МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра «Вычислительная техника»

Дисциплина «Высокопроизводительные вычисления»

**Лабораторная работа №2**

«Исследование параллельных реализаций алгоритма численного интегрирования»

**Вариант №3**

Выполнил:

студент 3 курса, гр. ИВТВМбд-31

Захарычев Н.А.

Проверил:

д. т. н., профессор кафедры ВТ

Негода В. Н.

г. Ульяновск, 2017

Цель работы

Изучение методов распараллеливания реализации вычисления определенного интеграла.

Вариант

Функция: *cos(x);*

**1. Выбор метода численного интегрирования**

В качестве метода численного интегрирования был выбран метод прямоугольников(средних).

**Метод прямоугольников** — метод [численного интегрирования](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%B8%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) функции одной переменной, заключающийся в замене подынтегральной функции на многочлен нулевой степени, то есть константу, на каждом элементарном отрезке. Если рассмотреть график подынтегральной функции, то метод будет заключаться в приближённом вычислении площади под графиком суммированием площадей конечного числа прямоугольников, ширина которых будет определяться расстоянием между соответствующими соседними узлами интегрирования, а высота — значением подынтегральной функции в этих узлах. [Алгебраический порядок точности](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA_%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D0%B8_%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B0) равен 1.

Если отрезок,{\displaystyle \left[a,b\right]} является элементарным и не подвергается дальнейшему разбиению, значение интеграла можно найти по формуле прямоугольников (средних):



**2. Исследование многопоточных реализаций выбранного метода численного интегрирования в среде одно-, двух- и четырехядерных микропроцессоров**

Был произведен замер при варьировании числа потоков от 1 до 8 и гранулярности задачи от 100 до 10 000 000 вычислений подинтегральной функции.

Для реализации многопоточного выполнения цикла вычислений было решено использовать библиотеку *OpenMP*, поскольку она позволяет крайне легко и просто выполнить распараллеливание цикла вычисления, без каких-либо усложнений исходного кода.

**Конфигурация №1**

|  |  |
| --- | --- |
| Название процессора | Intel® Core™ i5-7200U |
| Количество ядер | 2 |
| Количество потоков | 4 |
| Базовая частота | 2,5 ГГц |
| Кэш-память | 3 Мб |
| Компилятор | VisualStudio (64bit) |

**Время вычисления интеграла при значении аргумента от 0 до 1.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | time, мс | | | | |  |
| threads | 100,00 | 1000,00 | 10000,00 | 100000,00 | 1000000,00 |  |
| 1 | 8,3067 | 81,935 | 823,881 | 8244,09 | 90856,80 |  |
| 2 | 4,9085 | 41,534 | 409,675 | 4148,85 | 52159,70 |  |
| 3 | 3,7758 | 34,360 | 341,333 | 3406,53 | 36567,50 |  |
| 4 | 6,0413 | 26,431 | 259,398 | 2932,67 | 30359,40 |  |
| 5 | 6,4187 | 40,401 | 383,622 | 3842,26 | 41572,70 |  |
| 6 | 5,6636 | 35,493 | 331,893 | 3342,34 | 36428,20 |  |
| 7 | 6,7963 | 31,717 | 296,778 | 2943,24 | 34337,20 |  |
| 8 | 7,1740 | 28,696 | 260,531 | 2579,25 | 37043,70 |  |
|  |  |  |  |  |  |  |

**Конфигурация №2**

|  |  |
| --- | --- |
| Название процессора | AMD Athlon II X2 215 |
| Количество ядер | 2 |
| Количество потоков | 2 |
| Базовая частота | 2,7 ГГц |
| Кэш-память | 1 Мб |
| Компилятор | VisualStudio (64bit) |

**Время вычисления интеграла при значении аргумента от 0 до 1.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | time, мс | | | | |
| threads | 100,00 | 1000,00 | 10000,00 | 100000,00 | 1000000,00 |
| 1 | 15,64 | 140,400 | 1388,600 | 13987,10 | 147086,00 |
| 2 | 11,96 | 72,240 | 696,080 | 7316,96 | 120138,00 |
| 3 | 13,08 | 91,360 | 875,120 | 9834,68 | 94599,10 |
| 4 | 11,8 | 70,120 | 673,200 | 6603,60 | 72286,50 |
| 5 | 13,16 | 86,760 | 821,000 | 7835,76 | 120060,00 |
| 6 | 12,84 | 94,040 | 657,720 | 8648,48 | 97917,70 |
| 7 | 14,16 | 79,360 | 763,920 | 7414,48 | 120599,00 |
| 8 | 14,04 | 70,000 | 667,280 | 6947,64 | 77093,90 |
|  |  |  |  |  |  |

Исходя из полученных результатов можно заметить, что при малом количестве итераций время создания потока и синхронизация общих переменных, превосходит время, выигранное благодаря распараллеливанию.

Взглянув на характеристики процессоров, на которых проводились эксперименты, можно сказать что, минимальное время выполнения мы получаем, когда число порождаемых потоков равно числу ядер процессора.

**3. Вывод**

Анализируя полученные данные, можно сказать, что с увеличением числа потоков, уменьшается скорость роста производительности, так как последовательная часть программы, которая отвечает за создание и управление потоками не может быть распараллелена. Так же на вид графиков оказывает влияние непостоянная частота процессора. При использовании меньшего количества ядер частота выше, чем при полной нагрузке.

**4. Исходный код**

#include "stdafx.h"

#include <stdio.h>

#include <conio.h>

#include <iostream>

#include <cmath>

#include <time.h>

#include <omp.h>

int nMin = 100;

int nMax = 1000000;

double timing = 0;

double minTiming = 0;

float f(float x) {

return cos(x);

}

double CalcIntegral(double a, double b, int n)

{

int i;

double result, h;

result = 0;

h = (b - a) / n; //Шаг сетки

double startTiming = omp\_get\_wtime();

#pragma omp parallel for reduction(+:result)

for (i = 0; i < n; i++)

{

result += f(a + h \* (i + 0.5)); //Вычисляем в средней точке и добавляем в сумму

}

double endTiming = omp\_get\_wtime();

result \*= h;

timing = (endTiming - startTiming);

return result;

}

void getTest(int maxThreads) {

double temp1 = 0;

for (int n = nMin, k = 10000; n <= nMax; n \*= 10, k /= 10) {

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

printf("steps: %d\n", n);

printf("\nthreads , timing\n");

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

omp\_set\_dynamic(0); // запретить библиотеке openmp менять число потоков во время исполнения

omp\_set\_num\_threads(i); // установить число потоков в i

timing = 0;

minTiming = INT32\_MAX;

for (int j = 0; j < k; j++) {

CalcIntegral(0.0F, 1.0F, n);

if (timing < minTiming) {

minTiming = timing;

}

}

std::cout << " " << i << ", " << minTiming \* 1000000 << std::endl;

}

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

}

}

int main()

{

int n = 100;

std::cout << CalcIntegral(0.0F, 1.0F, n);

getTest(8);

\_getch();

return 0;

}