**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Федеральное ГОСУДАРСТВЕННОЕ бюджетное ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информатики и вычислительной техники

Кафедра ИиСП

**Курсовая работа**

**по дисциплине «Перспективное программное обеспечение»**

Выполнил:

студент группы ПС-31

факультета Информатики

и Вычислительной Техники

специальности «Программная инженерия»

Илдаркин А.

Проверил:

Филимонов А.

г. Йошкар-Ола

2016

# Постановка задачи

Найти алгебраическое дополнение для каждого элемента матрицы.

Входные данные: произвольная матрица А размерности n х n. Заполнение начальной матрицы реализовать в главном потоке приложения.

1. Реализовать последовательный вариант программы для указанного варианта.
2. Реализовать параллельный вариант программы. Количество потоков выполнения должно являться входным параметром задачи.
3. После завершения программа должна выдавать время своей работы. Подобрать размеры матриц таким образом, чтобы время работы последовательного варианта составляло не менее одной секунды.
4. Посчитать параметры качества вашей параллельной программы и построить в Excel графики для количества потоков = [1;16] и количестве задействованных ядер = [1;4] (используйте функцию SetAffinityMask).
5. Время выполнения
6. Ускорение
7. Эффективность распараллеливания.

# Листинг кода

main.cpp:

#include <chrono>

#include <iostream>

#include "MultithreadedCalculatorHelper.h"

#include <memory>

using namespace std;

void PrintMatrix(const Matrix & matrix)

{

for (size\_t i = 0; i != MATRIX.size(); ++i)

{

for (size\_t j = 0; j != MATRIX[0].size(); ++j)

{

cout << matrix[i][j] << " ";

}

cout << endl;

}

}

ThreadsChargeMap FillThreadsChargeMap(size\_t threadsCount)

{

map<size\_t, vector<size\_t>> threadsChargeMap;

for (size\_t i = 1; i <= threadsCount; i++)

{

threadsChargeMap.emplace(i, vector<size\_t>());

}

for (size\_t i = 0, threadNumber = 1; i != MATRIX.size(); i++, threadNumber++)

{

if (threadNumber > threadsCount)

{

threadNumber = 1;

}

threadsChargeMap.at(threadNumber).push\_back(i);

}

return threadsChargeMap;

}

int main(int argc, char \* argv[]) //TODO: выделить класс MatrixUtils; Ликвидировать из main() лишний код, связанный с созданием потков

{

const size\_t threadsCount = argc == 1 ? 1 : atoi(argv[1]);

CMultithreadedMatrixCalculator calc(FillThreadsChargeMap(threadsCount));

auto startTime = chrono::steady\_clock::now();

HANDLE \*hThreads = \*make\_shared<HANDLE\*>(new HANDLE[threadsCount]);

DWORD \*dwThreadsId = \*make\_shared<DWORD\*>(new DWORD[threadsCount]);

for (size\_t i = 1; i < threadsCount; i++)

{

SMultithreadedCalculatorHelper helper(i, calc);

hThreads[i - 1] = CreateThread(NULL, 0, StartMultithreadedCalculator, (PVOID)&helper, 0, &dwThreadsId[i - 1]);

}

calc.CalculateLine((PVOID)&threadsCount);

WaitForMultipleObjects(threadsCount - 1, hThreads, TRUE, INFINITE);

auto endTime = chrono::steady\_clock::now();

chrono::duration<double> diff = endTime - startTime;

cout << diff.count() << endl;

return 0;

}

# MultithreadedCalculatorHelper.h:

#include "matrix\_res.h"

#include <stdio.h>

#include <conio.h>

#include <windows.h>

class CMultithreadedMatrixCalculator

{

public:

CMultithreadedMatrixCalculator(const ThreadsChargeMap & threadsChargeMap);

const Matrix & GetOutputMatrix()const;

DWORD WINAPI CalculateLine(PVOID pvParam);

private:

void CalculateAlgebraicAddition(size\_t i, size\_t j);

private:

ThreadsChargeMap m\_threadsChargeMap;

Matrix m\_outputMatrix;

};

MultiThreadedMatrixCalculator.h:

#pragma once

#include "MultithreadedMatrixCalculator.h"

struct SMultithreadedCalculatorHelper

{

SMultithreadedCalculatorHelper(int threadNumber, CMultithreadedMatrixCalculator & matrixCalc)

: threadNumber(threadNumber)

, matrixCalc(matrixCalc)

{};

int threadNumber;

CMultithreadedMatrixCalculator & matrixCalc;

};

DWORD WINAPI StartMultithreadedCalculator(PVOID pvParam)

{

auto helper = static\_cast<SMultithreadedCalculatorHelper\*>(pvParam);

return helper->matrixCalc.CalculateLine((PVOID)&helper->threadNumber);

# }

MultiThreadedMatrixCalculator.cpp:

#include "MultithreadedMatrixCalculator.h"

#include <boost/optional.hpp>

namespace

{

void GetMatrixWithoutLineAndColumn(const Matrix & mas, Matrix & p, int i, int j, int m) {

int ki, kj, di, dj;

di = 0;

for (ki = 0; ki < m - 1; ki++)

{

if (ki == i)

{

di = 1;

}

dj = 0;

for (kj = 0; kj < m - 1; kj++)

{

if (kj == j) dj = 1;

p[ki][kj] = mas[ki + di][kj + dj];

}

}

}

double CalcDetByGausMethod(Matrix & matrix)

{

size\_t i, j, k;

double det = 1;

double b = 0;

for (i = 0; i < matrix.size(); i++)

{

for (j = i + 1; j < matrix[0].size(); j++)

{

if (matrix[i][i] == 0)

{

if (matrix[i][j] == 0)

{

b = 0;

}

else

{

return 0;

}

}

else

{

b = matrix[j][i] / matrix[i][i];

}

for (k = i; k < matrix.size(); k++)

{

matrix[j][k] = matrix[j][k] - matrix[i][k] \* b;

}

}

det \*= matrix[i][i];

}

return det;

}

double CalcMinor(size\_t iPos, size\_t jPos)

{

Matrix matrixWithoutIandJ(MATRIX.size() - 1, std::vector<double>(MATRIX.size() - 1));

GetMatrixWithoutLineAndColumn(MATRIX, matrixWithoutIandJ, iPos, jPos, MATRIX.size());

auto det = CalcDetByGausMethod(matrixWithoutIandJ);

return static\_cast<double>(std::pow(-1, iPos + jPos + 2)) \* (det);

}

}

CMultithreadedMatrixCalculator::CMultithreadedMatrixCalculator(const ThreadsChargeMap & threadsChargeMap)

: m\_threadsChargeMap(threadsChargeMap)

, m\_outputMatrix(Matrix(MATRIX.size(), std::vector<double>(MATRIX.size(), 0)))

{};

const Matrix & CMultithreadedMatrixCalculator::GetOutputMatrix() const

{

return m\_outputMatrix;

}

DWORD CMultithreadedMatrixCalculator::CalculateLine(PVOID pvParam)

{

auto threadNumber = \*static\_cast<int\*>(pvParam);

auto linesNumbers = m\_threadsChargeMap.at(threadNumber);

for (auto it = linesNumbers.begin(); it != linesNumbers.end(); ++it)

{

for (size\_t j = 0; j != MATRIX[0].size(); ++j)

{

CalculateAlgebraicAddition(\*it, j);

}

}

DWORD dwResult = 0;

return dwResult;

}

void CMultithreadedMatrixCalculator::CalculateAlgebraicAddition(size\_t i, size\_t j)

{

m\_outputMatrix[i][j] = CalcMinor(i, j);

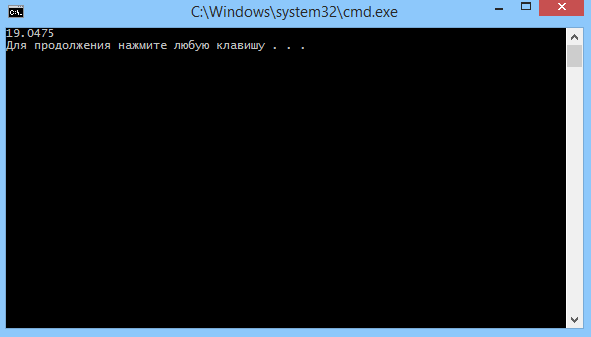
}

# 

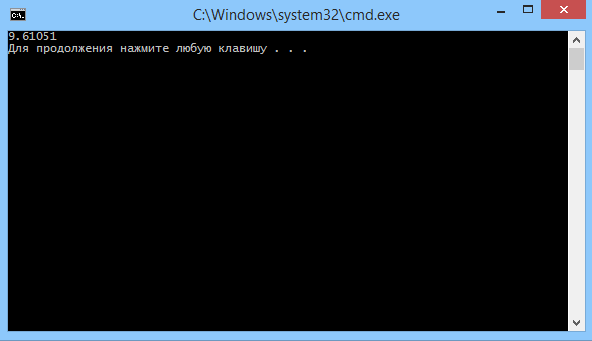
# Вывод

Прежде чем сделать выводы, хотелось бы приложить скриншоты с временем работы программы, выполняющей операции над матрицей, размером 100x100 в режиме Release.

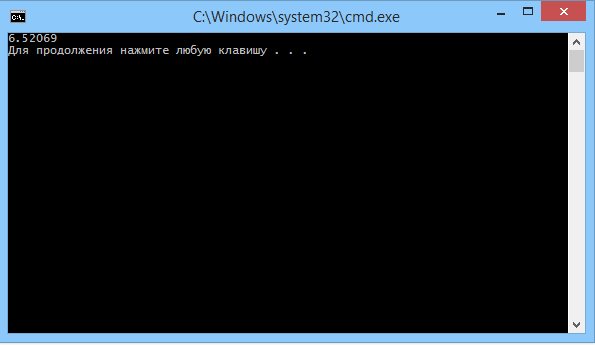
Последовательно:



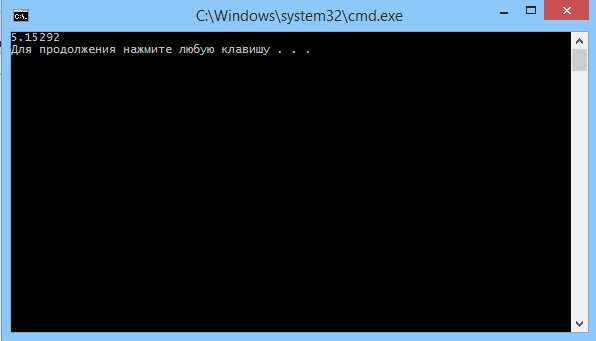
Параллельно (2 потока):



Параллельно (3 потока):



Параллельно (4 потока):



Таким образом, исходя из полученных данных, можно сделать вывод о том, что при параллельном исполнении программы, время выполнения уменьшается.

(после того, как подрефакторю код и запилю графики, сделаю более подробный вывод).