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 №3

Тема: «Решение СЛАУ с использованием OpenMP»

Выполнил: студент группы РИС-20-2б

Катаев М.М.   \_\_\_\_\_\_\_\_\_\_\_

Проверил: старший преподаватель кафедры ИТАС

Щапов В. А. \_\_\_\_\_\_\_\_\_\_\_

Дата \_\_\_\_\_\_

Пермь, 2024

**Цели и задачи**

1. Написать программу для решения СЛАУ с использованием OpenMP

2. Проанализировать результаты

3. Сделать выводы

**Выполнение работы**

OpenMP (Open Multi-Processing) - это набор директив, функций и средств для параллельного программирования на языках C, C++. Он предоставляет простой и удобный способ добавления параллелизма к программам, работающим на многоядерных и многопроцессорных системах.

В данном коде использованы директивы OpenMP для параллельного выполнения алгоритма Гаусса. Это позволяет использовать несколько потоков для одновременного выполнения итераций циклов, что ускоряет вычисления.

Метод Гаусса - это алгоритм для решения систем линейных уравнений и нахождения их корней. Он применяется для матриц, удовлетворяющих некоторым условиям, чтобы привести их к треугольному виду путем последовательного применения элементарных преобразований.

Роль OpenMP в методе Гаусса заключается в том, что он позволяет эффективно распараллеливать основные вычисления, такие как прямой и обратный ход метода. Это особенно полезно для крупных матриц, где вычисления могут занять значительное время.

Ниже представлен код программы.

Листинг 1 – Код программы для вычисления СЛАУ с использованием OpenMP

#include <iostream>

#include <omp.h>

#include <ctime>

#include <cstdlib>

using namespace std;

const int N = 1000; // Размерность матрицы

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

int i, j, k;

double koef;

// Прямой ход

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

#pragma omp parallel for private(j, koef) shared(A, k) schedule(dynamic)

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

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

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

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

}

}

}

// Обратный ход

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

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

A[k][k] = 1.0;

#pragma omp parallel for schedule(dynamic)

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

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

A[i][k] = 0.0;

}

}

for (int i = 0; i < 10; ++i) {

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

}//Вывод первых 10 элементов для сравнения результатов

}

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

int i, j, k;

double koef;

// Прямой ход

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

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

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

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

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

}

}

}

// Обратный ход

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

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

A[k][k] = 1.0;

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

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

A[i][k] = 0.0;

}

}

for (int i = 0; i < 10; ++i) {

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

}//Вывод первых 10 элементов для сравнения результатов

}

int main() {

setlocale(LC\_ALL, "Russian");

double A[N][N + 1];

double B[N][N + 1];

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];

}

}

double start\_sequential = omp\_get\_wtime();

gaussian(A);

double end\_sequential = omp\_get\_wtime();

cout << "Время при последовательном выполнении: " << end\_sequential - start\_sequential << " с." << endl;

double start\_parallel = omp\_get\_wtime();

gaussianOpenMP(B);

double end\_parallel = omp\_get\_wtime();

cout << "Время при распараллеленном выполнении: " << end\_parallel - start\_parallel << " с." << endl;

return 0;

}

Как видно в Листинге 1, мы используем #pragma omp parallel for private(j, koef) shared(A, k) schedule(dynamic), где указывается, что цикл for необходимо распараллелить, а также указаны приватные переменные, для которых каждый процесс создает свою локальную копию. Разделяемая (общая) переменная объявляется с помощью shared() и делает переменные разделяемыми (глобальными).

Schedule определяет, каким образом итерации цикла делятся между потоками группы. Правильность программы не должно зависеть от того, какой поток выполняет конкретную итерацию. Во время динамического распределения не существует предсказуемого порядка назначения итераций цикла нитям. Каждая итерация спрашивает, какие итерации цикла свободны для выполнения и в выполняет их, затем снова спрашивает, затем выполняет и.т.д.

Таким образом мы получили следующие результаты: при параллельном выполнении программы время вычисления корней СЛАУ меньше почти в 5 раз, чем вычисление корней СЛАУ без параллельности. Результаты приведены на рисунках 1 и 2.

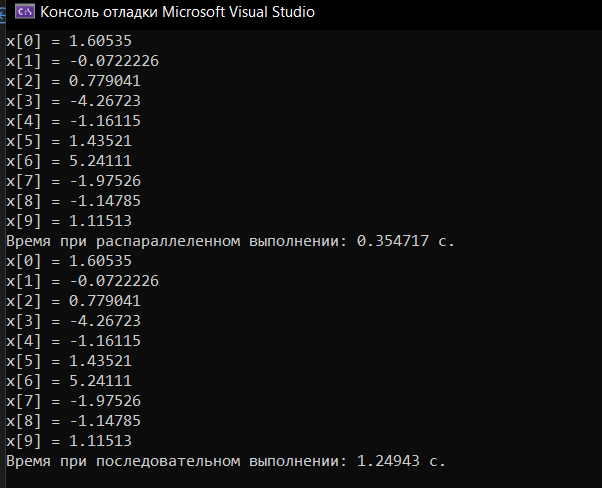


Рисунок 1- результат выполнения

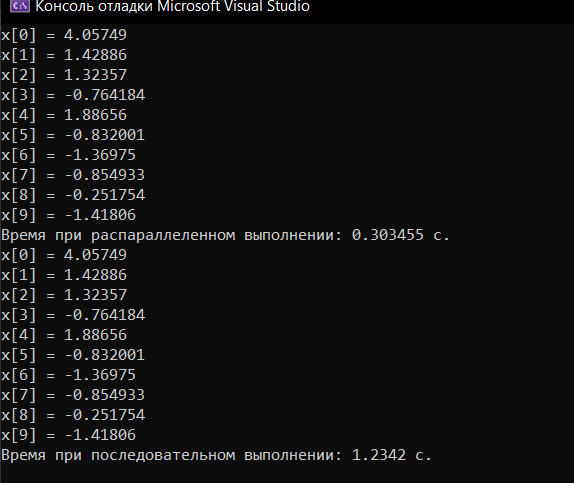


Рисунок 2- результат выполнения