Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования «Кубанский государственный университет»

Кафедра вычислительных технологий

**ОТЧЕТ**

о выполнении лабораторной работы № 2

по дисциплине «Теория параллельных алгоритмов»

Выполнил:

Корнилов К.А.

Проверил: преподаватель

Нигодин Е. А.

Краснодар

2024

**Тема работы:** Решение систем линейных уравнений методом Якоби.

**Цель:** Распараллеливание процесса решения систем линейных уравнений методом Якоби с использованием технологии OpenMP.

**Задание:**

1. Составить последовательный алгоритм. Построить граф алгоритма. Исследовать на нём зависимости и возможности распараллеливания.
2. Выполнить параллельную реализацию алгоритма
3. Составить таблицу, отражающую сравнительное время выполнения последовательного и параллельных алгоритмов для разных размеров матриц на 2-х и 4-х ядрах. (+ 3 ядра, + график в Excel).

**Ход работы:**

1. Был составлен и написан на языке С++ последовательный алгоритм решения систем линейных уравнений методом Якоби.

Входными данными алгоритма выступают: матрица A размерности ,где , и вектор B размерности , задающие коэффициенты для системы линейных уравнений.

Результатом выступает вектор X размерности , являющийся приближенным решением системы уравнений, заданной матрицами A и B, и полученный путем пошагового приближение решения Xk к действительному решению системы уравнений. Данные приближения выполняются, пока разница между значениями координат векторов Xk и X(k+1) не станут меньше какой-то заданной точности.

Элементы искомого вектора X на k+1 итерации приближения вычисляются следующим образом:

Тогда последовательный алгоритм вычисления k-ой итерации для алгоритма решения систем линейных уравнений методом Якоби схематично изображен на рисунке 1 для частного случая (3 неизвестных переменных):

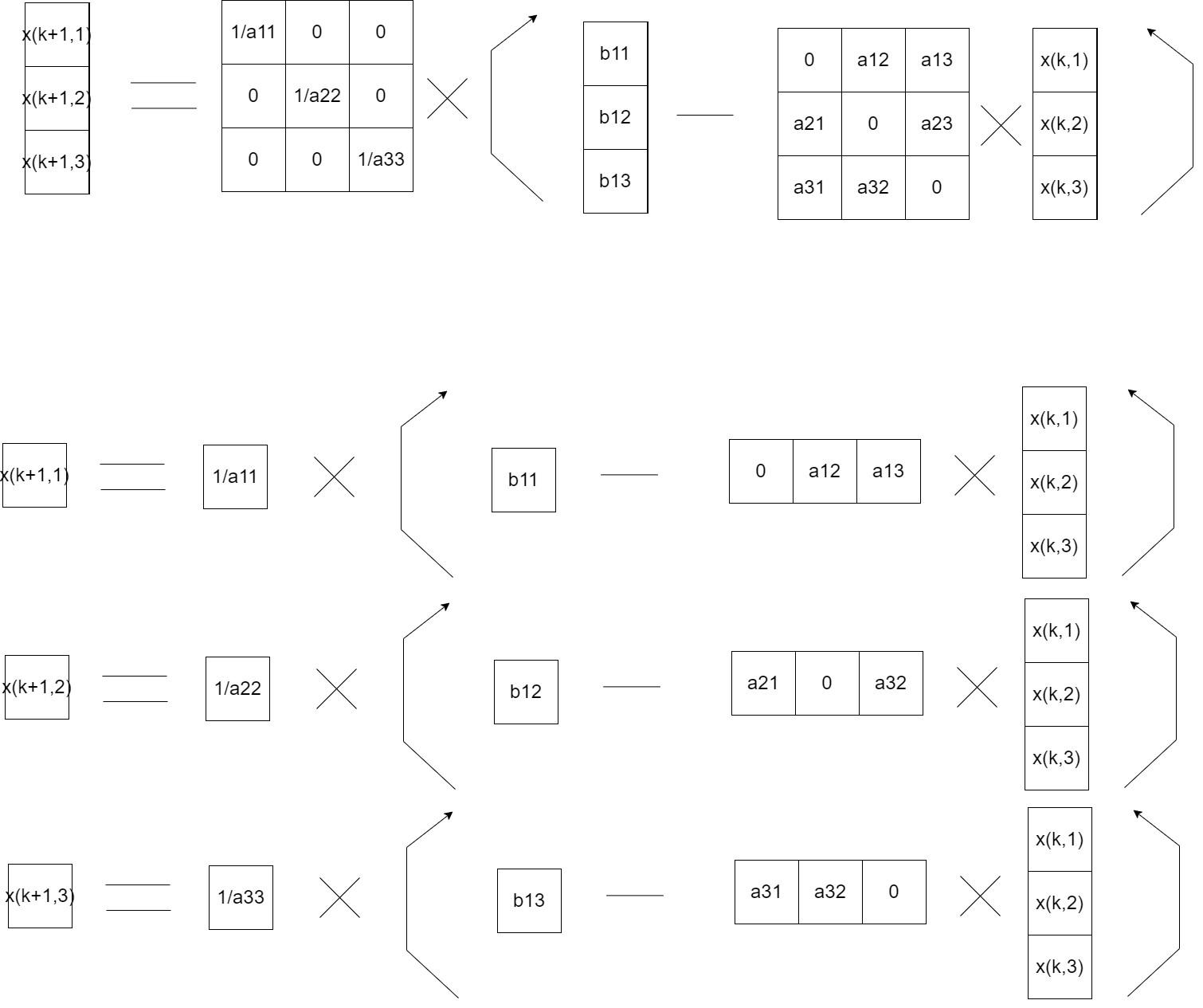


Рисунок 1 – Схема последовательного алгоритма вычисления k-ой итерации для алгоритма решения систем линейных уравнений методом Якоби

Код реализованного алгоритма имеет следующий вид:

double\* Jacobi(int N, double\*\* A, double\* F, double\* G)

{

double\* TempX = new double[N];

double\* X = new double[N];

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

X[i] = G[i];

}

double norm;

double eps = 0.0001;

do {

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

TempX[i] = F[i];

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

if (i != g)

TempX[i] -= A[i][g] \* X[g];

}

TempX[i] /= A[i][i];

}

norm = fabs(X[0] - TempX[0]);

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

if (fabs(X[h] - TempX[h]) > norm)

norm = fabs(X[h] - TempX[h]);

X[h] = TempX[h];

}

} while (norm > eps);

return X;

}

Согласно схеме алгоритма первой возможностью распараллеливания является распараллеливание вычисления значений нового вектора. Каждая строка вычисления координаты на k+1 итерации может быть выполнена не зависимо от других строк. Результаты этих вычислений затем могут быть объединены в один результирующий вектор. Схема алгоритма параллельного алгоритма вычислений k+1 итерации для алгоритма Якоби решения систем линейных уравнений представлена на рисунке 2.

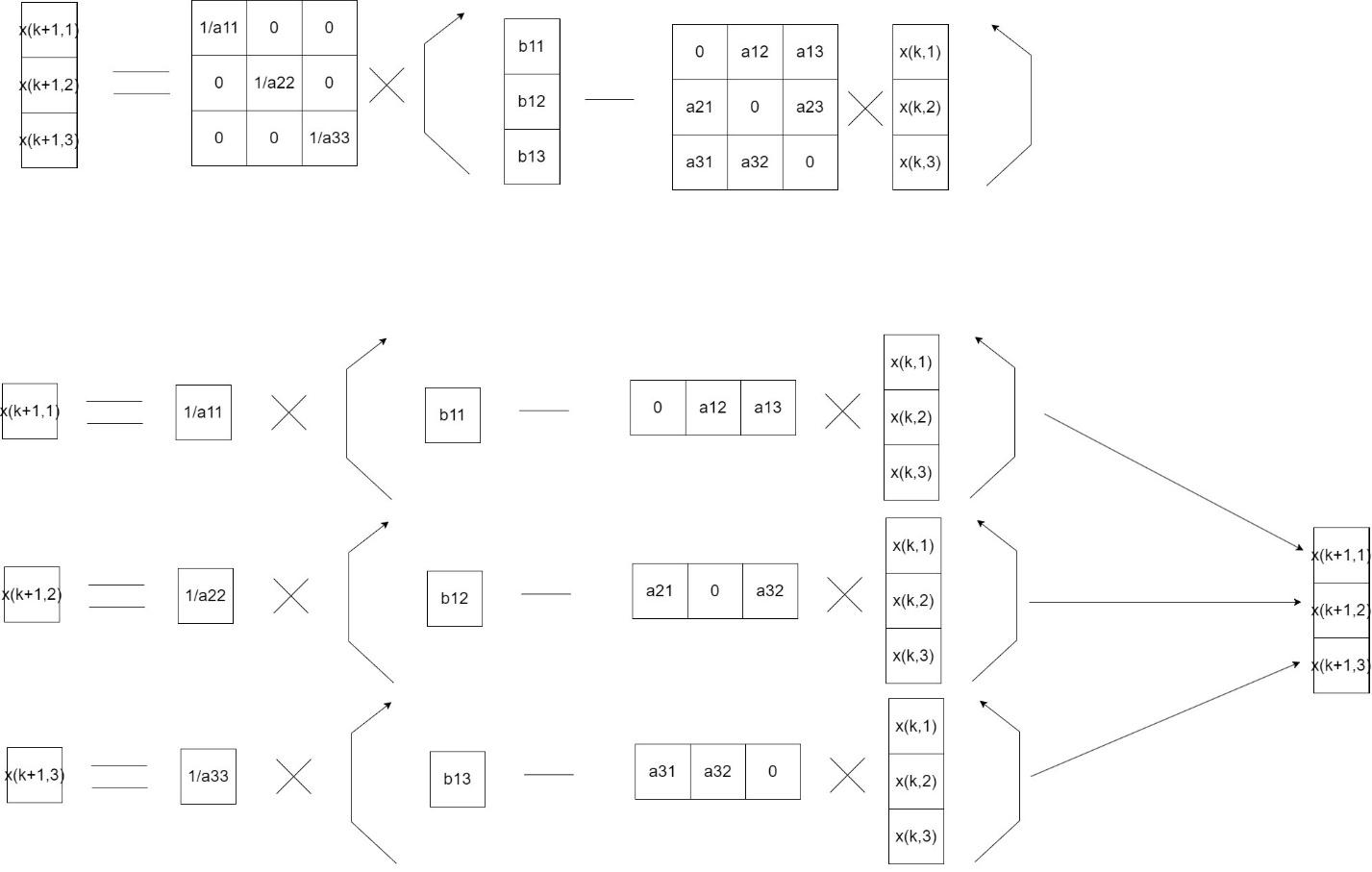


Рисунок 2 – Схема алгоритма параллельного алгоритма вычислений k+1 итерации для алгоритма Якоби решения систем линейных уравнений

Кроме того, можно таким же образом поступить с умножением двух векторов. Данное действие можно распараллелить, а затем использовать результат для формирование конечного значения координаты на данной итерации. Схема второго варианта алгоритма параллельного алгоритма вычислений k+1 итерации для алгоритма Якоби решения систем линейных уравнений представлена на рисунке 3.

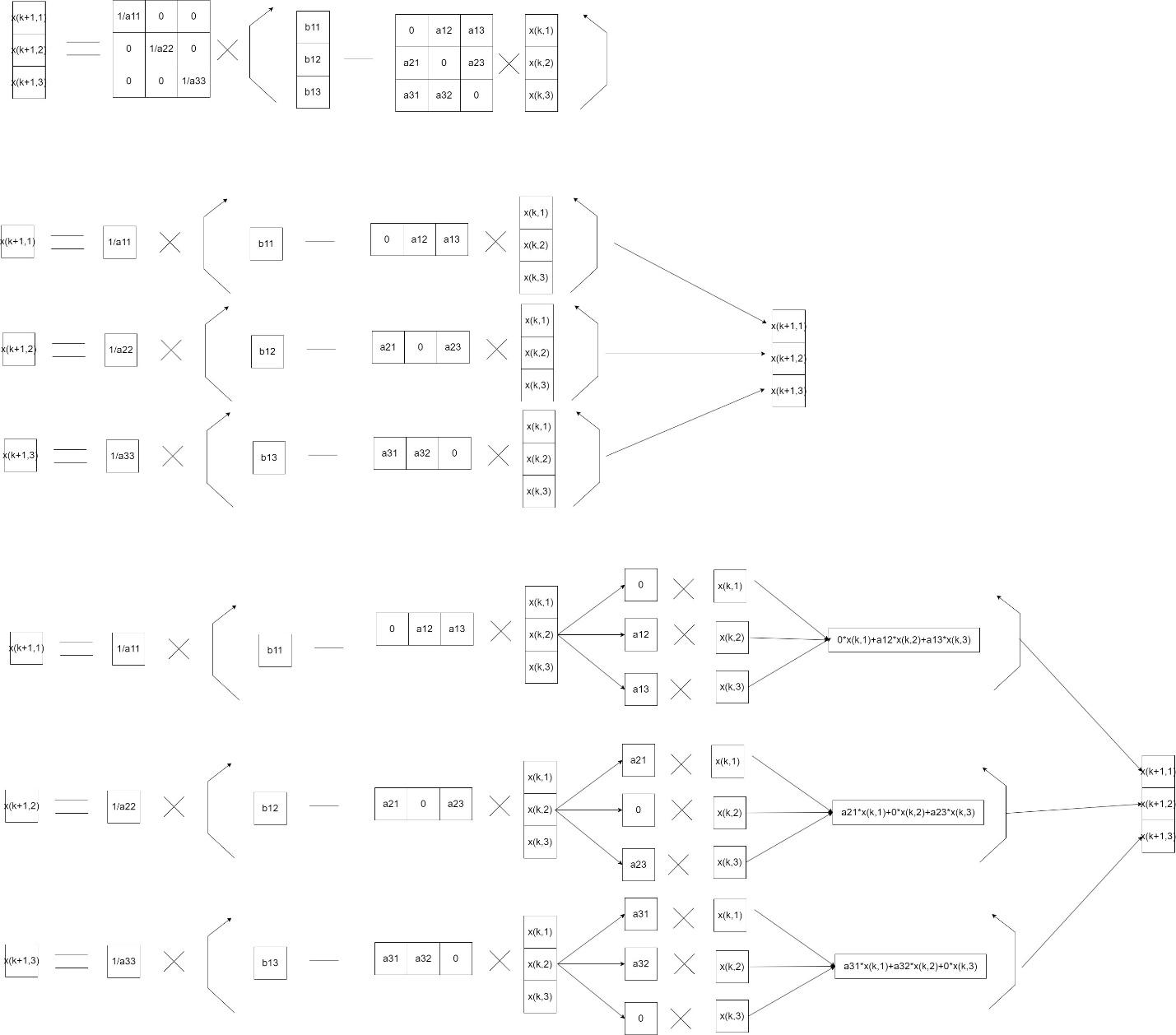


Рисунок 3 – Схема второго варианта параллельного алгоритма вычислений k+1 итерации для алгоритма Якоби решения систем линейных уравнений

1. Была написана параллельная реализация метода Якоби для решения систем линейных уравнений на языке С++ с использованием технологии OpenMP.

Код первого варианта параллельной реализации метода Якоби для решения систем линейных уравнений:

double\* Jacobi\_parallel(int N, double\*\* A, double\* F, double\* G,int n\_threads)

{

double\* TempX = new double[N];

double\* X = new double[N];

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

X[i] = G[i];

}

double norm;

double eps = 0.0001;

do {

#pragma omp parallel for num\_threads(n\_threads)

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

TempX[i] = F[i];

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

if (i != g)

TempX[i] -= A[i][g] \* X[g];

}

TempX[i] /= A[i][i];

}

norm = fabs(X[0] - TempX[0]);

#pragma omp parallel for num\_threads(n\_threads)

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

if (fabs(X[h] - TempX[h]) > norm)

norm = fabs(X[h] - TempX[h]);

X[h] = TempX[h];

}

} while (norm > eps);

return X;

}

Код второго варианта параллельной реализации метода Якоби для решения систем линейных уравнений:

double\* Jacobi\_parallel\_2(int N, double\*\* A, double\* F, double\* X, int n\_threads)

{

double\* TempX = new double[N];

double\* X = new double[N];

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

X[i] = G[i];

}

double norm;

double eps = 0.0001;

do {

#pragma omp parallel for collapse(2) num\_threads(n\_threads)

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

TempX[i] = F[i];

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

if (i != g)

TempX[i] -= A[i][g] \* X[g];

}

TempX[i] /= A[i][i];

}

norm = fabs(X[0] - TempX[0]);

#pragma omp parallel for num\_threads(n\_threads)

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

if (fabs(X[h] - TempX[h]) > norm)

norm = fabs(X[h] - TempX[h]);

X[h] = TempX[h];

}

} while (norm > eps);

return X;

}

1. Используя написанные программы, была проведена оценка времени работы алгоритмов для различных размеров систем и для различного количество ядер. Результаты выполненных оценок представлены в виде графика времени выполнения на рисунке 4.

По оси Ox изображены размеры системы уравнений. По оси Oy изображено время работы алгоритма.

Голубым цветом изображено время выполнения для последовательного алгоритма.

Оранжевым изображено время выполнения для первого параллельного алгоритма на 2 ядрах.

Светло–оранжевым изображено время выполнения для первого параллельного алгоритма на 3 ядрах.

Желтым изображено время выполнения для первого параллельного алгоритма на 4 ядрах.

Зеленым изображено время выполнения для второго параллельного алгоритма на 2 ядрах.

Светло–зеленым изображено время выполнения для второго параллельного алгоритма на 3 ядрах.

Темно–зеленым изображено время выполнения для второго параллельного алгоритма на 4 ядрах.

Рисунок 4 – График зависимости времени работы программы от размера системы и количества ядер

**Вывод:** Было осуществлено распараллеливание процесса вычисления приближенного решения системы линейных уравнений с помощью алгоритма Якоби с использованием технологии OpenMP и была осуществлена оценка времени работы данной программы. Согласно проведенными исследованиям максимальная скорость работы программы была достигнута на 4 потоках.

**ПРИЛОЖЕНИЕ.**

**Приложение 1. Полный код программы**

#include<iostream>

#include<string>

#include<sstream>

#include<omp.h>

using namespace std;

double\*\* input\_matr(int N, int M) {

double\*\* matr = new double\* [N];

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

matr[i] = new double[M];

}

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

string inp = "";

getline(std::cin, inp);

stringstream ss(inp);

int j = 0;

string word;

while (ss >> word) {

matr[i - 1][j] = stod(word);

j = j + 1;

}

}

return matr;

}

void print\_matr(double\*\* matr, int N, int M) {

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

for (int j = 0; j < M; j++) {

cout << matr[i][j] << ' ';

}

cout << endl;

}

}

double\* Jacobi(int N, double\*\* A, double\* F, double\* G)

{

double\* TempX = new double[N];

double\* X = new double[N];

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

X[i] = G[i];

}

double norm;

double eps = 0.0001;

do {

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

TempX[i] = F[i];

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

if (i != g)

TempX[i] -= A[i][g] \* X[g];

}

TempX[i] /= A[i][i];

}

norm = fabs(X[0] - TempX[0]);

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

if (fabs(X[h] - TempX[h]) > norm)

norm = fabs(X[h] - TempX[h]);

X[h] = TempX[h];

}

} while (norm > eps);

return X;

}

double\* Jacobi\_parallel(int N, double\*\* A, double\* F, double\* G,int n\_threads)

{

double\* TempX = new double[N];

double\* X = new double[N];

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

X[i] = G[i];

}

double norm;

double eps = 0.0001;

do {

#pragma omp parallel for num\_threads(n\_threads)

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

TempX[i] = F[i];

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

if (i != g)

TempX[i] -= A[i][g] \* X[g];

}

TempX[i] /= A[i][i];

}

norm = fabs(X[0] - TempX[0]);

#pragma omp parallel for num\_threads(n\_threads)

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

if (fabs(X[h] - TempX[h]) > norm)

norm = fabs(X[h] - TempX[h]);

X[h] = TempX[h];

}

} while (norm > eps);

return X;

}

double\* Jacobi\_parallel\_2(int N, double\*\* A, double\* F, double\* G, int n\_threads)

{

double\* TempX = new double[N];

double\* X = new double[N];

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

X[i] = G[i];

}

double norm;

double eps = 0.0001;

do {

#pragma omp parallel for collapse(2) num\_threads(n\_threads)

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

TempX[i] = F[i];

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

if (i != g)

TempX[i] -= A[i][g] \* X[g];

}

TempX[i] /= A[i][i];

}

norm = fabs(X[0] - TempX[0]);

#pragma omp parallel for num\_threads(n\_threads)

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

if (fabs(X[h] - TempX[h]) > norm)

norm = fabs(X[h] - TempX[h]);

X[h] = TempX[h];

}

} while (norm > eps);

return X;

}

int main()

{

setlocale(LC\_ALL, "russian");

int N1;

cin >> N1;

double\*\* matr1 = input\_matr(N1, N1 + 1);

print\_matr(matr1, N1, N1 + 1);

double\* start = new double[N1];

for (int i = 0; i < N1; i++) {

start[i] = rand() % 100;

}

double\* F = new double[N1];

for (int i = 0; i < N1; i++) {

F[i] = matr1[i][N1];

}

double start\_time = clock();

double\*result=Jacobi(N1,matr1, F,start);

for (int i = 0; i < N1; i++) {

cout << result[i] << " ";

}

double end\_time = clock();

cout << end\_time - start\_time << endl;

start\_time = clock();

result = Jacobi\_parallel(N1, matr1, F, start,2);

end\_time = clock();

cout << end\_time - start\_time << endl;

for (int i = 0; i < N1; i++) {

cout << result[i] << " ";

}

cout << endl;

start\_time = clock();

result = Jacobi\_parallel(N1, matr1, F, start, 3);

end\_time = clock();

cout << end\_time - start\_time << endl;

start\_time = clock();

result = Jacobi\_parallel(N1, matr1, F, start, 4);

end\_time = clock();

cout << end\_time - start\_time << endl;

start\_time = clock();

result = Jacobi\_parallel\_2(N1, matr1, F, start, 2);

end\_time = clock();

cout << end\_time - start\_time << endl;

start\_time = clock();

result = Jacobi\_parallel\_2(N1, matr1, F, start, 3);

end\_time = clock();

cout << end\_time - start\_time << endl;

start\_time = clock();

result = Jacobi\_parallel\_2(N1, matr1, F, start, 4);

end\_time = clock();

cout << end\_time - start\_time << endl;

for (int i = 0; i < N1; i++) {

cout << result[i] << " ";

}

cout << endl;

}