**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 №2**

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

студентки 2 курса, группы 21204

**Егоренко Ксении Владиславовны**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

(к.т.н., доцент)

А. Ю. Власенко

Новосибирск 2023

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ](#_25v28z2k7ip1)

[ЦЕЛЬ](#_jm875l4mpa1y)

[ЗАДАНИЕ](#_boaurhfjs03s)

[ХОД РАБОТЫ](#_4jbl5727ymfp)

[ЗАКЛЮЧЕНИЕ](#_oci01uapfmir)

[ПРИЛОЖЕНИЕ 1. Листинг последовательной программы](#_5ka2a1g0zwb)

[ПРИЛОЖЕНИЕ 2. Листинг параллельной программы](#_79n4216b0gh0)

[Листинг исправленного варианта параллельной программы с использованием меньшего количества коллективных функций](#_qfc1j4ckkt64)

[ПРИЛОЖЕНИЕ 3. Графики времени, ускорения, эффективности для параллельной программы](#_mlk7s6xkg4ih)

[ПРИЛОЖЕНИЕ 4. Профилирование](#_duvb2qwjpyb)

[ПРИЛОЖЕНИЕ 5. График времени для исправленного варианта параллельной программы](#_zeg6z6ikvjp8)

# ЦЕЛЬ

1. Ознакомиться с MPI (Message Passing Interface), сутью которого является создание нескольких процессов с целью параллельного исполнения вычислительных частей программы.
2. Выяснить, какой из 2-х вариантов программы, реализуемых в практической работе, на практике эффективнее и почему.

# 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:
   1. Вариант 1: векторы x и b дублируются в каждом MPI-процессе,
   2. 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).

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.

# ХОД РАБОТЫ

Вариант выполнения практической работы: Метод простой итерации.

1. Была написана программа на языке C++, реализующая итерационный алгоритм

решения СЛАУ вида Ax = b (см. Приложение 1);

1. Была написана параллельная программа реализующая итерационный алгоритм решения СЛАУ вида Ax = b (см. Приложение 2);
2. Были построены графики времени, ускорения и эффективности.
3. Было выполнено профилирование.

# ЗАКЛЮЧЕНИЕ

# 

При использовании параллельной программы время работы вычислительной части программы значительно меньше, чем у последовательной программы. Но нельзя бесконечно увеличивать количество процессов, т. к. в таком случае параллельная программа становится не столь эффективной, это связано с тем, что распараллеливание становится не столь эффективным в различных вычислительных частях программы и тратится больше времени на передачу информации между вычислительными узлами.

# 

# ПРИЛОЖЕНИЕ 1. Листинг последовательной программы

Для проверки корректности выполнения вычислительной части программы вектор x изначально заполняется нулями, матрица А заполнена значениями 1.0, на главной диагонали значения 2.0, вектор b заполнен значениями N+1. Результат выполнения программы вектор x целиком состоящий из 1.

**#include <iostream>**

**#include <cmath>**

**#include <time.h>**

**using namespace std;**

**const double eps = 0.000000001;**

**const double taul = 0.00001;**

**const int N = 6000;**

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

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

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

**A[i \* N + j] = i == j ? 2 : 1; // утяжеляем диагональ двойками, остальные значения в матрице 1**

**}**

**b[i] = N + 1;**

**x[i] = 0; // при первой инициализации заподняем вектор x нулями**

**}**

**}**

**void countVecMatrMult (double\* matr, double\* vec, double\* result) {**

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

**result[i] = 0;**

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

**result[i] += matr[i \* N + j] \* vec[j];**

**}**

**}**

**}**

**void countSubOfVectors(double\* result, double\* left, double\* right) {**

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

**result[i] = left[i] - right[i];**

**}**

**}**

**void countScalarMatrMult(double\* result, double\* vec) {**

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

**result[i] = vec[i] \* taul;**

**}**

**}**

**double countAbs(double\* vec) {**

**double abs = 0;**

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

**abs += pow(vec[i], 2);**

**}**

**return abs;**

**}**

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

**MPI\_Init(&argc, &argv);**

**double startTime = MPI\_Wtime();**

**// time\_t timeStart= time(NULL);**

**double\* A = new double[N \* N];**

**double\* b = new double[N];**

**double\* x = new double[N];**

**// создадим вектора для хранения промежуточных вычислений**

**double\* Ax = new double[N];**

**double\* subAx\_b = new double[N];**

**double\* multTaulAx\_b = new double[N];**

**// инициализируем переменные для проверки конца итераций**

**double absAx\_b = 0, abs\_b = 0;**

**firstInit(A, b, x);**

**countVecMatrMult(A, x, Ax);**

**countSubOfVectors(subAx\_b, Ax, b);**

**absAx\_b = countAbs(subAx\_b);**

**abs\_b = countAbs(b);**

**long double sqrEps = pow(eps, 2);**

**double g\_x = absAx\_b / abs\_b;**

**int itearationsNum = 0;**

**while (g\_x >= sqrEps && itearationsNum < 10000) {**

**countScalarMatrMult(multTaulAx\_b, subAx\_b);**

**countSubOfVectors(x, x, multTaulAx\_b);**

**countVecMatrMult(A, x, Ax);**

**countSubOfVectors(subAx\_b, Ax, b);**

**absAx\_b = countAbs(subAx\_b);**

**g\_x = absAx\_b / abs\_b;**

**itearationsNum++;**

**}**

**double endTime = MPI\_Wtime();**

**// time\_t timeEnd = time(NULL);**

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

**cout << x[i] << " ";**

**}**

**printf("Time spend: %lf", endTime - startTime);**

**// double totalTime = timeEnd - timeStart;**

**// printf("Time spend: %lf", totalTime);**

**MPI\_Finallize();**

**delete[] A;**

**delete[] x;**

**delete[] Ax;**

**delete[] b;**

**delete[] subAx\_b;**

**delete[] multTaulAx\_b;**

**return 0;**

**}**

# ПРИЛОЖЕНИЕ 2. Листинг параллельной программы

**#include <math.h>**

**#include <mpi.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**const int N = 7000;**

**const double eps = 0.00001;**

**const int maxIterationsCount = 10000;**

**const double taul = 0.0001;**

**void createMatrixPartForProcesses(int\* countOfLinesForProcess, int\* offsetArray, int\* countOfMatrElem\_s, int\* countOfSkipedElem, int N, int size) {**

**int currentOffset = 0;**

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

**countOfLinesForProcess[i] = N / size; // Гарантированно каждый получит число строк матрицы, равное целой части N / size**

**//Докидываем процессам от 1 до N % size по одной строке для равномерного распределения**

**//Если конечно же число N не кратно size**

**if (i < N % size) {**

**++countOfLinesForProcess[i];**

**}**

**//Начало каждого процесcа i содержится в offsets[i]**

**offsetArray[i] = currentOffset;**

**currentOffset += countOfLinesForProcess[i];**

**countOfMatrElem\_s[i] = countOfLinesForProcess[i] \* N;**

**countOfSkipedElem[i] = offsetArray[i] \* N;**

**}**

**}**

**double countAbsSqr(const double\* vec, const int N) {**

**double absSqr = 0.0;**

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

**absSqr += pow(vec[i], 2);**

**}**

**return absSqr;**

**}**

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

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

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

**A[i \* N + j] = 1.0;**

**}**

**A[i \* N + i] = 2.0;**

**x[i] = 0.0;**

**b[i] = N + 1;**

**}**

**}**

**void printMatrix(const double\* matrix) {**

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

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

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

**}**

**printf("\n");**

**}**

**printf("\n");**

**}**

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

**MPI\_Init(&argc, &argv);**

**int rank = 0;**

**int size = 0;**

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

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

**int\* countLinesForProc = new int[size];**

**int\* offsetArray = new int[size];**

**int\* countOfMatrElem\_s = new int[size];**

**int\* countOfSkipedElem = new int[size];**

**createMatrixPartForProcesses(countLinesForProc, offsetArray, countOfMatrElem\_s, countOfSkipedElem, N, size);**

**double\* partOfA = new double[countLinesForProc[rank] \* N];**

**double\* A = new double[N \* N];**

**double\* x = new double[N];**

**double\* b = new double[N];**

**double g\_x = 0.0;**

**double abs\_b = 0.0;**

**if (rank == 0) {**

**firstInit(A, b, x);**

**g\_x = 1;**

**abs\_b = sqrt(countAbsSqr(b, N));**

**}**

**// рассылаем вектор x, b, g(x)**

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

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

**MPI\_Bcast(&g\_x, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);**

**/\***

**здесь &g\_x - адрес начала буфера для приема сообщения**

**1 - число элементов принимаемого массива**

**MPI\_DOUBLE - тип элемента принимающего массива**

**0 - номер процесса-отправителя**

**MPI\_COMM\_WORLD - коммуникатор**

**\*/**

**// разрезаем матрицу А на части**

**MPI\_Scatterv(A, countOfMatrElem\_s, countOfSkipedElem, MPI\_DOUBLE, partOfA, countOfMatrElem\_s[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);**

**/\***

**здесь A - адрес с которого начинается пересылаемый участок информации**

**countOfMatrElem\_s - адрес массива размеров отправляемых сообщений**

**countOfSkipedElem - адрес массива смещений отправляемых сообщений**

**MPI\_DOUBLE - тип элемента данных отправляемого сообщения**

**partOfA - адрес начала буфера для приема сообщения**

**countOfMatrElem\_s[rank] - число элементов принимаемого сообщения**

**MPI\_DOUBLE - тип элемента данных принимаемого сообщения**

**0 - номер процесса-распылителя**

**MPI\_COMM\_WORLD - коммуникатор**

**\*/**

**double\* Ax\_b = new double[countLinesForProc[rank]];**

**double\* partOfX = new double[countLinesForProc[rank]];**

**double absOfSum = 0;**

**int iterationCount = 0;**

**double startTime = MPI\_Wtime();**

**while (g\_x > eps && iterationCount < maxIterationsCount) {**

**for (int i = 0; i < countLinesForProc[rank]; ++i) {**

**Ax\_b[i] = 0;**

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

**Ax\_b[i] += partOfA[i \* N + j] \* x[j];**

**}**

**Ax\_b[i] = Ax\_b[i] - b[offsetArray[rank] + i];**

**}**

**for (int i = 0; i < countLinesForProc[rank]; ++i) {**

**partOfX[i] = x[offsetArray[rank] + i] - taul \* Ax\_b[i];**

**}**

**// собираем по кусочкам вектор x**

**MPI\_Allgatherv(partOfX, countLinesForProc[rank], MPI\_DOUBLE, x, countLinesForProc, offsetArray, MPI\_DOUBLE, MPI\_COMM\_WORLD);**

**/\***

**здесь partOfX - адрес с которого начинается пересылаемый участок информации**

**countLinesForProc[rank] - число элементов отправляемого сообщения**

**MPI\_DOUBLE - тип элементов отправляемого сообщения**

**x - адрес начала буфера для приема сообщения**

**countLinesForProc - адрес массива размеров принимаемых сообщений**

**offsetArray - адрес массива смещений принимаемых сообщений**

**MPI\_DOUBLE - тип элемента данных принимаемых сообщений**

**MPI\_COMM\_WORLD - коммуникатор**

**\*/**

**// считаем норму по частям**

**double partAbs = countAbsSqr(Ax\_b, countLinesForProc[rank]);**

**MPI\_Reduce(&partAbs, &absOfSum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);**

**/\***

**здесь &partAbs - адрес с которого начинается пересылаемый участок информации**

**&absOfSum - адрес начала буфера для приема сообщения**

**1 - кол-во элементов отправляемого сообщения**

**MPI\_DOUBLE - тип элемента данных пересылаемого сообщения**

**MPI\_SUM - операция суммы**

**0 - номер принимающего процесса**

**MPI\_COMM\_WORLD - коммуникатор**

**\*/**

**// if (rank == 0) {**

**g\_x = sqrt(absOfSum) / abs\_b;**

**iterationCount++;**

**// }**

**// после каждой итерации Bcast рассылает обновленные значения переменных**

**MPI\_Bcast(&iterationCount, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);**

**// MPI\_Bcast(&g\_x, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);**

**}**

**if (rank == 0) {**

**if (iterationCount < maxIterationsCount) {**

**double endTime = MPI\_Wtime();**

**printf("Time taken %f\n", endTime - startTime);**

**} else {**

**//Нужно поменять знак, если метод не сошелся с заданным таул**

**printf("No converge, change taul sign\n");**

**}**

**}**

**delete[] countLinesForProc;**

**delete[] offsetArray;**

**delete[] x;**

**delete[] b;**

**delete[] A;**

**delete[] partOfA;**

**delete[] Ax\_b;**

**delete[] partOfX;**

**MPI\_Finalize();**

**return 0;**

**}**

# 

# Листинг исправленного варианта параллельной программы с использованием меньшего количества коллективных функций

**#include <math.h>**

**#include <mpi.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**const int N = 7000;**

**const double eps = 0.00001;**

**const int maxIterationsCount = 5000;**

**const double taul = 0.00001;**

**void createMatrixPartForProcesses(int\* countOfLinesForProcess, int\* offsetArray, int\* countOfMatrElem\_s, int\* countOfSkipedElem, int N, int size) {**

**int currentOffset = 0;**

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

**countOfLinesForProcess[i] = N / size; // Гарантированно каждый получит число строк матрицы, равное целой части N / size**

**//Докидываем процессам от 1 до N % size по одной строке для равномерного распределения**

**//Если конечно же число N не кратно size**

**if (i < N % size) {**

**++countOfLinesForProcess[i];**

**}**

**//Начало каждого процесcа i содержится в offsets[i]**

**offsetArray[i] = currentOffset;**

**currentOffset += countOfLinesForProcess[i];**

**countOfMatrElem\_s[i] = countOfLinesForProcess[i] \* N;**

**countOfSkipedElem[i] = offsetArray[i] \* N;**

**}**

**}**

**double countAbsSqr(const double\* vec, const int N) {**

**double absSqr = 0.0;**

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

**absSqr += pow(vec[i], 2);**

**}**

**return absSqr;**

**}**

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

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

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

**A[i \* N + j] = 1.0;**

**}**

**A[i \* N + i] = 2.0;**

**x[i] = 0.0;**

**b[i] = N + 1;**

**}**

**}**

**void printMatrix(const double\* matrix) {**

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

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

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

**}**

**printf("\n");**

**}**

**printf("\n");**

**}**

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

**MPI\_Init(&argc, &argv);**

**double startTime = MPI\_Wtime();**

**int rank = 0;**

**int size = 0;**

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

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

**int\* countLinesForProc = new int[size];**

**int\* offsetArray = new int[size];**

**int\* countOfMatrElem\_s = new int[size];**

**int\* countOfSkipedElem = new int[size];**

**createMatrixPartForProcesses(countLinesForProc, offsetArray, countOfMatrElem\_s, countOfSkipedElem, N, size);**

**double\* partOfA = new double[countLinesForProc[rank] \* N];**

**double\* A = new double[N \* N];**

**double\* x = new double[N];**

**double\* b = new double[N];**

**double g\_x = 0.0;**

**double abs\_b = 0.0;**

**if (rank == 0) {**

**firstInit(A, b, x);**

**g\_x = 1;**

**//abs\_b = sqrt(countAbsSqr(b, N));**

**}**

**// рассылаем вектор x, b, g(x)**

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

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

**MPI\_Bcast(&g\_x, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);**

**abs\_b = sqrt(countAbsSqr(b, N));**

**/\***

**здесь &g\_x - адрес начала буфера для приема сообщения**

**1 - число элементов принимаемого массива**

**MPI\_DOUBLE - тип элемента принимающего массива**

**0 - номер процесса-отправителя**

**MPI\_COMM\_WORLD - коммуникатор**

**\*/**

**// разрезаем матрицу А на части**

**MPI\_Scatterv(A, countOfMatrElem\_s, countOfSkipedElem, MPI\_DOUBLE, partOfA, countOfMatrElem\_s[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);**

**/\***

**здесь A - адрес с которого начинается пересылаемый участок информации**

**countOfMatrElem\_s - адрес массива размеров отправляемых сообщений**

**countOfSkipedElem - адрес массива смещений отправляемых сообщений**

**MPI\_DOUBLE - тип элемента данных отправляемого сообщения**

**partOfA - адрес начала буфера для приема сообщения**

**countOfMatrElem\_s[rank] - число элементов принимаемого сообщения**

**MPI\_DOUBLE - тип элемента данных принимаемого сообщения**

**0 - номер процесса-распылителя**

**MPI\_COMM\_WORLD - коммуникатор**

**\*/**

**double\* Ax\_b = new double[countLinesForProc[rank]];**

**double\* partOfX = new double[countLinesForProc[rank]];**

**double absOfSum = 0;**

**int iterationCount = 0;**

**while (g\_x > eps && iterationCount < maxIterationsCount) {**

**// заполняем Ax-b для каждого процесса по частям**

**for (int i = 0; i < countLinesForProc[rank]; ++i) {**

**Ax\_b[i] = 0;**

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

**Ax\_b[i] += partOfA[i \* N + j] \* x[j];**

**}**

**Ax\_b[i] = Ax\_b[i] - b[offsetArray[rank] + i];**

**}**

**// заполняем x для каждого процесса по частям**

**for (int i = 0; i < countLinesForProc[rank]; ++i) {**

**partOfX[i] = x[offsetArray[rank] + i] - taul \* Ax\_b[i];**

**}**

**// собираем по кусочкам вектор x**

**MPI\_Allgatherv(partOfX, countLinesForProc[rank], MPI\_DOUBLE, x, countLinesForProc, offsetArray, MPI\_DOUBLE, MPI\_COMM\_WORLD);**

**/\***

**здесь partOfX - адрес с которого начинается пересылаемый участок информации**

**countLinesForProc[rank] - число элементов отправляемого сообщения**

**MPI\_DOUBLE - тип элементов отправляемого сообщения**

**x - адрес начала буфера для приема сообщения**

**countLinesForProc - адрес массива размеров принимаемых сообщений**

**offsetArray - адрес массива смещений принимаемых сообщений**

**MPI\_DOUBLE - тип элемента данных принимаемых сообщений**

**MPI\_COMM\_WORLD - коммуникатор**

**\*/**

**// считаем норму по частям,**

**double partAbs = countAbsSqr(Ax\_b, countLinesForProc[rank]);**

**//не будем отсылать нулевому процессу absOfSum а сделаем так чтобы каждый процесс сам считал g\_x**

**MPI\_Allreduce(&partAbs, &g\_x, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);**

**/\***

**здесь &partAbs - адрес с которого начинается пересылаемый участок информации**

**&absOfSum - адрес начала буфера для приема сообщения**

**1 - кол-во элементов отправляемого сообщения**

**MPI\_DOUBLE - тип элемента данных пересылаемого сообщения**

**MPI\_SUM - операция суммы**

**0 - номер принимающего процесса**

**MPI\_COMM\_WORLD - коммуникатор**

**\*/**

**if (rank == 0) {**

**iterationCount++;**

**}**

**// после каждой итерации Bcast рассылает обновленные значения переменных**

**MPI\_Bcast(&iterationCount, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);**

**g\_x = sqrt(g\_x) / abs\_b;**

**}**

**if (rank == 0) {**

**if (iterationCount < maxIterationsCount) {**

**double endTime = MPI\_Wtime();**

**printf("Time taken %f\n", endTime - startTime);**

**}**

**else {**

**//Нужно поменять знак, если метод не сошелся с заданным таул**

**printf("No converge, change taul sign\n");**

**}**

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

**printf("%f ", x[i]);**

**}**

**}**

**delete[] countLinesForProc;**

**delete[] offsetArray;**

**delete[] x;**

**delete[] b;**

**delete[] A;**

**delete[] partOfA;**

**delete[] Ax\_b;**

**delete[] partOfX;**

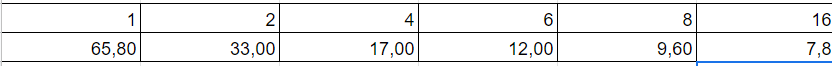
**MPI\_Finalize();**

**return 0;**

**}**

# ПРИЛОЖЕНИЕ 3. Графики времени, ускорения, эффективности для параллельной программы

# Диаграмма



# 

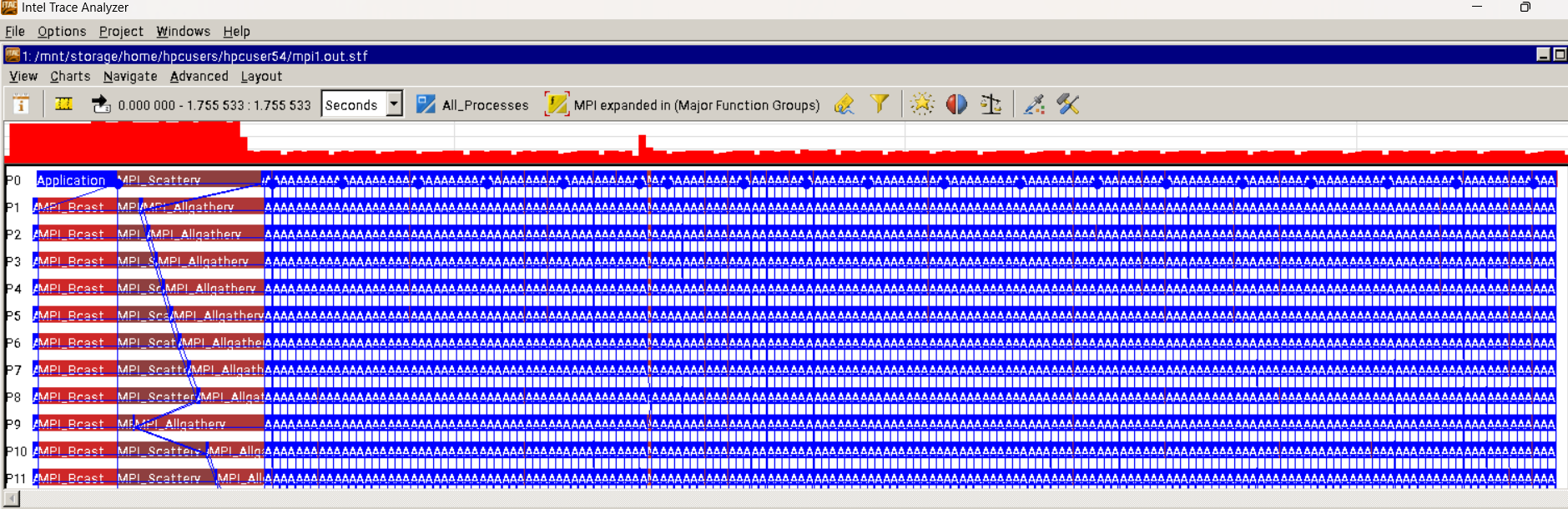
# Диаграмма

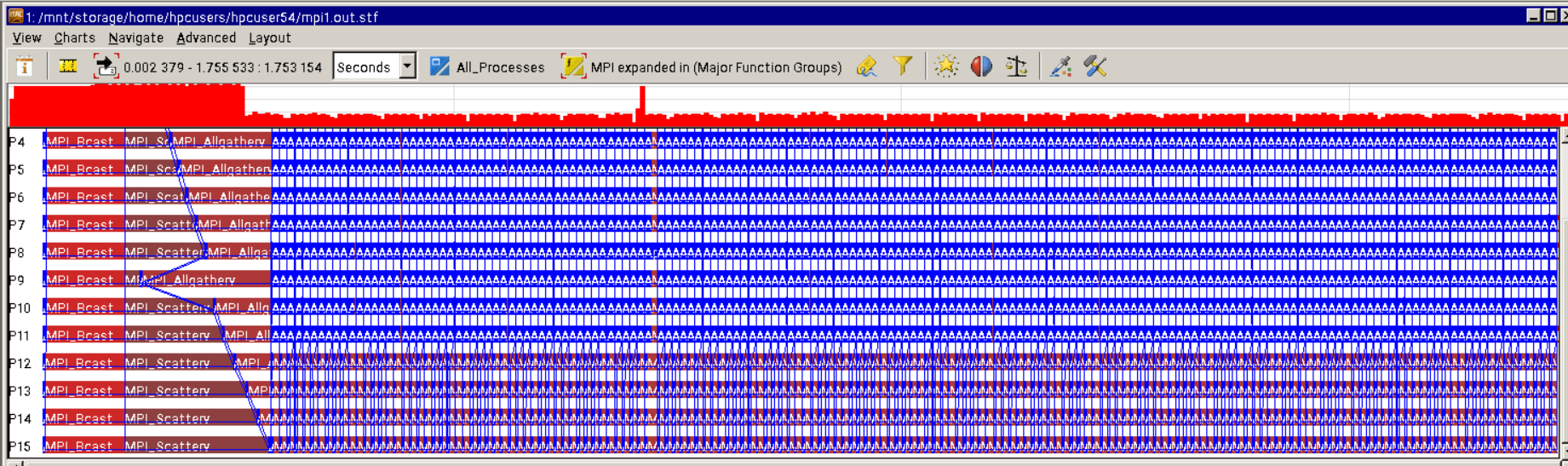


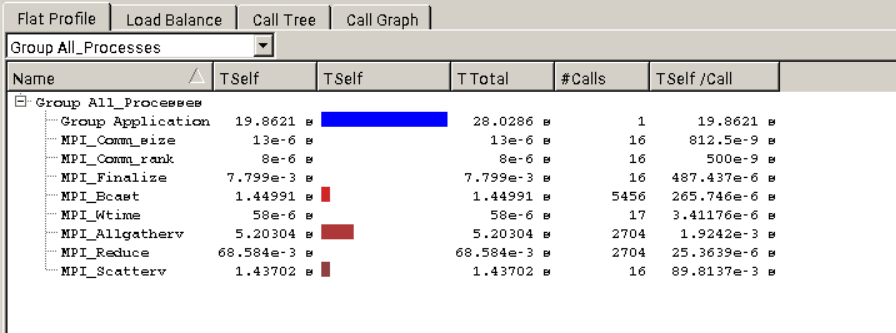
# Диаграмма

# 

# ПРИЛОЖЕНИЕ 4. Профилирование







# ПРИЛОЖЕНИЕ 5. График времени для исправленного варианта параллельной программы

Данный вариант программы был проверен на корректность вычислений на тестовых данных, диагональ матрицы утяжеленная, оба вектора x и b заполнены также как и в предыдущих измерениях времени. Для того чтобы увидеть насколько мы получим выигрыш во времени при уменьшении количества коллективных функций в параллельной программе размер матрицы и векторов и прочие параметры не изменялись.

