# Параллельное и распределённое программирование. Лабораторная работа №2. Вилков Е. С. ИВТ-21М. 2024

Задание к лабораторной работе: 2. «Решение уравнений методом Гаусса»

Принцип лабораторной работы аналогичен первой: составление и проверка алгоритма решения уравнений методом Гаусса на примерах с малым количеством уравнений (2-3), сравнение времени выполнения программы в разных режимах на примере с большим количеством уравнений (от 500).

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 "matrix.h"

#include "thread\_matrix.h"

#include "omp\_matrix.h"

using namespace std;

using namespace std::chrono;

int main()

{

bool printResults = true;

srand(time(NULL));

double numbers[12] = { 2, 1, -1, 8, -3, -1, 2, -11, -2, 1, 2, -3};

Matrix m1 = generate\_matrix({ 3000, 3001 }); //{ numbers, { 3, 4 } };

if (m1.size.m > 5 || m1.size.n > 6)

printResults = false;

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

Matrix m3 = solve\_gauss(m1);

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

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

Matrix m4 = solve\_gauss\_mt(m1);

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

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

Matrix m5 = solve\_gauss\_omp(m1);

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

if (printResults)

{

print\_matrix(m1);

cout << " -> " << endl;

print\_matrix(m3);

cout << "=" << endl;

print\_matrix(m4);

cout << "=" << endl;

print\_matrix(m5);

cout << endl;

cout << endl;

cout << endl;

}

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;

return 0;

}

Файл matrix.cpp:

#include "matrix.h"

#include <iomanip>

#include <iostream>

Matrix make\_upper\_triangle\_matrix(const Matrix& matrix)

{

Matrix result = { new double[matrix.size.m \* matrix.size.n], { matrix.size.m, matrix.size.n } };

memcpy(result.container, matrix.container, sizeof(double) \* matrix.size.m \* matrix.size.n);

//for every row

for (int row = 0; row < result.size.m - 1; row++)

{

// take the first item (assuming the previous items) are 0

double item = get\_matrix\_item(result, row, row);

// ...and subtract the [row] from every next row in a way, that makes the first coeffitient 0

for (int changeRow = row + 1; changeRow < result.size.m; changeRow++)

{

double item\_to\_null = get\_matrix\_item(result, changeRow, row);

double coeffitient = item\_to\_null / item;

for (int col = row; col < result.size.n; col++)

{

result.container[changeRow \* result.size.n + col] -= coeffitient \* result.container[row \* result.size.n + col];

}

}

}

return { result.container, result.size };

}

Matrix solve\_gauss(const Matrix& matrix)

{

Matrix triangle\_matrix = make\_upper\_triangle\_matrix(matrix);

Matrix result = { new double[matrix.size.m \* matrix.size.n], { matrix.size.m, matrix.size.n } };

memcpy(result.container, triangle\_matrix.container, sizeof(double) \* matrix.size.m \* matrix.size.n);

//for every row

for (int row = result.size.m - 1; row >= 0; row--)

{

// take the first item (assuming the previous items) are 0

double item = get\_matrix\_item(result, row, row);

// ...and subtract the [row] from every next row in a way, that makes the first coeffitient 0

for (int changeRow = row - 1; changeRow >= 0; changeRow--)

{

double item\_to\_null = get\_matrix\_item(result, changeRow, row);

double coeffitient = item\_to\_null / item;

for (int col = row; col < result.size.n; col++)

{

result.container[changeRow \* result.size.n + col] -= coeffitient \* result.container[row \* result.size.n + col];

}

}

// normalize current row

for (int col = 0; col < result.size.n; col++)

{

result.container[row \* result.size.n + col] /= item;

}

//std::cout << "Iteration #" << row + 1 << " = " << std::endl;

//print\_matrix(result);

}

return { result.container, result.size };

}

Файл omp\_matrix.cpp:

#include "omp\_matrix.h"

#include <iostream>

#include <omp.h>

Matrix make\_upper\_triangle\_matrix\_omp(const Matrix& matrix)

{

Matrix result = { new double[matrix.size.m \* matrix.size.n], { matrix.size.m, matrix.size.n } };

memcpy(result.container, matrix.container, sizeof(double) \* matrix.size.m \* matrix.size.n);

//for every row

for (int row = 0; row < result.size.m - 1; row++)

{

// take the first item (assuming the previous items) are 0

double item = result.container[row \* result.size.n + row]; //get\_matrix\_item(result, row, row);

// ...and subtract the [row] from every next row in a way, that makes the first coeffitient 0

#pragma omp parallel for

for (int changeRow = row + 1; changeRow < result.size.m; changeRow++)

{

double item\_to\_null = result.container[changeRow \* result.size.n + row]; // get\_matrix\_item(result, changeRow, row);

double coeffitient = item\_to\_null / item;

for (int col = row; col < result.size.n; col++)

{

#pragma vector always

result.container[changeRow \* result.size.n + col] -= coeffitient \* result.container[row \* result.size.n + col];

}

}

}

return { result.container, result.size };

}

Matrix solve\_gauss\_omp(const Matrix& matrix)

{

Matrix triangle\_matrix = make\_upper\_triangle\_matrix\_omp(matrix);

Matrix result = { new double[matrix.size.m \* matrix.size.n], { matrix.size.m, matrix.size.n } };

memcpy(result.container, triangle\_matrix.container, sizeof(double) \* matrix.size.m \* matrix.size.n);

//for every row

for (int row = result.size.m - 1; row >= 0; row--)

{

// take the first item (assuming the previous items) are 0

double item = result.container[row \* result.size.n + row]; // get\_matrix\_item(result, row, row);

// ...and subtract the [row] from every next row in a way, that makes the first coeffitient 0

#pragma omp parallel for

for (int changeRow = row - 1; changeRow >= 0; changeRow--)

{

double item\_to\_null = result.container[changeRow \* result.size.n + row]; // get\_matrix\_item(result, changeRow, row);

double coeffitient = item\_to\_null / item;

for (int col = row; col < result.size.n; col++)

{

#pragma vector always

result.container[changeRow \* result.size.n + col] -= coeffitient \* result.container[row \* result.size.n + col];

}

}

// normalize current row

#pragma omp parallel for

#pragma vector always

for (int col = row; col < result.size.n; col++)

{

result.container[row \* result.size.n + col] /= item;

}

//std::cout << "Iteration #" << row + 1 << " = " << std::endl;

//print\_matrix(result);

}

return { result.container, result.size };

}

Файл thread\_matrix.cpp:

#include "thread\_matrix.h"

#include <barrier>

#include <iostream>

#include <string>

#include <syncstream>

#include <thread>

#include <vector>

void calculate\_part\_triangle\_thread(double\* resultContainer, double\* sourceContainer, int rowSize, int colSize, int threadIndex, int threadSize, int pass)

{

// because we want to process all rows, we need to iterate over 'every other' item in column

// Imagine there is 4 cores (4 threads)

// thread 0 will process rows: [0], [4], [8], ...

// thread 1 will process rows: [1], [5], [9], ...

// thread 2 will process rows: [2], [6], [10], ...

// thread 3 will process rows: [3], [7], [11], ...

//

// all thread will 'cycle' through rows, creating a more uniform load distribution,

// than simply dividing every number of rows for a set number of threads

// (bc that way first batch will finish earlier, and start wait for 'heavier' threads to finish)

for (int rowIndex = pass + 1 + threadIndex; rowIndex < colSize; rowIndex += threadSize)

{

double coeffitient = resultContainer[rowIndex \* rowSize + pass] / resultContainer[pass \* rowSize + pass];

for (int col = pass; col < rowSize; col++)

{

resultContainer[rowIndex \* rowSize + col] -= coeffitient \* resultContainer[pass \* rowSize + col];

}

}

}

void calculate\_part\_gauss\_thread(double\* resultContainer, double\* sourceContainer, int rowSize, int colSize, int threadIndex, int threadSize, int pass)

{

for (int rowIndex = pass - 1 - threadIndex; rowIndex >= 0; rowIndex -= threadSize)

{

double coeffitient = resultContainer[rowIndex \* rowSize + pass] / resultContainer[pass \* rowSize + pass];

for (int col = rowIndex; col < rowSize; col++)

{

resultContainer[rowIndex \* rowSize + col] -= coeffitient \* resultContainer[pass \* rowSize + col];

}

}

}

Matrix make\_upper\_triangle\_matrix\_mt(const Matrix& matrix)

{

Matrix result = { new double[matrix.size.m \* matrix.size.n], { matrix.size.m, matrix.size.n } };

memcpy(result.container, matrix.container, sizeof(double) \* matrix.size.m \* matrix.size.n);

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

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

if (processor\_count == 0)

processor\_count = 1;

// if hardware\_concurrency is greater than data size

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

if (processor\_count > matrix.size.m)

processor\_count = matrix.size.m;

for (size\_t row = 0; row < matrix.size.m - 1; row++)

{

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

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

{

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

result.container,

matrix.container,

matrix.size.n,

matrix.size.m,

i,

processor\_count,

row);

}

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

{

threads[i].join();

}

delete[] threads;

}

return { result.container, result.size };

}

Matrix solve\_gauss\_mt(const Matrix& matrix)

{

Matrix triangleMatrix = make\_upper\_triangle\_matrix\_mt(matrix);

//print\_matrix(triangleMatrix);

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

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

if (processor\_count == 0)

processor\_count = 1;

// if hardware\_concurrency is greater than data size

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

if (processor\_count > matrix.size.m)

processor\_count = matrix.size.m;

for (int row = triangleMatrix.size.m - 1; row >= 0; row--)

{

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

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

{

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

triangleMatrix.container,

triangleMatrix.container,

triangleMatrix.size.n,

triangleMatrix.size.m,

i,

processor\_count, row);

}

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

{

threads[i].join();

}

delete[] threads;

}

return { triangleMatrix.container, triangleMatrix.size };

}

## Пояснения:

Метод решения систем линейных алгебраических уравнений (СЛАУ) Гаусса, довольно сложен для «распараллеливания». Так как, по определению, (и по статье в википедии) идея метода – в «упрощении» переменной путём методичного обнуления коэффициентов перед другими переменными используя эквивалентные преобразования системы.

Простыми словами: для того, чтобы получить «результат» одной переменной, необходимо выполнить полное преобразование строки с её номером, а для выполнения преобразования этой строки, необходимо ПОЛНОСТЬЮ закончить преобразование строки предыдущей.

То есть, необходимо для каждой строки в худшем случае итерировать по всей матрице. С точки зрения параллелизма, это плохо, так как разделение потоков по строкам сильно усложнено, потоку, который «хочет» работать над строкой, необходимо ждать, пока результат для него подготовит поток предыдущей строки. Такое решение не даст никакого преимущества над решением последовательным, единственное что, будет нагружено не одно физическое ядро на 100%, а все ядра поочерёдно, в среднем загрузка каждого ядра будет меньше.

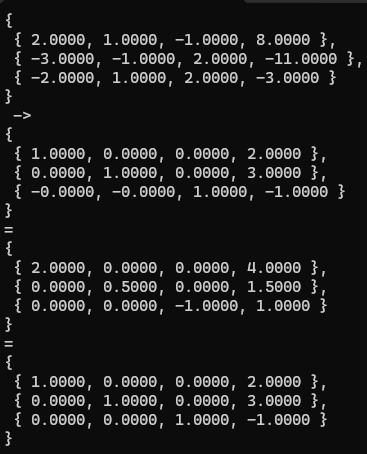
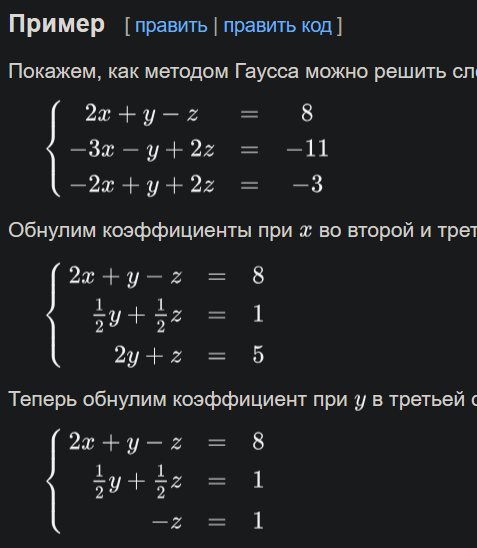
Разбить по строкам, выглядит изначально как хорошая идея, но, к сожалению, в эквивалентом преобразовании строк, константа на которую умножается вычитаемая строка, берётся из первого ненулевого элемента подготовленной строки. Всплывает такое-же ограничение на ожидание предыдущих потоков.

Остаётся два варианта: 1) выполнять параллельно, то, что не зависит от других потоков, само вычитание одной строки из другой по нескольким строкам, когда данные уже готовы. 2) развернуть алгоритм «в обратную сторону», так, чтобы множеству потоков не приходилось писать в одну и ту же область памяти из-за рассинхронизации.

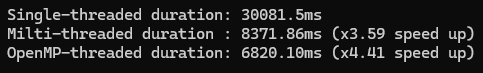
2й вариант исходит из идеи что: «результат треугольной матрицы, n-й строки, зависит только от строк предыдущих, то есть, для 3й строки, нам необходимо оперировать только строками 1 и 2: из 3-й строки вычесть строку 2 и строку 1, а строка 2 в свою очередь получается путём вычитывания строки 1 из изначальной. Однако такой алгоритм становится в худшем случае О(N4) так как для каждой строки матрицы, приходится не только смотреть на «все» строки до неё, но и на все рекурсивные уровни ниже. Не вариант.

Остановились на 1м варианте с идеей показать, что переключение между контекстами потоков – это плохо. Очень много потоков – это плохо, а создавать свой конвейер – ещё хуже, умные люди уже всё сделали за нас под капотом библиотек, и вряд ли получится что-то лучше.

Итак, выполнение:

   
Рис. 1. «Проверка результатов на верность»

Результаты хорошие, далее провека быстродействия на большой СЛАУ (3000 уравнений):

Рис. 2. «Быстродействие программы»

Видно, что выполнение своей реализации многопоточного приложения выполняется несколько хуже, чем такое же функциональное решение с использованием библиотеки OpenMP. Дело в том, что в отличии от лабораторной работы №1 где было показано, что лучше никакого конвейера потоков, чем хороший (когда работают все ядра системы на полную загрузку, и они внутри сами переключаются на новый сет данных, это всегда лучше, чем так или иначе переключать контекст выполнения). Однако здесь, наша реализация «спавнит» очень много потоков, которые должны и переключиться, и завершится, и создать их ещё все надо. А в OpenMP они создаются более умно – и потому, программа быстрее.

Полный исходный код можно найти в репозитории:

<https://github.com/ZecosMAX/yuujin.uni.cpp.parallel_computing>