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

Балашова Вячеслава Вадимовича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Кандидат технических наук
А. Ю. Власенко

Новосибирск 2023

Содержание

Цель

1. Научиться создавать параллельные программы на основе интерфейса Message Parsing Interface (MPI).

2. Научиться работать с кластерами, запускать на них параллельные программы с интерфейсом MPI.

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.

3. 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.

4. Выполнить профилирование двух вариантов программы с помощью jumpshot или ITAC (Intel Trace Analyzer and Collecter) при использовании 16-и или 24-х ядер.

5. На основании полученных результатов сделать вывод.

Описание работы

Ход выполнения работы:

1. Был использован метод простой итерации:
   a. Решение на каждом шаге задается формулой:

$$x_{n+1} = x_n - \tau(Ax_n - b)$$

   Где τ – константа, параметр метода.
   b. Критерий завершения счета:

$$\frac{\| Ax_n - b \|_{\mathbb{R}}}{\| b \|_{\mathbb{R}}} < \varepsilon$$

   c. $\| u \|_{\mathbb{R}} = \sqrt{\sum_{i=0}^{N-1} u_i^2}$ – норма вектора.
   d. $\varepsilon = 1 * 10^{-5}$

2. Алгоритм применялся на специально подготовленной матрице, которая заполняется следующим образом:
   a. Выбирается специальное число – seed – «семя» генератора, которое гарантирует одинаковые псевдослучайные значения в заполнении матрицы для разных запусков программ.
   b. Матрица имеет размер 3000 x 3000.
   c. Для каждой ячейки матрицы с помощью генератора выбирается число в промежутке [-1; 1].
   d. Если ячейка матрицы принадлежит главной диагонали, то к значению ячейки прибавляется число N*1.1, где N – размерность матрицы.
3. Также специально подготавливались вектор «b», ячейки которого заполнялись случайными целыми числами в промежутке [0; 99], и вектор начального приближения «x», ячейки которого заполнялись случайными целыми числами в промежутке [0; 9].
4. Была написана программа на языке программирования C, реализующая данный итерационный алгоритм последовательно (без использования распараллеливающих методов) (см. Приложение 1).
5. Была написана программа на языке программирования C, реализующая данный итерационный алгоритм с разбиением вычислений на несколько процессов, общающиеся между собой с помощью коллективных функций интерфейса MPI (см. Приложение 2). Была смоделирована схема получения матрицы и вектора на нулевом

процессе. Впоследствии матрица и вектор разрезаются на отличающиеся не более чем на 1 строчку части и рассылаются остальным процессам. За счет распараллеливания вычислений были ускорены вычисления:

a. Скалярного квадрата вектора
b. Умножение матрицы на вектор
c. Умножение вектора на скаляр
d. Разность векторов

6. В каждой программе был произведен замер времени работы алгоритма (в случае параллельных программ замером времени занимался процесс с рангом = 0). Также для параллельной программы был проведен замер времени из работы на разном количестве процессов: 1, 2, 4, 8, 16, 24. Каждая программа запускалась по 5 раз, из этих 5-ти запусков бралось минимальное время работы.

7. Были построены графики зависимости времени работы программы, эффективности (Ep = Sp / p= * 100%, где Sp – ускорение, p – количество процессов) и ускорения (Sp = T1 / Tp, где T1 – время работы последовательной программы, Tp – время работы программы на p процессах) от количества параллельных процессов.

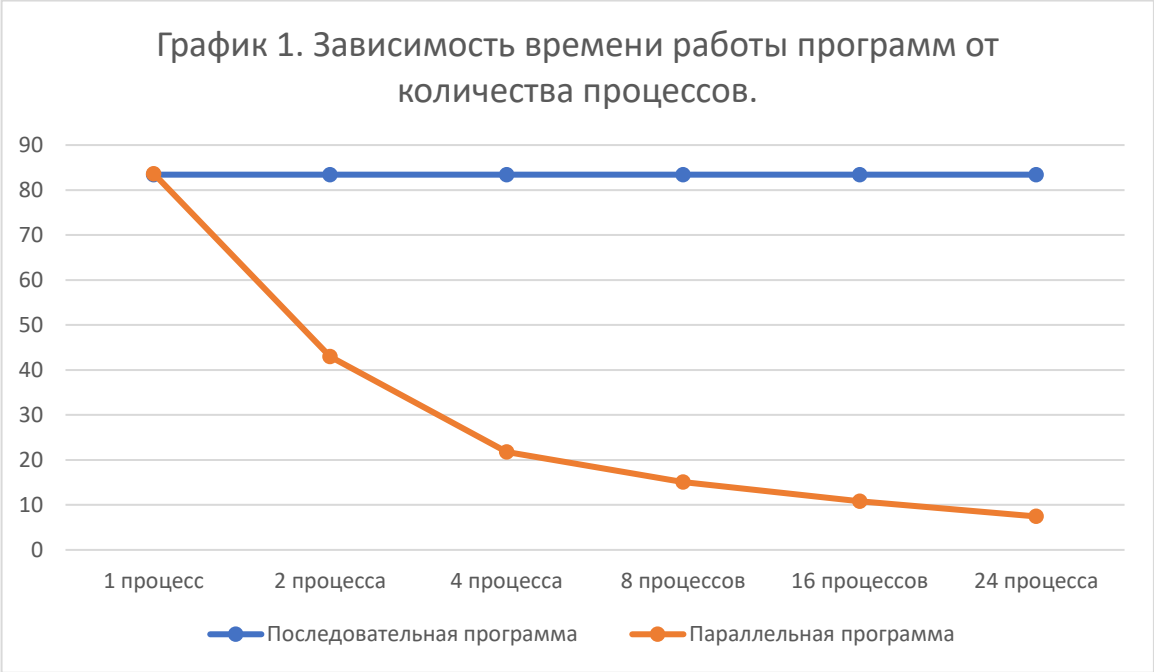


Таблица 1. Соответствие времени работы программы и количества процессов для параллельной программы.

Время, сек.	Время, сек	Количество процессов, шт.
83,646738	83,476821	1 процесс

42,986473	83,4868	2 процесса
21,787699	83,51432	4 процесса
15,089198	83,49331	8 процессов
10,804918	83,500123	16 процессов
7,43627	83,488901	24 процесса

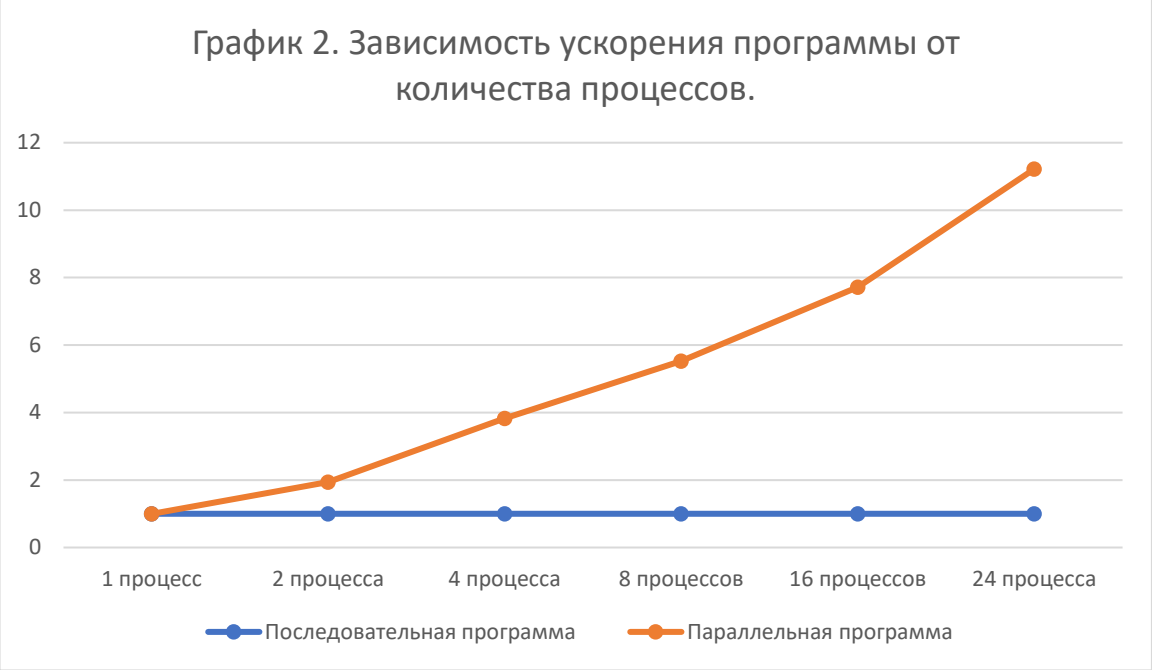
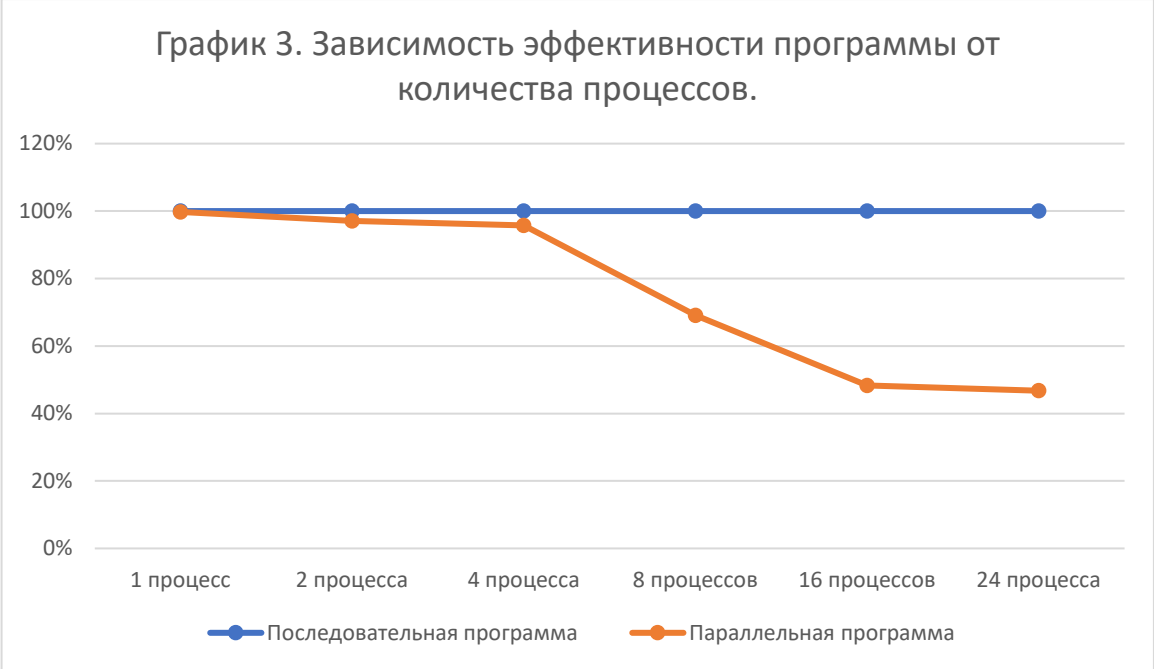


Таблица 2. Соответствие коэффициентов ускорения и количества процессов для параллельной программы.

Ускорение, коэфф.	Ускорение, коэфф.	Количество процессов, шт.
0,9973658148	1	1 процесс
1,940759294	1	2 процесса
3,829059553	1	4 процесса
5,528882118	1	8 процессов
7,721150406	1	16 процессов
11,21884991	1	24 процесса



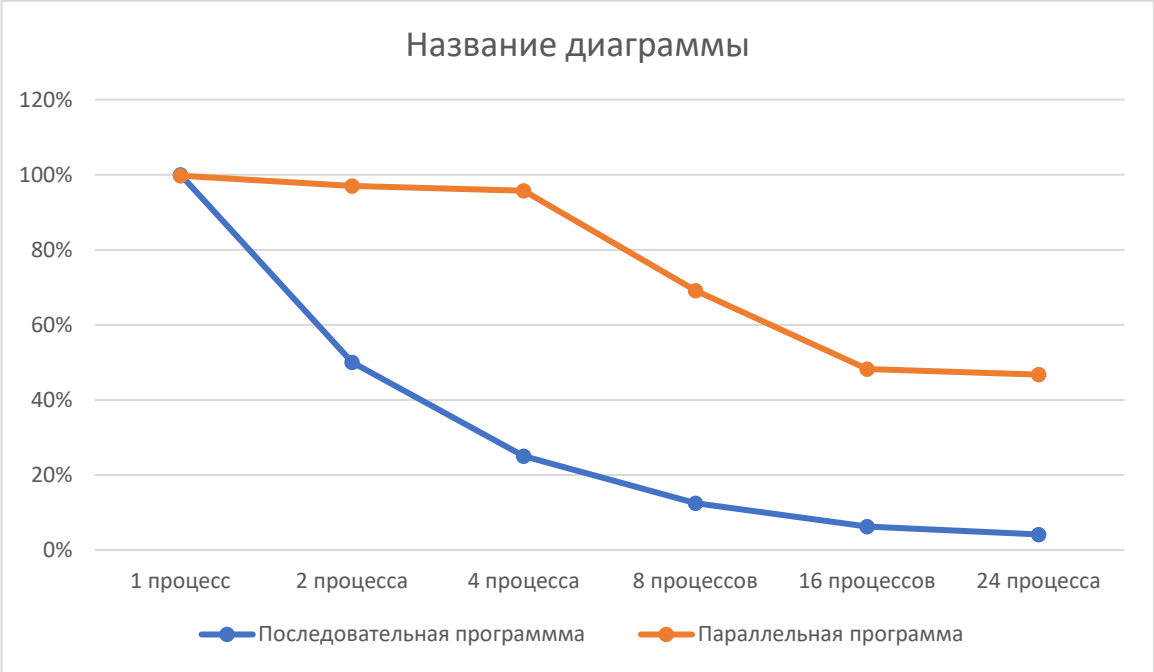
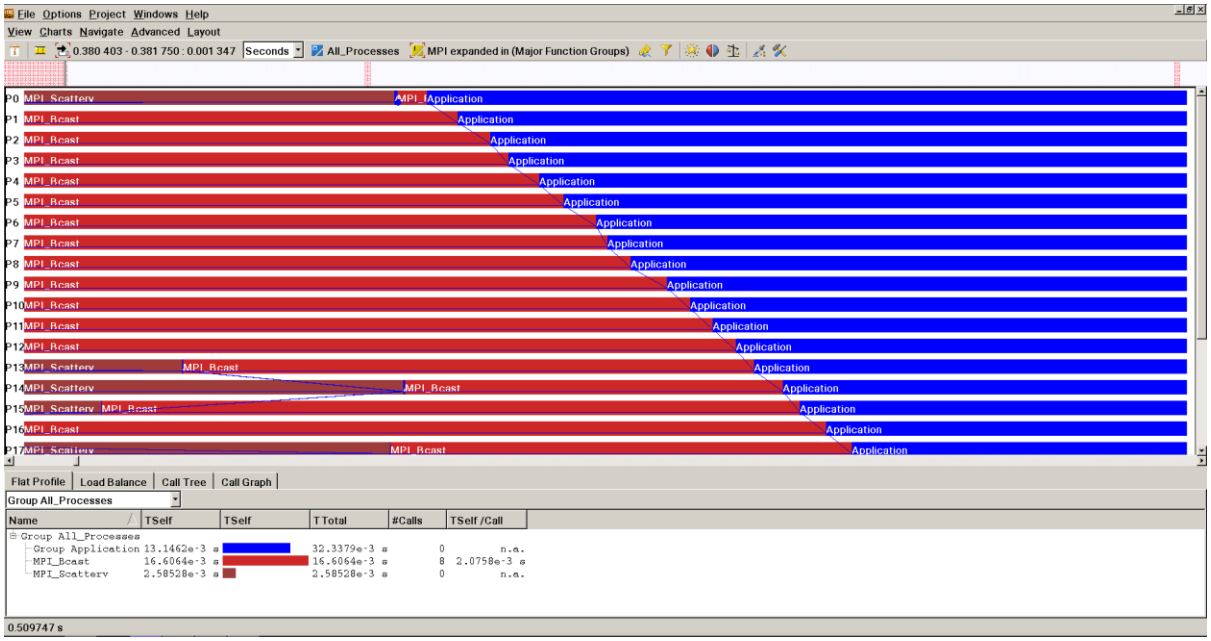


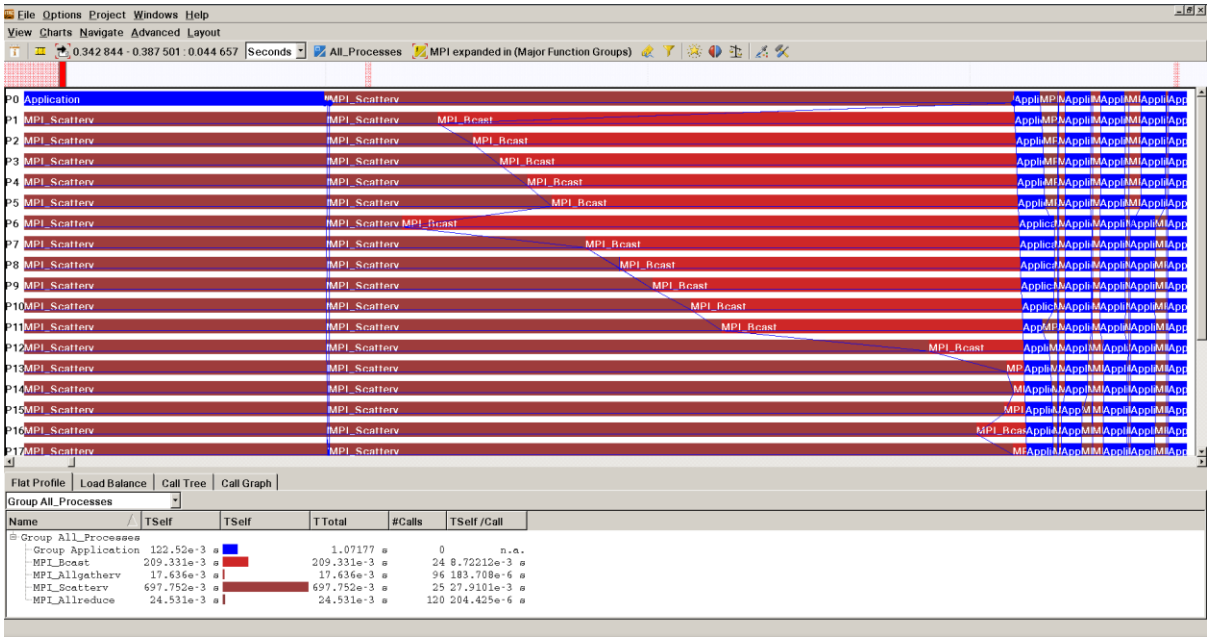
Таблица 3. Соответствие ускорения программы и количества процессов для параллельной программы.

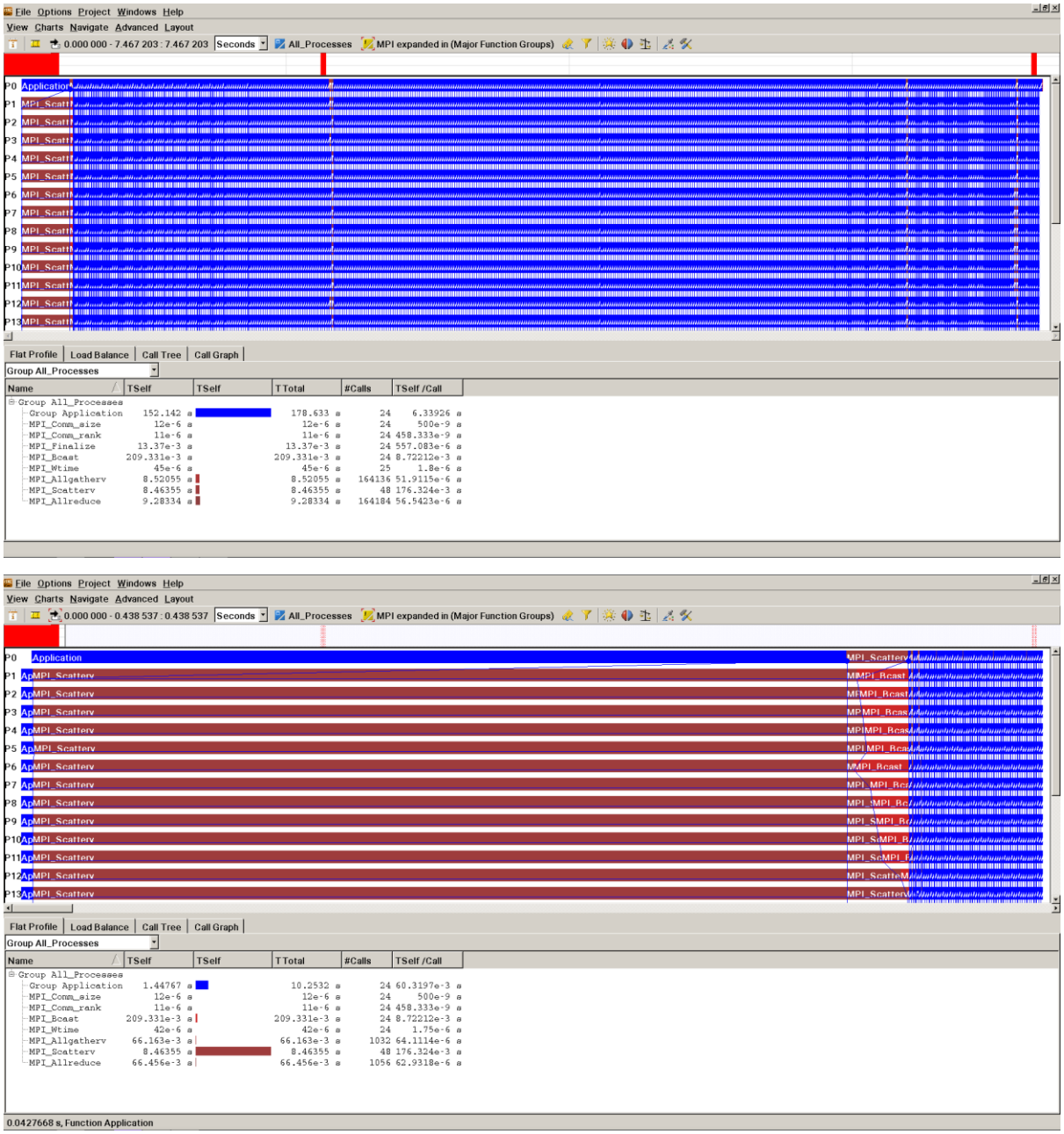
Эффективность, %	Эффективность, %	Количество процессов, шт.
99,74%	100%	1 процесс
97,04%	50%	2 процесса
95,73%	25%	4 процесса
69,11%	12,5%	8 процессов
48,26%	6,25%	16 процессов

46,75%	4,167%	24 процесса

8. Было выполнено профилирование параллельной программы, запущенной на 24 процессах с помощью инструмента ITAC (Intel Trace Analyzer and Collector). С помощью этой утилиты было выявлено: сколько времени ушло на различные этапы программы, какие функции блокирующие, а какие нет, какие выполняются функции MPI, когда выполняются функции MPI, а когда пользовательский код.






File Options Project Windows Help
View Charts Navigate Advanced Layout
0.000 000 - 7.467 203 : 7.467 203
Seconds
All_Processes
MPI expanded in (Major Function Groups)
P0 Application
P1 MPI_Scatt
P2 MPI_Scatt
P3 MPI_Scatt
P4 MPI_Scatt
P5 MPI_Scatt
P6 MPI_Scatt
P7 MPI_Scatt
P8 MPI_Scatt
P9 MPI_Scatt
P10 MPI_Scatt
P11 MPI_Scatt
P12 MPI_Scatt
P13 MPI_Scatt
Flat Profile
Load Balance
Call Tree
Call Graph
Group All_Processes
Name
TSelf
TSelf
T Total
#Calls
TSelf /Call
Group All_Processes
Group Application 152.142 s 178.633 s 24 6.33926 s
MPI_Comm_size 12e-6 s 12e-6 s 24 500e-9 s
MPI_Comm_rank 11e-6 s 11e-6 s 24 458.333e-9 s
MPI_Finalize 13.37e-3 s 13.37e-3 s 24 557.083e-6 s
MPI_Bcast 209.331e-3 s 209.331e-3 s 24 8.72212e-3 s
MPI_Wtime 45e-6 s 45e-6 s 25 1.8e-6 s
MPI_Allgatherv 8.52055 s 8.52055 s 164136 51.9115e-6 s
MPI_Scatterv 8.46355 s 8.46355 s 48 176.324e-3 s
MPI_Allreduce 9.28334 s 9.28334 s 164184 56.5423e-6 s

File Options Project Windows Help
View Charts Navigate Advanced Layout
0.000 000 - 0.438 537 : 0.438 537
Seconds
All_Processes
MPI expanded in (Major Function Groups)
P0 Application
P1 ApMPI_Scatterv
P2 ApMPI_Scatterv
P3 ApMPI_Scatterv
P4 ApMPI_Scatterv
P5 ApMPI_Scatterv
P6 ApMPI_Scatterv
P7 ApMPI_Scatterv
P8 ApMPI_Scatterv
P9 ApMPI_Scatterv
P10 ApMPI_Scatterv
P11 ApMPI_Scatterv
P12 ApMPI_Scatterv
P13 ApMPI_Scatterv
Flat Profile
Load Balance
Call Tree
Call Graph
Group All_Processes
Name
TSelf
TSelf
T Total
#Calls
TSelf /Call
Group All_Processes
Group Application 1.44767 s 10.2532 s 24 60.3197e-3 s
MPI_Comm_size 12e-6 s 12e-6 s 24 500e-9 s
MPI_Comm_rank 11e-6 s 11e-6 s 24 458.333e-9 s
MPI_Bcast 209.331e-3 s 209.331e-3 s 24 8.72212e-3 s
MPI_Wtime 42e-6 s 42e-6 s 24 1.75e-6 s
MPI_Allgatherv 66.163e-3 s 66.163e-3 s 1032 64.1114e-6 s
MPI_Scatterv 8.46355 s 8.46355 s 48 176.324e-3 s
MPI_Allreduce 66.456e-3 s 66.456e-3 s 1056 62.9318e-6 s
0.0427668 s, Function Application

Заключение

В ходе выполнения практической работы был получен опыт работы с вычислительным кластером и использование на нем параллельных программ, написанных с помощью интерфейса параллельного программирования MPI. На кластере запускать подобные программы гораздо удобнее, так как на кластерах присутствует система очередей, позволяющая избавиться от конкуренции за ресурсы компьютера, сохранив производительность.

Были изучены и применены новые функции интерфейса MPI, такие как:

a. MPI_Allreduce – собирает значения со всех процессов в коммуникаторе, производит над ними операцию (например, суммирование) и раздает всем.

b. MPI_Scatterv – позволяет раздавать неравные куски одного массива всем процессам коммуникатора.

c. MPI_Allgatherv – собирает части вектора со всех процессов и раздает всем.

Был получен опыт работы с инструментом профилирования ITAC.

Приложение 1. Последовательная программа, вычисляющая решение СЛАУ итерационным методом (main_default.c).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <mpi.h>

void makeMatrixRandomSymmetrical(double * matrix, int size)
{
   for (int i = 0; i < size; ++i)
   {
      for (int j = i; j < size; ++j)
      {
         matrix[i * size + j] = matrix[j * size + i] = 1. * ((rand() % 2) ? 1 : -1) * (rand() % 200)
/ (rand() % 200 + 1);
         if (i == j)
         {
            matrix[i * size + j] += 1.1 * size;
         }
      }
   }
}

void makeVectorZero(double * vector, int size)
{
   for (int i = 0; i < size; ++i)
   {
      vector[i] = 0.;
   }
}



void makeVectorRandom(double * vector, int size, int limit)
{
   for (int i = 0; i < size; ++i)
   {
      vector[i] = rand() % limit;
   }
}

void printSLE(double * matrix, int matrixHeight, int matrixWidth, double * vector)
{
   for (int i = 0; i < matrixHeight; ++i)
   {
      for (int j = 0; j < matrixWidth; ++j)
      {
         printf("%6.2lf ", matrix[i * matrixWidth + j]);
```

```
        }
        printf("| %6.2lf\n", vector[i]);
    }
}

void printMatrix(double * matrix, int matrixHeight, int matrixWidth)
{
    for (int i = 0; i < matrixHeight; ++i)
    {
        for (int j = 0; j < matrixWidth; ++j)
        {
            printf("%6.2lf ", matrix[i * matrixWidth + j]);
        }
        printf("\n");
    }
}

void printVector(double * vector, int size)
{
    printf("[");
    for (int i = 0; i < size; ++i)
    {
        printf("%6.2lf, ", vector[i]);
    }
    printf("]\n");
}

void mulMatrixWithVector(double * matrix, int matrixHeight, int matrixWidth, double *
vector, double * result)
{
    for (int i = 0; i < matrixHeight; ++i)
    {
        result[i] = 0;
        for (int j = 0; j < matrixWidth; ++j)
        {
            result[i] += matrix[i * matrixWidth + j] * vector[j];
        }
    }
}

void sumVectors(double * v1, double * v2, double * res, int size)
{
    for (int i = 0; i < size; ++i)
    {
        res[i] = v1[i] + v2[i];
    }
}

void subVectors(double * v1, double * v2, double * res, int size)
{
```

```
    for (int i = 0; i < size; ++i)
    {
        res[i] = v1[i] - v2[i];
    }
}

void mulVectorWithScalar(double * vector, double * res, int size, double scalar)
{
    for (int i = 0; i < size; ++i)
    {
        res[i] = vector[i] * scalar;
    }
}

double countScalarSquare(double * vector, int size)
{
    double res = 0;
    for (int i = 0; i < size; ++i)
    {
        res += vector[i] * vector[i];
    }
    return res;
}



int simpleIterationMethod(double * matrix, double * vector, double * result, int size, double
eps, double tao)
{
    printf("size=%d\n\n\n", size);

    int iterationsCounter = 0;
    double vectorBScalarSquare = countScalarSquare(vector, size);
    double * iterationVector = (double *) malloc(sizeof(double) * size); //* Ax^n - b

    mulMatrixWithVector(matrix, size, size, result, iterationVector);
    printf("\n\n\nvb = %lf\n\n\n", vectorBScalarSquare);

    subVectors(iterationVector, vector, iterationVector, size);
    double newEps = eps * eps * vectorBScalarSquare;

    while (iterationsCounter < 10000 && countScalarSquare(iterationVector, size) >= newEps)
    {
        mulVectorWithScalar(iterationVector, iterationVector, size, tao);
        subVectors(result, iterationVector, result, size);


        mulMatrixWithVector(matrix, size, size, result, iterationVector);
        subVectors(iterationVector, vector, iterationVector, size);
        ++iterationsCounter;
```

```
    }
    free(iterationVector);
    return iterationsCounter;
}



int findMismatchInVectors(double * v1, double * v2, int size)
{
    for (int i = 0; i < size; ++i)
    {
        if (v1[i] - v2[i] >= 0.01)
        {
            return i;
        }
    }
    return -1;
}

int main(int argc, char * argv[])
{
    srand(1678536002);

    int matrixSize = 3000;
    if (argc > 1)
    {
        matrixSize = atoi(argv[1]);
    }

    double tao = 0.000001;
    if (argc > 2)
    {
        sscanf(argv[2], "%lf", &tao);
    }

    printf("\n\ntao = %lf\n\n", tao);

    double start;
    MPI_Init(&argc, &argv);
    start = MPI_Wtime();

    double * A = (double *) malloc(sizeof(double) * matrixSize * matrixSize);
    double * b = (double *) malloc(sizeof(double) * matrixSize);
    double * x = (double *) malloc(sizeof(double) * matrixSize);

    makeMatrixRandomSymmetrical(A, matrixSize);
    makeVectorRandom(b, matrixSize, 100);
    makeVectorRandom(x, matrixSize, 10);
```

```
    printf("Answer found for %d iterations in %lf seconds\n", simpleIterationMethod(A, b, x,
matrixSize, 1e-5, tao), MPI_Wtime() - start);

    double * buf = (double *) malloc(sizeof(double) * matrixSize);

    mulMatrixWithVector(A, matrixSize, matrixSize, x, buf);

    printf("b: ");
    printVector(b, matrixSize);
    printf("Ax: ");
    printVector(buf, matrixSize);

    free(A);
    free(b);
    free(x);
    free(buf);

    MPI_Finalize();
    return 0;
}
```

Приложение 2. Параллельная программа, вычисляющая решение СЛАУ итерационным методом (main_parallel.c).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "mpi.h"

void makeMatrixRandomSymmetrical(double * matrix, int size)
{
   for (int i = 0; i < size; ++i)
   {
      for (int j = i; j < size; ++j)
      {
         matrix[i * size + j] = matrix[j * size + i] = 1. * ((rand() % 2) ? 1 : -1) * (rand() % 200)
/ (rand() % 200 + 1);
         if (i == j)
         {
            matrix[i * size + j] += 1.1 * size;
         }
      }
   }
}

void makeVectorRandom(double * vector, int size, int limit)
{
   for (int i = 0; i < size; ++i)
   {
      vector[i] = rand() % limit;
   }
}

double countScalarSquare(double * vector, int size)
{
   double res = 0;
   for (int i = 0; i < size; ++i)
   {
      res += vector[i] * vector[i];
   }
   return res;
}

void printSLE(double * matrix, int matrixHeight, int matrixWidth, double * vector)
{
   for (int i = 0; i < matrixHeight; ++i)
   {
      for (int j = 0; j < matrixWidth; ++j)
      {
         printf("%6.2lf ", matrix[i * matrixWidth + j]);
```

```
        }
        printf("| %6.2lf\n", vector[i]);
    }
}

void printMatrix(double * matrix, int matrixHeight, int matrixWidth)
{
    for (int i = 0; i < matrixHeight; ++i)
    {
        for (int j = 0; j < matrixWidth; ++j)
        {
            printf("%6.2lf ", matrix[i * matrixWidth + j]);
        }
        printf("\n");
    }
}

void printVector(double * vector, int size)
{
    printf("[");
    for (int i = 0; i < size; ++i)
    {
        printf("%6.2lf, ", vector[i]);
    }
    printf("]\n");
}

void printVectorInt(int * vector, int size)
{
    printf("[");
    for (int i = 0; i < size; ++i)
    {
        printf("%6d, ", vector[i]);
    }
    printf("]\n");
}

void mulMatrixWithVector(double * matrix, int matrixHeight, int matrixWidth, double *
vector, double * result)
{
    for (int i = 0; i < matrixHeight; ++i)
    {
        result[i] = 0;
        for (int j = 0; j < matrixWidth; ++j)
        {
            result[i] += matrix[i * matrixWidth + j] * vector[j];
        }
    }
}
```

```
void subVectors(double * v1, double * v2, double * res, int size)
{
   for (int i = 0; i < size; ++i)
   {
      res[i] = v1[i] - v2[i];
   }
}

void mulVectorWithScalar(double * vector, double * res, int size, double scalar)
{
   for (int i = 0; i < size; ++i)
   {
      res[i] = vector[i] * scalar;
   }
}

int main(int argc, char * argv[])
{
   srand(1678536002);

   int matrixSize = 3000;

   double * A = NULL;
   double * b = NULL;
   double * x = NULL;

   double * partOfA = NULL;
   double * partOfB = NULL;
   double * partOfIterationVector = NULL;

   int * partsMatrix;
   int * positionsMatrix;
   int * partsVector;
   int * positionsVector;

   int mpiRank;
   int mpiSize;

   int iterationsCounter = 0;

   double iterationVectorScalarSquarePart = 0.;
   double iterationVectorScalarSquare = 0.;


   double vectorBScalarSquarePart = 0.;
   double vectorBScalarSquare = 0.;

   double eps = 1e-5;
   double tao = 0.000001;
   double newEps;
```

```
int rootRank = 0;
double start;


if (argc > 1)
{
   matrixSize = atoi(argv[1]);
}
if (argc > 2)
{
   sscanf(argv[2], "%lf", &tao);
}

MPI_Init(&argc, &argv);
start = MPI_Wtime();
MPI_Comm_size(MPI_COMM_WORLD, &mpiSize);
MPI_Comm_rank(MPI_COMM_WORLD, &mpiRank);

if (mpiRank == rootRank)
{
   A = (double *) malloc(sizeof(double) * matrixSize * matrixSize);
   b = (double *) malloc(sizeof(double) * matrixSize);
   x = (double *) malloc(sizeof(double) * matrixSize);

   partsMatrix     = (int *) malloc(sizeof(int) * mpiSize);
   positionsMatrix = (int *) malloc(sizeof(int) * mpiSize);

   partsVector     = (int *) malloc(sizeof(int) * mpiSize);
   positionsVector = (int *) malloc(sizeof(int) * mpiSize);


   partsMatrix[0] = matrixSize / mpiSize * matrixSize;
   partsVector[0] = matrixSize / mpiSize;
   positionsMatrix[0] = 0;
   positionsVector[0] = 0;

   for (int i = 1; i < mpiSize; i++)
   {
      partsMatrix[i] = (matrixSize * (i + 1) / mpiSize - matrixSize * i / mpiSize) * matrixSize;
      partsVector[i] = (matrixSize * (i + 1) / mpiSize - matrixSize * i / mpiSize);
      positionsMatrix[i] = partsMatrix[i - 1] + positionsMatrix[i - 1];
      positionsVector[i] = partsVector[i - 1] + positionsVector[i - 1];
   }

   partOfA              = (double *) malloc(sizeof(double) * partsMatrix[mpiRank]);
   partOfB              = (double *) malloc(sizeof(double) * partsVector[mpiRank]);
   partOfIterationVector = (double *) malloc(sizeof(double) * partsVector[mpiRank]);
```

```
    makeMatrixRandomSymmetrical(A, matrixSize);
    makeVectorRandom(b, matrixSize, 100);
    makeVectorRandom(x, matrixSize, 10);

    MPI_Scatterv(b, partsVector, positionsVector, MPI_DOUBLE, partOfB, partsVector[mpiRank], MPI_DOUBLE, rootRank, MPI_COMM_WORLD);

    vectorBScalarSquarePart = countScalarSquare(partOfB, partsVector[mpiRank]);
    MPI_Allreduce(&vectorBScalarSquarePart, &vectorBScalarSquare, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    newEps = eps * eps * vectorBScalarSquare;

    MPI_Scatterv(A, partsMatrix, positionsMatrix, MPI_DOUBLE, partOfA, partsMatrix[mpiRank], MPI_DOUBLE, rootRank, MPI_COMM_WORLD);

    MPI_Bcast(x, matrixSize, MPI_DOUBLE, rootRank, MPI_COMM_WORLD);

    mulMatrixWithVector(partOfA, partsVector[mpiRank], matrixSize, x, partOfIterationVector);
    subVectors(partOfIterationVector, partOfB, partOfIterationVector, partsVector[mpiRank]);

    iterationVectorScalarSquarePart = countScalarSquare(partOfIterationVector, partsVector[mpiRank]);
    MPI_Allreduce(&iterationVectorScalarSquarePart, &iterationVectorScalarSquare, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    while (iterationsCounter < 10000 && iterationVectorScalarSquare >= newEps)
    {
      mulVectorWithScalar(partOfIterationVector, partOfIterationVector, partsVector[mpiRank], tao);
      subVectors(x + positionsVector[mpiRank], partOfIterationVector, partOfIterationVector, partsVector[mpiRank]);

      MPI_Allgatherv(partOfIterationVector, partsVector[mpiRank], MPI_DOUBLE, x, partsVector, positionsVector, MPI_DOUBLE, MPI_COMM_WORLD);

      mulMatrixWithVector(partOfA, partsVector[mpiRank], matrixSize, x, partOfIterationVector);
      subVectors(partOfIterationVector, partOfB, partOfIterationVector, partsVector[mpiRank]);

      iterationVectorScalarSquarePart = countScalarSquare(partOfIterationVector, partsVector[mpiRank]);
      MPI_Allreduce(&iterationVectorScalarSquarePart, &iterationVectorScalarSquare, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

      ++iterationsCounter;
    }
```

```
    printf("Answer found for %d iterations in %lf seconds\n", iterationsCounter, MPI_Wtime() - start);

    printf("b:  ");
    printVector(b, matrixSize);
    makeVectorRandom(b, matrixSize, 1);
    mulMatrixWithVector(A, matrixSize, matrixSize, x, b);
    printf("Ax: ");
    printVector(b, matrixSize);


    free(A);
    free(b);
    free(x);
    free(partsMatrix);
    free(positionsMatrix);
    free(partsVector);
    free(positionsVector);
    free(partOfA);
    free(partOfB);
    free(partOfIterationVector);
  }

  else
  {
    partsVector     = (int *) malloc(sizeof(int) * mpiSize);
    positionsVector = (int *) malloc(sizeof(int) * mpiSize);

    partsVector[0] = matrixSize / mpiSize;
    positionsVector[0] = 0;

    for (int i = 1; i < mpiSize; i++)
    {
      partsVector[i] = (matrixSize * (i + 1) / mpiSize - matrixSize * i / mpiSize);
      positionsVector[i] = partsVector[i - 1] + positionsVector[i - 1];
    }

    x = (double *) malloc(sizeof(double) * matrixSize);
    partOfA = (double *) malloc(sizeof(double) * partsVector[mpiRank] * matrixSize);
    partOfB = (double *) malloc(sizeof(double) * partsVector[mpiRank]);
    partOfIterationVector = (double *) malloc(sizeof(double) * partsVector[mpiRank]);

    MPI_Scatterv(NULL, NULL, NULL, MPI_DOUBLE, partOfB, partsVector[mpiRank], MPI_DOUBLE, rootRank, MPI_COMM_WORLD);

    vectorBScalarSquarePart = countScalarSquare(partOfB, partsVector[mpiRank]);
    MPI_Allreduce(&vectorBScalarSquarePart, &vectorBScalarSquare, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

```
        newEps = eps * eps * vectorBScalarSquare;

        MPI_Scatterv(NULL, NULL, NULL, MPI_DOUBLE, partOfA, partsVector[mpiRank] *
matrixSize, MPI_DOUBLE, rootRank, MPI_COMM_WORLD);

        MPI_Bcast(x, matrixSize, MPI_DOUBLE, rootRank, MPI_COMM_WORLD);

        mulMatrixWithVector(partOfA, partsVector[mpiRank], matrixSize, x,
partOfIterationVector);
        subVectors(partOfIterationVector, partOfB, partOfIterationVector,
partsVector[mpiRank]);

        iterationVectorScalarSquarePart = countScalarSquare(partOfIterationVector,
partsVector[mpiRank]);
        MPI_Allreduce(&iterationVectorScalarSquarePart, &iterationVectorScalarSquare, 1,
MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

        while (iterationsCounter < 10000 && iterationVectorScalarSquare >= newEps)
        {
            mulVectorWithScalar(partOfIterationVector, partOfIterationVector,
partsVector[mpiRank], tao);
            subVectors(x + positionsVector[mpiRank], partOfIterationVector,
partOfIterationVector, partsVector[mpiRank]);

            MPI_Allgatherv(partOfIterationVector, partsVector[mpiRank], MPI_DOUBLE, x,
partsVector, positionsVector, MPI_DOUBLE, MPI_COMM_WORLD);

            mulMatrixWithVector(partOfA, partsVector[mpiRank], matrixSize, x,
partOfIterationVector);
            subVectors(partOfIterationVector, partOfB, partOfIterationVector,
partsVector[mpiRank]);

            iterationVectorScalarSquarePart = countScalarSquare(partOfIterationVector,
partsVector[mpiRank]);
            MPI_Allreduce(&iterationVectorScalarSquarePart, &iterationVectorScalarSquare, 1,
MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

            ++iterationsCounter;
        }

        free(x);
        free(partOfA);
        free(partOfB);
        free(partOfIterationVector);
        free(partsVector);
        free(positionsVector);
    }
    MPI_Finalize();
    return 0;
}
```