###### 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, 23210 группы

**Лаухина Егора Денисовича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Артюхов А.А.

Новосибирск 2025

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc98584822)

[ЗАДАНИЕ 3](#_Toc98584823)

[ХОД РАБОТЫ 4](#_Toc98584824)

[ЗАКЛЮЧЕНИЕ 5](#_Toc98584825)

[Приложение 1. Графики 6](#_Toc98584826)

[Приложение 2. Последовательная программа 11](#_Toc98584829)

[Приложение 3. Параллельная программа 1 15](#_Toc98584830)

[Приложение 4. Параллельная программа 2 21](#_Toc98584831)

# *ЦЕЛЬ*

Ознакомиться с основными функциями MPI. Реализовать программу решения системы линейных алгебраических уравнений, используя функционал MPI. Оценить эффективность и ускорение программ при различном числе процессов, предоставленных для решения задачи.

# *ЗАДАНИЕ*

1. Написать 3 программы (одну последовательную и 2 параллельные) на языке C или 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. Параллельные программы реализовать с помощью MPI с разрезанием матрицы A по строкам (в первой программе) или по столбцам (во второй) на близкие по размеру, возможно не одинаковые, части.   
   •Вариант 1: векторы x и b дублируются в каждом MPI-процессе,  
   •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).
3. Замерить время работы последовательного варианта программы и 2-х параллельных при использовании различного числа процессорных ядер. Минимально на 2, 4, 8, 16 (на каждом из наших узлов кластера по 12 ядер), но чем на большем количестве процессов будут выполнены замеры, тем лучше. Также чем больше замеров будет выполнено на одном и том же количестве 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. Выполнить профилирование двух вариантов программы с помощью  
   MPE при использовании 16-и ядер
5. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программ

# ХОД РАБОТЫ

В первую очередь был реализован последовательный алгоритм решения СЛАУ методом минимальных невязок. Затем на основе последовательной программы были написаны два варианта MPI-программ, реализующих разные подходы к разрезанию данных между процессами, описанные в задании. В последующем все варианты алгоритмов были подвергнуты тестированию на корректность решения при разном числе процессов и разных размерах исходной матрицы (параметр N). Также были проведены замеры времени исполнения каждой программы при варьировании числа процессов и при одном размере матрицы для всех вариантов. Предварительно был подобран параметр N (размер матрицы) так, чтобы время исполнения последовательной программы не было меньше 30 секунд. На основании полученных данных были построены таблицы и графики времени исполнения, эффективности и ускорения в зависимости от числа процессов при одинаковом параметре N (см. Приложение). В заключение было проведено профилирование двух вариантов параллельных программ при использовании 16-и ядер.

Параметр N = 1500

|  |  |  |  |
| --- | --- | --- | --- |
| ***MPI\_1*** | | | |
| **Число процессов** | **Время выполнения** | **Эффективность** | **Ускорение** |
| 1 | 50,55771 | 100% | 1 |
| 2 | 24,25912 | 104% | 2,084069981 |
| 4 | 13,17984 | 96% | 3,835986981 |
| 8 | 6,86043 | 92% | 7,369467711 |

|  |  |  |  |
| --- | --- | --- | --- |
| ***MPI\_2*** | | | |
| **Число процессов** | **Время выполнения** | **Эффективность** | **Ускорение** |
| 1 | 50,870756 | 100% | 1 |
| 2 | 24,62584 | 103% | 2,065747036 |
| 4 | 13,263544 | 96% | 3,835381856 |
| 8 | 6,743541 | 94% | 7,543626709 |

# ***ЗАКЛЮЧЕНИЕ***

После анализа данных, полученных в результате замеров времени исполнения программ, в которых используются MPI-команды, можно сделать вывод о том, что использование функций библиотеки MPI и исполнения программы на большом количестве процессов может оказаться довольно эффективным методом ускорения исходной программы

График:

Приложение 2. *Последовательная программа*

#include <iostream>  
#include <cmath>  
#include <vector>  
#include <fstream>  
#define Nx 30  
#define Ny 50  
#define N (Nx\*Ny)  
  
double scalyar(std::vector<double>& y, std::vector<double>& A) {  
 std::vector<double> multi(N, 0);  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 multi[i]+=A[i\*N+j]\*y[j];  
 }  
 }  
 double numerator = 0;  
 for (int i = 0; i < N; i++) {  
 numerator += (y[i]\*multi[i]);  
 }  
 double denominator = 0;  
 for (int i = 0; i < N; i++) {  
 denominator += (multi[i] \* multi[i]);  
 }  
 return numerator/denominator;  
}  
  
void method(std::vector<double>& A, std::vector<double>& x, std::vector<double>& b) {  
 std::vector<double> y(N, 0);  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 y[i]+=A[i\*N + j]\*x[j];  
 }  
 y[i]-=b[i];  
 }  
 double tau = scalyar(y, A);  
 for (int i = 0; i < N; i++) {  
 x[i]-=y[i]\*tau;  
 }  
}  
  
double get\_test(std::vector<double>& A, std::vector<double>& x, std::vector<double>& b) {  
 std::vector<double> help(N, 0);  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 help[i]+=A[i\*N+j]\*x[j];  
 }  
 help[i]-=b[i];  
 }  
 double numerator = 0.0, denominator = 0.0;  
 for (int i = 0; i < N; i++) {  
 numerator += pow(help[i], 2);  
 denominator += pow(b[i], 2);  
 }  
 numerator = pow(numerator, 0.5);  
 denominator = pow(denominator, 0.5);  
 return numerator/denominator;  
}  
  
void intialize(std::vector<double>& A, std::vector<double>& b) {  
  
 for (int i = 0; i < N; i++) {  
 A[i \* N + i] = -4.0;  
 if (i % Nx != 0) {  
 A[i \* N + (i - 1)] = 1.0;  
 }  
 if ((i + 1) % Nx != 0) {  
 A[i \* N + (i + 1)] = 1.0;  
 }  
 if (i >= Nx) {  
 A[i \* N + (i - Nx)] = 1.0;  
 }  
 if (i < N - Nx) {  
 A[i \* N + (i + Nx)] = 1.0;  
 }  
 }  
 for (int i = 0; i < N; i++) {  
 if (i % 2 == 0) {  
 b[i] = (i + 63) % 50;  
 }  
 else {  
 b[i] = -((i + 63) % 50);  
 }  
 }  
}  
  
  
  
  
  
void start() {  
 double epsilon = 1e-5;  
 std::vector<double> A(N\*N, 0.0);  
 std::vector<double> b(N, 0.0);  
 std::vector<double> x(N, 0.0);  
 intialize(A, b);  
 double test = pow(10, 10);  
 while (epsilon < test) {  
 method(A, x, b);  
 test = get\_test(A, x, b);  
 }  
 std::cout << std::endl << "x" << std::endl;  
 for (int i = 0; i < N; i++) {  
 std::cout << x[i] << " ";  
 }  
 std::cout << std::endl << "all is good" << std::endl;  
}  
  
  
  
int main() {  
 start();  
 return 0;  
}

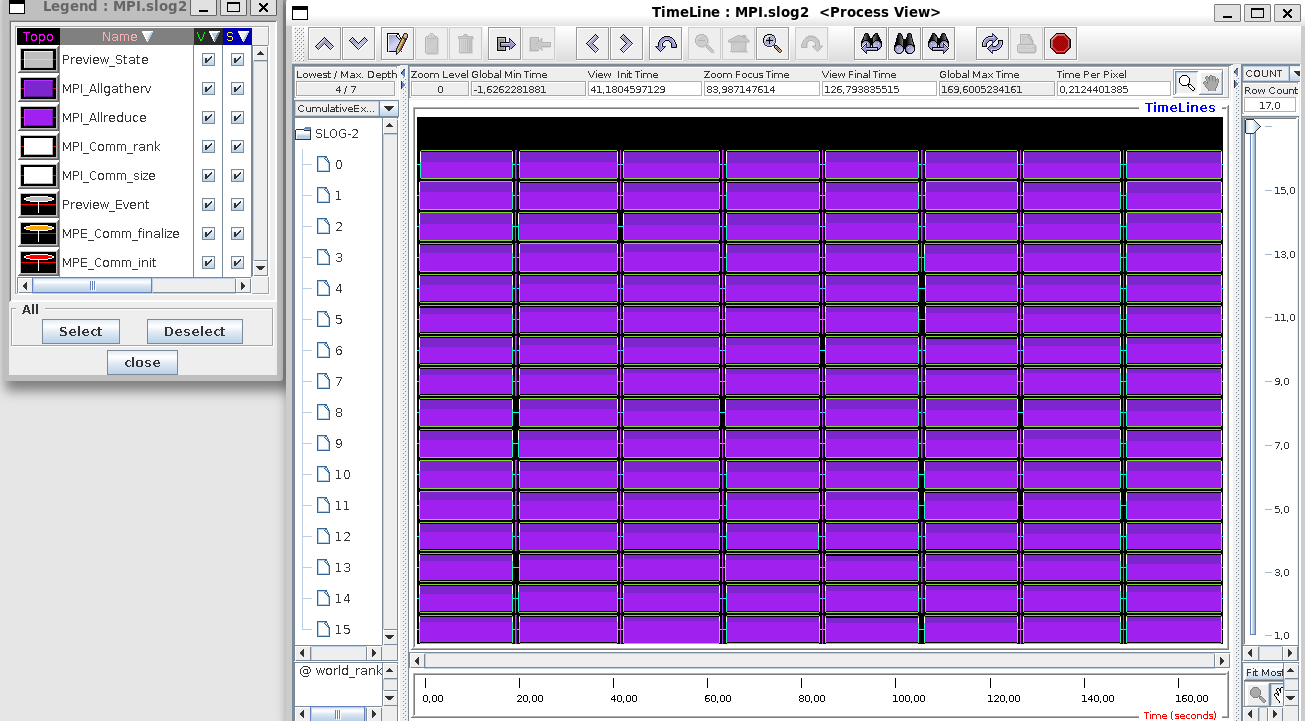
Приложение 3. *Параллельная программа 1*

#include <iostream>  
#include <vector>  
#include <cmath>  
#include <mpi.h>  
  
#define Nx 30  
#define Ny 50  
#define N (Nx \* Ny)  
  
  
double scalyar(std::vector<double>& y, std::vector<std::vector<double>>& A, std::vector<int> size, std::vector<int> shift, int rank) {  
 std::vector<double> multi(N, 0);  
 for (int i = 0; i < size[rank]; i++) {  
 int global\_index = shift[rank] + i;  
 for (int j = 0; j < N; j++) {  
 multi[global\_index] += A[i][j] \* y[j];  
 }  
 }  
  
 double local\_numerator = 0, local\_denominator = 0;  
 for (int i = 0; i < N; i++) {  
 local\_numerator += y[i] \* multi[i];  
 local\_denominator += multi[i] \* multi[i];  
 }  
  
 double global\_numerator = 0, global\_denominator = 0;  
 MPI\_Allreduce(&local\_numerator, &global\_numerator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
 MPI\_Allreduce(&local\_denominator, &global\_denominator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
 double global\_tau = global\_numerator / global\_denominator;  
 return global\_tau;  
}  
  
  
void method(std::vector<std::vector<double>>& A, std::vector<double>& x, std::vector<double>& b, std::vector<int> size, std::vector<int> shift, int rank) {  
 std::vector<double> mpi\_x(N, 0);  
 MPI\_Allgatherv(x.data() + shift[rank], size[rank], MPI\_DOUBLE, mpi\_x.data(), size.data(), shift.data(), MPI\_DOUBLE, MPI\_COMM\_WORLD);  
 std::vector<double> y(N, 0);  
 for (int i = 0; i < size[rank]; i++) {  
 int global\_index = shift[rank] + i;  
 for (int j = 0; j < N; j++) {  
 y[global\_index] += A[i][j] \* mpi\_x[j];  
 }  
 y[global\_index] -= b[global\_index];  
 }  
 double tau = scalyar(y, A, size, shift, rank);  
 for (int i = 0; i < size[rank]; i++) {  
 x[i+shift[rank]] -= y[i+shift[rank]] \* tau;  
 }  
}  
  
  
double get\_test(std::vector<std::vector<double>>& A, std::vector<double>& x, std::vector<double>& b, std::vector<int> size, std::vector<int> shift, int rank) {  
 std::vector<double> help(N, 0);  
 std::vector<double> mpi\_x(N, 0);  
 MPI\_Allgatherv(x.data() + shift[rank], size[rank], MPI\_DOUBLE, mpi\_x.data(), size.data(), shift.data(), MPI\_DOUBLE, MPI\_COMM\_WORLD);  
 for (int i = 0; i < size[rank]; i++) {  
 int global\_index = i + shift[rank];  
 for (int j = 0; j < N; j++) {  
 help[global\_index] += A[i][j] \* mpi\_x[j];  
 }  
 help[global\_index] -= b[global\_index];  
 }  
 double local\_numerator = 0, local\_denominator = 0;  
 for (int i = 0; i < N; i++) {  
 local\_numerator += pow(help[i], 2);  
 local\_denominator += pow(b[i], 2);  
 }  
 double global\_numerator = 0, global\_denominator = 0;  
 MPI\_Allreduce(&local\_numerator, &global\_numerator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
 MPI\_Allreduce(&local\_denominator, &global\_denominator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
 double global\_test = sqrt(global\_numerator) / sqrt(global\_denominator);  
  
 return global\_test;  
}  
  
  
void initialize(std::vector<std::vector<double>>& A, std::vector<double>& b, int size, int shift) {  
 for (int i = 0; i < size; i++) {  
 int global\_index = i + shift;  
 A[i][global\_index] = -4.0;  
 if (global\_index % Nx != 0) {  
 A[i][global\_index - 1] = 1.0;  
 }  
 if ((global\_index + 1) % Nx != 0) {  
 A[i][global\_index + 1] = 1.0;  
 }  
 if (global\_index >= Nx) {  
 A[i][global\_index - Nx] = 1.0;  
 }  
 if (global\_index < N - Nx) {  
 A[i][global\_index + Nx] = 1.0;  
 }  
 }  
  
 for (int i = 0; i < N; i++) {  
 if (i % 2 == 0) {  
 b[i] = (i + 63) % 50;  
 } else {  
 b[i] = -(i + 63) % 50;  
 }  
 }  
}  
  
  
std::vector<int> getRows(int size) {  
 std::vector<int> rows(size);  
 int count1 = N / size;  
 int count2 = N % size;  
 for (int i = 0; i < size; ++i) {  
 rows[i] = count1 + (i < count2);  
 }  
 return rows;  
}  
  
std::vector<int> getShift(int size, std::vector<int>& rows) {  
 std::vector<int> shift(size);  
 int count = 0;  
 for (int i = 0; i < size; i++) {  
 shift[i] = count;  
 count+=rows[i];  
 }  
 return shift;  
}  
  
void start(int rank, int size) {  
 double epsilon = 1e-5;  
 std::vector<int> rows = getRows(size);  
 std::vector<int> shift = getShift(size, rows);  
 std::vector<std::vector<double>> A(rows[rank], std::vector<double>(N, 0.0));  
 std::vector<double> b(N, 0.0);  
 std::vector<double> x(N, 0.0);  
 initialize(A, b, rows[rank], shift[rank]);  
 double test = 1e10;  
 int a = 0;  
 while (test > epsilon) {  
 method(A, x, b, rows, shift, rank);  
 test = get\_test(A, x, b, rows, shift, rank);  
 a++;  
 }  
// std::cout << std::endl << "x" << std::endl;  
// for (int i = 0; i < N; i++) {  
// std::cout << x[i] << " ";  
// }  
// std::cout << std::endl;  
}  
  
  
  
int main(int argc, char\*\* argv) {  
 MPI\_Init(&argc, &argv);  
 int rank, size;  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 auto startTime = MPI\_Wtime();  
 start(rank, size);  
 auto endTime = MPI\_Wtime();  
 if (rank == 0) {  
 std::cout << "Время выполнения: " << (endTime-startTime) << " с" << std::endl;  
 }  
 MPI\_Finalize();  
 return 0;  
}

Приложение 4. *Параллельная программа 2*

#include <iostream>  
#include <vector>  
#include <cmath>  
#include <mpi.h>  
#include <chrono>  
  
#define Nx 30  
#define Ny 50  
#define N (Nx \* Ny)  
  
  
double scalyar(std::vector<double>& y, std::vector<std::vector<double>>& A, std::vector<int> size, std::vector<int> shift, int rank) {  
 std::vector<double> multi(N, 0);  
 for (int i = 0; i < size[rank]; i++) {  
 int global\_index = shift[rank] + i;  
 for (int j = 0; j < N; j++) {  
 multi[global\_index] += A[i][j] \* y[j];  
 }  
 }  
  
 double local\_numerator = 0, local\_denominator = 0;  
 for (int i = 0; i < N; i++) {  
 local\_numerator += y[i] \* multi[i];  
 local\_denominator += multi[i] \* multi[i];  
 }  
  
 double global\_numerator = 0, global\_denominator = 0;  
 MPI\_Allreduce(&local\_numerator, &global\_numerator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
 MPI\_Allreduce(&local\_denominator, &global\_denominator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
 double global\_tau = global\_numerator / global\_denominator;  
 return global\_tau;  
}  
  
  
void method(std::vector<std::vector<double>>& A, std::vector<double>& x, std::vector<double>& b, std::vector<int> size, std::vector<int> shift, int rank, int num\_procs) {  
 std::vector<double> y(size[rank], 0);  
 for (int src = 0; src < num\_procs; src++) {  
 std::vector<double> local\_x(size[src]);  
 if (rank == src) {  
 local\_x = x;  
 }  
 MPI\_Bcast(local\_x.data(), size[src], MPI\_DOUBLE, src, MPI\_COMM\_WORLD);  
 for (int i = 0; i < size[rank]; i++) {  
 for (int j = 0; j < size[src]; j++) {  
 y[i] += A[i][shift[src] + j] \* local\_x[j];  
 }  
 }  
 }  
 for (int i = 0; i < size[rank]; i++) {  
 y[i] -= b[i];  
 }  
  
 double tau = scalyar(y, A, size, shift, rank);  
 for (int i = 0; i < size[rank]; i++) {  
 x[i] -= y[i] \* tau;  
 }  
}  
  
  
  
double get\_test(std::vector<std::vector<double>>& A, std::vector<double>& x, std::vector<double>& b, std::vector<int> size, std::vector<int> shift, int rank, int num\_procs) {  
 std::vector<double> help(size[rank], 0);  
 for (int src = 0; src < num\_procs; src++) {  
 std::vector<double> local\_x(size[src]);  
 if (rank == src) {  
 local\_x = x;  
 }  
 MPI\_Bcast(local\_x.data(), size[src], MPI\_DOUBLE, src, MPI\_COMM\_WORLD);  
 for (int i = 0; i < size[rank]; i++) {  
 for (int j = 0; j < size[src]; j++) {  
 help[i] += A[i][shift[src] + j] \* local\_x[j];  
 }  
 }  
 }  
 for (int i = 0; i < size[rank]; i++) {  
 help[i] -= b[i];  
 }  
 double local\_numerator = 0, local\_denominator = 0;  
 for (int i = 0; i < size[rank]; i++) {  
 local\_numerator += pow(help[i], 2);  
 local\_denominator += pow(b[i], 2);  
 }  
  
 double global\_numerator = 0, global\_denominator = 0;  
 MPI\_Allreduce(&local\_numerator, &global\_numerator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
 MPI\_Allreduce(&local\_denominator, &global\_denominator, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
 return sqrt(global\_numerator) / sqrt(global\_denominator);  
}  
  
  
  
void initialize(std::vector<std::vector<double>>& A, std::vector<double>& b, int size, int shift) {  
 for (int i = 0; i < size; i++) {  
 int global\_index = i + shift;  
 A[i][global\_index] = -4.0;  
 if (global\_index % Nx != 0) {  
 A[i][global\_index - 1] = 1.0;  
 }  
 if ((global\_index + 1) % Nx != 0) {  
 A[i][global\_index + 1] = 1.0;  
 }  
 if (global\_index >= Nx) {  
 A[i][global\_index - Nx] = 1.0;  
 }  
 if (global\_index < N - Nx) {  
 A[i][global\_index + Nx] = 1.0;  
 }  
 }  
  
 for (int i = 0; i < size; i++) {  
 if ((i+shift) % 2 == 0) {  
 b[i] = (i + 63 + shift) % 50;  
 } else {  
 b[i] = -(i + 63 + shift) % 50;  
 }  
 }  
}  
  
  
std::vector<int> getRows(int size) {  
 std::vector<int> rows(size);  
 int count1 = N / size;  
 int count2 = N % size;  
 for (int i = 0; i < size; ++i) {  
 rows[i] = count1 + (i < count2);  
 }  
 return rows;  
}  
  
std::vector<int> getShift(int size, std::vector<int>& rows) {  
 std::vector<int> shift(size);  
 int count = 0;  
 for (int i = 0; i < size; i++) {  
 shift[i] = count;  
 count+=rows[i];  
 }  
 return shift;  
}  
  
void start(int rank, int size) {  
 double epsilon = 1e-5;  
 std::vector<int> rows = getRows(size);  
 std::vector<int> shift = getShift(size, rows);  
 std::vector<std::vector<double>> A(rows[rank], std::vector<double>(N, 0.0));  
 std::vector<double> b(rows[rank], 0.0);  
 std::vector<double> x(rows[rank], 0.0);  
 initialize(A, b, rows[rank], shift[rank]);  
 double test = 1e10;  
 int a = 0;  
 while (test > epsilon) {  
 method(A, x, b, rows, shift, rank, size);  
 test = get\_test(A, x, b, rows, shift, rank, size);  
 a++;  
 }  
// std::cout << "Rank: " << rank << std::endl << "x" << std::endl;  
// for (int i = 0; i < rows[rank]; i++) {  
// std::cout << x[i] << " ";  
// }  
// std::cout << std::endl;  
}  
  
  
  
int main(int argc, char\*\* argv) {  
 MPI\_Init(&argc, &argv);  
 int rank, size;  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 auto startTime = MPI\_Wtime();  
 start(rank, size);  
 auto endTime = MPI\_Wtime();  
 if (rank == 0) {  
 std::cout << "Время выполнения: " << (endTime-startTime) << " с" << std::endl;  
 }  
 MPI\_Finalize();  
 return 0;  
}

**Профилирование**

****

**На скриншоте видно, что много времени занимает MPI\_Allreduce. Это связано с тем, что используется активная пересылка данных от одного процесса ко всем остальным.**