###### 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, 18204 группы

Хорошавина Алексея Константиновича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук,

Доцент.

А.Ю. Власенко

Новосибирск 2019

**Содержание**

[Задание 3](#_Toc33817844)

[Заключение 7](#_Toc33817845)

[Приложение 8](#_Toc33817846)

**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:

* Вариант 1: векторы x и b дублируются в каждом MPI-процессе
* Вариант 2: векторы x и b разрезаются между MPI-процессами аналогично матрице A

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.

**График зависимости времени выполнения**

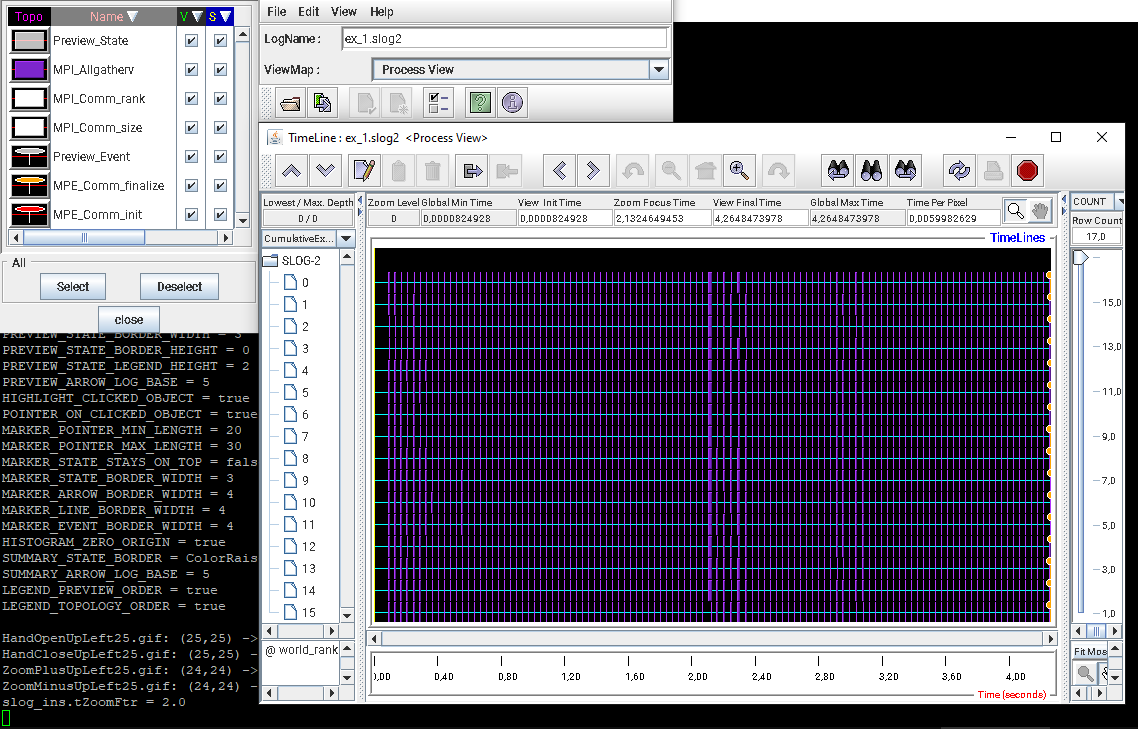
Время работы последовательной программы: 64,02 сек.

**График зависимости ускорения**

**График зависимости эффективности**

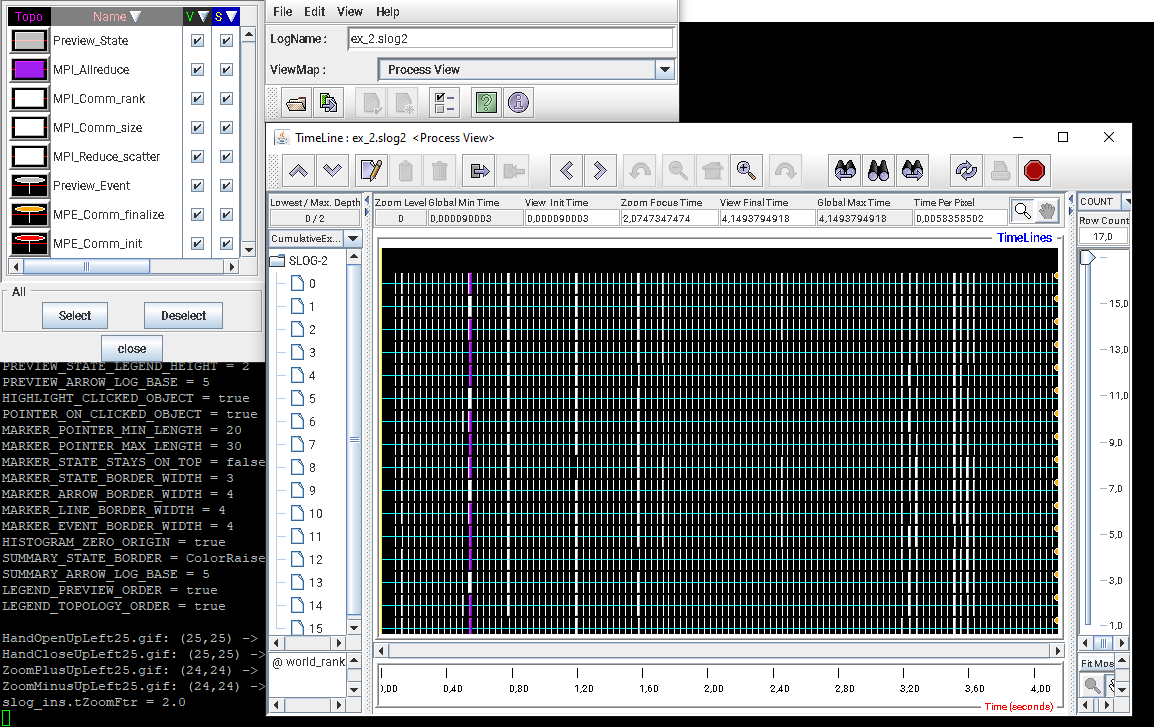
**Профилирование**

Вариант 1: векторы x и b дублируются в каждом MPI-процессе



**Профилирование**

Вариант 2: векторы x и b разрезаются между MPI-процессами аналогично матрице A



**Заключение**

Был реализован алгоритм решения СЛАУ итерационным методом. Были реализованы два варианта распараллелить программу, построены графики зависимости времени работы, ускорения и эффективности. По результатам на графиках видно, что второй вариант с разрезанием векторов аналогично матрице оказался немного эффективней, чем первый.

**Приложение**

Листинг программы.

(Последовательная программа)

#include <cmath>  
#include <cstdio>  
#include <cstdlib>  
#include <time.h>  
  
*void* Matrix\_by\_vector(*int* N, *double* \*\*M, *const double* \*V, *double* \*R)  
{  
 *for*(*int* i = 0; i < N; i++){  
 R[i] = 0;  
 *for*(*int* j = 0; j < N; j++)  
 {  
 R[i] += M[i][j]\*V[j];  
 }  
 }  
}  
  
*void* Minimal\_Nevazki(*int* N, *double* \*\*A, *const double* \*b, *double* \*X, *double* eps)  
{  
 *double* \*R = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*Y = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*Xn = (*double*\*)malloc(*sizeof*(*double*)\*N);  
  
 *double* crit\_module;  
 *double* chisl\_Tau;  
 *double* del\_Tau;  
  
 *for*(*int* i = 0; i < N; i++){  
 Xn[i] = 0;  
 }  
  
 *do*{  
 Matrix\_by\_vector(N, A, Xn, R);  
 *for*(*int* i = 0; i < N; i++){  
 Y[i] = R[i] - b[i];  
 }  
  
 Matrix\_by\_vector(N, A, Y, R);  
  
 chisl\_Tau = 0.0;  
 del\_Tau = 0.0;  
  
 *for*(*int* i = 0; i < N; i++)  
 {  
 chisl\_Tau += R[i]\*Y[i];  
 del\_Tau += R[i]\*R[i];  
 }  
 chisl\_Tau = chisl\_Tau/del\_Tau;  
  
 printf("%lf\n", chisl\_Tau);  
  
 *for*(*int* i = 0; i < N; i++){  
 X[i] = Xn[i] - chisl\_Tau\*Y[i];  
 }  
  
  
 Matrix\_by\_vector(N, A, X, R);  
  
 *double* crit\_1 = 0.0;  
 *double* crit\_2 = 0.0;  
  
 *for*(*int* i = 0; i < N; i++){  
 crit\_1 += pow(R[i] - b[i], 2);  
 crit\_2 += pow(b[i], 2);  
 }  
 crit\_1 = sqrt(crit\_1);  
 crit\_2 = sqrt(crit\_2);  
 crit\_module = crit\_1/crit\_2;  
  
 *for*(*int* i = 0; i < N; i++){  
 Xn[i] = X[i];  
 }  
 }  
 *while* (crit\_module >= eps);  
  
 free(R);  
 free(Y);  
 free(Xn);  
}  
  
*int* main(*int* argc, *char* \*\*argv) {  
 *int* N = 10000;  
  
 *struct* timespec start, end;  
  
 *double* \*\*A;  
 A = (*double*\*\*)malloc(N \**sizeof*(*double*\*));  
 *for*(*int* i = 0; i < N; ++i){  
 A[i] = (*double*\*)malloc(*sizeof*(*double*) \* N);  
 }  
  
 *for*(*int* i = 0; i < N; i++){  
 *for*(*int* j = 0; j < N; j++){  
 *if*(i == j){  
 A[i][j] = 2.0;  
 } *else*{  
 A[i][j] = 1.0;  
 }  
 }  
 }  
  
 *double* \*u = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *for*(*int* i = 0; i < N; i++){  
 u[i] = sin((2\*M\_1\_PI\*i));  
 }  
  
 *double* \*b = (*double*\*)malloc(*sizeof*(*double*) \* N);  
 Matrix\_by\_vector(N, A, u, b);  
  
 *double* \*X = (*double*\*) malloc(*sizeof*(*double*) \* N);  
 *double* epsilon = pow(10, -5);  
  
 clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);  
 Minimal\_Nevazki(N, A, b, X, epsilon);  
 clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);  
  
  
 printf("Time: %lf sec \n", end.tv\_sec - start.tv\_sec + 0.000000001\*(end.tv\_nsec - start.tv\_nsec));  
  
 */\*for(int i = 0; i < N; i++){  
 printf("X[%d] = %lf u[%d] = %lf\n",i, X[i], i, u[i]);  
 }\*/  
  
 for* (*int* i = 0; i < N; ++i)  
 free(A[i]);  
 free(A);  
  
 free(u);  
 free(b);  
 free(X);  
  
 *return* 0;  
}

(MPI Вариант 1: векторы x и b дублируются в каждом MPI-процессе)

#include <cmath>  
#include <cstdio>  
#include <cstdlib>  
#include <mpi.h>  
#include <time.h>  
  
*int* chunk\_size(*int* rank, *int* size, *int* N){  
 *int* basic = N/size;  
 *int* rest = N % size;  
 *return* basic + (rank < rest ? 1 : 0);  
}  
  
*int* get\_offset(*int* rank, *int* size, *int* N){  
 *int* offset = 0;  
 *for*(*int* i = 0; i < rank; i++){  
 offset += chunk\_size(rank, size, N);  
 }  
 *return* offset;  
}  
  
*void* Matrix\_by\_vector(*int* N, *double* \*\*M, *const double* \*V, *double* \*R, *int* rank, *int* size)  
{  
 *int* row\_count = chunk\_size(rank, size, N);  
 *for*(*int* i = 0; i < row\_count; i++){  
 R[i] = 0;  
 *for*(*int* j = 0; j < N; j++)  
 {  
 R[i] += M[i][j]\*V[j];  
 }  
 }  
}  
  
*void* Minimal\_Nevazki(*int* N, *double* \*\*A, *const double* \*b, *double* \*X, *double* eps, *int* rank, *int* size, *int*\* sizes, *int*\* offsets)  
{  
 *int* row\_count = chunk\_size(rank, size, N);  
 *double* \*R\_part = (*double*\*)malloc(*sizeof*(*double*)\*row\_count);  
 *double* \*R = (*double*\*)malloc(*sizeof*(*double*)\*N);  
  
  
 *double* \*Y = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*Xn = (*double*\*)malloc(*sizeof*(*double*)\*N);  
  
 *double* crit\_module;  
 *double* chisl\_Tau;  
 *double* del\_Tau;  
  
 *for*(*int* i = 0; i < N; i++){  
 Xn[i] = 0;  
 }  
  
 *do*{  
 *//Ax* Matrix\_by\_vector(N, A, Xn, R\_part, rank, size);  
 MPI\_Allgatherv(R\_part, sizes[rank], MPI\_DOUBLE, R, sizes, offsets, MPI\_DOUBLE, MPI\_COMM\_WORLD);  
  
 *for*(*int* i = 0; i < N; i++){  
 Y[i] = R[i] - b[i];  
 }  
  
 Matrix\_by\_vector(N, A, Y, R\_part, rank, size);  
 MPI\_Allgatherv(R\_part, sizes[rank], MPI\_DOUBLE, R, sizes, offsets, MPI\_DOUBLE, MPI\_COMM\_WORLD);  
  
 chisl\_Tau = 0.0;  
 del\_Tau = 0.0;  
  
 *for*(*int* i = 0; i < N; i++)  
 {  
 chisl\_Tau += R[i]\*Y[i];  
 del\_Tau += R[i]\*R[i];  
 }  
  
  
 chisl\_Tau = chisl\_Tau/del\_Tau;  
  
 *for*(*int* i = 0; i < N; i++){  
 X[i] = Xn[i] - chisl\_Tau\*Y[i];  
 }  
  
 Matrix\_by\_vector(N, A, X, R\_part, rank, size);  
 MPI\_Allgatherv(R\_part, sizes[rank], MPI\_DOUBLE, R, sizes, offsets, MPI\_DOUBLE, MPI\_COMM\_WORLD);  
  
 *double* crit\_1 = 0.0;  
 *double* crit\_2 = 0.0;  
  
 *for*(*int* i = 0; i < N; i++){  
 crit\_1 += pow(R[i] - b[i], 2);  
 crit\_2 += pow(b[i], 2);  
 }  
  
 crit\_1 = sqrt(crit\_1);  
 crit\_2 = sqrt(crit\_2);  
 crit\_module = crit\_1/crit\_2;  
  
  
 *for*(*int* i = 0; i < N; i++){  
 Xn[i] = X[i];  
 }  
  
  
 }  
 *while* (crit\_module >= eps);  
  
 free(R);  
 free(Y);  
 free(Xn);  
 free(R\_part);  
}  
  
*int* main(*int* argc, *char* \*\*argv) {  
 *int* N = 10000;  
  
 *struct* timespec start, end;  
  
 *int* size, rank;  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
  
 *int*\* sizes = (*int*\*)malloc(*sizeof*(*int*) \* size);  
 *for*(*int* i = 0; i < size; i++){  
 sizes[i] = chunk\_size(i, size, N);  
 }  
  
 *int*\* offsets = (*int*\*)malloc(*sizeof*(*double*) \* size);  
 *for*(*int* i = 0; i < size; i++){  
 offsets[i] = get\_offset(i, size, N);  
 }  
  
 *double* \*\*A;  
 A = (*double*\*\*)malloc(sizes[rank] \* *sizeof*(*double*\*));  
 *for*(*int* i = 0; i < sizes[rank]; ++i){  
 A[i] = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 }  
  
  
 *for*(*int* i = 0; i < sizes[rank]; i++){  
 *for*(*int* j = 0; j < N; j++){  
 *if*(i + offsets[rank] == j){  
 A[i][j] = 2.0;  
 } *else*{  
 A[i][j] = 1.0;  
 }  
 }  
 }  
  
 *double* \*u = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *for*(*int* i = 0; i < N; i++){  
 u[i] = sin((2\*M\_1\_PI\*i));  
 }  
  
 *double* \*b\_part = (*double*\*)malloc(*sizeof*(*double*) \* sizes[rank]);  
 *double* \*b = (*double*\*)malloc(*sizeof*(*double*)\*N);  
  
 Matrix\_by\_vector(N, A, u, b\_part, rank, size);  
 MPI\_Allgatherv(b\_part, sizes[rank], MPI\_DOUBLE, b, sizes, offsets, MPI\_DOUBLE, MPI\_COMM\_WORLD);  
  
 *double* \*X = (*double*\*) malloc(*sizeof*(*double*) \* N);  
 *double* epsilon = pow(10, -5);  
  
 clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);  
 Minimal\_Nevazki(N, A, b, X, epsilon, rank, size, sizes, offsets);  
 clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);  
  
 */\*for(int i = 0; i < N; i++){  
 printf("X[%d] = %lf u[%d] = %lf\n",i, X[i], i, u[i]);  
 }\*/* printf("Time:%lf\n", end.tv\_sec - start.tv\_sec + 0.000000001\*(end.tv\_nsec - start.tv\_nsec));  
  
 *for* (*int* i = 0; i < sizes[rank]; ++i)  
 free(A[i]);  
 free(A);  
  
 free(u);  
 free(b);  
 free(X);  
 free(b\_part);  
 MPI\_Finalize();  
  
 *return* 0;  
}

(MPI Вариант 2: векторы x и b разрезаются между MPI-процессами аналогично матрице A)

#include <cmath>  
#include <cstdio>  
#include <cstdlib>  
#include <mpi.h>  
  
*int* chunk\_size(*int* rank, *int* size, *int* N){  
 *int* basic = N/size;  
 *int* rest = N % size;  
 *return* basic + (rank < rest ? 1 : 0);  
}  
  
*int* get\_offset(*int* rank, *int* size, *int* N){  
 *int* offset = 0;  
 *for*(*int* i = 0; i < rank; i++){  
 offset += chunk\_size(rank, size, N);  
 }  
 *return* offset;  
}  
  
*void* Matrix\_by\_vector(*int* N, *double* \*\*M, *const double* \*V, *double* \*R, *int* rank, *int* size)  
{  
 *for*(*int* i = 0; i < N; i++){  
 R[i] = 0;  
 }  
  
 *int* column\_count = chunk\_size(rank, size, N);  
 *for*(*int* i = 0; i < column\_count; i++){  
 *for*(*int* j = 0; j < N; j++)  
 {  
 R[j] += M[i][j]\*V[i];  
 }  
 }  
}  
  
*void* Minimal\_Nevazki(*int* N, *double* \*\*A, *const double* \*b, *double* \*X, *double* eps, *int* rank, *int* size, *int*\* sizes, *int*\* offsets)  
{  
 *int* column\_count = chunk\_size(rank, size, N);  
 *double* \*R\_part = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*R = (*double*\*)malloc(*sizeof*(*double*)\*column\_count);  
  
  
 *double* \*Y = (*double*\*)malloc(*sizeof*(*double*)\*column\_count);  
 *double* \*Xn = (*double*\*)malloc(*sizeof*(*double*)\*column\_count);  
  
 *double* crit\_module;  
 *double* chisl\_Tau;  
 *double* del\_Tau;  
  
 *for*(*int* i = 0; i < column\_count; i++){  
 Xn[i] = 0;  
 }  
  
 *do*{  
 *//Ax* Matrix\_by\_vector(N, A, Xn, R\_part, rank, size);  
 MPI\_Reduce\_scatter(R\_part, R, sizes, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
 *for*(*int* i = 0; i < column\_count; i++){  
 Y[i] = R[i] - b[i];  
 }  
  
 Matrix\_by\_vector(N, A, Y, R\_part, rank, size);  
 MPI\_Reduce\_scatter(R\_part, R, sizes, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
 *double* chisl\_Tau\_part = 0.0;  
 *double* del\_Tau\_part = 0.0;  
  
 chisl\_Tau = 0.0;  
 del\_Tau = 0.0;  
  
 *for*(*int* i = 0; i < column\_count; i++)  
 {  
 chisl\_Tau\_part += R[i]\*Y[i];  
 del\_Tau\_part += R[i]\*R[i];  
 }  
 MPI\_Allreduce(&chisl\_Tau\_part, &chisl\_Tau, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
 MPI\_Allreduce(&del\_Tau\_part, &del\_Tau, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
  
 chisl\_Tau = chisl\_Tau/del\_Tau;  
 *for*(*int* i = 0; i < column\_count; i++){  
 X[i] = Xn[i] - chisl\_Tau\*Y[i];  
 }  
  
 Matrix\_by\_vector(N, A, X, R\_part, rank, size);  
 MPI\_Reduce\_scatter(R\_part, R, sizes, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
  
 *double* crit\_1 = 0.0;  
 *double* crit\_2 = 0.0;  
  
 *double* crit\_1\_part = 0.0;  
 *double* crit\_2\_part = 0.0;  
  
 *for*(*int* i = 0; i < column\_count; i++){  
 crit\_1\_part += pow(R[i] - b[i], 2);  
 crit\_2\_part += pow(b[i], 2);  
 }  
 MPI\_Allreduce(&crit\_1\_part, &crit\_1, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
 MPI\_Allreduce(&crit\_2\_part, &crit\_2, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
 crit\_1 = sqrt(crit\_1);  
 crit\_2 = sqrt(crit\_2);  
 crit\_module = crit\_1/crit\_2;  
  
  
 *for*(*int* i = 0; i < column\_count; i++){  
 Xn[i] = X[i];  
 }  
  
 }  
 *while* (crit\_module >= eps);  
  
 free(R);  
 free(Y);  
 free(Xn);  
 free(R\_part);  
}  
  
*int* main(*int* argc, *char* \*\*argv) {  
 *int* N = 10000;  
  
 *int* size, rank;  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
  
 *//printf("Size:%d\n", size);  
  
 double* \*\*A;  
 *int* column\_count = chunk\_size(rank, size, N);  
 *//printf("colm:%d\n", column\_count);  
 int* offset = get\_offset(rank, size, N);  
 A = (*double*\*\*)malloc(column\_count \* *sizeof*(*double*\*));  
 *for*(*int* j = 0; j < column\_count; ++j){  
 A[j] = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 }  
  
  
  
 *for*(*int* j = 0; j < column\_count; j++){  
 *for*(*int* i = 0; i < N; i++){  
 *if*(j + offset == i){  
 A[j][i] = 2.0;  
 } *else*{  
 A[j][i] = 1.0;  
 }  
 }  
 }  
  
 *double* \*u = (*double*\*)malloc(*sizeof*(*double*)\*column\_count);  
 *for*(*int* i = 0; i < column\_count; i++){  
 u[i] = sin((2\*M\_1\_PI\*(i + offset)));  
 }  
  
 *double* \*b\_part = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*b = (*double*\*)malloc(*sizeof*(*double*)\*column\_count);  
  
 *int*\* sizes = (*int*\*)malloc(*sizeof*(*int*) \* size);  
 *for*(*int* i = 0; i < size; i++){  
 sizes[i] = chunk\_size(i, size, N);  
 }  
  
 *int*\* offsets = (*int*\*)malloc(*sizeof*(*double*) \* size);  
 *for*(*int* i = 0; i < size; i++){  
 offsets[i] = get\_offset(i, size, N);  
 }  
  
 Matrix\_by\_vector(N, A, u, b\_part, rank, size);  
 MPI\_Reduce\_scatter(b\_part, b, sizes, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);  
  
 *double* \*X = (*double*\*) malloc(*sizeof*(*double*)\*column\_count);  
 *double* epsilon = pow(10, -5);  
  
 *double* start = MPI\_Wtime();  
 Minimal\_Nevazki(N, A, b, X, epsilon, rank, size, sizes, offsets);  
 *double* end = MPI\_Wtime();  
  
 printf("Time:%lf\n", end - start);  
  
 */\*for(int i = 0; i < column\_count; i++){  
 printf("X[%d] = %lf u[%d] = %lf\n",i, X[i], i, u[i]);  
 }\*/  
  
 for* (*int* i = 0; i < column\_count; ++i)  
 free(A[i]);  
 free(A);  
  
 free(u);  
 free(b);  
 free(X);  
 free(b\_part);  
 MPI\_Finalize();  
  
 *return* 0;  
}