Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

 «Пермский национальный исследовательский политехнический университет»

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

Направление 09.03.04 – «Программная инженерия»

Дисциплина: «Технологии блокчейн и распределенные информационные системы»

Профиль: «Разработка программно-информационных систем»

ОТЧЕТ

по лабораторной работе №5

Тема: «Решение СЛАУ методом Гаусса с использованием MPI»

Выполнил: студент группы РИС-20-2б

Катаев М.М.   \_\_\_\_\_\_\_\_\_\_\_

Проверил: старший преподаватель кафедры ИТАС

Щапов В. А. \_\_\_\_\_\_\_\_\_\_\_

Дата \_\_\_\_\_\_

Пермь, 2024

**Цели и задачи**

1. Реализовать программу для решения СЛАУ с использованием MPI

2. Проанализировать результаты

3. Сделать выводы

**Выполнение работы**

MPI - это стандарт на программный инструментарий для обеспечения связи между ветвями параллельного приложения.

Первой была написана функция для вычисления произведения матриц.

Получение общего количества параллельных задач и порядковый номер вызывающей задачи происходит с помощью **MPI\_Comm\_size()** и

**MPI\_Comm\_rank()**.

**MPI\_Bcast** транслирует данные от одного участника группы всем участникам группы, например, матрицу **A**.

**MPI\_Gather** противоположность MPI\_Scatter, собирает с каждой задачи части и формирует финальную матрицу **A** в родительской задаче.

Логика работы программы следующая.

Программа инициализирует MPI, определяет номер текущего процесса и общее количество процессов.

Матрица A заполняется значениями в корневом потоке(0), а затем с помощью MPI\_Bcast рассылает всем процессам данные длинной N\*(N+1) из ранка с номером 0.

При прямом обходе в методе Гаусса было принято решение разделить строки матрицы между процессами на равное количество. Таким образом, необходимо чтобы размер матрицы был кратен количеству процессов. За количество строк на процесс отвечает переменная localRows.

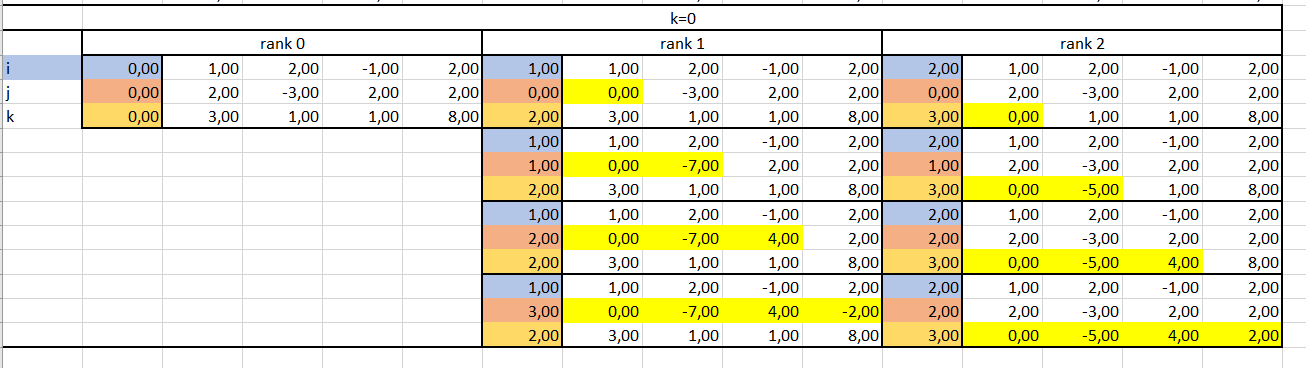
Разделение по строкам продемонстрированно на рисунке ниже. 

Рисунок 1 – Параллельное выполнение прямого хода

Жёлтым цветом выделены данные, которые изменились в каждом процессе, в данном случае было 3 процесса, то есть каждый процесс обрабатывал по одной строке.

После прохождения итерации по k необходимо синхронизировать данные, чтобы каждый процесс получил обновленные данные. Синхронизация реализована с помощью MPI\_Gather, который выполняет сбор данных длинной localRows\*(N+1) с каждого процесса в буфер temp корневого процесса (0). После записи данных в буфер заносим обновленные данные в матрицу A и рассылаем широковещательным сообщением её всем остальным процессам с помощью MPI\_Bcast.

Обратный ход реализован с такой же логикой: каждый процесс изменяет данные по своим строкам, собираются все строки в буфер, значения из которого заносятся в матрицу A корневого процесса, который после этого рассылает эту матрицу всем остальным процессам.

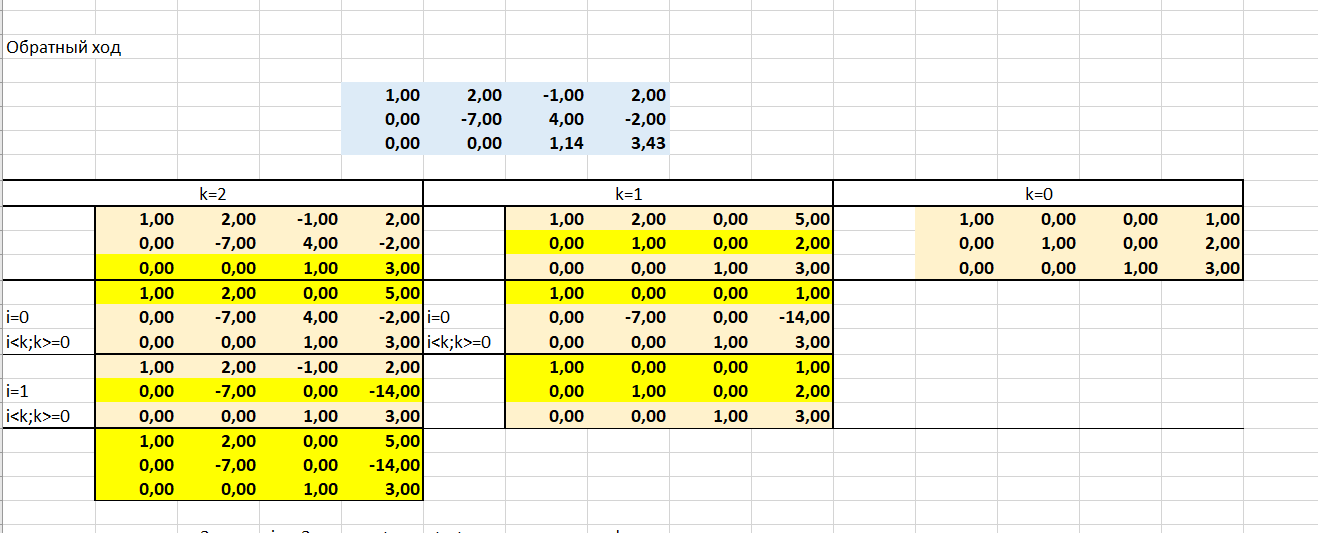


Рисунок 2 – Параллельное выполнение обратного хода

Ниже приведет полный код программы.

Листинг 1 – Код программы

#include <iostream>

#include <mpi.h>

#include <ctime>

#include <cstdlib>

using namespace std;

const int N = 4; // Размерность матрицы

double A[N][N + 1], B[N][N + 1];

void gauss(double A[N][N + 1]) {

// Прямой ход

for (int k = 0; k < N - 1; k++) {

for (int i = k + 1; i < N; i++) {

double koef = A[i][k] / A[k][k];

for (int j = k; j < N + 1; j++) {

A[i][j] -= koef \* A[k][j];

}

}

}

// Обратный ход

for (int k = N - 1; k >= 0; k--) {

A[k][N] /= A[k][k];

A[k][k] = 1.0;

for (int i = 0; i < k; i++) {

A[i][N] -= A[i][k] \* A[k][N];

A[i][k] = 0.0;

}

}

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

cout << "x[" << i << "] = " << A[i][N] << endl;

}

}

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

int rank, size;

int localRows;

MPI\_Init(&argc, &argv);

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

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

localRows = N / size;//Кол-во строк на процесс

if (rank == 0)

{

//double values[N][N + 1] = { {1, 2, -1, 2}, {2, -3, 2, 2}, {3, 1, 1, 8} };

double values[N][N + 1] = { {1, 1, 2, 3, 1}, { 1, 2, 3, -1,-4 }, { 3, -1, -1, -2,-4 }, { 2, 3, -1, -1,-6 } };

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

for (int j = 0; j < N + 1; ++j) {

A[i][j] = values[i][j];//инициализируем матрицу на 0 процессе

B[i][j] = values[i][j];//инициализируем матрицу на 0 процессе

}

}

/\*srand(time(0));

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

for (int j = 0; j < N + 1; ++j) {

A[i][j] = rand() % 10 + 1;

B[i][j] = A[i][j];

}

}\*/

}

MPI\_Bcast(&A[0][0], N \* (N + 1), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Прямой ход

int start = rank \* localRows;

int end = start + localRows;

double temp[N][N + 1];

for (int k = 0; k < N - 1; k++) {

if (end - 1 > k) {

cout << "k " << k << " rank" << rank << endl;

for (int i = start; i < end; i++) {

if (i > k) {

double koef = A[i][k] / A[k][k];

for (int j = k; j < N + 1; j++) {

A[i][j] -= koef \* A[k][j];

}

}

}

}

MPI\_Gather(&A[start][0], localRows \* (N + 1), MPI\_DOUBLE, &temp[start][0], localRows \* (N + 1), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);//собираем изменения во временный буфер

if (rank == 0)

{

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

{

for (int j = 0; j < N + 1; j++)

{

A[i][j] = temp[i][j];//актуализируем данные в матрцице A

}

}

}

MPI\_Bcast(&A[0][0], N \* (N + 1), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);//рассылаем с 0 процесса данные всем остальным процессам с целью актуализации

}

if (rank == 0)

{

cout << endl << " Results after a forward elimination " << endl;

for (int f = 0; f < N; ++f) {

for (int r = 0; r < N + 1; ++r) {

cout << A[f][r] << " ";

}

cout << endl;

}

}

// Обратный ход

for (int k = N - 1; k >= 0; k--) {

if (rank == 0)

{

A[k][N] /= A[k][k];

A[k][k] = 1.0;

}

MPI\_Bcast(&A[k][0], (N + 1), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);//рассылаем с 0 процесса данные всем остальным процессам с целью актуализации

for (int i = start; i < end; i++) {

if (i < k)

{

cout << "kBack " << k << " rankBack " << rank << endl;

A[i][N] -= A[i][k] \* A[k][N];

A[i][k] = 0.0;

}

}

MPI\_Gather(&A[start][0], localRows \* (N + 1), MPI\_DOUBLE, &temp[start][0], localRows \* (N + 1), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);//собираем изменения во временный буфер

if (rank == 0)

{

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

{

for (int j = 0; j < N + 1; j++)

{

A[i][j] = temp[i][j];//актуализируем данные в матрцице A

}

}

}

MPI\_Bcast(&A[0][0], N \* (N + 1), MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);//рассылаем с 0 процесса данные всем остальным процессам с целью актуализации

}

if (rank == 0) {

cout << endl << " Final matrix " << endl;

for (int f = 0; f < N; ++f) {

for (int r = 0; r < N + 1; ++r) {

cout << A[f][r] << " ";

}

cout << endl;

}

}

// Вывод результатов на процессе с рангом 0

if (rank == 0) {

cout << endl;

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

cout << "x[" << i << "] = " << A[i][N] << endl;

}

cout << endl;

}

// Решение слау для проверки результата работы параллельной части

if (rank == 0) {

gauss(B);

}

MPI\_Finalize();

return 0;

}

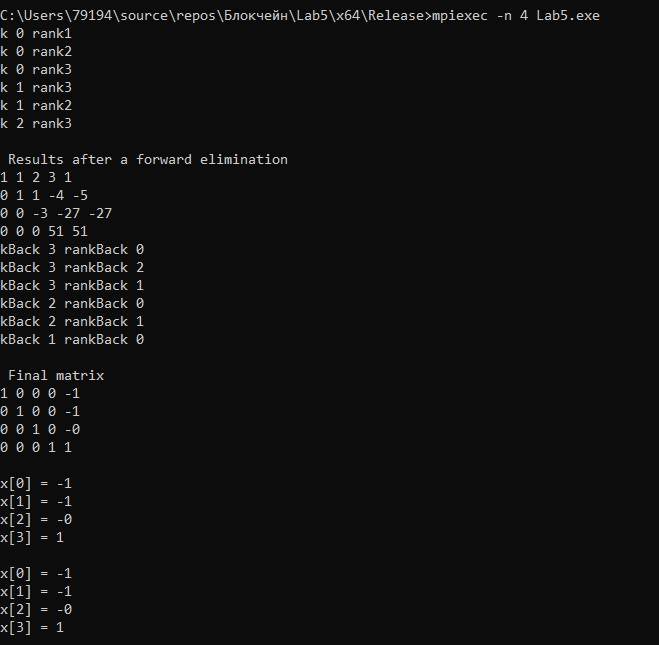
Ниже показаны результаты выполнения программы. 

Рисунок 3 – Результат выполнения программы(N=4, size=4)

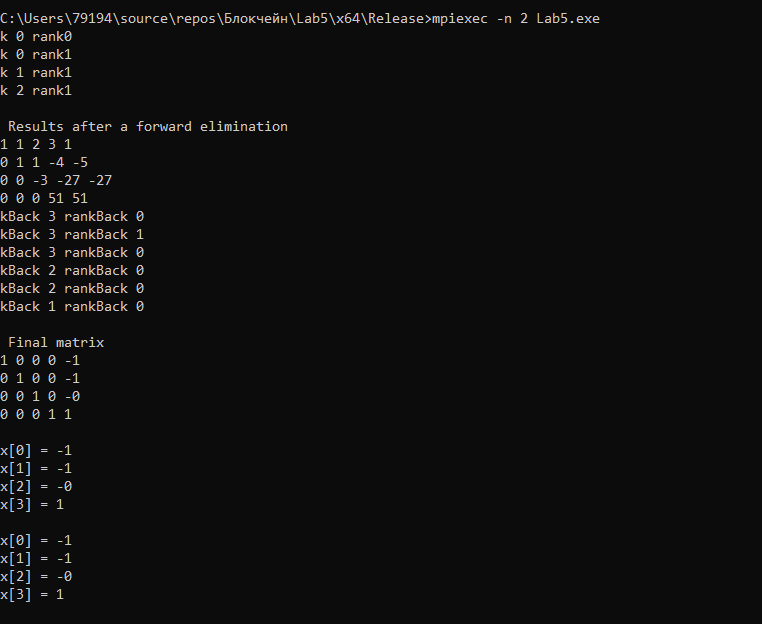


Рисунок 4 – Результат выполнения программы(N=4, size=2)

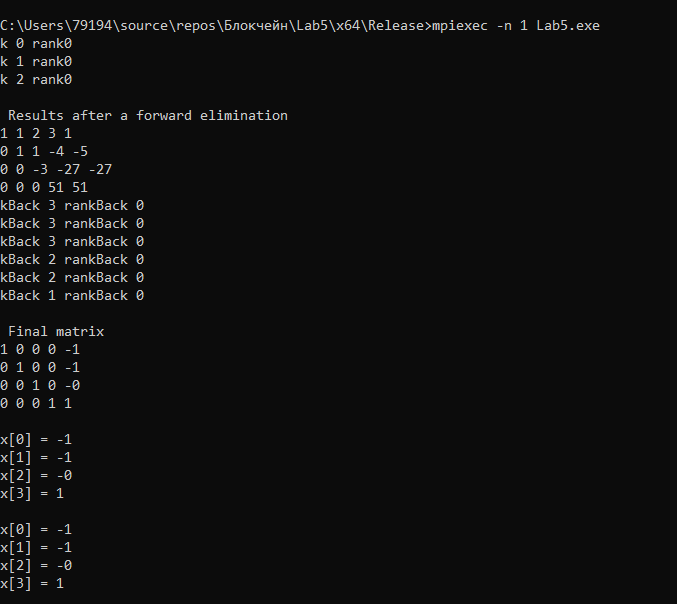


Рисунок 5 – Результат выполнения программы(N=4, size=1)

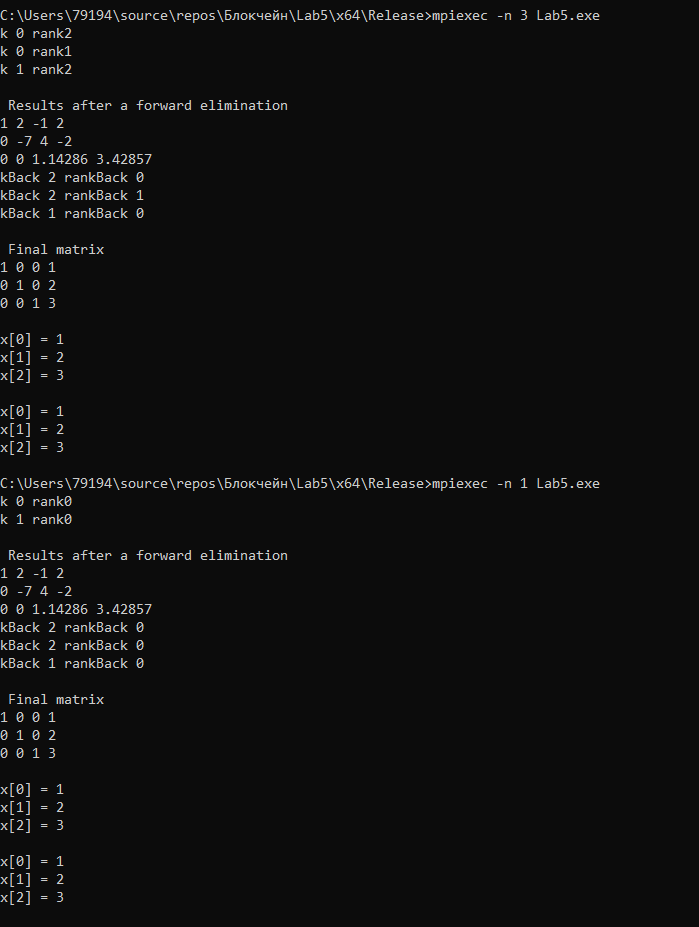


Рисунок 2 – Результат выполнения программы (N=3,size=3 и N=3,size=1)

Также на рисунках видно, что не все процессы производят вычисления во внешнем цикле с k, что говорит нам о том, что параллельность не нарушена.