МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ ДНР

ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт КНТ

Факультет ФИСП

Кафедра ПИ им. Л.П. Фельдмана

Лабораторная работа №1

Тема: «Итерационный параллелизм OpenMP»

Курс: ПРСОД

Выполнил

ст. гр. ПИ-18Б

Моргунов А.Г.

Проверил

Щедрин С.В.

Донецк – 2022

Задание

1. Написать функцию, перемножающую матрицы размером n\*n. Cтуденты с четными номерами вариантов разрабатывают алгоритм, параллельный по столбцам, с нечетными – по строкам.

2. Протестировать реализованную функцию из п.1 для размерностей матрицы от 2 до 290. Каждый раз необходимо получить время, за которое выполняется умножение. По полученным показаниям времени построить график T(n) .

3. Написать функцию, в которой умножение матриц распараллеливается.

4. Для п. 3 выполнить действия, описанные в п.2.

5. Сравнить результаты, полученные в п.2 и п.4.

6. Функцию, описанную в п. 4, выполнить с использованием 2, 3, 4 и 5 нитей. При этом для каждого из количеств нитей провести исследование, описанное в п.2. Сравнить результаты.

Функция перемножающая матрицы:

Matrix operator\*(const Matrix& m1, const Matrix& m2) {

Matrix result = CreateSquareMatrix(m1.size());

#pragma omp parallel for

for (int i = 0; i < m1.size(); ++i) {

for (int j = 0; j < m1[i].size(); ++j) {

result[i][j] = getRow(m1, i) \* getColumn(m2, j);

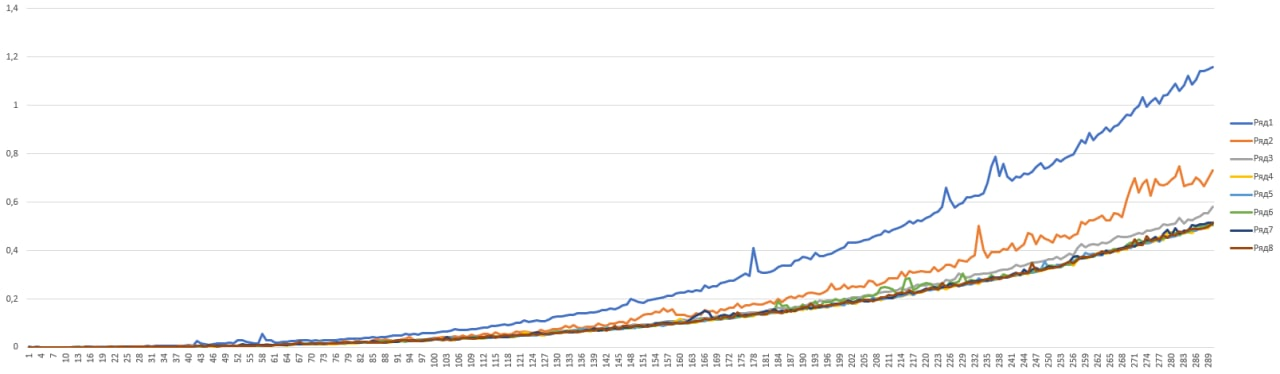
}

}

return result;

}

Результат работы программы с количеством потоков от 1 до 8



Можем сделать вывод, что с 1 до 4 потоков выполнение ускоряется, а начиная с 5 и более потоков ускорение не происходит.

Листинг программы

#include <iostream>

#include <omp.h>

#include <unistd.h>

#include <vector>

#include <random>

#include <fstream>

using Matrix = std::vector<std::vector<double>>;

#include <chrono> // для функций из std::chrono

class Timer

{

private:

// Псевдонимы типов используются для удобного доступа к вложенным типам

using clock\_t = std::chrono::high\_resolution\_clock;

using second\_t = std::chrono::duration<double, std::ratio<1> >;

std::chrono::time\_point<clock\_t> m\_beg;

public:

Timer() : m\_beg(clock\_t::now())

{

}

void reset()

{

m\_beg = clock\_t::now();

}

double elapsed() const

{

return std::chrono::duration\_cast<second\_t>(clock\_t::now() - m\_beg).count();

}

};

void SetSize(Matrix& matrix, int raw\_count, int column\_count) {

matrix.resize(raw\_count);

for (auto& row : matrix) {

row.resize(column\_count);

}

}

Matrix CreateMatrix(int raw\_count, int column\_count) {

Matrix result;

SetSize(result, raw\_count, column\_count);

return result;

}

Matrix CreateSquareMatrix(int raw\_count) {

return CreateMatrix(raw\_count, raw\_count);

}

std::vector<double> getColumn(const Matrix& m, int index) {

std::vector<double> result;

for(auto& row : m) {

result.push\_back(row[index]);

}

return result;

}

std::vector<double> getRow(const Matrix& m, int index) {

return m[index];

}

double operator\*(const std::vector<double>& d1, const std::vector<double>& d2) {

double result = 0;

for (int i = 0; i < d1.size(); ++i) {

result += d1[i] \* d2[i];

}

return result;

}

Matrix operator\*(const Matrix& m1, const Matrix& m2) {

Matrix result = CreateSquareMatrix(m1.size());

#pragma omp parallel for

for (int i = 0; i < m1.size(); ++i) {

for (int j = 0; j < m1[i].size(); ++j) {

result[i][j] = getRow(m1, i) \* getColumn(m2, j);

}

}

return result;

}

std::ostream& operator<<(std::ostream& out, const Matrix& m) {

for (auto& raw : m) {

out << "[ ";

for (auto& element : raw) {

out << element << " ";

}

out << "]" << std::endl;

}

return out;

}

double fRand(double fMin, double fMax)

{

double f = (double)rand() / RAND\_MAX;

return fMin + f \* (fMax - fMin);

}

Matrix GenerateMatrix(int row\_count) {

Matrix result = CreateSquareMatrix(row\_count);

for (auto& row : result)

for (auto& elem : row) {

elem = fRand(0.0, 10.0);

}

return result;

}

void Process(int thread\_count, std::ofstream& file) {

omp\_set\_num\_threads(thread\_count);

for (int i = 1; i < 291; ++i) {

Timer t;

Matrix mat = GenerateMatrix(i);

Matrix mat2 = GenerateMatrix(i);

mat \* mat2;

std::cout << "Iteration[" << i << ", " << thread\_count << "] : " << t.elapsed() << std::endl;

file << t.elapsed() << ",";

}

file << std::endl;

}

int main()

{

std::ofstream file("parallel.csv", std::fstream::out);

srand(0);

for (int i = 1; i < 9; ++i) {

Process(i, file);

}

file.close();

return 0;

}