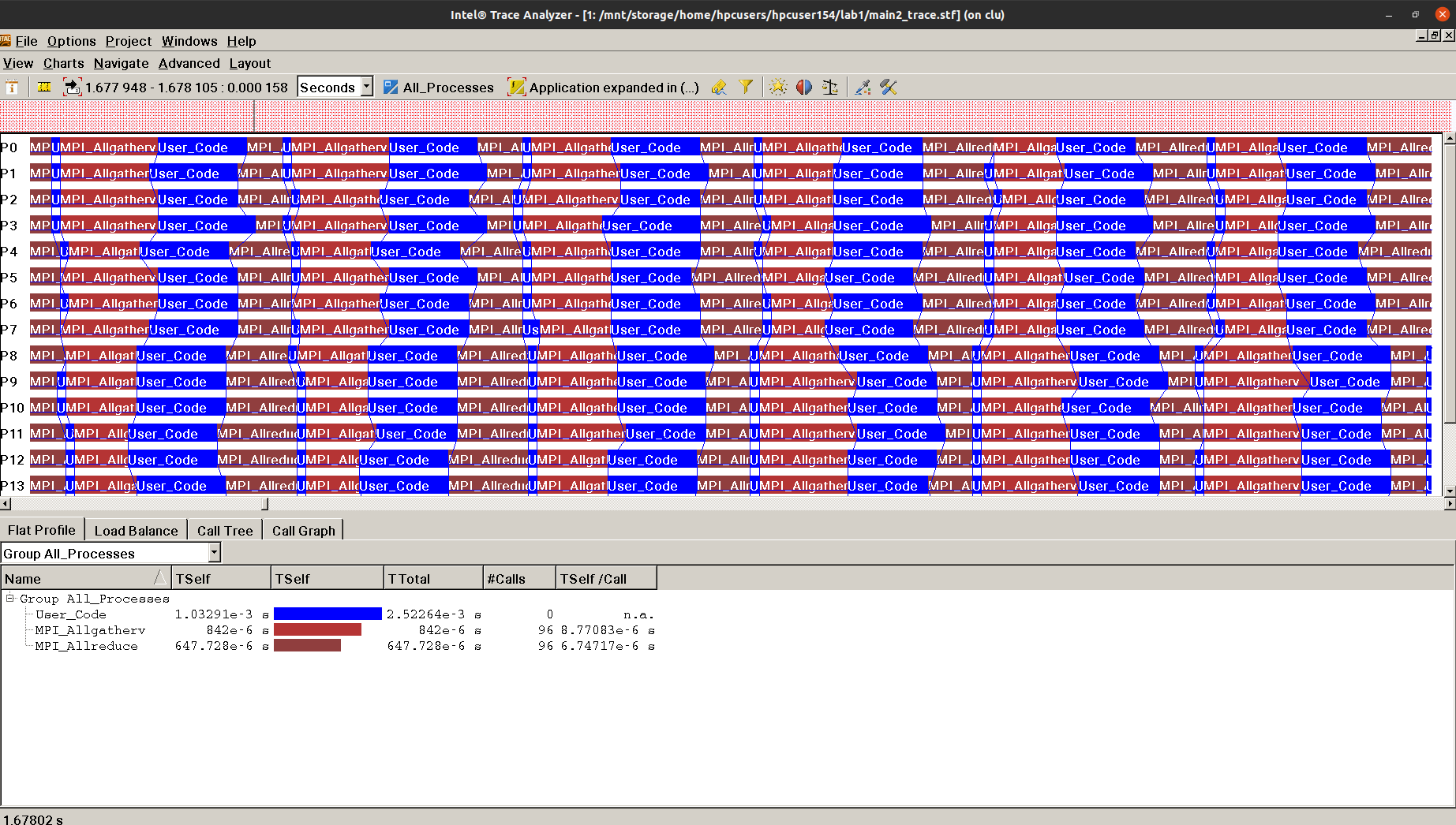


# ПРИЛОЖЕНИЕ 11

Результат профилирования первого варианта программы с помощью MPE:







# ПРИЛОЖЕНИЕ 12

Результат профилирования второго варианта программы с помощью MPE:

