МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ФИЗИКИ

КАФЕДРА РАДИОАСТРАНОМИИ

Лабораторная работа на тему:

**Высокопроизводительные вычислительные системы**

Выполнил студент группы 06-722

Дуничев Кирилл

г.Казань

2020

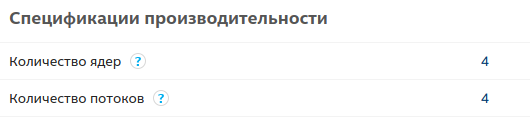
**Цель работы:**  провести эксперимент по анализу ускорения вычислений на кластере.

**Ход работы:**

**Работа проводится на OS Linux mint 19.3 Cinnamon**

**Процессор Intel© Core™ i5-7300HQ CPU @ 2.50GHz × 4**

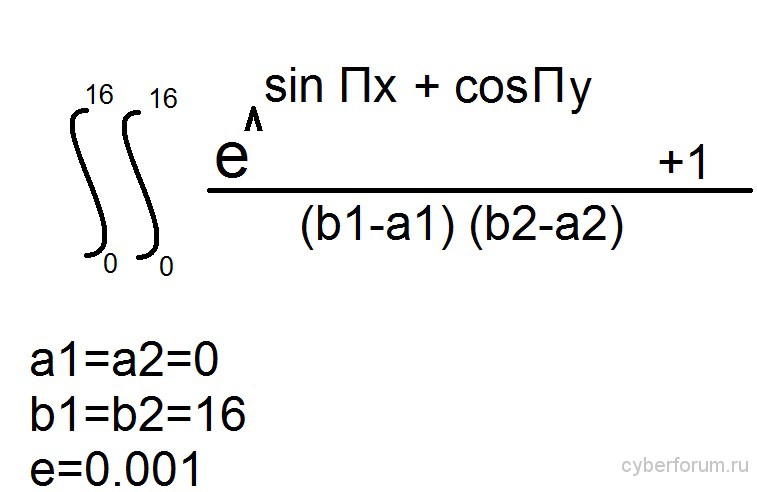
Из характеристик с официального сайта intel.com, узнали, что данный процессор имеет 4 физических ядра по 1 логическому ядру в каждом.



Значит в нашей задаче будем использовать от 1 до 4 потоков при параллельных вычислениях.

Поставленная перед нами задача формулируется следующим образом:

**Вычисление двумерного интеграла с заданной точностью. Мной был выбран данный интеграл**



Составили и отладили программу, решающую задачу, на языке С++ в обычном, последовательном и параллельном режиме вычислений.

#include <iostream>

#include <omp.h>

#include <cstdlib>

#include <time.h>

#include <stdio.h>

#define \_USE\_MATH\_DEFINES

#define \_USE\_GENERIC\_MATH1

#include <math.h>

#include <unistd.h>

using namespace std;

double f(double x, double y)

{

return (exp(sin(3.141592\*x)+cos(3.141592\*y))+1)/16;

}

double func(double a1,double b1,double a2,double b2, int n1,int n2, int nthreads)

{

double I=0,h1=(b1-a1)/n1,h2=(b2-a2)/n2;

int how\_many\_x = (b1-a1)/h1;

int how\_many\_y = (b2-a2)/h2;

double x = a1, y=a2;

#pragma omp parallel for num\_threads(nthreads) reduction(+:I)

for (int board1 = 0; board1<how\_many\_x; board1++)

{

for (int board2 = 0; board2<how\_many\_y;board2++)

{

I+=h1\*h2\*f(x+h1/2,y+h2/2);

y+=h1;

}

x+=h1;

}

return I;

}

int main() {

clock\_t clock\_timer;

double wall\_timer;

for (int nthreads = 1; nthreads <=4; ++nthreads) {

// cout << "I = " << func(0,16,0,16,1000,1000, nthreads) << endl;

clock\_timer = clock();

wall\_timer = omp\_get\_wtime();

std::cout << "Результат программы = " << func(0,16,0,16,40000,40000, nthreads) << " Количество потоков: " << nthreads << " Время выполнения: " << omp\_get\_wtime() - wall\_timer << "сек" << "\n";

}

cout << "Результат истинный = " << 41.6467599099991 << endl;

return 0;

}

Для распараллеливания была выбрана библиотека openMP.

Для этого в шапку программы добавили строку

#include <omp.h>

Очевидно, что цикл for можно распараллелить и обрабатывать сразу несколькими ядрами процессора, поскольку вычисление значения любого элемента площади прямоугольника никак не зависит от остальных элементов подынтегральной кривой. В листинге 1 показано, как можно это сделать с помощью OpenMP.

Листинг 1.

#pragma omp parallel

for (int board1 = 0; board1<how\_many\_x; board1++)

{

for (int board2 = 0; board2<how\_many\_y;board2++)

{

I+=h1\*h2\*f(x+h1/2,y+h2/2);

y+=h1;

}

x+=h1;

}

Количеством потоков можно легко управлять с помощью прагмы с аргументом num\_threads. Ниже представлен код из листинга 2 с заданным количеством потоков (int nthreads потоков):

Листинг 2.

#pragma omp parallel for num\_threads(nthreads)

for (int board1 = 0; board1<how\_many\_x; board1++)

{

for (int board2 = 0; board2<how\_many\_y;board2++)

{

I+=h1\*h2\*f(x+h1/2,y+h2/2);

y+=h1;

}

x+=h1;

}

**Директива reduction**

Директива OpenMP reduction позволяет собрать вместе в главном потоке результаты вычислений частичных сумм, разностей и т. п. из параллельных потоков последующего параллельного структурного блока. В нашей программе в каждом параллельном потоке определена локальная переменная I для вычисления частичных сумм интеграла. После завершения параллельных потоков все локальные переменные I суммируются, а результат сохраняется в одноименной общей (глобальной) переменной I.

В листинге 3 показано, как это реализовано в нашей программе с помощью OpenMP

Листинг 3

#pragma omp parallel for num\_threads(nthreads) reduction(+:I)

for (int board1 = 0; board1<how\_many\_x; board1++)

{

for (int board2 = 0; board2<how\_many\_y;board2++)

{

I+=h1\*h2\*f(x+h1/2,y+h2/2);

y+=h1;

}

x+=h1;

}

Для того чтобы значительно увеличить временную сложность вычисления функций мы решили разбить интервал интегрирования на 40000 отрезков по оси oX и 40000 отрезков по оси oY.

Количество разбиений передаем в фунцию 5-6 параметром (см. листинг 4)

Листинг 4

func(0,16,0,16,40000,40000, nthreads)

Прагма parallel for помогает распределить рабочую нагрузку цикла for между несколькими потоками, каждый из которых может обрабатываться отдельным ядром процессора; таким образом общее время вычислений существенно снижается. Это подтверждается в листинге 5.

Листинг 5. Пример с использованием API-функции omp\_get\_wtime

clock\_t clock\_timer;

double wall\_timer;

for (int nthreads = 1; nthreads <=4; ++nthreads) {

// cout << "I = " << func(0,16,0,16,1000,1000, nthreads) << endl;

clock\_timer = clock();

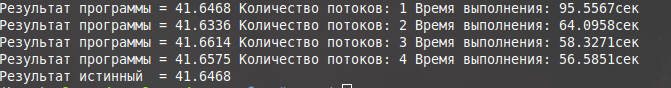
wall\_timer = omp\_get\_wtime();

std::cout << "Результат программы = " << func(0,16,0,16,40000,40000, nthreads) << " Количество потоков: " << nthreads << " Время выполнения: " << omp\_get\_wtime() - wall\_timer << "сек" << "\n";

}

В листинге 5 мы измеряем время выполнения двух циклов for, увеличивая при этом количество потоков. API-функция omp\_get\_wtime возвращает затраченное фактическое время (в секундах), прошедшее с начала заданной точки отсчета. Таким образом, значение omp\_get\_wtime() - wall\_timer возвращает фактическое время выполнения цикла for. Системный вызов clock() используется для оценки времени, затраченного центральным процессором на выполнение всей программы, т. е. прежде чем получить итоговый результат, мы суммируем все эти временные интервалы с учетом потоков. На моем компьютере с процессором Intel Core i5 я получил результаты, приведенные в листинге 6.

Листинг 6. Статистика выполнения нашей программы



**ВЫВОД:** Время выполнения программы постоянно уменьшалось при увеличении количества потоков, которые выполнялись параллельно отдельными процессорными ядрами. **Мы экспериментально продемонстрировали, что при увеличении потоков можно добиться существенного уменьшения времени выполнения программы. В нашем случае, увеличение числа потоков до 4х дало прибавку в скорости в 41.83% (39.9716 сек). Мы почти в два раза выиграли в скорости!**