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, группы 22206

***Тропина Никиты Васильевича***

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

ассистент,

А.А.Ажбаков

Новосибирск 2023

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 2](#_Toc165015689)

[ЦЕЛИ 3](#_Toc165015690)

[ЗАДАНИЕ 3](#_Toc165015691)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc165015692)

[ЗАКЛЮЧЕНИЕ 7](#_Toc165015693)

[ПРИЛОЖЕНИЕ 1 9](#_Toc165015694)

[ПРИЛОЖЕНИЕ 2 10](#_Toc165015696)

[ПРИЛОЖЕНИЕ 3 15](#_Toc165015697)

# ЦЕЛИ

Определить эффективность распараллеливания программы с помощью MPI от числа используемых ядер.

# ЗАДАНИЕ

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.

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.

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

Программа, реализующая итерационный алгоритм решения систем линейных алгебраических уравнений вида Ax=b, была распараллелена двумя вариантами: векторы x и b дублируются в каждом MPI-процессе (Приложение 2), векторы x и b разрезаются между MPI-процессами аналогично матрице A (Приложение 3).

Выполнение программы производилось на вычислительном кластере НОЦ «Газпромнефть-НГУ» с помощью скрипта системы пакетной обработки SLURM (Приложение 1).   
 Для измерения времени использовалась функция MPI\_Wtime().

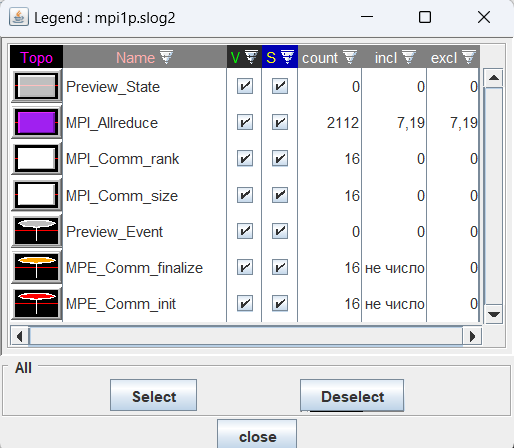
В качестве исходных данных для тестирования взята модельная задача с заданным решением: Элементы главной диагонали матрицы A размера N×N, значение N = 16 384, равны 2.0, остальные - 1.0. Начальные значения элементов вектора b равны N+1, вектора x – 0. В этом случае решением системы будет вектор, элементы которого равны 1.0.

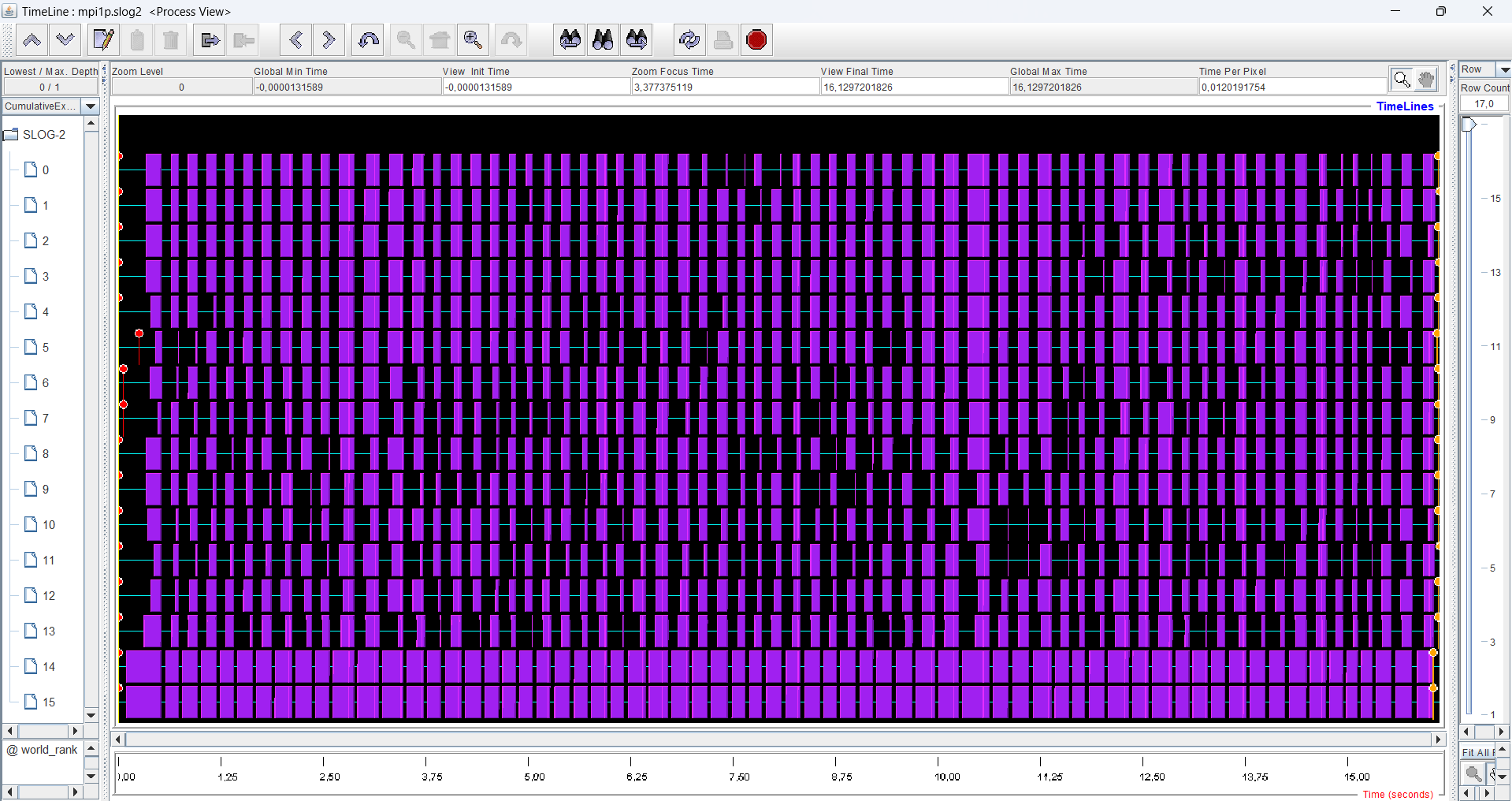
Результаты измерений представлены на диаграммах:

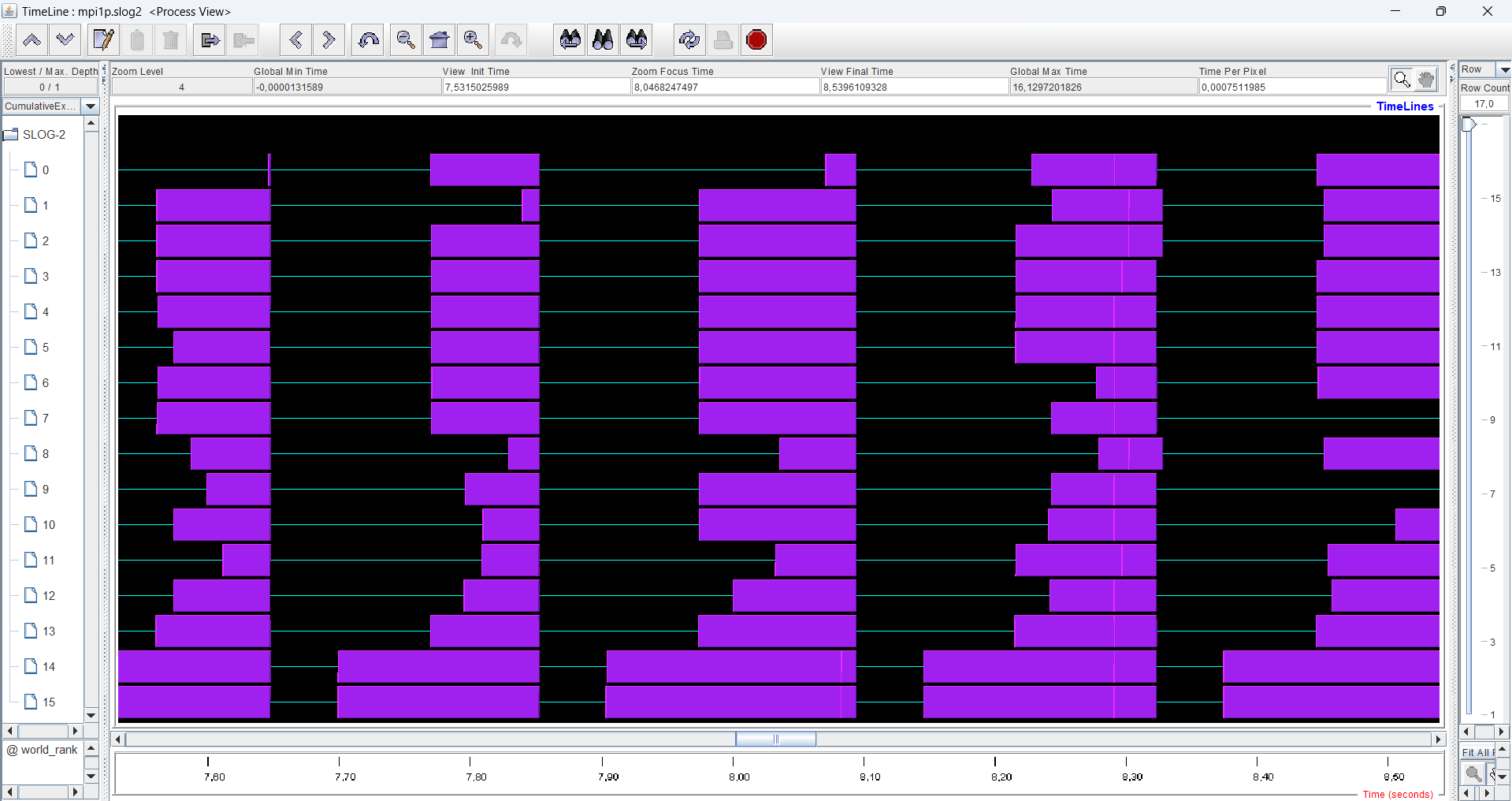
После измерения времени работы программ было произведено их профилирование с помощью MPE при использовании 16-и ядер.

## **Профилирование:**

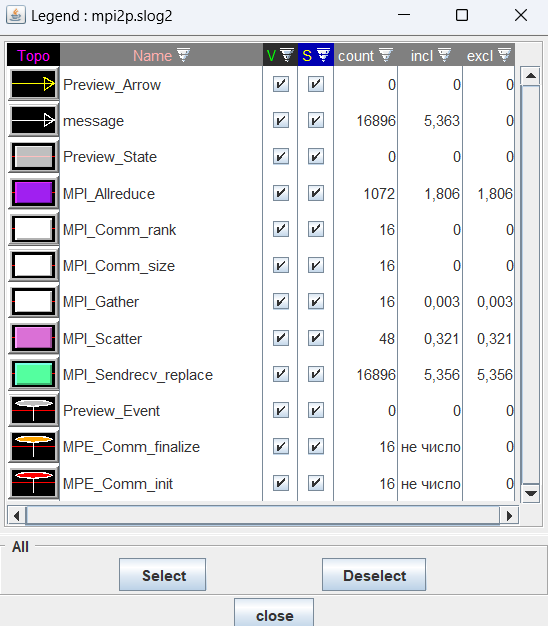
Вариант 1:

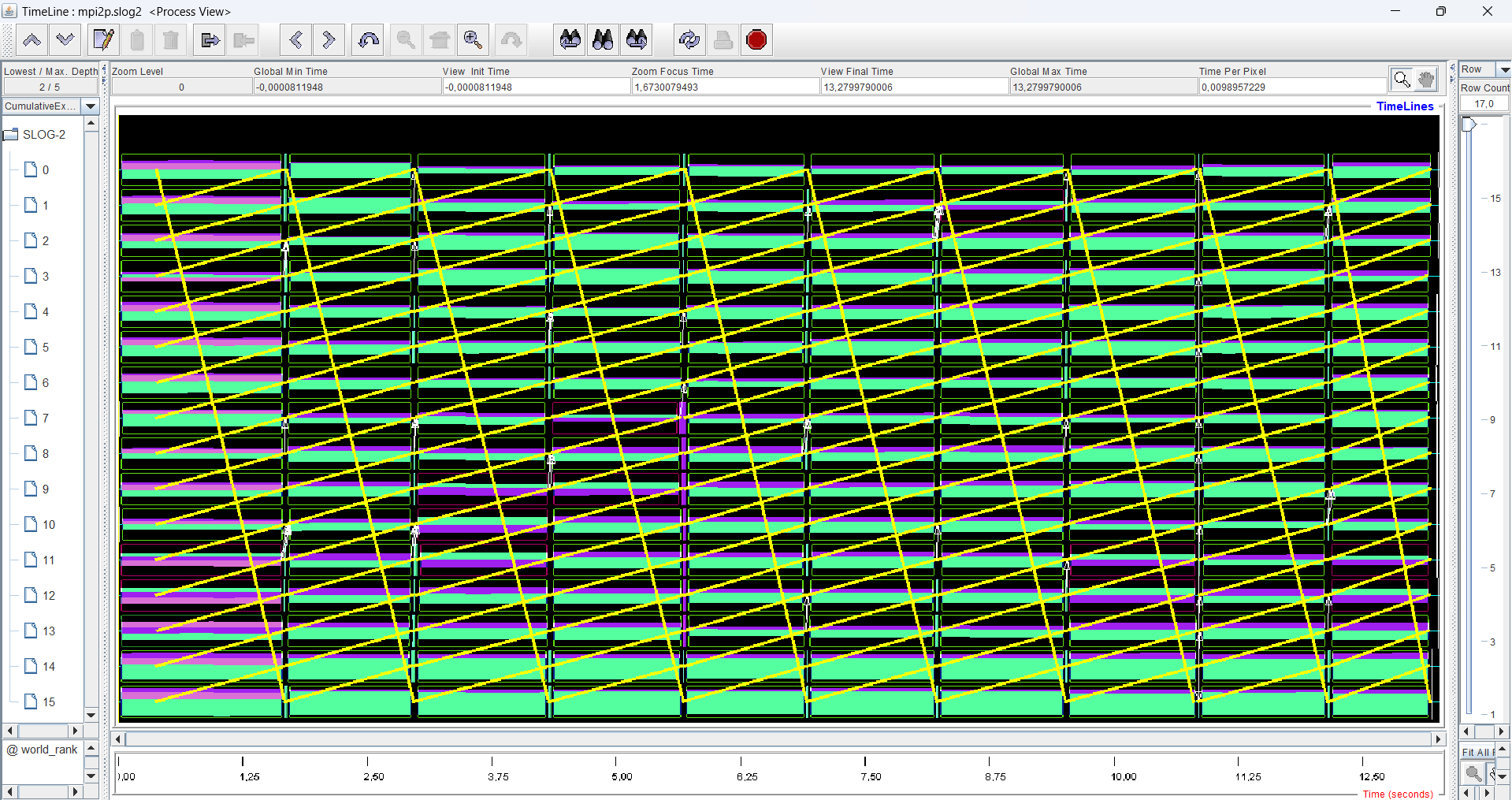


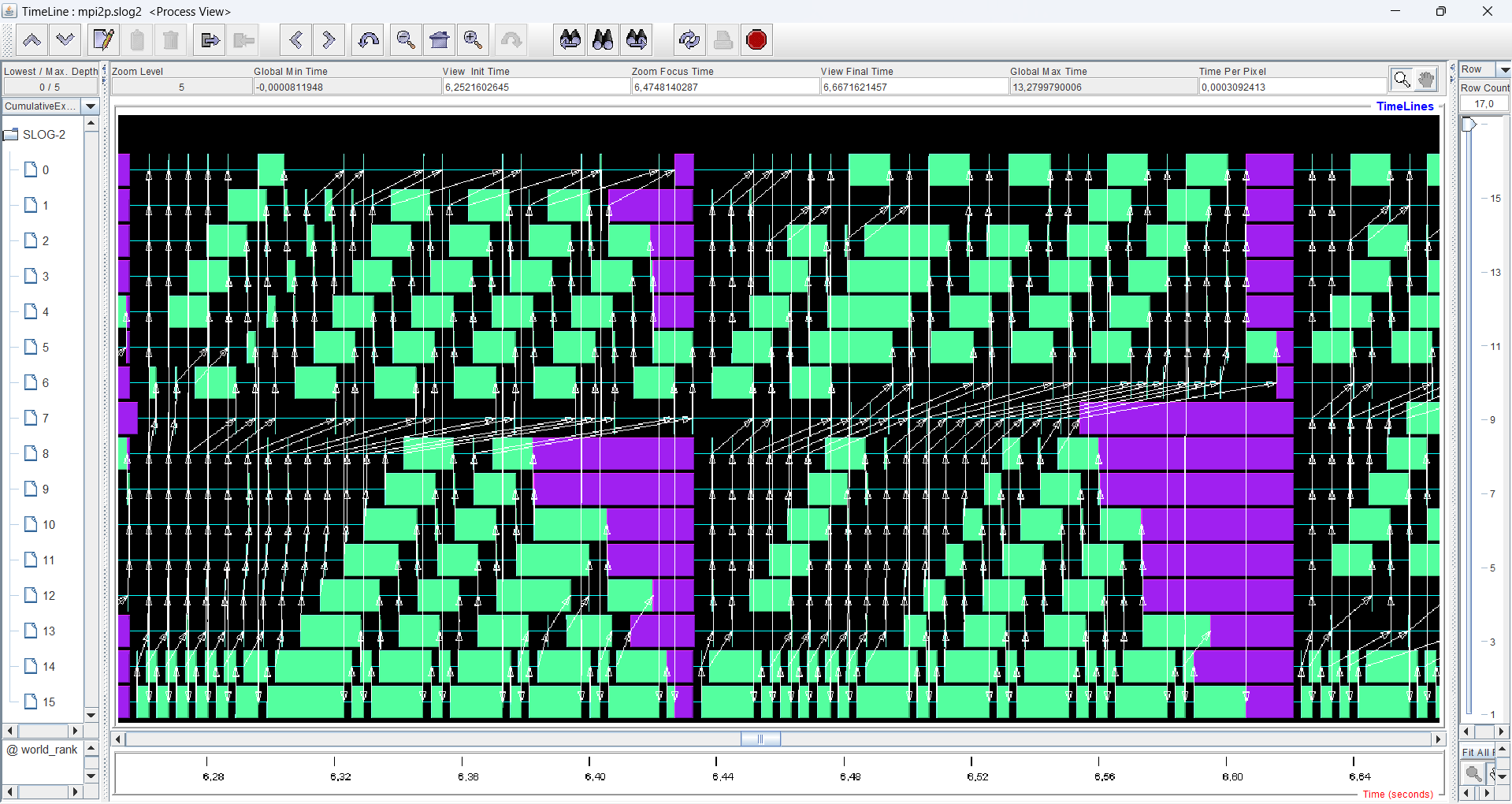




Вариант 2:



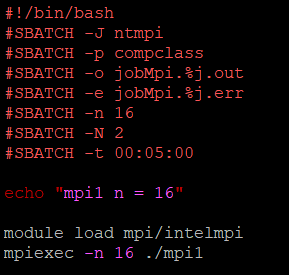




# ЗАКЛЮЧЕНИЕ

В ходе работы были построены графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых потоков. Также выполнено профилирование обоих вариантов. Видно, что во втором варианте программы время, затраченное на обмен между процессами, сопоставимо с временем, затраченным на вычисления. Однако в первом варианте доля обмена занимает большую часть. Эффективность и ускорение распараллеливания второго варианта программы оказалось выше. Таким образом, эффективность, ускорение распараллеливания и масштабируемость второго варианта программы оказалось выше.

# ПРИЛОЖЕНИЕ 1



# ПРИЛОЖЕНИЕ 2

#include "mpi.h"

#include <cstring>

#include <iostream>

#include <math.h>

#include <omp.h>

enum {

SIZE\_VECTOR = 1024,//4096,

ZERO\_VALUE = 0,

ROOT = 0

};

const double TAU = 1e-5;

const double EPSILON = 1e-5;

void printVector(double\* vector, int size, int rank)

{

printf("rank:%d\n", rank);

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

printf("%f ", vector[i]);

}

printf("\n");

}

void printPartMatrix(double\* matrix, int partSize, int shift, int matrixSideSize,

int nProcesses, int rank)

{

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

MPI\_Barrier(MPI\_COMM\_WORLD);

if (i == rank) {

//printf("rank = %d\n", rank);

//printVector(partMatrix, partMatrixSize);

printf("\n");

//free space

for (int i = 0; i < shift % matrixSideSize; i++) {

printf(" ");

}

//part of matrix

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

if (i != 0 && (i + shift) % matrixSideSize == 0) {

printf("\n");

}

printf("%1.0f ", matrix[i]);

}

printf("\n");

}

MPI\_Barrier(MPI\_COMM\_WORLD);

}

}

double calculateDeterminant(double\* matrix, int size)

{

double determinant = 0;

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

determinant += matrix[i] \* matrix[i];

}

return determinant;

}

double calculateNorm(double\* matrix, int size)

{

double determinant = calculateDeterminant(matrix, size);

return sqrt(determinant);

}

void fillVectorWithValue(double\* vector, int size, int value)

{

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

vector[i] = value;

}

}

void vectorPartSubtraction(double\* minuend, double\* subtrahend, double\* result, int vectorSize, int rank, int nProcesses, int partSize)

{

int shift = rank \* partSize;

if (rank + 1 == nProcesses && partSize != 0) {

partSize = vectorSize - shift;

}

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

result[shift + i] = minuend[shift + i] - subtrahend[shift + i];

}

}

void multiplyVectorPartByScalar(double\* vector, double\* result, int vectorSize, double scalar, int rank, int nProcesses, int partSize)

{

int shift = rank \* partSize;

if (rank + 1 == nProcesses && partSize != 0) {

partSize = vectorSize - shift;

}

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

result[shift + i] = vector[shift + i] \* scalar;

}

}

//vector != result

void multiplyPartMatrixByVector(double\* partMatrix, int partMatrixSize, double\* multiplyVector, double\* result, int vectorSize,

int shift)

{

fillVectorWithValue(result, vectorSize, ZERO\_VALUE);

int multiplyVectorLineIndex = shift % vectorSize;

int resultLineIndex = shift / vectorSize;

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

result[resultLineIndex] += partMatrix[i] \* multiplyVector[multiplyVectorLineIndex];

++multiplyVectorLineIndex;

if (multiplyVectorLineIndex % vectorSize == 0) {

multiplyVectorLineIndex = 0;

resultLineIndex++;

}

}

}

double\* generateZeroVector(int vectorSize)

{

double\* vector = new double[vectorSize];

fillVectorWithValue(vector, vectorSize, ZERO\_VALUE);

return vector;

}

double\* generateSolutionVector(int vectorSize)

{

double\* solutionVector = generateZeroVector(vectorSize);

fillVectorWithValue(solutionVector, vectorSize, ZERO\_VALUE);

return solutionVector;

}

double\* generateRightSideVector(int vectorSize)

{

double\* rightSideVector = generateZeroVector(vectorSize);

fillVectorWithValue(rightSideVector, vectorSize, vectorSize + 1);

return rightSideVector;

}

double\* generateSquareMatrix(int sideSize)

{

double\* matrix = new double[sideSize \* sideSize];

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

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

if (i == j)

matrix[i \* sideSize + j] = 2;

else

matrix[i \* sideSize + j] = 1;

}

}

return matrix;

}

int calculatePartMatrixSize(int matrixSize, int rank, int nProcesses)

{

int partMatrixSize = matrixSize / nProcesses;

int valuesLeft = matrixSize % nProcesses;

if (valuesLeft != 0 && rank < valuesLeft) {

partMatrixSize++;

}

return partMatrixSize;

}

int calculateMatrixShift(int matrixSize, int partMatrixSize, int rank, int nProcesses)

{

int valuesLeft = matrixSize % nProcesses;

int shift = 0;

if (rank < valuesLeft) {

shift = rank \* partMatrixSize;

}

else {

shift = rank \* partMatrixSize + valuesLeft;

}

return shift;

}

int calculateVectorProcessingPart(int vectorSize, int rank, int nProcesses)

{

int processPart;

if (vectorSize < nProcesses) {

rank < vectorSize ? processPart = 1 : processPart = 0;

}

else {

processPart = vectorSize / nProcesses;

}

return processPart;

}

double\* generatePartMatrix(double\* matrix, int matrixSize, int shift, int partMatrixSize)

{

double\* partMatrix = generateZeroVector(partMatrixSize);

memcpy(partMatrix, &matrix[shift], partMatrixSize \* sizeof(double));

return partMatrix;

}

bool isAccuracyAchieved(double numerator, double denominator)

{

return ((numerator / denominator) < EPSILON);

}

void copyVector(double\* source, double\* destination, int size)

{

memcpy(destination, source, size \* sizeof(double));

}

void prepareForReduce(double\* vector, int vectorSize, int matrixShift, int partMatrixSize, int rank)

{

int firstPart = matrixShift / vectorSize;

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

vector[i] = 0;

}

int lastPartBegin = (matrixShift + partMatrixSize) / vectorSize;

if ((matrixShift + partMatrixSize) % vectorSize != 0) {

lastPartBegin++;

}

for (int i = lastPartBegin; i < vectorSize; i++) {

vector[i] = 0;

}

}

//x^(n+1) = x^n – τ(Ax^n – b)

double\* iterationMethod(double\* matrix, double\* rightSideVector, int matrixSideSize, int rank, int nProcesses)

{

int partMatrixSize = calculatePartMatrixSize(matrixSideSize \* matrixSideSize, rank, nProcesses);

int matrixShift = calculateMatrixShift(matrixSideSize \* matrixSideSize, partMatrixSize, rank, nProcesses);

double\* partMatrix = generatePartMatrix(matrix, matrixSideSize \* matrixSideSize, matrixShift, partMatrixSize);

int vectorProcessingPart = calculateVectorProcessingPart(matrixSideSize, rank, nProcesses);

double\* solutionVector = generateSolutionVector(matrixSideSize);

double\* solutionReduceVector = generateZeroVector(matrixSideSize);

double\* processingVector = generateZeroVector(matrixSideSize);

double\* normVector = generateZeroVector(matrixSideSize);

double solutionNorm = 0;

double rightSideNorm = calculateNorm(rightSideVector, matrixSideSize);

bool run = true;

while (run) {

//Ax

multiplyPartMatrixByVector(partMatrix, partMatrixSize, solutionVector,

processingVector, matrixSideSize, matrixShift);

//Ax-b

vectorPartSubtraction(processingVector, rightSideVector, processingVector, matrixSideSize, rank, nProcesses, vectorProcessingPart);

prepareForReduce(processingVector, matrixSideSize, matrixShift, partMatrixSize, rank);

MPI\_Allreduce(processingVector, normVector, matrixSideSize, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

solutionNorm = calculateNorm(normVector, matrixSideSize);

//t(Ax-b)

multiplyVectorPartByScalar(processingVector, processingVector, matrixSideSize, TAU, rank, nProcesses, vectorProcessingPart);

//x-t(Ax-b)

vectorPartSubtraction(solutionVector, processingVector, solutionReduceVector, matrixSideSize, rank, nProcesses, vectorProcessingPart);

prepareForReduce(solutionReduceVector, matrixSideSize, matrixShift, partMatrixSize, rank);

MPI\_Allreduce(solutionReduceVector, solutionVector, matrixSideSize, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

run = !isAccuracyAchieved(solutionNorm, rightSideNorm);

}

delete[] partMatrix;

delete[] solutionReduceVector;

delete[] processingVector;

delete[] normVector;

return solutionVector;

}

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

{

double\* matrix = generateSquareMatrix(SIZE\_VECTOR);

double\* rightPartVector = generateRightSideVector(SIZE\_VECTOR);

int rank, nProcesses;

MPI\_Init(&argc, &argv);

double start = MPI\_Wtime();

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

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

double\* solutionVector = iterationMethod(matrix, rightPartVector, SIZE\_VECTOR, rank, nProcesses);

double finish = MPI\_Wtime();

if (rank == 0) {

printf("Time: %f\n", finish - start);

//printVector(solutionVector, SIZE\_VECTOR, 0);

}

delete[] matrix;

delete[] solutionVector;

delete[] rightPartVector;

MPI\_Finalize();

}

# ПРИЛОЖЕНИЕ 3

#include "mpi.h"

#include <cmath>

#include <cstring>

#include <iostream>

#include <math.h>

enum {

SIZE\_VECTOR = 8,//4096,

ZERO\_VALUE = 0,

ROOT = 0

};

const double TAU = 1e-5;

const double EPSILON = 1e-5;

void printVector(double\* vector, int size, int rank)

{

printf("rank:%d\n", rank);

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

printf("%f ", vector[i]);

}

printf("\n");

}

void printPartMatrix(double\* matrix, int partSize, int shift, int matrixSideSize,

int nProcesses, int rank)

{

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

MPI\_Barrier(MPI\_COMM\_WORLD);

if (i == rank) {

printf("\n");

//free space

for (int i = 0; i < shift % matrixSideSize; i++) {

printf(" ");

}

//part of matrix

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

if (i != 0 && (i + shift) % matrixSideSize == 0) {

printf("\n");

}

printf("%f ", matrix[i]);

}

printf("\n");

}

MPI\_Barrier(MPI\_COMM\_WORLD);

}

}

double calculateDeterminant(double\* matrix, int size)

{

double a = 4;

double b = sqrt(a);

double determinant = 0;

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

determinant += matrix[i] \* matrix[i];

}

return determinant;

}

void fillVectorWithValue(double\* vector, int size, int value)

{

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

vector[i] = value;

}

}

void vectorSubtraction(double\* minuend, double\* subtrahend, double\* result, int vectorSize)

{

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

result[i] = minuend[i] - subtrahend[i];

}

}

void multiplyVectorByScalar(double\* vector, double\* result, int vectorSize, double scalar)

{

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

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

}

}

//vector != result

void multiplyPartMatrixByPartVector(double\* partMatrix, int partMatrixSize, double\* multiplyVector,

double\* resultVector, int vectorSize, int rank, int nProcesses)

{

fillVectorWithValue(resultVector, vectorSize, ZERO\_VALUE);

MPI\_Status status;

int recvRank = (rank + 1) % nProcesses;

int sendRank = (rank + nProcesses - 1) % nProcesses;

int columnShift = rank \* vectorSize;

int lineSize = partMatrixSize / vectorSize;

for (int partsDone = 0; partsDone < nProcesses; partsDone++) {

int lineShift = ((rank + partsDone) % nProcesses) \* vectorSize;

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

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

resultVector[i] += partMatrix[i \* lineSize + lineShift + j] \* multiplyVector[j];

}

}

MPI\_Sendrecv\_replace(multiplyVector, vectorSize, MPI\_DOUBLE, sendRank, 0, recvRank, 0, MPI\_COMM\_WORLD, &status);

}

}

double\* generateZeroVector(int vectorSize)

{

double\* vector = new double[vectorSize];

fillVectorWithValue(vector, vectorSize, ZERO\_VALUE);

return vector;

}

double\* generateSolutionVector(int vectorSize)

{

double\* solutionVector = generateZeroVector(vectorSize);

fillVectorWithValue(solutionVector, vectorSize, ZERO\_VALUE);

return solutionVector;

}

double\* generateRightSideVector(int vectorSize)

{

double\* rightSideVector = generateZeroVector(vectorSize);

fillVectorWithValue(rightSideVector, vectorSize, vectorSize + 1);

return rightSideVector;

}

double\* generateSquareMatrix(int sideSize)

{

double\* matrix = new double[sideSize \* sideSize];

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

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

if (i == j)

matrix[i \* sideSize + j] = 2;

else

matrix[i \* sideSize + j] = 1;

}

}

return matrix;

}

int calculatePartMatrixSize(int matrixSize, int rank, int nProcesses)

{

int partMatrixSize = matrixSize / nProcesses;

if (matrixSize % nProcesses != 0) {

partMatrixSize++;

}

return partMatrixSize;

}

bool isAccuracyAchieved(double numerator, double denominator)

{

return ((numerator / denominator) < EPSILON);

}

void copyVector(double\* source, double\* destination, int size)

{

memcpy(destination, source, size \* sizeof(double));

}

int calculatePartVectorSize(int partMatrixSize, int nProcesses)

{

return partMatrixSize / nProcesses;

}

//x^(n+1) = x^n – τ(Ax^n – b)

double\* iterationMethod(double\* matrix, double\* rightSideVector, int matrixSideSize, int rank, int nProcesses)

{

int partMatrixSize = calculatePartMatrixSize(matrixSideSize \* matrixSideSize, rank, nProcesses);

double\* partMatrix = generateZeroVector(partMatrixSize);

MPI\_Scatter(matrix, partMatrixSize, MPI\_DOUBLE, partMatrix, partMatrixSize, MPI\_DOUBLE, ROOT, MPI\_COMM\_WORLD);

int partVectorSize = calculatePartVectorSize(matrixSideSize, nProcesses);

double\* rightPartVector = generateZeroVector(partVectorSize);

MPI\_Scatter(rightSideVector, partVectorSize, MPI\_DOUBLE, rightPartVector, partVectorSize, MPI\_DOUBLE, ROOT, MPI\_COMM\_WORLD);

double\* solutionVector = generateSolutionVector(matrixSideSize);

double\* solutionPartVector = generateZeroVector(partVectorSize);

MPI\_Scatter(solutionVector, partVectorSize, MPI\_DOUBLE, solutionPartVector, partVectorSize, MPI\_DOUBLE, ROOT, MPI\_COMM\_WORLD);

double\* scatteredVector = generateZeroVector(partVectorSize);

double\* multiplyResultVector = generateZeroVector(partVectorSize);

double partDeterminant = 0, rightSideNorm = 0, resultNorm = 0;

partDeterminant = calculateDeterminant(rightPartVector, partVectorSize);

MPI\_Allreduce(&partDeterminant, &rightSideNorm, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

rightSideNorm = sqrt(rightSideNorm);

bool run = true;

while (run) {

//Ax

multiplyPartMatrixByPartVector(partMatrix, partMatrixSize, solutionPartVector, multiplyResultVector,

partVectorSize, rank, nProcesses);

//Ax-b

vectorSubtraction(multiplyResultVector, rightPartVector, multiplyResultVector, partVectorSize);

partDeterminant = calculateDeterminant(multiplyResultVector, partVectorSize);

MPI\_Allreduce(&partDeterminant, &resultNorm, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

resultNorm = sqrt(resultNorm);

//t(Ax-b)

multiplyVectorByScalar(multiplyResultVector, multiplyResultVector, partVectorSize, TAU);

//x-t(Ax-b)

vectorSubtraction(solutionPartVector, multiplyResultVector, multiplyResultVector, partVectorSize);

copyVector(multiplyResultVector, solutionPartVector, partVectorSize);

run = !isAccuracyAchieved(resultNorm, rightSideNorm);

//run = false;

}

MPI\_Gather(solutionPartVector, partVectorSize, MPI\_DOUBLE, solutionVector, partVectorSize, MPI\_DOUBLE, ROOT, MPI\_COMM\_WORLD);

delete[] partMatrix;

delete[] rightPartVector;

delete[] solutionPartVector;

delete[] scatteredVector;

delete[] multiplyResultVector;

return solutionVector;

}

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

{

double\* matrix = generateSquareMatrix(SIZE\_VECTOR);

double\* rightPartVector = generateRightSideVector(SIZE\_VECTOR);

int rank, nProcesses;

MPI\_Init(&argc, &argv);

double start = MPI\_Wtime();

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

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

double\* solutionVector = iterationMethod(matrix, rightPartVector, SIZE\_VECTOR, rank, nProcesses);

double finish = MPI\_Wtime();

if (rank == 0) {

printf("Time: %f\n", finish - start);

printVector(solutionVector, SIZE\_VECTOR, 0);

}

delete[] matrix;

delete[] solutionVector;

delete[] rightPartVector;

MPI\_Finalize();

}