**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 ПРАКТИЧЕСКОЙ РАБОТЫ**

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

студента 2 курса, группы 21206

**Мельникова Никиты Сергеевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

к.т.н., доцент

А.Ю. Власенко

Новосибирск 2023

**СОДЕРЖАНИЕ**

[**ЦЕЛЬ**](#_heading=h.79v9ra25yi03) 3

[**ЗАДАНИЕ**](#_heading=h.5z6twkpbirop) 3

[**ОПИСАНИЕ РАБОТЫ**](#_heading=h.hm69z85m7z0q) 4

[**ЗАКЛЮЧЕНИЕ**](#_heading=h.85qrjnqdo1v3) 8

[**Приложение 1.** Метод минимальных невязок](#_heading=h.i6onqhcnticq)9

[**Приложение 2.** Листинг последовательной программы на языке Си](#_heading=h.y7ghj6grs9vq)10

[**Приложение 3.** Листинг параллельной программы на языке Си](#_heading=h.oybk5c32m0mm)13

# ЦЕЛЬ

1. Реализовать итерационный алгоритм решения системы линейных алгебраических уравнений.
2. Изучить межпроцессорное взаимодействие MPI-программы, познакомиться с профилированием.

# ЗАДАНИЕ

1. Написать 2 программы (последовательную и параллельную с использованием MPI) на языке C/C++, которые реализуют 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. Параллельную программу реализовать с тем условием, что матрица A и вектор b инициализируются на каком-либо одном процессе, а затем матрица A «разрезается» по строкам на близкие по размеру, возможно не одинаковые, части, а вектор b раздается каждому процессу. Уделить внимание тому, чтобы при запуске программы на различном числе MPI-процессов решалась одна и та же задача (исходные данные заполнялись одинаковым образом).
3. Замерить время работы последовательного варианта программы, а также время работы параллельного при использовании различного числа процессорных ядер. Минимально на 2, 4, 8, 16, 24 (на каждом из наших узлов кластера по 12 ядер), но чем на большем количестве различного числа процессов будут выполнены замеры, тем лучше. Также чем больше замеров будет выполнено на одном и том же числе процессов, тем лучше. В этом случае для построения графиков следует брать минимальное время. 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.
4. Выполнить профилирование двух вариантов программы с помощью jumpshot или ITAC (Intel Trace Analyzer and Collector) при использовании 16-и или 24-х ядер.
5. На основании полученных результатов сделать вывод.

# ОПИСАНИЕ РАБОТЫ

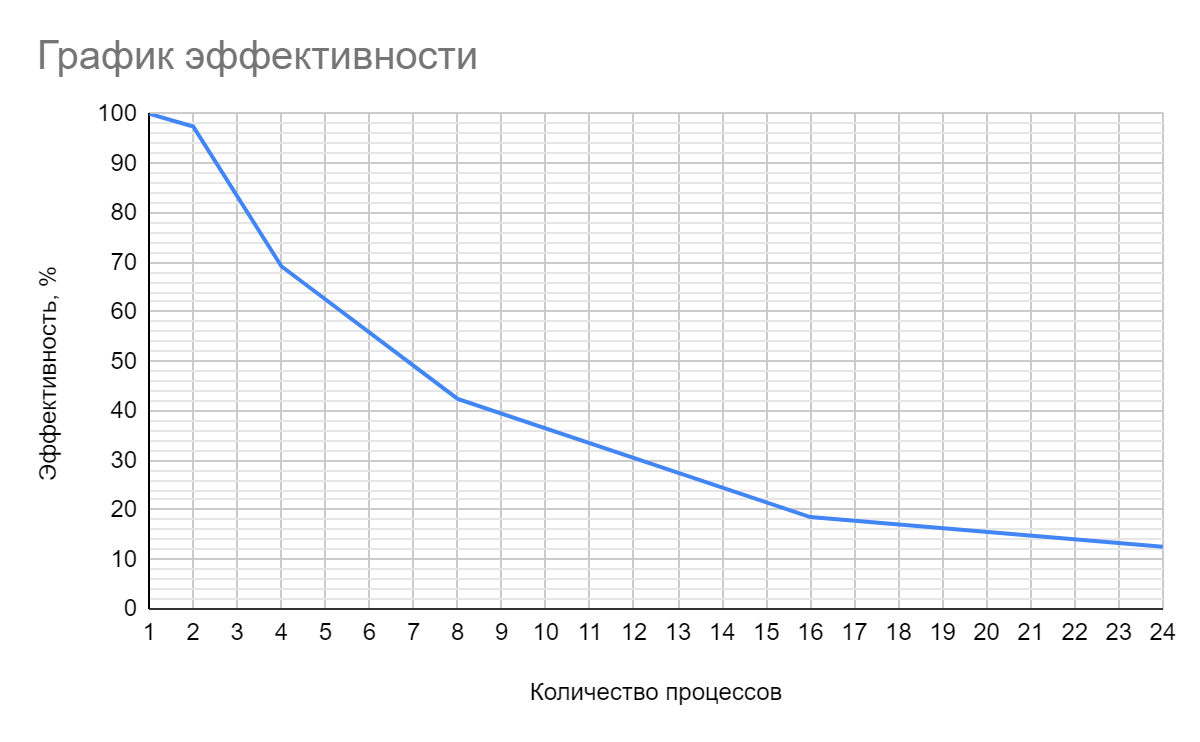
1. Было написано две программы на языке C (последовательная (листинг 1) и параллельная с использованием MPI (листинг 2)), реализующих итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b с использованием метода минимальных невязок (приложение 1). Здесь A– матрица размером N×N, x и b – векторы длины N. Тип элементов – double.
2. Было замерено время работы последовательной программы. Данные выбраны таким образом, чтобы время работы последовательной программы было не менее 30 сек (взята симметричная матрица со случайными элементами на отрезке [-1;1] с утяжеленной главной диагональю).
3. Было замерено время работы параллельной программы на 2, 4, 8, 16, 24 процессах.
4. Были построены графики ускорения и эффективности, где

**Ускорение**: Sp = T1 / Tp, где T1 - время работы ПОСЛЕДОВАТЕЛЬНОЙ программы. Tp - время работы параллельной программы на p процессах/потоках.

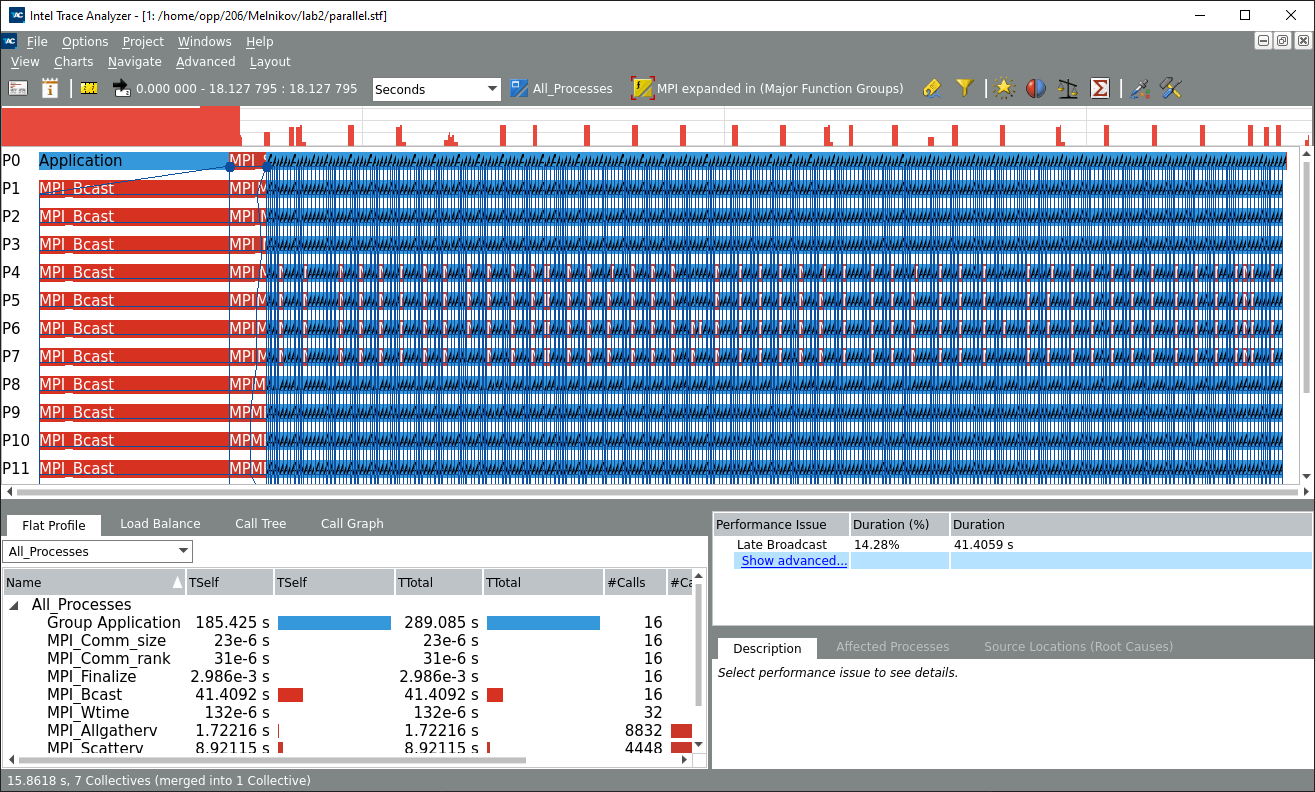
**Эффективность**: Ep = Sp / p \* 100%

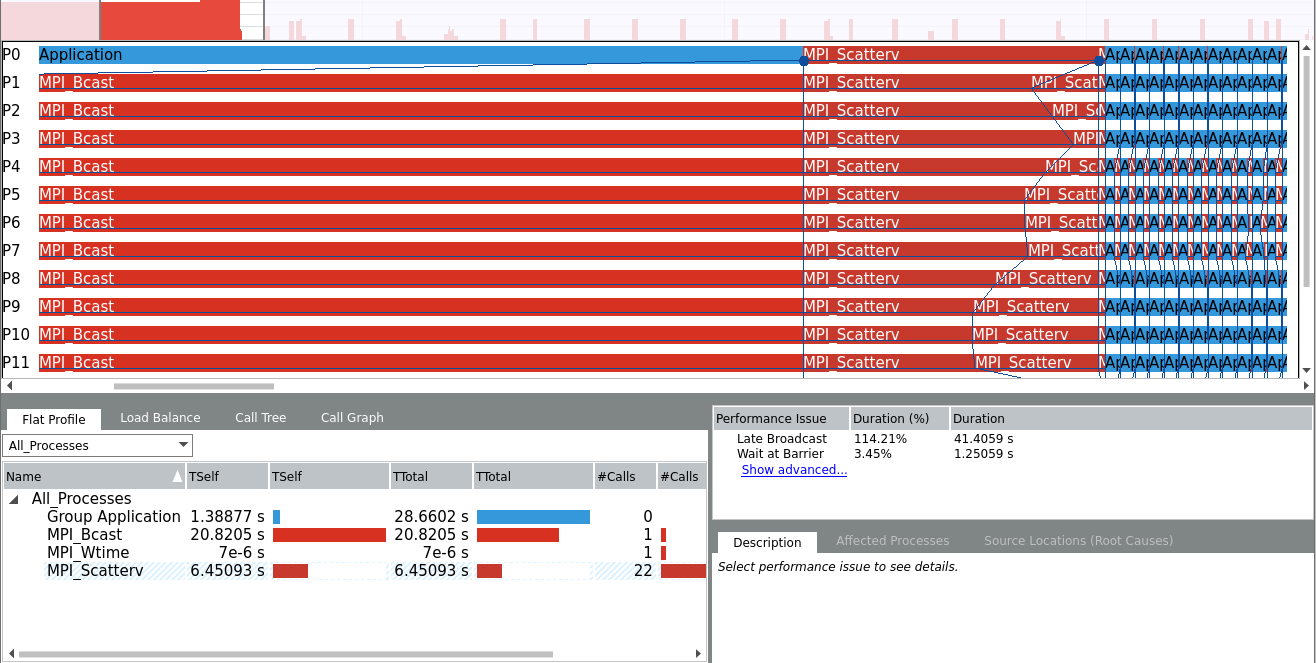


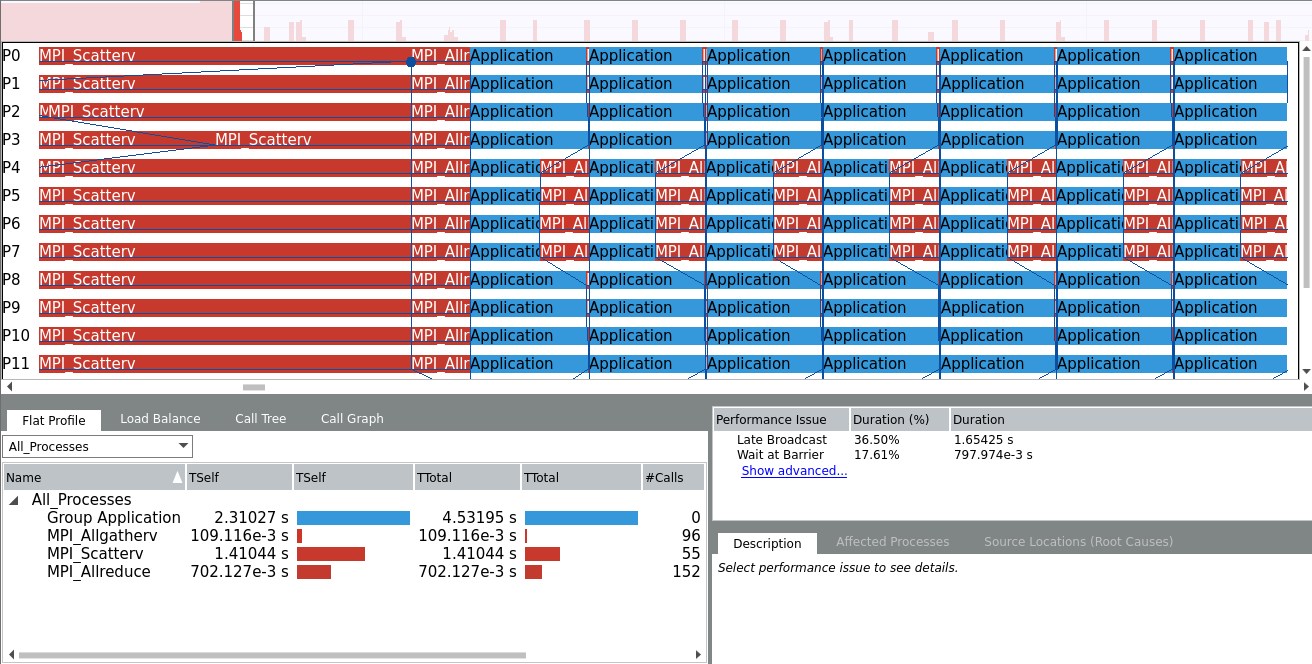




1. Было выполнено профилирование с помощью ITAC (Intel Trace Analyzer and Collector).





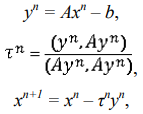


# ЗАКЛЮЧЕНИЕ

В ходе практической работы удалось выявить существенное существенное повышение производительности при использовании параллельных программ по сравнению с последовательными. При увеличении количества процессов время работы программы уменьшается. В процессе профилирования замечено, что операции коммуникации могут потратить много времени на ожидание данных от корневого процесса (в частности MPI\_Bcast в начале программы на некорневых процессах ожидает отправку вектора x из корневого процесса).

# **Приложение 1.***Метод минимальных невязок*

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



Критерий завершения счета:



# **Приложение 2.***Листинг последовательной программы на языке Си*

*Листинг 1 — Исходный код программы*

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define N 10000

#define EPSILON 0.000001

#define MAX\_ITERATION\_COUNT 10000

#define DIAGONAL\_DOMINANCE 120

#define MEASURES 3

double \*fill\_matrix(double \*A) {

srand(N);

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

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

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

}

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

A[i \* N + j] = (double) rand() / RAND\_MAX \* 2.0 - 1.0; //in [-1; 1]

if (i == j) A[i \* N + j] += DIAGONAL\_DOMINANCE;

}

}

return A;

}

double \*fill\_vect(double \*vect) {

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

vect[i] = (double) rand() / RAND\_MAX \* (double) rand();

return vect;

}

void matrix\_to\_vect\_mul(const double \*matrix, const double \*vect, double \*result) {

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

double tmp = 0;

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

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

}

result[i] = tmp;

}

}

void vect\_to\_scalar\_mul(const double \*vect, double scalar, double \*result) {

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

result[i] = vect[i] \* scalar;

}

void vect\_sub(const double \*vect1, const double \*vect2, double \*result) {

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

result[i] = vect1[i] - vect2[i];

}

double scalar\_mul(const double \*vect1, const double \*vect2) {

double result = 0;

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

result += vect1[i] \* vect2[i];

return result;

}

void solve(double \*A, double \*x, double \*b) {

double \*y\_n = (double \*) malloc(N \* sizeof(double));

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

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

double tau;

size\_t iterations = 0;

double epsilonb\_norm\_squared = EPSILON \* EPSILON \* scalar\_mul(b, b);

//calculate y\_n

matrix\_to\_vect\_mul(A, x, tmp);

vect\_sub(tmp, b, y\_n);

while (iterations < MAX\_ITERATION\_COUNT) {

matrix\_to\_vect\_mul(A, y\_n, Ay\_n); //Ay\_n = A \* y\_n

//calculate tau

tau = scalar\_mul(y\_n, Ay\_n) / scalar\_mul(Ay\_n, Ay\_n);

//calculate x\_{n+1}

vect\_to\_scalar\_mul(y\_n, tau, tmp);

vect\_sub(x, tmp, x);

//calc y\_{n+1}

matrix\_to\_vect\_mul(A, x, tmp); //tmp = A \* x

vect\_sub(tmp, b, y\_n); //y\_n = A \* x - b

++iterations;

if (scalar\_mul(y\_n, y\_n) < epsilonb\_norm\_squared) break;

}

free(y\_n);

free(Ay\_n);

free(tmp);

}

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

double time\_start = 0;

double time\_end = 0;

double min\_time = ULLONG\_MAX;

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

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

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

MPI\_Init(&argc, &argv);

for (int i = 0; i < MEASURES; ++i) {

fill\_matrix(A);

fill\_vect(x);

fill\_vect(b);

time\_start = MPI\_Wtime();

solve(A, x, b);

time\_end = MPI\_Wtime();

if (time\_end - time\_start < min\_time) min\_time = time\_end - time\_start;

}

printf("Time taken: %f sec.\n", min\_time);

free(A);

free(x);

free(b);

MPI\_Finalize();

return EXIT\_SUCCESS;

}

# **Приложение 3.***Листинг параллельной программы на языке Си*

*Листинг 2 — Исходный код программы*

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define N 10000

#define EPSILON 0.000001

#define MAX\_ITERATION\_COUNT 10000

#define DIAGONAL\_DOMINANCE 120

#define MEASURES 3

double \*fill\_matrix(double \*A) {

srand(N);

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

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

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

}

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

A[i \* N + j] = (double) rand() / RAND\_MAX \* 2.0 - 1.0; //in [-1; 1]

if (i == j) A[i \* N + j] += DIAGONAL\_DOMINANCE;

}

}

return A;

}

double \*fill\_vect(double \*vect) {

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

vect[i] = (double) rand() / RAND\_MAX \* (double) rand();

return vect;

}

void matrix\_to\_vect\_mul(const double \*matrix, int num\_of\_lines, const double \*vect, double \*result) {

for (int i = 0; i < num\_of\_lines; ++i) {

double tmp = 0;

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

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

}

result[i] = tmp;

}

}

void vect\_to\_scalar\_mul(const double \*vect, int len, double scalar, double \*result) {

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

result[i] = vect[i] \* scalar;

}

void vect\_sub(const double \*vect1, const double \*vect2, int len, double \*result) {

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

result[i] = vect1[i] - vect2[i];

}

double scalar\_mul(const double \*vect1, const double \*vect2, int len) {

double result = 0;

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

result += vect1[i] \* vect2[i];

return result;

}

void set\_matrix\_part(int \*sendcounts, int \*displs, int \*linecounts, int \*offsets, int size) {

int offset = 0;

for (int i = 0; i < size; ++i) {

linecounts[i] = N / size;

if (i < N % size)

++linecounts[i];

offsets[i] = offset;

offset += linecounts[i];

sendcounts[i] = linecounts[i] \* N;

displs[i] = offsets[i] \* N;

}

}

inline void solve(double \*A, double \*x, double \*b, int rank, int size) {

double epsilonb\_norm\_squared;

double \*y\_n = (double \*) malloc(N \* sizeof(double));

double tau;

double temp;

MPI\_Bcast(x, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

int \*sendcounts = (int \*) malloc(size \* sizeof(int));

int \*displs = (int \*) malloc(size \* sizeof(int));

int \*linecounts = (int \*) malloc(size \* sizeof(int));

int \*offsets = (int \*) malloc(size \* sizeof(int));

set\_matrix\_part(sendcounts, displs, linecounts, offsets, size);

double \*A\_part = (double \*) malloc(sendcounts[rank] \* sizeof(double));

double \*b\_part = (double \*) malloc(linecounts[rank] \* sizeof(double));

double \*x\_part = (double \*) malloc(linecounts[rank] \* sizeof(double));

double \*y\_part = (double \*) malloc(linecounts[rank] \* sizeof(double));

double \*Ay\_part = (double \*) malloc(linecounts[rank] \* sizeof(double));

double \*tmp = (double \*) malloc(linecounts[rank] \* sizeof(double));

size\_t iter = 0;

MPI\_Scatterv(A, sendcounts, displs, MPI\_DOUBLE, A\_part, sendcounts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatterv(b, linecounts, offsets, MPI\_DOUBLE, b\_part, linecounts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

temp = EPSILON \* EPSILON \* scalar\_mul(b\_part, b\_part, linecounts[rank]);

MPI\_Allreduce(&temp, &epsilonb\_norm\_squared, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

//calculate y\_part

matrix\_to\_vect\_mul(A\_part, linecounts[rank], x, tmp);

vect\_sub(tmp, b\_part, linecounts[rank], y\_part);

while (iter < MAX\_ITERATION\_COUNT) {

MPI\_Allgatherv(y\_part, linecounts[rank], MPI\_DOUBLE, y\_n, linecounts, offsets, MPI\_DOUBLE, MPI\_COMM\_WORLD);

matrix\_to\_vect\_mul(A\_part, linecounts[rank], y\_n, Ay\_part);

double tau\_numerator, tau\_denominator;

temp = scalar\_mul(y\_part, Ay\_part, linecounts[rank]);

MPI\_Allreduce(&temp, &tau\_numerator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

temp = scalar\_mul(Ay\_part, Ay\_part, linecounts[rank]);

MPI\_Allreduce(&temp, &tau\_denominator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

//calculate tau

tau = tau\_numerator / tau\_denominator;

//calculate x\_{n+1}

vect\_to\_scalar\_mul(y\_part, linecounts[rank], tau, tmp);

MPI\_Scatterv(x, linecounts, offsets, MPI\_DOUBLE, x\_part, linecounts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

vect\_sub(x\_part, tmp, linecounts[rank], x\_part);

MPI\_Allgatherv(x\_part, linecounts[rank], MPI\_DOUBLE, x, linecounts, offsets, MPI\_DOUBLE, MPI\_COMM\_WORLD);

//calc y\_{n+1}

matrix\_to\_vect\_mul(A\_part, linecounts[rank], x, tmp);

vect\_sub(tmp, b\_part, linecounts[rank], y\_part);

temp = scalar\_mul(y\_part, y\_part, linecounts[rank]);

double res;

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

++iter;

if (res < epsilonb\_norm\_squared) break;

}

free(y\_n);

free(sendcounts);

free(displs);

free(linecounts);

free(offsets);

free(A\_part);

free(b\_part);

free(x\_part);

free(y\_part);

free(tmp);

}

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

int rank = 0;

int size = 0;

MPI\_Init(&argc, &argv);

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

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

double \*A;

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

double \*b;

double min\_time = ULLONG\_MAX;

if (rank == 0) {

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

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

}

for (int i = 0; i < MEASURES; ++i) {

if (rank == 0) {

fill\_matrix(A);

fill\_vect(x);

fill\_vect(b);

}

double time\_start = MPI\_Wtime();

solve(A, x, b, rank, size);

double time\_end = MPI\_Wtime();

if (time\_end - time\_start < min\_time) min\_time = time\_end - time\_start;

}

if (rank == 0) {

printf("Time taken: %f sec.\n", min\_time);

free(A);

free(b);

}

free(x);

MPI\_Finalize();

return EXIT\_SUCCESS;

}