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

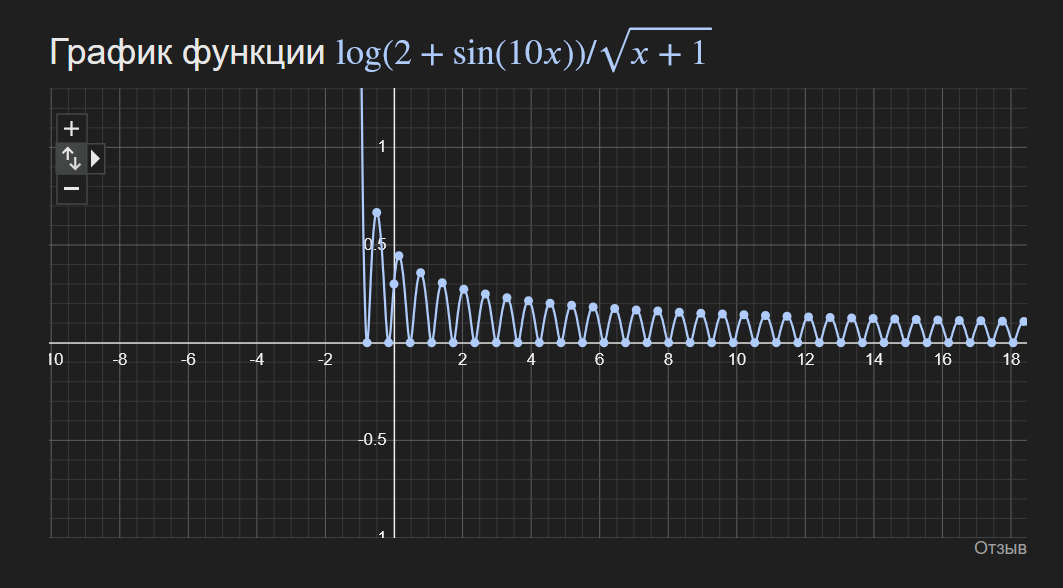
Задание к лабораторной работе: 2. «Решение интеграла»

В работе необходимо вычислить случайный интеграл средней сложности на диапазоне от (например) 1 до 1000 с шагом 1 (от 1 до 2, от 2 до 3 и т.д.) для проверки алгоритма вычисления. В дальнейшем шаг вычислений уменьшить в 100-1000 раз и запустить в последовательном и параллельном режимах работы аналогично предыдущим лабораторным работам.

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.

Для вычисления интеграла была выбрана следующая функция:

Выглядит она следующим образом:

****Рис. 1. «График выбранной функции»

Критерий того какая функция даст интеграл «средней сложности», потому, был выбран такой интеграл, от всего сердца. И сердце подсказывает что интеграл средней сложности.

Файл main.cpp:

#include <iomanip>

#include <iostream>

#include <chrono>

#include <thread>

#include <omp.h>

#include "common.h"

using namespace std;

using namespace std::chrono;

double calc\_integral\_st(double start, double end, double step);

double calc\_integral\_mt(double start, double end, double step);

double calc\_integral\_omp(double start, double end, double step);

int main()

{

double start = 0.0;

double end = 100.0;

double step = 1e-6;

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

double m3 = calc\_integral\_st(start, end, step);

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

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

double m4 = calc\_integral\_mt(start, end, step);

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

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

double m5 = calc\_integral\_omp(start, end, step);

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

cout << "Single-threaded value: " << m3 << endl;

cout << "Milti-threaded value : " << m4 << endl;

cout << "OpenMP-threaded value: " << m5 << 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;

}

double calc\_integral\_st(double start, double end, double step)

{

double area = 0.0;

for (double pos = start; pos < end; pos += step)

{

area += (function(pos) + function(pos + step)) \* 0.5 \* step;

}

return area;

}

void calc\_integral\_thread(double start, double end, double step, double threadIndex, double threadSize, double\* returnValue)

{

double area = 0.0;

double threadStep = (end - start) / threadSize;

double threadStart = start + threadIndex \* threadStep;

double threadEnd = start + (threadIndex + 1) \* threadStep;

for (double pos = threadStart; pos < threadEnd; pos += step)

{

area += (function(pos) + function(pos + step)) \* 0.5 \* step;

}

\*returnValue = area;

}

double calc\_integral\_mt(double start, double end, double step)

{

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

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

if (processor\_count == 0)

processor\_count = 1;

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

double\* returnValues = new double[processor\_count];

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

{

threads[i] = std::thread(calc\_integral\_thread,

start,

end,

step,

i,

processor\_count,

&returnValues[i]);

}

double area = 0.0;

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

{

threads[i].join();

area += returnValues[i];

}

delete[] threads;

return area;

}

double calc\_integral\_omp(double start, double end, double step)

{

double area = 0.0;

int n = (end - start) / step;

double rem = (end - start) - (n \* step);

#pragma omp parallel for reduction(+:area)

for (int pos = 0; pos < n; pos += 1)

{

area += (function(pos \* step) + function((pos + 1) \* step)) \* 0.5 \* step;

}

//area += (function(n \* step) + function(n \* step + rem)) \* 0.5 \* rem;

return area;

}

Файл common.h

#pragma once

#include <iostream>

double function(double x)

{

return log(2.0 + sin(10.0 \* x)) / sqrt(x + 1.0);

}

## Пояснения:

Был выбран простейший метод приближения конечного интеграла: Суммы Римана. Так как геометрический смысл интеграла – знаковая площадь под графиком, то тогда интеграл можно представить как предел сумм «прямоугольников», высота которых между значениями функции в точках расположения боковых сторон такого прямоугольника, умноженных на ширину данных прямоугольников. И при ширине таких прямоугольников, стремящихся к нулю, такая сумма стремится к значению интеграла функции.

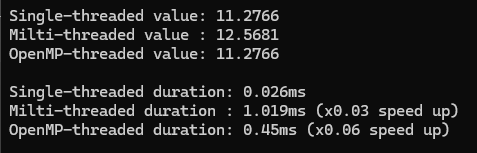
Можно было бы выбрать и другой метод, например аппроксимация трапецией, но зачем напрягаться, когда можно не напрягаться? (в задании лабораторной работы не уточнено какой метод необходимо использовать)

В случае простых поток был применён метод разделения зон, в которых считается интеграл: то есть, если интеграл считается от 0 до 100, а потоков десять, то первый поток будет считать диапазон от 0 до 10, второй поток от 10 до 20, третий от 20 до 30 и так далее… Нет никакого смысла «распределять» точки по всему диапазону, так как в конкретно данной реализации, нет заготовленных данных, матриц и массивов, и оптимизировать кэш процессора нет необходимости.

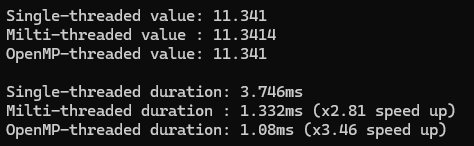
В случае библиотеки OpenMP был использован оператор редукции: «omp parallel for reduction(+:area)» который делит цикл на потоки, создаёт внутренний временный массив, и по окончанию выполнения цикла, применяет оператор указанный в директиве на весь массив (в данном случае сложение)

Выполнение программы с разными шагами:

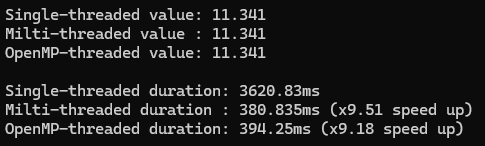
Шаг 1:



Шаг 1e-3:



Шаг 1е-6:



Видно, что с ростом количества итераций, линейно растёт лишь однопоточное выполнение, что не удиивительно. В шагах «1» и «0.001» выполнение программы на 16 потоках занимает сильно меньше времени, чем подготовка необходимого колиечства потоков, и отсюда, скорость выполнения одного потока в 100 раз больше, и лишь в 3 раза меньше соответсвтенно.

Однако интересно то, что при выполнении с шагом «1», решение OpenMP стабильно в 2 раза быстрее, чем сырые потоки. Скорее всего это свяано с тем, что OpenMP использует некое динамическое решение диспатча потоков, и на таком кол-ве итераций не нужно выделять все 16 потоков. Они и не выделяются, в отличии от решение с простыми потоками, где всегда создаётся ровно столько, сколько есть ядер в системе. Отсюда прирост в 0.5мс на создание и включение контекста выполения. При этом, с шагом «0.001» разница меньше, скорее всего из-за того что OpenMP отправил на выполнение больше потоков.

Полный исходный код можно найти в репозитории:

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