**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**

Лабораторная работа 1

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью MPI»

Студента 2 курса, 19210 группы

**Пирожков Андрей Константинович**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

П.О.Холявко

Новосибирск 2021

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc65234499)

[ЗАДАНИЕ 3](#_Toc65234500)

[ВАРИАНТ 3](#_Toc65234501)

[ОПИСАНИЕ РАБОТЫ 5](#_Toc65234502)

[ЗАКЛЮЧЕНИЕ 12](#_Toc65234503)

[Приложение 1 *Листинг файла lab1.c* 13](#_Toc65234504)

[Приложение 2 *Листинг файла lab1\_1.c* 16](#_Toc65234505)

[Приложение 3 *Листинг файла lab1\_2.c* 20](#_Toc65234506)

[Приложение 4 *Листинг файла run.sh* 24](#_Toc65234507)

# ЦЕЛЬ

* Изучение библиотеки MPI как средство распараллеливания программы на работающие части
* Научиться работать с кластером
* Проанализировать полученную программу и выявить зависимость производительности программы от входных данных, варианта решения, количество ядер

# ЗАДАНИЕ

1. Написать программу на языке C или C++, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b в соответствии с выбранным вариантом. Здесь A – матрица размером N×N, x и b – векторы длины N. Тип элементов – double.
2. Программу распараллелить с помощью MPI с разрезанием матрицы A по строкам 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.

Уделить внимание тому, чтобы при запуске программы на различном числе MPI-процессов решалась одна и та же задача (исходные данные заполнялись одинаковым образом). Каждый вариант программы оптимизировать по скорости, насколько это возможно.

1. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 1, 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.
3. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программы.

# ВАРИАНТ

***Метод минимальных невязок.***

В методе минимальных невязок преобразование решения на каждом шаге задается следующими формулами:

В отличие от метода простой итерации, метод минимальных невязок содержит два умножения матрицы на вектор на каждой итерации, однако сходится до нужной точности за меньшее число итераций. Критерий завершения счета можно взять такой же, как в методе простой итерации:

# **ОПИСАНИЕ РАБОТЫ**

Выбрал язык программирования Си стандарта 1999 года. Решил, что буду использовать второй метод решения: «метод минимальных невязок». Написал сначала просто программу, которая работает без библиотеки MPI. Далее с помощью простых калькуляторов в интернете убедился в правильности решения СЛАУ.

В качестве входных данных, я решил, что буду генерировать их сам, с помощью функции rand(). Благодаря ей при каждом запуске программы получаются одинаковые рандомные входные данные (в моём случае рандом выдаёт цифры от -9 до 9). Это гарантирует, что при каждом запуске на различном числе MPI-процессов будет решаться одна и та же задача.

Сначала я создаю матрицу , затем генерирую вектор , далее я считаю, чему будет равен вектор . А после всего этого, я сбрасываю вектор на другие рандомные значения. Таким образом, получаю максимально рандомную СЛАУ и быстро генерирую исходные данные, которую нужно решить. А правильность работы написанного алгоритма легко проверить, если сохранить исходный вектор и сравнить его с конечным полученным результатом.

Сразу отмечу, что я создаю симметрическую матрицу, потому что несимметрические матрицы расходятся в решении (алгоритм начинает работать вечно). Причины такого феномена я не нашёл, однако в 3-ем методе («метод сопряженных градиентов») сказано, что он подходит только для симметрических матриц. Думаю, это применимо ко всем остальным методам.

Далее я искал подходящие и для моей задачи. Оказалось, что лучше всего подойдёт и . Такие входные данные обеспечивают не очень продолжительное время на одном ядре (около 42 в очереди и 26 секунд просто на сервере) и не самое большое количество итераций: 385908.

Кстати, определение количество итераций – очень хороший критерий проверки правильности выполнения программы. Я думаю, её точность сравнима с хэш-функций, которая преобразует строку в цифру. В дальнейшем правильность выполнения 2-ух вариантов программ на больших данных отслеживал именно сравнением числа количества итераций.

Далее начал заниматься разработкой программы с MPI. Мне удалось обойтись только 3-мя основными функциями:

* MPI\_Scatter – эта функция помогла мне делить на кусочки готовую матрицу и отправлять её разным процессам.
* MPI\_Allgather – собирала воедино вектор из кусочков в умножении матрицы на вектор.
* MPI\_Allreduce – суммировала результаты из разных процессов.
* MPI\_Init, MPI\_Comm\_rank, MPI\_Comm\_size, MPI\_Finalize() – не беру в счёт, их и так почти везде нужно использовать, если подключать MPI.

Сразу отмечу, что скорость работы программ конкретно на кластере без очереди работает быстрее, чем способом с очередью. Возможно, это связанно с тем, что через очередь программы запускаются на другом процессоре. Либо счётчик секунд работает некорректно (хотя до этого он меня не подводил).

Решил, что честнее прогонять программу через очередь (файл [run.sh](#_Приложение_4_1)). По крайней мере это гораздо проще, можно написать один единственный файлик на сразу все программы. А через терминал прогонять ненадёжно, процессы могут быть убиты.

Таблица с компиляцией программы:

|  |  |  |
| --- | --- | --- |
| Файл | Описание | Команда компиляции |
| [lab1.c](#_Приложение_1) | Исходный файл без MPI | gcc lab1.c -std=gnu99 -o lab1.exe -lm |
| [lab1\_1.c](#_Приложение_2) | Вариант 1-ый с MPI | mpicc lab1\_1.c -std=gnu99 -o lab1\_1.exe -lm |
| [lab1\_2.c](#_Приложение_3) | Вариант 2-ой с MPI | mpicc lab1\_2.c -std=gnu99 -o lab1\_2.exe -lm |

Пояснение к команде компиляции:

* -std=gnu99 – используемый мною стандарт языка си
* -lm – позволяет использовать sqrt() из библиотеки “math.h”

Пояснение что за варианты программ:

* Вариант 1-ый с MPI – векторы x и b дублируются в каждом MPI-процессе
* Вариант 2-ой с MPI – векторы x и b разрезаются между MPI-процессами аналогично матрице A.

Таблица с результатами:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | без MPI | 1 | 2 | 4 | 8 | 16 |
| lab1\_1 | 42,478553 | 42,496013 | 22,825935 | 12,943475 | 8,431231 | 10,479614 |
| lab1\_2 | 43,174492 | 23,80283 | 14,382684 | 10,519845 | 17,477601 |

Таблица с ускорением и эффективностью:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | без MPI | 1 | 2 | 4 | 8 | 16 |
| lab1\_1 | 1,0 | 0,999589 | 1,860977 | 3,281850 | 5,038238 | 4,053446 |
| lab1\_2 | 0,983880 | 1,784600 | 2,953451 | 4,037944 | 2,430456 |

Графики с результатами:

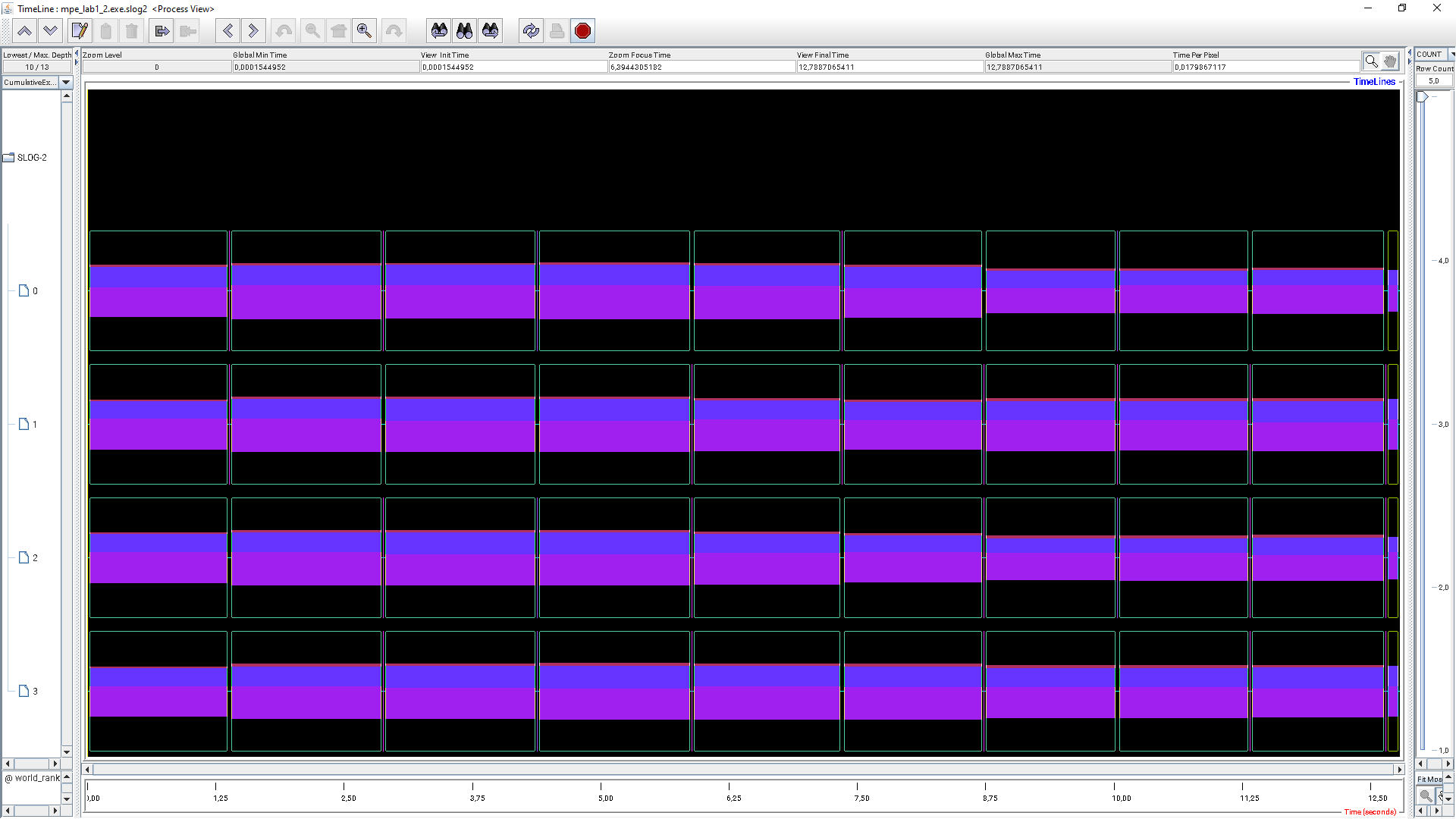
Далее я занялся профилированием. Команды компиляции были следующими:

* mpecc -mpilog -std=gnu99 -o mpe\_lab1\_1.exe lab1\_1.c -lm
* mpecc -mpilog -std=gnu99 -o mpe\_lab1\_2.exe lab1\_2.c -lm

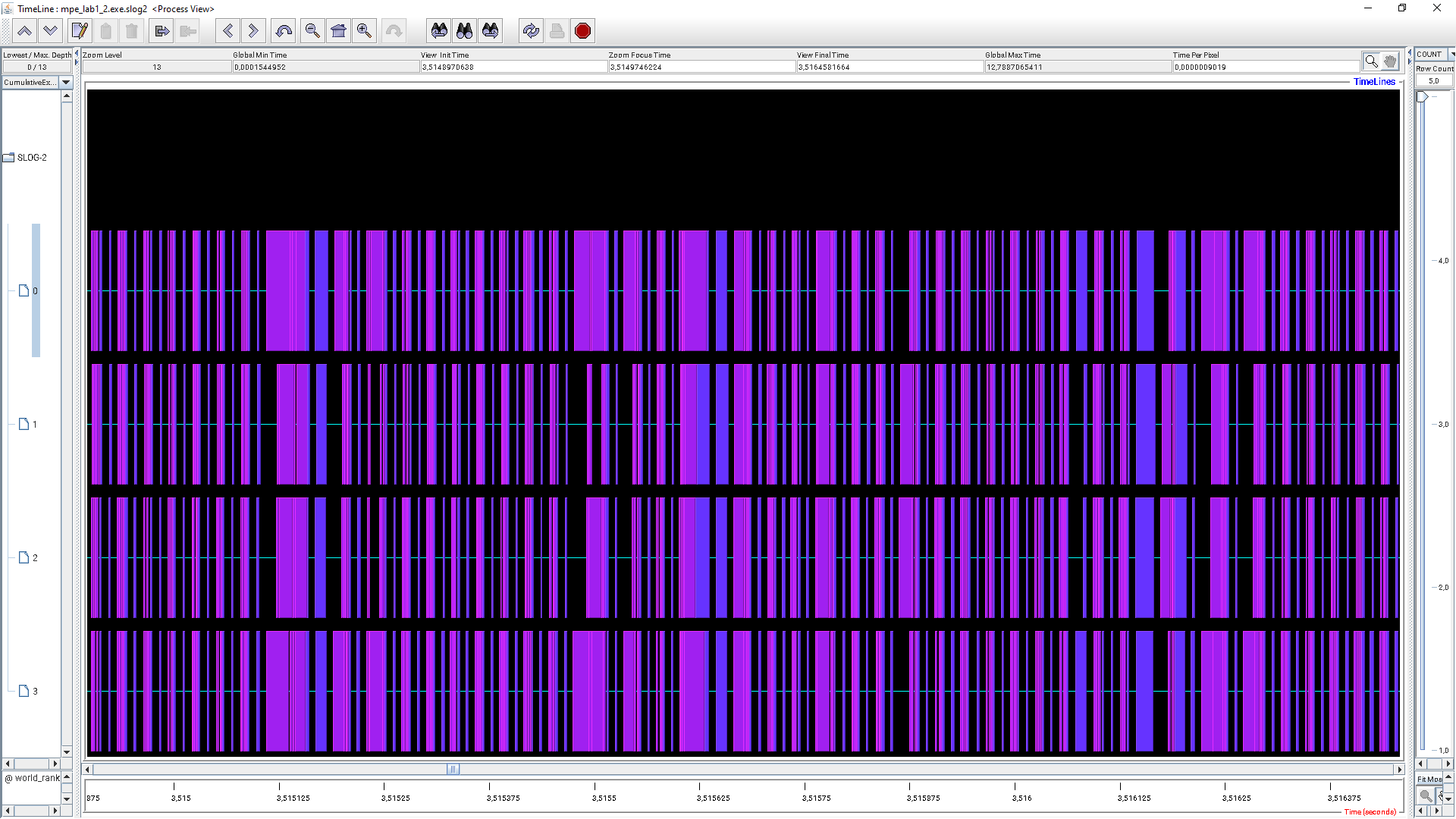
Профилирование практически одинаковые, за исключением того, что во 2ом варианте программы не используется MPI\_Allreduce. Поэтому там нет розовых полосок

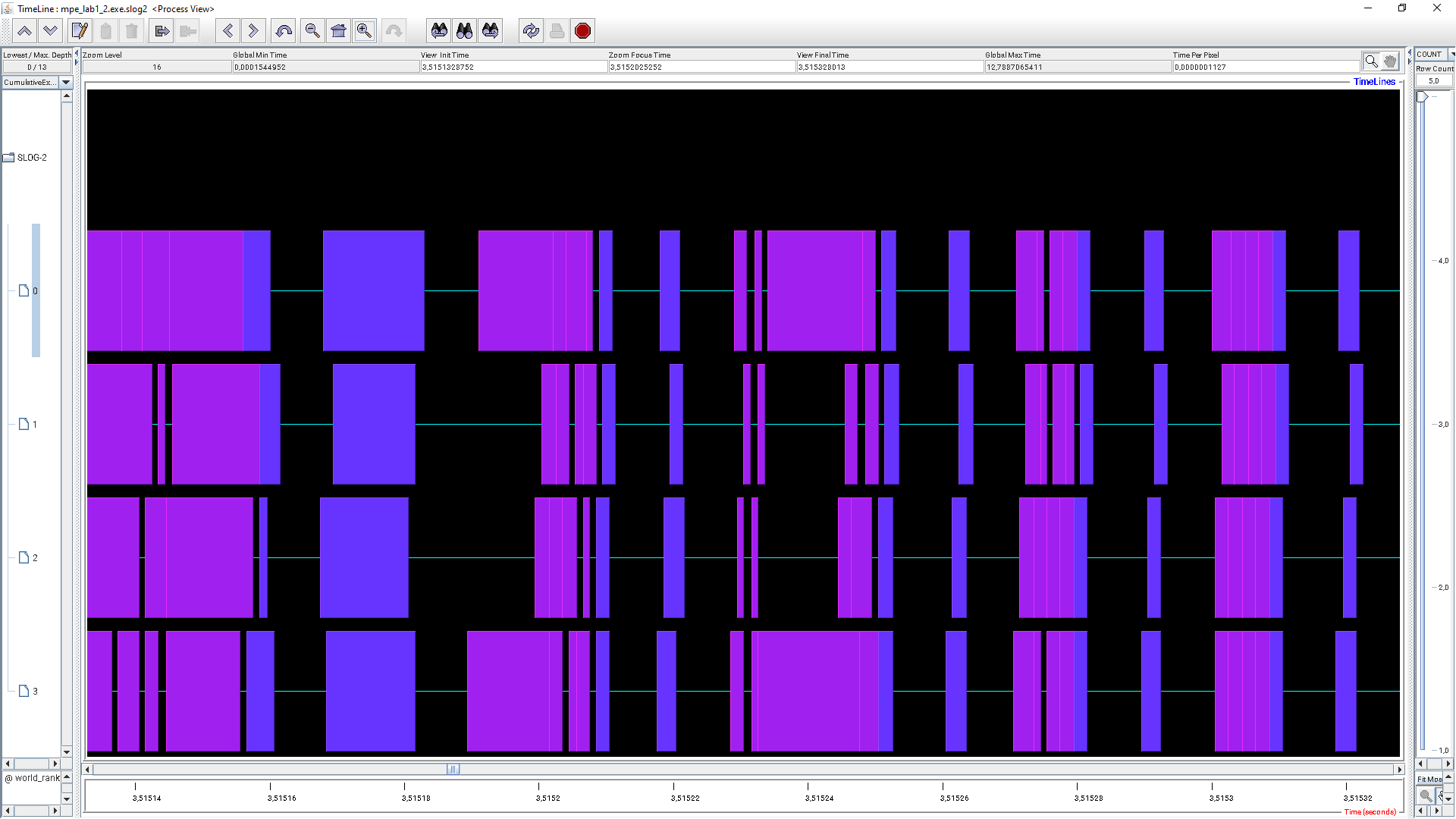
Скриншоты с профилировщика:

Zoom 0:

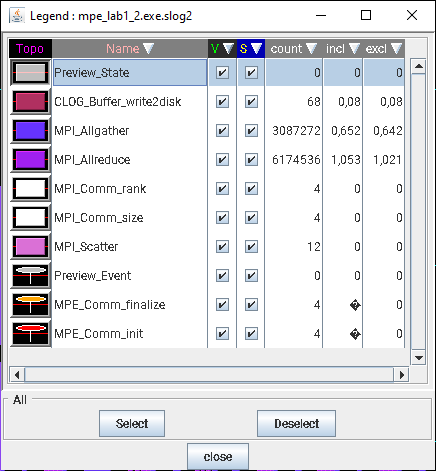


Zoom 13:



Zoom 16:

Легенда:



По скриншотам можно примерно понять, что вызовы функций, происходят приблизительно одновременно. Разница между вызовами только на секундах. На мой взгляд всё проходит относительно синхронно и задержек между процессами особо не наблюдается.

По заданию нужно проанализировать профилировщиком 16-и ядерную программу, однако памяти мне не хватает на кластере. Файл получается размером практически в 600МБ при 4-ёх ядрах. А дельнейшую переконвертацию уже не получается сделать. Пришлось использовать папку tmp для того, чтобы создать нужный файл для профилировщика.

По диаграммам времени получается так, что 2ой вариант программы работает медленнее. Думаю, это объясняется тем, что каждый раз на каждом процессе мы собираем вектор, чтобы потом посчитать произведение, когда как в 1ом варианте, такого делать не нужно.

Также стоит отметить, что производительность заканчивается на 8ом ядре, далее идёт только отрицательная динамика. Возможно, это связано с тем, что там уже используется 2 процессора, а не один, и наладить связь между ними становится сложнее. Нужно понимать, что итераций в цикле у меня 385908 (, ), вызовов функций MPI в сумме практически 10 млн (это только на 4-ех ядрах, на 16-и в 4 раза больше, а они все блокирующие). Получается это всё не только не ускоряет, а даже замедляет. И получается самые лучшие показатели на 8-и ядрах 1-ого варианта.

Возможно, нужно чтобы был 1 процессор с 16 ядрами, и тогда показатели могли быть другими. Или взять большой исчисляемый сотнями или тысячами и маленький , чтобы количество итераций было меньше 10000.

# **ЗАКЛЮЧЕНИЕ**

В ходе лабораторной работы я научился работать с библиотекой MPI и распараллеливать процессы с помощью неё. Понял, как работать с кластером и профилировщиком. Также я проанализировал полученные графики. И по ним можно сделать вывод о том, что 1-ый вариант работает лучше на больших количествах итераций. Но и с количеством ядер здесь тоже неоднозначно. Надо анализировать сколько их выбирать. Слишком большое число ядер может привести к увеличению времени работы программы.

# Приложение 1

*Листинг файла lab1.c*

*Исходный код без MPI*

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <time.h>

//----------Operations----------

// |x|=sqrt(sum(x\_i^2))

double modul(double\* x, int N)

{

double sum = 0.0;

for (int i = 0; i < N; i++)

{

sum += x[i] \* x[i];

}

return sqrt(sum);

}

// (x,y)=sum(x\_i\*y\_i)

double skobki(double\* x, double\* y, int N)

{

double res = 0.0;

for (int i = 0; i < N; i++)

{

res += x[i] \* y[i];

}

return res;

}

//multiplicate matrix to vector: result = x \* y (N - size; x - matrix; result, y - vector)

void mul\_M\_v(double\* result, double\* x, double\* y, int N)

{

double\* res = (double\*)malloc(N \* sizeof(double));

for (int i = 0; i < N; i++)

{

res[i] = 0.0;

}

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

res[i] += x[i \* N + j] \* y[j];

}

}

for (int i = 0; i < N; i++)

{

result[i] = res[i];

}

free(res);

}

//multiplicate constant to vector: result = c \* y (N - size; c - constant; result, y - vector)

void mul\_c\_v(double\* result, double c, double\* y, int N)

{

for (int i = 0; i < N; i++)

{

result[i] = c \* y[i];

}

}

//substraction vector to vector: result = x - y (N - size; result, x, y - vector)

void sub(double\* result, double\* x, double\* y, int N)//elited!!!

{

for (int i = 0; i < N; i++)

{

result[i] = x[i] - y[i];

}

}

//----------Setters/Getters/Print----------

void random\_vector(double\* x, int N)

{

for (int i = 0; i < N; i++)

{

x[i] = (double)(rand() % 18 - 9);

}

}

void symmetrical\_matrix(double\* A, int N)

{

for (int i = 0; i < N; i++)

{

A[i \* N + i] = (double)(rand() % 18 - 9);

}

for (int i = 0; i < N; i++)

{

for (int j = 0; j < i; j++)

{

A[i \* N + j] = (double)(rand() % 18 - 9);

A[j \* N + i] = A[i \* N + j];

}

}

}

void print\_matrix(double\* A, int N)

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

printf("%2g ", A[i \* N + j]);

}

printf("\n");

}

}

void print\_vector(double\* V, int N)

{

printf("(");

for (int i = 0; i < N; i++)

{

printf("%g ", V[i]);

}

printf(")\n");

}

//----------Algorithm----------

void y\_pow(double\* y, double\* A, double\* x, double\* b, int N)

{

double\* Ax = (double\*)malloc(N \* sizeof(double));

mul\_M\_v(Ax, A, x, N);

sub(y, Ax, b, N);

free(Ax);

}

double t\_pow(double\* y, double\* A, int N)

{

double\* Ay = (double\*)malloc(N \* sizeof(double));

mul\_M\_v(Ay, A, y, N);

double t = skobki(y, Ay, N) / skobki(Ay, Ay, N);

free(Ay);

return t;

}

void x\_pow(double\* y, double\* A, double\* x, double\* b, int N)

{

y\_pow(y, A, x, b, N);

double t = t\_pow(y, A, N);

double\* ty = (double\*)malloc(N \* sizeof(double));

mul\_c\_v(ty, t, y, N);

sub(x, x, ty, N);

free(ty);

}

int test(double\* y, double\* b, double epsilon, int N)

{

double t = modul(y, N) / modul(b, N);

if (t < epsilon)

{

return 1; //остановка

}

else

{

return 0; //продолжай, пока не 1

}

}

int main()

{

//Фиксируем время начала

struct timespec start, finish;

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

//Входные данные

srand(1); //чтобы входные данные при каждом запуске были разные

int N = 96;

double epsilon = 0.0001;

double\* A = (double\*)malloc(N \* N \* sizeof(double));

double\* x = (double\*)malloc(N \* sizeof(double));

double\* b = (double\*)malloc(N \* sizeof(double));

double\* y = (double\*)malloc(N \* sizeof(double)); // y=Ax-b

//Создаём матрицу A и x, считаем b, а затем переписываем x (как раз далее будем этот вектор находить)

symmetrical\_matrix(A, N);

random\_vector(x, N);

mul\_M\_v(b, A, x, N);

random\_vector(x, N);

y\_pow(y, A, x, b, N);

//Начало алгоритма

int i = 0; //счётчик итераций

while (test(y, b, epsilon, N) == 0)

{

x\_pow(y, A, x, b, N);

i++;

}

printf("Steps: %d\n", i);

free(A);

free(x);

free(b);

free(y);

//Фиксируем время конца

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

printf("Time: %lf\n", (finish.tv\_sec - start.tv\_sec + 0.000000001 \* (finish.tv\_nsec - start.tv\_nsec)));

return 0;

}

//http://rsusu1.rnd.runnet.ru/tutor/method/m2/page09.html

//https://www.opennet.ru/docs/RUS/MPI\_intro/#Barrier

# Приложение 2

*Листинг файла lab1\_1.c*

*1-ый вариант программы*

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <time.h>

#include "mpi.h"

//----------Operations----------

// |x|=sqrt(sum(x\_i^2))

double modul(double\* x, int N)

{

double sum = 0.0;

for (int i = 0; i < N; i++)

{

sum += x[i] \* x[i];

}

return sqrt(sum);

}

// (x,y)=sum(x\_i\*y\_i)

double skobki(double\* x, double\* y, int N)

{

double res = 0.0;

for (int i = 0; i < N; i++)

{

res += x[i] \* y[i];

}

return res;

}

//multiplicate matrix to vector: result = x \* y (N - size; x - matrix; result, y - vector)

void mul\_M\_v(double\* result, double\* x, double\* y, int N, int M, int rank)

{

double\* res = (double\*)calloc(M, sizeof(double));

for (int i = 0; i < M; i++)

{

res[i] = 0.0;

}

for (int i = 0; i < M; i++)

{

for (int j = 0; j < N; j++)

{

res[i] += x[i \* N + j] \* y[j];

}

}

MPI\_Allgather(res, M, MPI\_DOUBLE, result, M, MPI\_DOUBLE, MPI\_COMM\_WORLD);

free(res);

}

//multiplicate constant to vector: result = c \* y (N - size; c - constant; result, y - vector)

void mul\_c\_v(double\* result, double c, double\* y, int N)

{

for (int i = 0; i < N; i++)

{

result[i] = c \* y[i];

}

}

//substraction vector to vector: result = x - y (N - size; result, x, y - vector)

void sub(double\* result, double\* x, double\* y, int N)

{

for (int i = 0; i < N; i++)

{

result[i] = x[i] - y[i];

}

}

//----------Setters/Getters/Print----------

void random\_vector(double\* x, int N)

{

for (int i = 0; i < N; i++)

{

x[i] = (double)(rand() % 18 - 9);

}

}

void symmetrical\_matrix(double\* A, int N, int M)

{

double\* matrix = (double\*)calloc(N \* N, sizeof(double));

for (int i = 0; i < N; i++)

{

matrix[i \* N + i] = (double)(rand() % 18 - 9);

}

for (int i = 0; i < N; i++)

{

for (int j = 0; j < i; j++)

{

matrix[i \* N + j] = (double)(rand() % 18 - 9);

matrix[j \* N + i] = matrix[i \* N + j];

}

}

MPI\_Scatter(matrix, M \* N, MPI\_DOUBLE, A, M \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

free(matrix);

}

void print\_matrix(double\* A, int N, int M, int rank, int size)

{

for (int h = 0; h < size; h++)

{

MPI\_Barrier(MPI\_COMM\_WORLD);

if (rank == h)

{

for (int i = 0; i < M; i++)

{

for (int j = 0; j < N; j++)

{

printf("%2g ", A[i \* N + j]);

}

printf("\n");

}

}

}

}

void print\_vector(double\* V, int N)

{

printf("(");

for (int i = 0; i < N; i++)

{

printf("%g ", V[i]);

}

printf(")\n");

}

//----------Algorithm----------

void y\_pow(double\* y, double\* A, double\* x, double\* b, int N, int M, int rank)

{

double\* Ax = (double\*)calloc(N, sizeof(double));

mul\_M\_v(Ax, A, x, N, M, rank);

sub(y, Ax, b, N);

free(Ax);

}

double t\_pow(double\* y, double\* A, int N, int M, int rank)

{

double\* Ay = (double\*)calloc(N, sizeof(double));

mul\_M\_v(Ay, A, y, N, M, rank);

double t = skobki(y, Ay, N) / skobki(Ay, Ay, N);

free(Ay);

return t;

}

void x\_pow(double\* y, double\* A, double\* x, double\* b, int N, int M, int rank)

{

y\_pow(y, A, x, b, N, M, rank);

double t = t\_pow(y, A, N, M, rank);

double\* ty = (double\*)calloc(N, sizeof(double));

mul\_c\_v(ty, t, y, N);

sub(x, x, ty, N);

free(ty);

}

int test(double\* y, double\* b, double epsilon, int N)

{

double t = modul(y, N) / modul(b, N);

if (t < epsilon)

{

return 1; //остановка

}

else

{

return 0; //продолжай, пока не 1

}

}

int main(int argc, char\* argv[])

{

//Фиксируем время начала

struct timespec start, finish;

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

//MPI

int size;

int rank;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

printf("Process %d of %d\n", rank, size);

//Входные данные

srand(1); //чтобы входные данные при каждом запуске были разные

int N = 96;

if (N % size == 0)

{

int M = N / size; //делим на ровные кусочки

double epsilon = 0.0001;

double\* A = (double\*)calloc(N \* M, sizeof(double));

double\* x = (double\*)calloc(N, sizeof(double));

double\* b = (double\*)calloc(N, sizeof(double));

double\* y = (double\*)calloc(N, sizeof(double)); // y=Ax-b

//Создаём матрицу A и x, считаем b, а затем переписываем x (как раз далее будем этот вектор находить)

symmetrical\_matrix(A, N, M);

random\_vector(x, N);

mul\_M\_v(b, A, x, N, M, rank);

random\_vector(x, N);

y\_pow(y, A, x, b, N, M, rank);

//Начало алгоритма

int i = 0; //счётчик итераций

while (test(y, b, epsilon, N) == 0)

{

x\_pow(y, A, x, b, N, M, rank);

i++;

}

printf("Steps: %d\n", i);

free(A);

free(x);

free(b);

free(y);

}

else

{

printf("Error! Matrix size is not a multiple of the number of threads: size % threads != 0\n");

}

MPI\_Finalize();

//Фиксируем время конца

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

printf("Time: %lf\n", (finish.tv\_sec - start.tv\_sec + 0.000000001 \* (finish.tv\_nsec - start.tv\_nsec)));

return 0;

}

# Приложение 3

*Листинг файла lab1\_2.c*

*Второй вариант программы*

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <time.h>

#include "mpi.h"

//----------Operations----------

// |x|=sqrt(sum(x\_i^2))

double modul(double\* x, int M)

{

double sum = 0.0;

for (int i = 0; i < M; i++)

{

sum += x[i] \* x[i];

}

double allsum = 0.0;

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

return sqrt(allsum);

}

// (x,y)=sum(x\_i\*y\_i)

double skobki(double\* x, double\* y, int M)

{

double res = 0.0;

for (int i = 0; i < M; i++)

{

res += x[i] \* y[i];

}

double allres = 0.0;

MPI\_Allreduce(&res, &allres, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

return allres;

}

//multiplicate matrix to vector: result = x \* y (N - size; x - matrix; result, y - vector)

void mul\_M\_v(double\* result, double\* x, double\* y, int N, int M, int rank)

{

double\* res = (double\*)calloc(M, sizeof(double));

for (int i = 0; i < M; i++)

{

res[i] = 0.0;

}

double\* Y = (double\*)calloc(N, sizeof(double));

MPI\_Allgather(y, M, MPI\_DOUBLE, Y, M, MPI\_DOUBLE, MPI\_COMM\_WORLD);

for (int i = 0; i < M; i++)

{

for (int j = 0; j < N; j++)

{

res[i] += x[i \* N + j] \* Y[j];

}

}

for (int i = 0; i < M; i++)

{

result[i] = res[i];

}

free(Y);

free(res);

}

//multiplicate constant to vector: result = c \* y (N - size; c - constant; result, y - vector)

void mul\_c\_v(double\* result, double c, double\* y, int M)

{

for (int i = 0; i < M; i++)

{

result[i] = c \* y[i];

}

}

//substraction vector to vector: result = x - y (N - size; result, x, y - vector)

void sub(double\* result, double\* x, double\* y, int M)

{

for (int i = 0; i < M; i++)

{

result[i] = x[i] - y[i];

}

}

//----------Setters/Getters/Print----------

void random\_vector(double\* x, int N, int M)

{

double\* vector = (double\*)calloc(N, sizeof(double));

for (int i = 0; i < N; i++)

{

vector[i] = (double)(rand() % 18 - 9);

}

MPI\_Scatter(vector, M, MPI\_DOUBLE, x, M, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

free(vector);

}

void symmetrical\_matrix(double\* A, int N, int M)

{

double\* matrix = (double\*)calloc(N \* N, sizeof(double));

for (int i = 0; i < N; i++)

{

matrix[i \* N + i] = (double)(rand() % 18 - 9);

}

for (int i = 0; i < N; i++)

{

for (int j = 0; j < i; j++)

{

matrix[i \* N + j] = (double)(rand() % 18 - 9);

matrix[j \* N + i] = matrix[i \* N + j];

}

}

MPI\_Scatter(matrix, M \* N, MPI\_DOUBLE, A, M \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

free(matrix);

}

void print\_matrix(double\* A, int N, int M, int rank, int size)

{

for (int h = 0; h < size; h++)

{

MPI\_Barrier(MPI\_COMM\_WORLD);

if (rank == h)

{

for (int i = 0; i < M; i++)

{

for (int j = 0; j < N; j++)

{

printf("%2g ", A[i \* N + j]);

}

printf("\n");

}

}

}

}

void print\_vector(double\* V, int M, int rank, int size)

{

if (rank == 0)

{

printf("(");

}

for (int h = 0; h < size; h++)

{

MPI\_Barrier(MPI\_COMM\_WORLD);

if (rank == h)

{

for (int i = 0; i < M; i++)

{

printf("%g ", V[i]);

}

}

}

if (rank == 0)

{

printf(")\n");

}

}

//----------Algorithm----------

void y\_pow(double\* y, double\* A, double\* x, double\* b, int N, int M, int rank)

{

double\* Ax = (double\*)calloc(M, sizeof(double));

mul\_M\_v(Ax, A, x, N, M, rank);

sub(y, Ax, b, M);

free(Ax);

}

double t\_pow(double\* y, double\* A, int N, int M, int rank)

{

double\* Ay = (double\*)calloc(M, sizeof(double));

mul\_M\_v(Ay, A, y, N, M, rank);

double t = skobki(y, Ay, M) / skobki(Ay, Ay, M);

free(Ay);

return t;

}

void x\_pow(double\* y, double\* A, double\* x, double\* b, int N, int M, int rank)

{

y\_pow(y, A, x, b, N, M, rank);

double t = t\_pow(y, A, N, M, rank);

double\* ty = (double\*)calloc(M, sizeof(double));

mul\_c\_v(ty, t, y, M);

sub(x, x, ty, M);

free(ty);

}

int test(double\* y, double\* b, double epsilon, int M)

{

double t = modul(y, M) / modul(b, M);

if (t < epsilon)

{

return 1; //остановка

}

else

{

return 0; //продолжай, пока не 1

}

}

int main(int argc, char\* argv[])

{

//Фиксируем время начала

struct timespec start, finish;

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

//MPI

int size;

int rank;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

printf("Process %d of %d\n", rank, size);

//Входные данные

srand(1); //чтобы входные данные при каждом запуске были разные

int N = 96;

if (N % size == 0)

{

int M = N / size; //делим на ровные кусочки

double epsilon = 0.0001;

double\* A = (double\*)calloc(N \* M, sizeof(double));

double\* x = (double\*)calloc(M, sizeof(double));

double\* b = (double\*)calloc(M, sizeof(double));

double\* y = (double\*)calloc(M, sizeof(double)); // y=Ax-b

//Создаём матрицу A и x, считаем b, а затем переписываем x (как раз далее будем этот вектор находить)

symmetrical\_matrix(A, N, M);

random\_vector(x, N, M);

mul\_M\_v(b, A, x, N, M, rank);

random\_vector(x, N, M);

y\_pow(y, A, x, b, N, M, rank);

//Начало алгоритма

int i = 0; //счётчик итераций

while (test(y, b, epsilon, M) == 0)

{

x\_pow(y, A, x, b, N, M, rank);

i++;

}

printf("Steps: %d\n", i);

free(A);

free(x);

free(b);

free(y);

}

else

{

printf("Error! Matrix size is not a multiple of the number of threads: size % threads != 0\n");

}

MPI\_Finalize();

//Фиксируем время конца

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

printf("Time: %lf\n", (finish.tv\_sec - start.tv\_sec + 0.000000001 \* (finish.tv\_nsec - start.tv\_nsec)));

return 0;

}

# Приложение 4

*Листинг файла run.sh*

#!/bin/bash

#PBS -l walltime=00:10:00

#PBS -l select=2:ncpus=8:mpiprocs=8:mem=2000m,place=free

#PBS -m n

cd $PBS\_O\_WORKDIR

MPI\_NP=$(wc -l $PBS\_NODEFILE | awk '{ print $1 }')

echo "Number of MPI process: $MPI\_NP"

echo 'File $PBS\_NODEFILE:'

cat $PBS\_NODEFILE

echo

./lab1.exe

mpirun -machinefile $PBS\_NODEFILE -np 1 ./lab1\_1.exe

mpirun -machinefile $PBS\_NODEFILE -np 2 ./lab1\_1.exe

mpirun -machinefile $PBS\_NODEFILE -np 4 ./lab1\_1.exe

mpirun -machinefile $PBS\_NODEFILE -np 8 ./lab1\_1.exe

mpirun -machinefile $PBS\_NODEFILE -np 16 ./lab1\_1.exe

mpirun -machinefile $PBS\_NODEFILE -np 1 ./lab1\_2.exe

mpirun -machinefile $PBS\_NODEFILE -np 2 ./lab1\_2.exe

mpirun -machinefile $PBS\_NODEFILE -np 4 ./lab1\_2.exe

mpirun -machinefile $PBS\_NODEFILE -np 8 ./lab1\_2.exe

mpirun -machinefile $PBS\_NODEFILE -np 16 ./lab1\_2.exe

*Результат*

Number of MPI process: 16

File $PBS\_NODEFILE:

cn274.hpc.nusc.ru

cn274.hpc.nusc.ru

cn274.hpc.nusc.ru

cn274.hpc.nusc.ru

cn274.hpc.nusc.ru

cn274.hpc.nusc.ru

cn274.hpc.nusc.ru

cn274.hpc.nusc.ru

cn201.hpc.nusc.ru

cn201.hpc.nusc.ru

cn201.hpc.nusc.ru

cn201.hpc.nusc.ru

cn201.hpc.nusc.ru

cn201.hpc.nusc.ru

cn201.hpc.nusc.ru

cn201.hpc.nusc.ru

Steps: 385908

Time: 42.478553

Process 0 of 1

Steps: 385908

Time: 42.496013

Process 0 of 2

Process 1 of 2

Steps: 385908

Steps: 385908

Time: 22.825935

Time: 22.826366

Process 0 of 4

Process 1 of 4

Process 2 of 4

Process 3 of 4

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Time: 12.943475

Time: 12.942082

Time: 12.944334

Time: 12.944479

Process 3 of 8

Process 6 of 8

Process 7 of 8

Process 0 of 8

Process 1 of 8

Process 4 of 8

Process 5 of 8

Process 2 of 8

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Time: 8.431231

Time: 8.430812

Time: 8.420449

Time: 8.427406

Time: 8.431130

Time: 8.427792

Time: 8.412554

Time: 8.428255

Process 0 of 16

Process 1 of 16

Process 2 of 16

Process 3 of 16

Process 4 of 16

Process 5 of 16

Process 6 of 16

Process 8 of 16

Process 7 of 16

Process 9 of 16

Process 10 of 16

Process 11 of 16

Process 12 of 16

Process 13 of 16

Process 14 of 16

Process 15 of 16

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Time: 10.479614

Time: 10.480190

Time: 10.479350

Time: 10.480878

Time: 10.479869

Time: 10.027515

Time: 10.479026

Time: 10.476945

Time: 10.027796

Time: 10.028261

Time: 10.028010

Time: 10.028173

Time: 10.030147

Time: 10.028440

Time: 10.480835

Time: 10.030290

Process 0 of 1

Steps: 385908

Time: 43.174492

Process 0 of 2

Process 1 of 2

Steps: 385908

Steps: 385908

Time: 23.802830

Time: 23.803232

Process 0 of 4

Process 1 of 4

Process 2 of 4

Process 3 of 4

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Time: 14.382684

Time: 14.386401

Time: 14.393357

Time: 14.393510

Process 0 of 8

Process 1 of 8

Process 2 of 8

Process 3 of 8

Process 4 of 8

Process 5 of 8

Process 6 of 8

Process 7 of 8

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Time: 10.519845

Time: 10.520401

Time: 10.512342

Time: 10.516590

Time: 10.517042

Time: 10.512591

Time: 10.520394

Time: 10.517623

Process 0 of 16

Process 1 of 16

Process 2 of 16

Process 3 of 16

Process 4 of 16

Process 5 of 16

Process 6 of 16

Process 7 of 16

Process 8 of 16

Process 9 of 16

Process 10 of 16

Process 11 of 16

Process 12 of 16

Process 13 of 16

Process 14 of 16

Process 15 of 16

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Steps: 385908

Time: 17.477601

Time: 17.477553

Time: 17.476795

Time: 17.474250

Time: 17.025444

Time: 17.025238

Time: 17.477399

Time: 17.025600

Time: 17.474473

Time: 17.476612

Time: 17.025909

Time: 17.026084

Time: 17.027694

Time: 17.026898

Time: 17.481412

Time: 17.027113