# Параллельное и распределённое программирование. Лабораторная работа №1. Вилков Е. С. ИВТ-21М. 2024

Задание к лабораторной работе: «Перемножение матриц»

В работе необходимо составить и проверить алгоритм перемножения матриц. Для проверки алгоритма целесообразно использовать небольшие матрицы (до 3х3 включительно), результат перемножения вывести на экран. После успешной проверки задаются большие (1000х1000) матрицы как входные данные, наполнение матриц случайно и необязательно к выводу на экран. На скриншотах выполнения программы должна быть видна разница во времени исполнения в последовательном и в параллельном режимах.

Код разделён на несколько исходных файлов, разделяющих методы выполнения на методики, указанные в задании (сырые потоки, OpenMP). Вместо сырых POSIX потоков, была взята ответственность, и лабораторная работа была выполнена с использованием std::thread. Так как выполнение производится на машине под управлением ОС Microsoft Windows и стандартный компилятор Microsoft (MSVC) сложно «подружить» с библиотекой «pthread.h». Общий алгоритм тем не менее не был бы изменён, и в лабораторной работе показано умение работать как с библиотеками, так и с «сырыми» интерфейсами многопоточного выполнения.

Файл main.cpp:

#include <iomanip>

#include <iostream>

#include <chrono>

#include <thread>

#include <omp.h>

#include "matrix.h"

#include "thread\_matrix.h"

#include "omp\_matrix.h"

using namespace std;

using namespace std::chrono;

int main()

{

srand(time(NULL));

bool printResults = true;

Matrix m1 = generate\_matrix({ 1500, 1300 });

Matrix m2 = generate\_matrix({ 1300, 1500 });

//Matrix m1 = generate\_matrix({ 2, 3 });

//Matrix m2 = generate\_matrix({ 3, 5 });

if (m1.size.m > 5 || m1.size.n > 5)

printResults = false;

CacheOptimizedMatrix c2 = make\_cache\_matrix(m2);

CacheOptimizedMatrix c1 = make\_cache\_matrix(m1);

auto startSingleThread = high\_resolution\_clock::now();

Matrix m3 = multiply\_matrix\_st(c1, c2);

auto endSingleThread = high\_resolution\_clock::now();

auto startMultiThreaded = high\_resolution\_clock::now();

Matrix m4 = multiply\_matrix\_mt(c1, c2);

auto endMultiThreaded = high\_resolution\_clock::now();

auto startOpenMP = high\_resolution\_clock::now();

Matrix m5 = multiply\_matrix\_omp(c1, c2);

auto endOpenMP = high\_resolution\_clock::now();

if (printResults)

{

print\_matrix(m1);

cout << "\*" << endl;

print\_matrix(m2);

cout << "=" << endl;

print\_matrix(m3);

cout << "=" << endl;

print\_matrix(m4);

cout << "=" << endl;

print\_matrix(m5);

}

auto durationSt = duration\_cast<microseconds>(endSingleThread - startSingleThread);

auto durationMt = duration\_cast<microseconds>(endMultiThreaded - startMultiThreaded);

auto durationMp = duration\_cast<microseconds>(endOpenMP - startOpenMP);

cout << "Single-threaded duration: " << durationSt.count() / 1000.0 << "ms" << endl;

cout << "Milti-threaded duration : " << durationMt.count() / 1000.0 << "ms (x" << fixed << setprecision(2) << ((double)durationSt.count() / durationMt.count()) << " speed up)" << endl;

cout << "OpenMP-threaded duration: " << durationMp.count() / 1000.0 << "ms (x" << fixed << setprecision(2) << ((double)durationSt.count() / durationMp.count()) << " speed up)" << endl;

}

Файл matrix.h:

#pragma once

typedef struct MatrixDimension {

int m; // vertical size aka row count

int n; // horizontal size aka column count

};

typedef struct Matrix {

double\* container;

MatrixDimension size;

};

typedef struct CacheOptimizedMatrix {

Matrix rowMajorMatrix;

Matrix colMajorMatrix;

};

Matrix generate\_matrix(MatrixDimension);

CacheOptimizedMatrix make\_cache\_matrix(const Matrix&);

double get\_matrix\_item(const Matrix&, int, int);

void set\_matrix\_item(const Matrix&, int, int, double);

void print\_matrix(const Matrix&);

Matrix multiply\_matrix\_st(const CacheOptimizedMatrix&, const CacheOptimizedMatrix&);

Файл matrix.cpp:

#include "matrix.h"

#include <iomanip>

#include <iostream>

// Умножение матриц в одном потоке

Matrix multiply\_matrix\_st(const CacheOptimizedMatrix& m1, const CacheOptimizedMatrix& m2)

{

if (m1.rowMajorMatrix.size.n != m2.rowMajorMatrix.size.m)

{

std::cout << "Matrices are not compatable to multiplication!" << std::endl;

return Matrix{};

}

MatrixDimension resultDim = { m1.rowMajorMatrix.size.m, m2.rowMajorMatrix.size.n };

double\* resultMatrixContainter = new double[resultDim.m \* resultDim.n];

for (int i = 0; i < resultDim.m; i++)

{

for (int j = 0; j < resultDim.n; j++)

{

double sum = 0.0;

for (int k = 0; k < m1.rowMajorMatrix.size.n; k++)

{

sum += m1.rowMajorMatrix.container[i \* m1.rowMajorMatrix.size.n + k] \* m2.colMajorMatrix.container[j \* m1.rowMajorMatrix.size.n + k];

}

resultMatrixContainter[i \* resultDim.n + j] = sum;

}

}

return { resultMatrixContainter, resultDim };

}

Matrix generate\_matrix(MatrixDimension dim)

{

int size = dim.m \* dim.n;

double\* container = new double[size];

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

{

container[i] = rand() / (double)RAND\_MAX;

}

return { container, dim };

}

void print\_matrix(const Matrix& matrix)

{

std::cout << "{" << std::endl;

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

{

std::cout << " { ";

for (int j = 0; j < matrix.size.n; j++)

{

double item = get\_matrix\_item(matrix, i, j);

std::cout << std::fixed << std::setprecision(4) << item;

if (j != matrix.size.n - 1)

std::cout << ", ";

}

if (i != matrix.size.m - 1)

std::cout << " },\n";

else

std::cout << " }\n";

}

std::cout << "}" << std::endl;

}

double get\_matrix\_item(const Matrix& m, int m\_index, int n\_index)

{

if (m\_index > m.size.m - 1)

return 0;

if (n\_index > m.size.n - 1)

return 0;

return m.container[m\_index \* m.size.n + n\_index];

}

void set\_matrix\_item(const Matrix& m, int m\_index, int n\_index, double item)

{

if (m\_index > m.size.m - 1)

return;

if (n\_index > m.size.n - 1)

return;

m.container[m\_index \* m.size.n + n\_index] = item;

}

CacheOptimizedMatrix make\_cache\_matrix(const Matrix& matrix)

{

double\* newContainer = new double[matrix.size.n \* matrix.size.m];

Matrix colMajorMatrix{ newContainer, {matrix.size.n, matrix.size.m} };

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

{

for (int j = 0; j < matrix.size.n; j++)

{

double item = get\_matrix\_item(matrix, i, j);

set\_matrix\_item(colMajorMatrix, j, i, item);

}

}

return { matrix, colMajorMatrix };

}

Файл omp\_matrix.h:

#pragma once

#include "matrix.h"

Matrix multiply\_matrix\_omp(const CacheOptimizedMatrix&, const CacheOptimizedMatrix&);

Файл omp\_matrix.cpp:

#include "omp\_matrix.h"

#include <iostream>

#include <omp.h>

// Умножение матриц в нескольких потоках с использованием OpenMP (параллельно)

double calculate\_sum\_task(double\* m1RowArr, double\* m2ColArr, int n, int i, int j)

{

double sum = 0.0;

for (int k = 0; k < n; k++)

{

sum += m1RowArr[i \* n + k] \* m2ColArr[j \* n + k];

}

return sum;

}

Matrix multiply\_matrix\_omp(const CacheOptimizedMatrix& m1, const CacheOptimizedMatrix& m2)

{

if (m1.rowMajorMatrix.size.n != m2.rowMajorMatrix.size.m)

{

std::cout << "Matrices are not compatable to multiplication!" << std::endl;

return Matrix{};

}

MatrixDimension resultDim = { m1.rowMajorMatrix.size.m, m2.rowMajorMatrix.size.n };

double\* resultMatrixContainter = new double[resultDim.m \* resultDim.n];

for (int i = 0; i < resultDim.m; i++)

{

#pragma omp parallel for

for (int j = 0; j < resultDim.n; j++)

{

double sum = calculate\_sum\_task(

m1.rowMajorMatrix.container,

m2.colMajorMatrix.container,

m1.rowMajorMatrix.size.n,

i,

j);

resultMatrixContainter[i \* resultDim.n + j] = sum;

}

}

return { resultMatrixContainter, resultDim };

}

Файл thread\_matrix.h:

#pragma once

#include "matrix.h"

Matrix multiply\_matrix\_mt(const CacheOptimizedMatrix&, const CacheOptimizedMatrix&);

Файл thread\_matrix.cpp:

#include "thread\_matrix.h"

#include <iostream>

#include <thread>

// Умножение матриц в нескольких потоках с использованием стандартных потоков (параллельно)

void calculate\_part\_thread(double\* m1RowArr, double\* m2ColArr, double\* rRowArr, int n\_start, int n\_end, int m, int p)

{

for (int i = n\_start; i < n\_end; i++)

{

for (int j = 0; j < p; j++)

{

double sum = 0.0;

for (int k = 0; k < m; k++)

{

sum += m1RowArr[i \* m + k] \* m2ColArr[j \* m + k];

}

rRowArr[i \* p + j] = sum;

}

}

}

Matrix multiply\_matrix\_mt(const CacheOptimizedMatrix& m1, const CacheOptimizedMatrix& m2)

{

if (m1.rowMajorMatrix.size.n != m2.rowMajorMatrix.size.m)

{

std::cout << "Matrices are not compatable to multiplication!" << std::endl;

return Matrix{};

}

MatrixDimension resultDim = { m1.rowMajorMatrix.size.m, m2.rowMajorMatrix.size.n };

int overallSize = resultDim.m \* resultDim.n;

double\* resultMatrixContainter = new double[overallSize];

auto processor\_count = std::thread::hardware\_concurrency();

// if hardware\_concurrency return 0 (unable to detect)

// then set the processor count to the size of resulting data (1 row per thread)

// and let the sheduler do the hard work, switching the context

if (processor\_count == 0)

processor\_count = resultDim.m;

// if hardware\_concurrency is greater than data size

// also set the processor count to the size of resulting data (1 row per thread)

if (processor\_count > resultDim.m)

processor\_count = resultDim.m;

double rowsPerThread = (double)resultDim.m / processor\_count; // -- 1000 / 16 = 62.5; avg rows per thread

int minimalRowsPerThread = rowsPerThread; // floor the 62.5 to 62, so every single thread will calculate at least 62 rows

int remainderRows = resultDim.m - (minimalRowsPerThread \* processor\_count); // 1000 - (62 \* 16) = 8, how many rows are left

// make the first 8 (remainder) thread handle 63 rows then

std::thread\* threads = new std::thread[processor\_count];

for (int i = 0, rowStartCounter = 0; i < processor\_count; i++)

{

int rowCount = minimalRowsPerThread;

if (i < remainderRows)

rowCount += 1;

threads[i] = std::thread(calculate\_part\_thread,

m1.rowMajorMatrix.container,

m2.colMajorMatrix.container,

resultMatrixContainter,

rowStartCounter,

rowStartCounter + rowCount,

m1.rowMajorMatrix.size.n,

m2.rowMajorMatrix.size.n);

rowStartCounter += rowCount;

}

// wait for threads

for (int i = 0; i < processor\_count; i++)

{

threads[i].join();

}

return { resultMatrixContainter, resultDim };

}

В случае OpenMP был применён подход «распараллеливания» внутреннего цикла, что приводит к дополнительным вычислениям процессора на переключение контекста выполнения, это будет заметно при выполнении программы на больших объёмах данных.

В случае использования сырых потоков, я помимо простого «разбития» данных на независимые участки, учёл крайние случаи, чтобы потоки не отличались сильно по объёму обрабатываемых данных.

Проверка верности алгоритма:

Для проверки того, правильно ли программа рассчитала перемножение матриц, была использована помощь сервиса WolframAlpha:

Лог выполнения программы:

{

{ 0.7408, 0.7165, 0.0567 },

{ 0.9426, 0.9397, 0.3608 }

}

\*

{

{ 0.2311, 0.6069, 0.8548, 0.4793, 0.5205 },

{ 0.0954, 0.4462, 0.1531, 0.2969, 0.9448 },

{ 0.6456, 0.4631, 0.1913, 0.1415, 0.3550 }

}

=

{

{ 0.2762, 0.7956, 0.7538, 0.5759, 1.0827 },

{ 0.5405, 1.1585, 1.0187, 0.7819, 1.5065 }

}

=

{

{ 0.2762, 0.7956, 0.7538, 0.5759, 1.0827 },

{ 0.5405, 1.1585, 1.0187, 0.7819, 1.5065 }

}

=

{

{ 0.2762, 0.7956, 0.7538, 0.5759, 1.0827 },

{ 0.5405, 1.1585, 1.0187, 0.7819, 1.5065 }

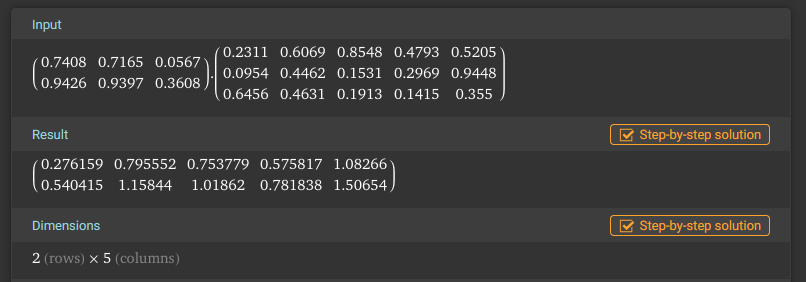
}

Single-threaded duration: 0.0030ms

Milti-threaded duration : 3.2310ms (x0.00 speed up)

OpenMP-threaded duration: 2.63ms (x0.00 speed up)

Вывод WolframAlpha:



Далее проверим выполнение программы на больших матрицах:

Matrix m1 = generate\_matrix({ 1500, 1300 });

Matrix m2 = generate\_matrix({ 1300, 2500 });

Лог выполнения программы:

Single-threaded duration: 13135.2ms

Milti-threaded duration : 1090.6ms (x12.04 speed up)

OpenMP-threaded duration: 1397.61ms (x9.40 speed up)

Видно, что выполнение на сырых потоках достаточно быстрее чем выполнение через OpenMP, моё предположение в том, что общее количество создаваемых потоков в OpenMP в моём случае сильно больше, чем в прямом случае использования потоков.