###### 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, 21203 группы

**Неретина Степана Ивановича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Власенко

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЗАДАНИЕ 3](#_heading=h.gjdgxs)

[ОПИСАНИЕ РАБОТЫ 4](#_heading=h.30j0zll)

Вывод9

[Приложение 1](#_heading=h.1fob9te) 10

Приложение 2 15

# 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:
3. Замерить время работы 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.
4. Выполнить профилирование двух вариантов программы с помощью MPE при использовании 24-и ядер.

# ОПИСАНИЕ РАБОТЫ

# Написан код (Приложение 1, 2), который реализует поставленную задачу(решение слау методом минимальных невязок).

Компиляция производилась командой mpicc для параллельной, а для последовательной при помощи gcc

Тестирование производилось на кластере. Подключение осуществлялось

посредством протокола ssh с логином и паролем.

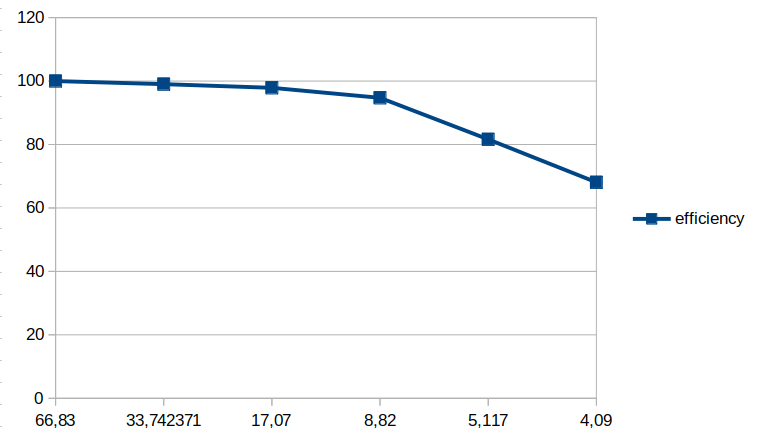
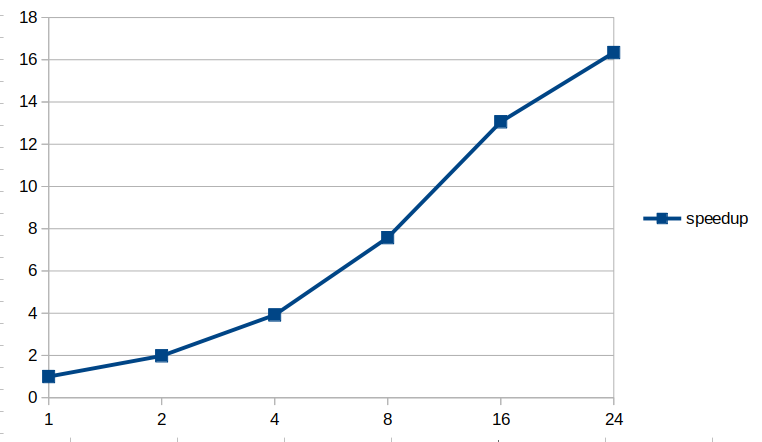
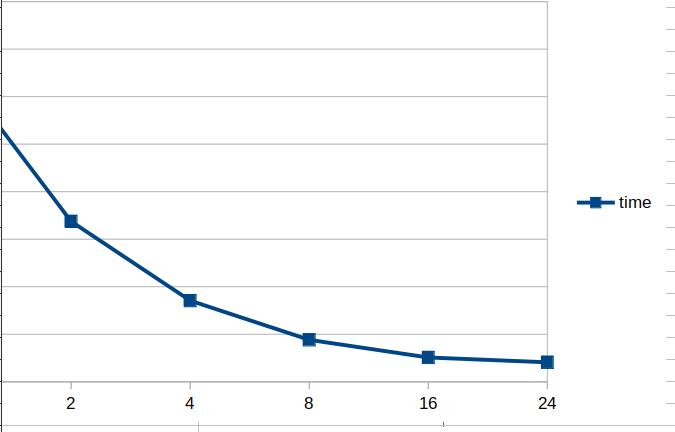
Для выполнения условия «... решение задачи на одном ядре занимало не

менее 30 секунд.» подобраны значения N = 1000, ε = 10^(-5), ITERATIONS\_COUNT 10000

В результате замеров получены следующие результаты:

|  |  |  |  |
| --- | --- | --- | --- |
| N | Время | Ускорение | Эффективность |
| 1 | 66,83 | 1 | 100 |
| 2 | 33,742371 | 1,98059585083692 | 99,029792541846 |
| 4 | 17,07 | 3,91505565319274 | 97,8763913298184 |
| 8 | 8,82 | 7,57709750566893 | 94,7137188208617 |
| 16 | 5,117 | 13,0603869454759 | 81,6274184092242 |
| 24 | 4,09 | 16,3398533007335 | 68,0827220863896 |

**Графики**



# Профилирование:

# 

# 

# 

# 

# **Вывод**

Из результатов замеров видно, что эффективность и скорость работы программы улучшается при увеличении количества процессов, но разница при переходе на 24 процесса становится уже не такой большой.

# **Приложение 1. *Последовательная программа***

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#include <math.h>

#include <stdbool.h>

#define N 1000

#define START (-100)

#define STOP (100)

#define ITERATIONS\_COUNT 10000

double \* generate\_double\_arr(size\_t n){

return (double \*) malloc(n \* sizeof(double));

}

double generate\_random\_in\_range(double min, double max) {

double range = (max - min);

double div = RAND\_MAX / range;

return min + (rand() / div);

}

double \*generate\_vector\_with\_filled\_value(double value){

double \* res = generate\_double\_arr(N);

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

res[i] = value;

}

return res;

}

double \*fill\_vector(){

double \* res = generate\_double\_arr(N);

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

res[i] = i\*i;

}

return res;

}

double \*generate\_vector\_with\_random\_value(){

double \* res = generate\_double\_arr(N);

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

res[i] = generate\_random\_in\_range(START, STOP);

}

return res;

}

double\* generate\_matrix\_a() {

double \*res = generate\_double\_arr(N\*N);

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

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

//res[i]= generate\_random\_in\_range(START, STOP);

res[j\*N +i] = (i == j ? i : N);

// if(i == j){

// //res[i\*N + j] += (N / 8.0);

// res[i\*N + j] = (N\*8);

// }

}

}

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

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

// if(i == j){

// res[i\*N + j] =2;

// }else{

// res[i\*N + j] =1;

// }

// }

// }

return res;

}

void print\_matrix(double \*arr, size\_t size){

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

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

printf("%lf ", arr[i\*size + j]);

}

printf("\n");

}

printf("\n\n");

}

void print\_vec(double \*vec, size\_t size){

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

printf("%lf\n", vec[i]);

}

printf("\n\n");

}

double scalar\_dot(const double \*vec1, const double \*vec2, size\_t size){

double res = 0;

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

res += vec1[i]\*vec2[i];

}

return res;

}

void multiple\_vec\_with\_num(const double \*vec, double num, double \*res, size\_t size){

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

res[i] = vec[i] \* num;

}

}

double calculate\_squared\_norm(double \*vec, size\_t size){

double res = 0;

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

res += (double)powl(vec[i], 2);

}

return res;

}

bool check\_convergence(double \*left, double right\_part){

return calculate\_squared\_norm(left, N) < right\_part;

}

double calculate\_right\_part(double \*b){

double epsilon = powl(10, -5);

double a = calculate\_squared\_norm(b, N);

return a \* epsilon \* epsilon;

}

void subtract\_vectors(const double \*vec1, const double \*vec2, double \*res, int size){

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

res[i] = vec1[i] - vec2[i];

}

}

void multiple\_matrix\_and\_vector(const double \*matrix, const double \*vec, double \*res){

memset(res, 0, sizeof(double) \* N);

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

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

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

}

}

}

void calculate\_vector\_y(const double \*matrix\_a, double\* x, double \*b, double \*res) {

multiple\_matrix\_and\_vector(matrix\_a, x, res);

subtract\_vectors(res, b, res, N);

}

double \*calculate\_ay\_n(double \*a, double \*y) {

double \*res = generate\_double\_arr(N);

multiple\_matrix\_and\_vector(a,y, res);

return res;

}

double tau\_update(double \*a, double \*y) {

double \*ay\_n = calculate\_ay\_n(a, y);

double numerator = scalar\_dot(y, ay\_n, N);

double denumerator = scalar\_dot(ay\_n, ay\_n, N);

return numerator / denumerator;

}

void save\_matrix\_to\_file(double\* matrix, char\* filename) {

// Open the file for writing

FILE\* file = fopen(filename, "w");

// Write the matrix dimensions to the file

fprintf(file, "%d ", N);

// Write the matrix elements to the file

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

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

fprintf(file, "%f ", matrix[i\*N + j]);

}

//fprintf(file, "\n");

}

// Close the file

fclose(file);

}

void save\_vector\_to\_file(const char\* filename, double \* vector) {

FILE\* fp = fopen(filename, "w");

if (fp == NULL) {

printf("Error opening file %s\n", filename);

return;

}

fprintf(fp, "%d ", N);

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

fprintf(fp, "%f\n", vector[i]);

}

fclose(fp);

}

double\* read\_matrix\_from\_file(const char\* filename) {

FILE\* fp = fopen(filename, "r");

if (fp == NULL) {

printf("Error opening file %s\n", filename);

return NULL;

}

size\_t n;

fscanf(fp, "%zd", &n);

char c = fgetc(fp);

// Allocate memory for the matrix

double\* arr = generate\_double\_arr(n\*n);

// Read the matrix elements from the file

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

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

double res;

fscanf(fp, "%lf", &res);

arr[i\* n + j] = res;

fgetc(fp);

}

}

fclose(fp);

return arr;

}

double\* read\_vector\_from\_file(const char\* filename) {

FILE\* fp = fopen(filename, "r");

if (fp == NULL) {

printf("Error opening file %s\n", filename);

return NULL;

}

size\_t n;

fscanf(fp, "%zd", &n);

fgetc(fp);

// Allocate memory for the matrix

double\* arr = generate\_double\_arr(n);

// Read the matrix elements from the file

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

double res;

fscanf(fp, "%lf", &res);

arr[i] = res;

fgetc(fp);

}

fclose(fp);

return arr;

}

int main() {

srand (time ( NULL));

double\* a = generate\_matrix\_a();

double\* b = fill\_vector();

double right\_part = calculate\_right\_part(b);

//printf("Right part %f\n", right\_part);

double\* x = generate\_vector\_with\_filled\_value(0);

double \*y = generate\_double\_arr(N);

double \*tau\_y = generate\_double\_arr(N);

// print\_matrix(a, N);

// print\_vec(b, N);

calculate\_vector\_y(a, x, b, y);

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

//printf("%zu\n", i+1);

double tau = tau\_update(a, y);

//printf("Tau %f\n", tau);

multiple\_vec\_with\_num(y, tau, tau\_y, N);

subtract\_vectors(x, tau\_y, x, N);

//print\_vec(x, N);

calculate\_vector\_y(a, x, b, y);

if(check\_convergence(y, right\_part)){

printf("Finished on: %zu\n", i+1);

// if(i+1 > 1500){

// save\_matrix\_to\_file(a, "matrix.txt");

// save\_vector\_to\_file("vector.txt", b);

// }

break;

}

}

//print\_vec(x, N);

free(a);

free(b);

free(x);

free(y);

free(tau\_y);

return 0;

}

# **Приложение 2. *Параллельная программа.***

#include <stdio.h>

#include <math.h>

#include <mpi.h>

#include <stdlib.h>

#include <string.h>

#define N 1000

#define START (-100)

#define STOP (100)

#define ITERATIONS\_COUNT 10000

void calculate\_right\_part(int rank, const int \*b\_send\_counts, const int \*b\_displs, const double \*b, double \*right\_part) {

double \*start\_b = (double \*)malloc(b\_send\_counts[rank] \* sizeof(double));

MPI\_Scatterv(b, b\_send\_counts, b\_displs, MPI\_DOUBLE, start\_b, b\_send\_counts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

size\_t local\_sum = 0;

size\_t global\_sum = 0;

for (int i = 0; i < b\_send\_counts[rank]; i++) {

local\_sum += start\_b[i] \* start\_b[i];

}

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

double epsilon = powl(10, -5);

if(rank == 0){

(\*right\_part) = global\_sum \* epsilon \* epsilon;

}

MPI\_Bcast(right\_part, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

free(start\_b);

}

double \* generate\_double\_arr(size\_t n){

return (double \*) malloc(n \* sizeof(double));

}

int \* generate\_int\_arr(size\_t n){

return (int \*) malloc(n \* sizeof(int));

}

double\* read\_matrix\_from\_file(const char\* filename) {

FILE\* fp = fopen(filename, "r");

if (fp == NULL) {

printf("Error opening file %s\n", filename);

return NULL;

}

size\_t n;

fscanf(fp, "%zd", &n);

char c = fgetc(fp);

// Allocate memory for the matrix

double\* arr = generate\_double\_arr(n\*n);

// Read the matrix elements from the file

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

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

double res;

fscanf(fp, "%lf", &res);

arr[i\* n + j] = res;

fgetc(fp);

}

}

fclose(fp);

return arr;

}

double\* read\_vector\_from\_file(const char\* filename) {

FILE\* fp = fopen(filename, "r");

if (fp == NULL) {

printf("Error opening file %s\n", filename);

return NULL;

}

size\_t n;

fscanf(fp, "%zd", &n);

fgetc(fp);

// Allocate memory for the matrix

double\* arr = generate\_double\_arr(n);

// Read the matrix elements from the file

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

double res;

fscanf(fp, "%lf", &res);

arr[i] = res;

fgetc(fp);

}

fclose(fp);

return arr;

}

void fill\_vector\_with\_value(double \*res, double value) {

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

res[i] = value;

}

}

double scalar\_dot(double \*row, double \*column, size\_t size){

double res = 0;

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

res += row[i] \* column[i];

}

return res;

}

void print\_matrix(double \*arr, size\_t size){

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

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

printf("%lf ", arr[i\*size + j]);

}

printf("\n");

}

printf("\n\n");

}

void print\_vec(double \*vec, size\_t size){

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

printf("%lf\n", vec[i]);

}

printf("\n\n");

}

void subtract\_vectors(const double \*vec1, const double \*vec2, double \*res, int size){

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

res[i] = vec1[i] - vec2[i];

}

}

void multiple\_matrix\_and\_vector(const double \*matrix, const double \*vec, double \*res){

memset(res, 0, sizeof(double) \* N);

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

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

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

}

}

}

double\* generate\_matrix\_a() {

double \*res = generate\_double\_arr(N\*N);

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

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

//res[i]= generate\_random\_in\_range(START, STOP);

res[j\*N +i] = (i == j ? i : N);

// if(i == j){

// //res[i\*N + j] += (N / 8.0);

// res[i\*N + j] = (N\*8);

// }

}

}

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

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

// if(i == j){

// res[i\*N + j] =2;

// }else{

// res[i\*N + j] =1;

// }

// }

// }

return res;

}

double \*fill\_vector(){

double \* res = generate\_double\_arr(N);

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

res[i] = i\*i;

}

return res;

}

void calculateY(int rank, const int \*matrix\_send\_counts, const int \*matrix\_displs, const int \*b\_send\_counts,

const int \*b\_displs, double \*x, double \*y,

double \*matrix\_rec\_row, double \*vec\_rec1, double \*vec\_rec2) {

int num\_handle\_rows = matrix\_send\_counts[rank] / N;

for(size\_t i = 0; i < num\_handle\_rows; i++){

vec\_rec2[i] =

scalar\_dot(&matrix\_rec\_row[i\*N], x, N) - vec\_rec1[i];

}

MPI\_Allgatherv(vec\_rec2, num\_handle\_rows, MPI\_DOUBLE, y, b\_send\_counts, b\_displs, MPI\_DOUBLE, MPI\_COMM\_WORLD);

}

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

double start, end;

int rank, size, n, rest;

MPI\_Init( & argc, & argv);

start = MPI\_Wtime();

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

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

n = N / size;

rest = N % size;

int \* matrix\_send\_counts = generate\_int\_arr(size);

int \* matrix\_displs = generate\_int\_arr(size);

int \* b\_send\_counts = generate\_int\_arr(size);

int \* b\_displs = generate\_int\_arr(size);

int offset = 0;

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

matrix\_send\_counts[i] = n \* N;

if (i < rest) {

matrix\_send\_counts[i] += N;

}

matrix\_displs[i] = offset;

offset += matrix\_send\_counts[i];

}

offset = 0;

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

b\_send\_counts[i] = n;

if (i < rest) {

b\_send\_counts[i] += 1;

}

b\_displs[i] = offset;

offset += b\_send\_counts[i];

}

//

// if (rank == 0) {

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

// printf("matrix\_send\_counts[%d] = %d\tdispls[%d] = %d\n", i, matrix\_send\_counts[i], i, matrix\_displs[i]);

// }

// printf("\n");

// }

//

// if (rank == 0) {

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

// printf("b\_send\_counts[%d] = %d\tb\_displs[%d] = %d\n", i, b\_send\_counts[i], i, b\_displs[i]);

// }

// printf("\n");

// }

double \*a, \*b, \*x, \*y;

//double \*tau\_y;

x = generate\_double\_arr(N);

fill\_vector\_with\_value(x, 0);

y = generate\_double\_arr(N);

if(rank == 0){

// a = read\_matrix\_from\_file("matrix.txt");

// b = read\_vector\_from\_file("vector.txt");

a = generate\_matrix\_a();

b = fill\_vector();

//tau\_y = generate\_double\_arr(N);

}

double right\_part;

calculate\_right\_part(rank, b\_send\_counts, b\_displs, b, &right\_part);

if(rank == 0){

printf("%f", right\_part);

}

double \*matrix\_rec\_row = generate\_double\_arr(matrix\_send\_counts[rank]);

double \*vec\_rec1 = generate\_double\_arr(b\_send\_counts[rank]);

double \*vec\_rec2 = generate\_double\_arr(b\_send\_counts[rank]);

double \*vec\_rec3= generate\_double\_arr(b\_send\_counts[rank]);

MPI\_Scatterv(a, matrix\_send\_counts, matrix\_displs, MPI\_DOUBLE, matrix\_rec\_row, matrix\_send\_counts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatterv(b, b\_send\_counts, b\_displs, MPI\_DOUBLE, vec\_rec1, b\_send\_counts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

calculateY(rank, matrix\_send\_counts, matrix\_displs, b\_send\_counts, b\_displs, x, y,

matrix\_rec\_row, vec\_rec1, vec\_rec2);

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

//double tau = tau\_update(a, y);

//double \*ay\_n = calculate\_ay\_n(a, y);

// vec\_rec2 = y

// vec\_rec1 ay part

double local\_ay\_repeated\_scalar\_sum = 0;

double global\_ay\_repeated\_scalar\_sum = 0;

double local\_ay\_scalar\_sum = 0;

double global\_ay\_scalar\_sum = 0;

int num\_handle\_rows = matrix\_send\_counts[rank] / N;

for(size\_t j = 0; j < b\_send\_counts[rank]; j++) {

//printf("%d %f\n", rank, matrix\_rec\_row[num\_handle\_rows\*(i + 2)]);

vec\_rec3[j] =

scalar\_dot(&matrix\_rec\_row[j \* N], y, N); // ay

}

local\_ay\_scalar\_sum += scalar\_dot(vec\_rec3, vec\_rec3, b\_send\_counts[rank]);

local\_ay\_repeated\_scalar\_sum += scalar\_dot(vec\_rec3, vec\_rec2, b\_send\_counts[rank]); // числитель

MPI\_Reduce(&local\_ay\_scalar\_sum, &global\_ay\_scalar\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&local\_ay\_repeated\_scalar\_sum, &global\_ay\_repeated\_scalar\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

double tau;

if(rank == 0){

tau = global\_ay\_repeated\_scalar\_sum / global\_ay\_scalar\_sum;

if(i % 1000 == 0)

printf("Tau: %f\n", tau);

}

MPI\_Bcast(&tau, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

for(size\_t j = 0; j < num\_handle\_rows; j++) {

vec\_rec3[j] = x[b\_displs[rank]+j] - vec\_rec2[j] \* tau;

}

MPI\_Gatherv(vec\_rec3, num\_handle\_rows, MPI\_DOUBLE, x, b\_send\_counts, b\_displs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

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

calculateY(rank, matrix\_send\_counts, matrix\_displs, b\_send\_counts, b\_displs, x, y,

matrix\_rec\_row, vec\_rec1, vec\_rec2);

double local\_left\_scalar\_y = 0;

double global\_left\_scalar\_y = 0;

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

local\_left\_scalar\_y += y[j]\*y[j];

}

// MPI\_Reduce(&local\_left\_scalar\_y, &global\_left\_scalar\_y, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

// MPI\_Bcast(&global\_left\_scalar\_y, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// MPI\_Bcast(&right\_part, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

if(local\_left\_scalar\_y < right\_part){

if(rank == 0){

printf("Finished on: %zu\n", i+1);

}

break;

}

}

// if(rank == 0){

// print\_vec(x, 2);

// }

free(matrix\_send\_counts);

free(b\_send\_counts);

free(matrix\_displs);

free(b\_displs);

free(matrix\_rec\_row);

free(vec\_rec1);

free(vec\_rec2);

free(vec\_rec3);

free(y);

if(rank == 0){

free(a);

free(b);

//free(tau\_y);

}

if(rank == 0){

end = MPI\_Wtime();

printf("Runtime = %f\n", end-start);

}

MPI\_Finalize();

}