**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»***

студентки 2 курса, группы 22203

***Никулиной Анны Андреевны***

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

*А.Ю. Власенко*

Новосибирск 2024

## СОДЕРЖАНИЕ

[**СОДЕРЖАНИЕ 2**](#_jmvfz8n56gd3)

[**ЦЕЛЬ 3**](#_gwjhe0mv7mln)

[**ЗАДАНИЕ 3**](#_pa24bvjw2tlq)

[**ОПИСАНИЕ РАБОТЫ 4**](#_xdr7o9cqc5br)

[**ЗАКЛЮЧЕНИЕ 5**](#_vit67dohzc0e)

[**Приложение 1. Код последовательной программы. 6**](#_7g0hj4t35jra)

[**Приложение 2. Код параллельной программы. 8**](#_1aw9129a8n43)

[**Приложение 3. График времени. 11**](#_s21fk3w2hbip)

[**Приложение 4. График ускорения. 12**](#_eedwk78lemn6)

[**Приложение 5. График эффективности. 13**](#_1ci93xb)

[**Приложение 6. Профилирование. 14**](#_3whwml4)

## ЦЕЛЬ

Научиться применять MPI на практике, в данном случае для решения системы линейных алгебраических уравнений

## ЗАДАНИЕ

1. 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. 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. 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. 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. 5. На основании полученных результатов сделать вывод.

Вариант: Метод минимальных невязок

## ОПИСАНИЕ РАБОТЫ

1. Была написана последовательная программа, находящая решение примерно за 43 секунды. Число N для такого результата было взято равным 6144, а количество итераций было 925. Код данной программы приведен в Приложении 1.
2. Была написана параллельная программа, использующая коллективные коммуникации. Код данной программы приведен в Приложении 2.
3. Были построены графики времени, ускорения и эффективности. Данные графики приведены в Приложениях 3, 4 и 5 соответственно.
4. Было выполнено профилирование с помощью утилиты Intel Trace Analyzer and Collector. Результаты приведены в Приложении 6.

## ЗАКЛЮЧЕНИЕ

1. В ходе выполнения практической работы были изучены приемы параллелизации метода минимальных невязок с помощью MPI. Из приведенных ниже графиков можно заметить, что время после 8 процессов начинает расти, что, скорее всего, связано с количеством ядер на одном узле кластера. Также, на графике эффективности видно, что она уменьшается, так как больше времени начинает тратиться на коллективные операции между большим кол-вом процессов, нежели на сами операции. А также, в ходе работы был изучен инструмент профилирования Intel Trace Analyzer and Collector.

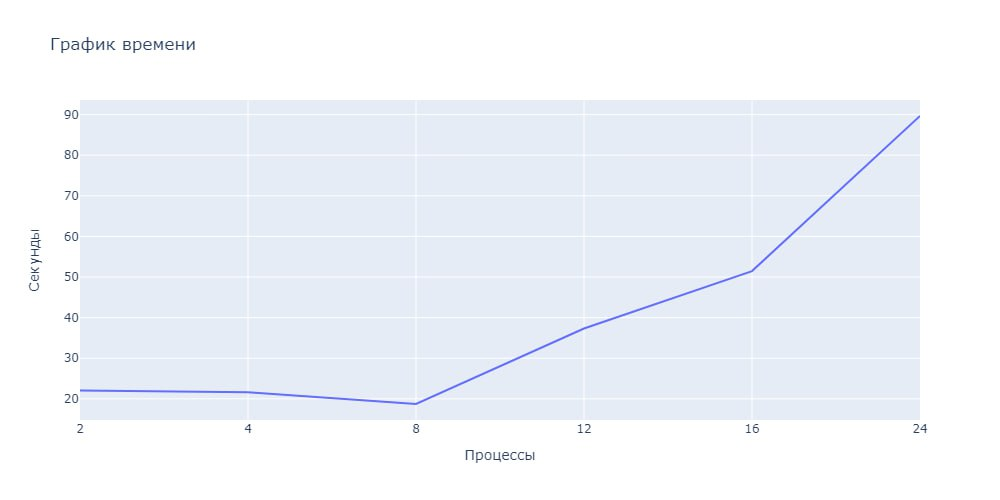
## Приложение 1. *Код последовательной программы.*

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <math.h>  
#include "mpi.h"  
  
void MatrixOnVector(double\* matrix, double\* vec, double\* res, int size){  
 for(int i = 0; i < size; i++){  
 res[i] = 0;  
 for(int j = 0; j < size; j++){  
 res[i] += matrix[i\*size+j] \* vec[j];  
 }  
 }  
}  
  
void VecMinusVec(double\* vec1, double\* vec2, double\* res, int size){  
 for(int i = 0; i < size; i++){  
 res[i] = vec1[i] - vec2[i];  
 }  
}  
  
double ScalarMul(double\* vec1, double\* vec2, int size){  
 double res = 0;  
 for(int i = 0; i < size; i++){  
 res += vec1[i] \* vec2[i];  
 }  
 return res;  
}  
  
void VectorOnScalar(double\* vec, double\* res, double scalar, int size){  
 for(int i = 0; i < size; i++){  
 res[i] = vec[i] \* scalar;  
 }  
}  
  
double Norm(double\* vec, int size){  
 double res = 0;  
 for(int i = 0; i < size; i++){  
 res += vec[i] \* vec[i];  
 }  
 return sqrt(res);  
}  
  
void Filling(double\* matrix, double\* vec, double\* x\_n, int size){  
 for(int i = 0; i < size; i++){  
 srand(i);  
 for(int j = 0; j < size; j++){  
 if(i == j){  
 matrix[i\*size+j] = (rand() % 1000) + size;  
 } else {  
 matrix[i\*size+j] = size/2;  
 }  
 }  
 x\_n[i] = 0;  
 }  
  
 double\* u = (double\*)malloc(sizeof(double) \* size);  
  
 for(int i = 0; i < size; ++i) {  
 u[i] = sin(2 \* M\_PI \* i / size);  
 }  
  
 MatrixOnVector(matrix, u, vec, size);  
 free(u);  
}  
  
int main(int argc, char \*\*argv){  
  
 MPI\_Init(&argc, &argv);  
 int N = 6144;  
 double epsilon = 0.00001;  
 double tao = 0.1;  
 double ending\_coeff = 0;  
 double start\_time;  
 double end\_time;  
  
 double\* A = (double\*)malloc(sizeof(double) \* N\*N);  
 double\* b = (double \*)malloc(sizeof(double) \* N);  
 double\* x\_n = (double \*)malloc(sizeof(double) \* N);  
 double\* y = (double \*)malloc(sizeof(double) \* N);  
  
 memset(x\_n, 1, sizeof(double) \* N);  
  
 double\* tmp = (double \*)malloc(sizeof(double) \* N);  
 start\_time = MPI\_Wtime();  
 Filling(A, b, x\_n, N);  
 double b\_norm = Norm(b, N);  
  
 for(int i = 0; i < 10000; i++){  
 MatrixOnVector(A, x\_n, tmp, N);  
 VecMinusVec(tmp, b, y, N);  
 double index = Norm(y, N)/ b\_norm;  
 if(index <= epsilon){  
 ending\_coeff ++;  
 if(ending\_coeff == 3){  
 printf("iterations: %d", i);  
 break;  
 }  
 } else ending\_coeff = 0;  
 MatrixOnVector(A, y, tmp, N);  
 tao = ScalarMul(y, tmp, N) / (ScalarMul(tmp, tmp, N) + epsilon);  
 VectorOnScalar(y, tmp, tao, N);  
 VecMinusVec(x\_n, tmp, x\_n, N);  
 }  
 end\_time = MPI\_Wtime();  
 printf("Время выполнения функции: %f секунд\n", end\_time-start\_time);  
 free(A);  
 free(b);  
 free(x\_n);  
 free(y);  
 free(tmp);  
 MPI\_Finalize();  
 return 0;  
}

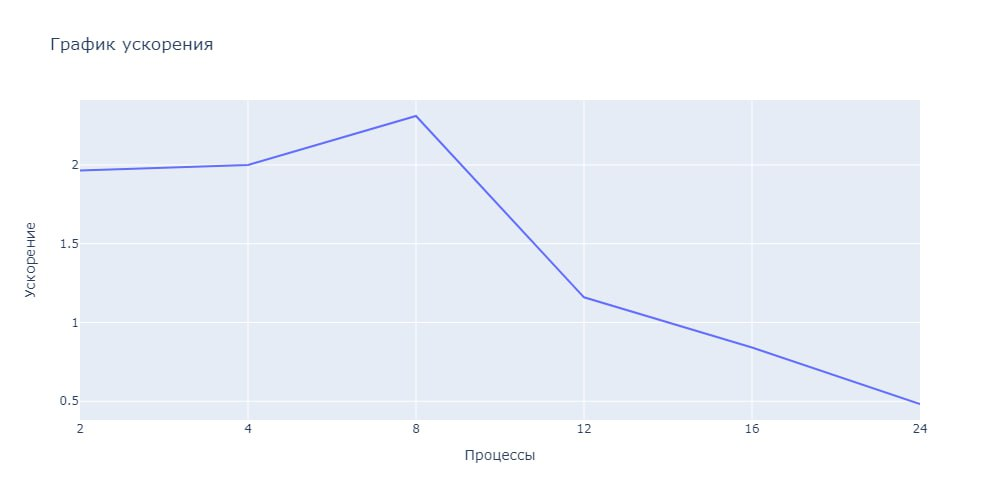
## Приложение 2. *Код параллельной программы.*

1. #include <stdio.h>  
   #include <stdlib.h>  
   #include <string.h>  
   #include <math.h>  
   #include "mpi.h"  
     
   void MatrixOnVector(double\* matrix, double\* vec, double\* res, int size, int part\_size){  
    for(int i = 0; i < part\_size; i++){  
    res[i] = 0;  
    for(int j = 0; j < size; j++){  
    res[i] += matrix[i\*size+j] \* vec[j];  
    }  
    }  
   }  
     
   void VecMinusVec(double\* vec1, double\* vec2, double\* res, int size){  
    for(int i = 0; i < size; i++){  
    res[i] = vec1[i] - vec2[i];  
    }  
   }  
     
   double ScalarMul(double\* vec1, double\* vec2, int size){  
    double res = 0;  
    for(int i = 0; i < size; i++){  
    res += vec1[i] \* vec2[i];  
    }  
    return res;  
   }  
     
   void VectorOnScalar(double\* vec, double\* res, double scalar, int size){  
    for(int i = 0; i < size; i++){  
    res[i] = vec[i] \* scalar;  
    }  
   }  
     
   double Norm(double\* vec, int size){  
    double res = 0;  
    for(int i = 0; i < size; i++){  
    res += vec[i] \* vec[i];  
    }  
    return res;  
   }  
     
   void TestFilling(double\* matrix, double\* vec, double\* x\_n, int size){  
    for(int i = 0; i < size; i++){  
    for(int j = 0; j < size; j++){  
    if(i == j){  
    matrix[i\*size+j] = 2.0;  
    } else {  
    matrix[i\*size+j] = 1.0;  
    }  
    }  
    vec[i] = size + 1;  
    x\_n[i] = 0;  
    }  
   }  
     
   void Filling(double\* matrix, double\* vec, double\* x\_n, int size){  
    for(int i = 0; i < size; i++){  
    srand(i);  
    for(int j = 0; j < size; j++){  
    if(i == j){  
    matrix[i\*size+j] = (rand() % 1000) + size;  
    } else {  
    matrix[i\*size+j] = size/2;  
    }  
    }  
    x\_n[i] = 0;  
    }  
     
    double\* u = (double\*)malloc(sizeof(double) \* size);  
     
    for(int i = 0; i < size; ++i) {  
    u[i] = sin(2 \* M\_PI \* i / size);  
    }  
     
    MatrixOnVector(matrix, u, vec, size, size);  
    free(u);  
   }  
     
   int main(int argc, char \*\*argv){  
    double start\_time;  
    double end\_time;  
    MPI\_Init(&argc, &argv);  
    int rank;  
    int number\_of\_processes;  
    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
    MPI\_Comm\_size(MPI\_COMM\_WORLD, &number\_of\_processes);  
    int N = 6144;  
    int part\_size = N/number\_of\_processes;  
    double epsilon = 0.00001;  
    //double o = 0.0000001;  
    double tao = 0.1;  
    double ending\_coeff = 0;  
     
    double\* x\_n = (double \*)malloc(sizeof(double) \* N);  
    double\* y = (double \*)malloc(sizeof(double) \* N);  
    double\* A;  
    double\* b;  
     
    double\* tmp = (double \*)malloc(sizeof(double) \* part\_size);  
     
    if(rank == 0){  
    start\_time = MPI\_Wtime();  
    A = (double\*)malloc(sizeof(double) \* N\*N);  
    b = (double \*)malloc(sizeof(double) \* N);  
    Filling(A, b, x\_n, N);  
    }  
     
    double\* b\_part = (double \*)malloc(sizeof(double) \* part\_size);  
    MPI\_Scatter(b, part\_size, MPI\_DOUBLE, b\_part, part\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
     
    double\* A\_part = (double \*) malloc(sizeof(double) \* part\_size\*N);  
    MPI\_Scatter(A, part\_size \* N, MPI\_DOUBLE, A\_part, part\_size \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
     
    MPI\_Bcast(x\_n, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
     
    double b\_norm = 0;  
    double part\_of\_b\_norm = Norm(b\_part, part\_size);  
    MPI\_Allreduce(&part\_of\_b\_norm, &b\_norm, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
    b\_norm = sqrt(b\_norm);  
     
    for(int i = 0; i < 50000; i++){  
    MatrixOnVector(A\_part, x\_n, tmp, N, part\_size);  
    VecMinusVec(tmp, b\_part, tmp, part\_size);  
    double y\_norm = 0;  
    double part\_of\_y\_norm = Norm(tmp, part\_size);  
    MPI\_Allreduce(&part\_of\_y\_norm, &y\_norm, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
    y\_norm = sqrt(y\_norm);  
    double index = y\_norm / b\_norm;  
    /\*if(rank == 0){  
    printf("index = %lf\n", index);  
    }\*/  
    if(index < epsilon){  
    ending\_coeff ++;  
    if(ending\_coeff == 3){  
    if(rank == 0){  
    printf("iterations: %d\n", i);  
    }  
    break;  
    }  
    } else ending\_coeff = 0;  
    MPI\_Allgather(tmp, part\_size, MPI\_DOUBLE, y, part\_size, MPI\_DOUBLE, MPI\_COMM\_WORLD);  
    MatrixOnVector(A\_part, y, tmp, N, part\_size);  
    double tao1 = 0;  
    double tao2 = 0;  
    double part1\_tao = 0;  
    double part2\_tao = 0;  
    part1\_tao = ScalarMul(y+part\_size\*rank, tmp, part\_size) ;  
    part2\_tao = ScalarMul(tmp, tmp, part\_size);  
    MPI\_Allreduce(&part1\_tao, &tao1, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
    MPI\_Allreduce(&part2\_tao, &tao2, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
    tao = tao1/tao2;  
    VectorOnScalar(y+part\_size\*rank, tmp, tao, part\_size);  
    VecMinusVec(x\_n+part\_size\*rank, tmp, tmp, part\_size);  
    MPI\_Allgather(tmp, part\_size, MPI\_DOUBLE, x\_n, part\_size, MPI\_DOUBLE, MPI\_COMM\_WORLD);  
    }  
    if (rank == 0){  
    end\_time = MPI\_Wtime();  
    printf("Время выполнения функции: %f секунд\n", end\_time-start\_time);  
    free(A);  
    free(b);  
    }  
    free(b\_part);  
    free(A\_part);  
    free(x\_n);  
    free(y);  
    free(tmp);  
    MPI\_Finalize();  
    return 0;  
   }

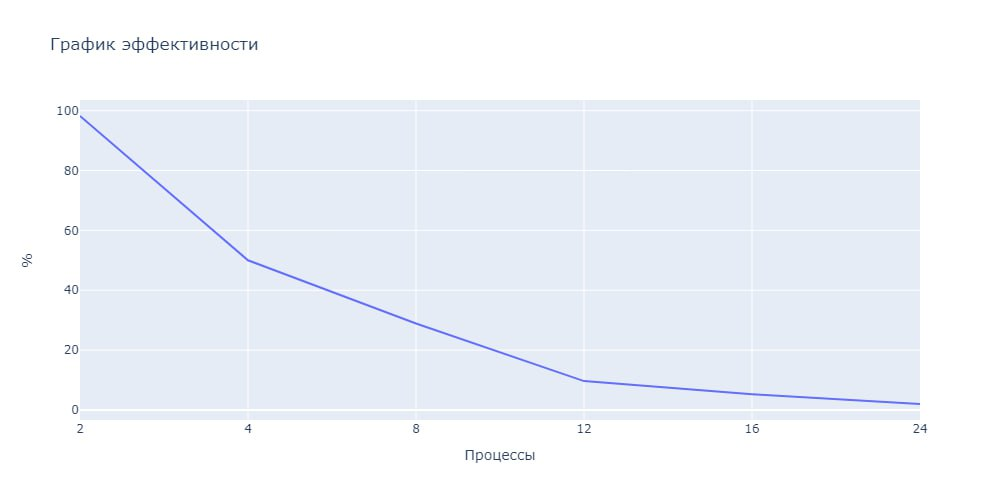
## Приложение 3. *График времени.*



## Приложение 4. *График ускорения.*



## Приложение 5. *График эффективности.*



## Приложение 6. *Профилирование.*

