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

Быстров Сергей

г.Казань

2020

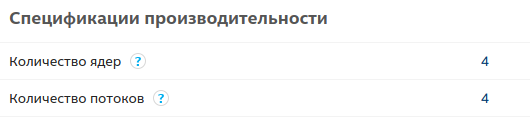
**Цель работы:**  провести эксперимент по анализу ускорения вычислений на кластере.

**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 суммы двумерных массивов. (Вычислить матрицу С = А+В). Для усложнения вычислять элементы массива с помощью сложной функции в процессе суммирования.**

**Мною была выбрана сложная функция следующего вида:**

**Element =**

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;

// функция генерации случайного числа из диапазона от range\_min до range\_max включительно

int rrand(int range\_min, int range\_max) {

return rand() % (range\_max - range\_min + 1) + range\_min;

}

const unsigned int ARRSIZE = 10000;

const int ABSLIMIT = 9;

int A[ARRSIZE][ARRSIZE];

int B[ARRSIZE][ARRSIZE];

int RESULT[ARRSIZE][ARRSIZE];

// void print\_matrix(int A[][ARRSIZE])

// {

// int i = 0;

// int j = 0;

// while (i < ARRSIZE)

// {

// while (j < ARRSIZE)

// {

// // printf("%-5d ", A[i][j]);

// j++;

// }

// j = 0;

// i++;

// // printf("\n");

// }

// }

int main(void)

{

int i = 0;

int j = 0;

int nthreads;

clock\_t clock\_timer;

double wall\_timer;

// Заполняем двумерные массивы A и В рандомными числами

while (i < ARRSIZE)

{

while (j < ARRSIZE)

{

A[i][j] = rrand(-ABSLIMIT, ABSLIMIT);

B[i][j] = rrand(-ABSLIMIT, ABSLIMIT);

j++;

}

j = 0;

i++;

}

//запускаем в расспараллеливание операци сложения массивов

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

clock\_timer = clock();

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

#pragma omp parallel for num\_threads(nthreads)

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

{

for (j = 0; j < ARRSIZE; j++)

{

RESULT[i][j] = tan(cos(sin(cos(tan(A[i][j] + B[i][j])))));

}

}

std::cout << "Задействовано потоков: " << nthreads

<< " Время выполнения программы: " << omp\_get\_wtime() - wall\_timer

<< " сек" << "\n";

}

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 двумерного массива никак не зависит от вычисления остальных элементов матрицы результата. В листинге 1 показано, как можно это сделать с помощью OpenMP.

Листинг 1.

#pragma omp parallel

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

{

for (j = 0; j < ARRSIZE; j++)

{

RESULT[i][j] = tan(cos(sin(cos(tan(A[i][j] + B[i][j])))));

}

}

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 (i = 0; i < ARRSIZE; i++)

{

for (j = 0; j < ARRSIZE; j++)

{

RESULT[i][j] = tan(cos(sin(cos(tan(A[i][j] + B[i][j])))));

}

}

Для того чтобы значительно увеличить временную сложность вычисления функций мы решили использовать матрицы размером 10000х10000 элементов, а также использовать сложную функцию, для вычисления суммы каждого из элементов. Наша функция выглядит следующим образом

**Element =**

Размеры массивов передаем в функцию глобальной переменной ARRSIZE (см. листинг 4)

Листинг 4

const unsigned int ARRSIZE = 10000;

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

int nthreads;

clock\_t clock\_timer;

double wall\_timer;

//запускаем в расспараллеливание операци сложения массивов

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

clock\_timer = clock();

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

#pragma omp parallel for num\_threads(nthreads)

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

{

for (j = 0; j < ARRSIZE; j++)

{

RESULT[i][j] = tan(cos(sin(cos(tan(A[i][j] + B[i][j])))));

}

}

std::cout << "Задействовано потоков: " << nthreads

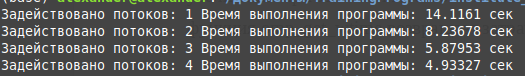
<< " Время выполнения программы: " << omp\_get\_wtime() - wall\_timer

<< " 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 65.05% (9.18283 сек). Мы более чем в два раза выиграли в скорости при использовании многопоточности, по сравнению с однопоточным режимом вычислений!**