Национальный исследовательский ядерный университет
“МИФИ”

Лабораторная работа №2: “Выделение ресурса параллелизма. Технология OpenMp”

ИИКС ИБ

Б20-505

Барабанов Андрей

2022 год

1. Рабочая среда

```
~ lscpu
Архитектура:             x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Порядок байт:          Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
ID прроизводителя:       GenuineIntel
  Имя модели:            Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    Семейство ЦПУ:       6
    Модель:              165
    Thread(s) per core:  2
    Ядер на сокет:       6
    Сокетов:             1
    Степпинг:            2
    CPU(s) scaling MHz:  24%
    CPU max MHz:         5000,0000
    CPU min MHz:         800,0000
    BogoMIPS:            5202,65
```

```
~ echo |cpp -fopenmp -dM |grep -i open
#define _OPENMP 201511
```

2. Анализ алгоритма

Алгоритм работает в наилучшем случае за O(1), в наихудшем за O($\frac{n}{m}$), где n – количество данных, m – количество потоков.

#pragma omp parallel num_threads(threads) shared(array, count, i, target, index) default(none)

#pragma - директива компилятора

• *omp* - принадлежность директивы к OpenMp

• Параллельная область задаётся при помощи директивы *parallel*

• *num_threads(целочисленное выражение)* – явное задание количества потоков, которые будут выполнять параллельную область; по умолчанию выбирается последнее значение, установленное с помощью функции omp_set_num_threads(), или значение переменной OMP_NUM_THREADS

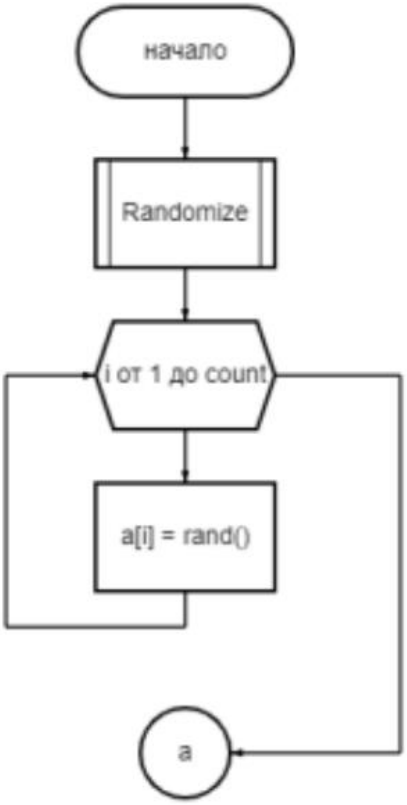
• *shared(список)* – задаёт список переменных, общих для всех потоков

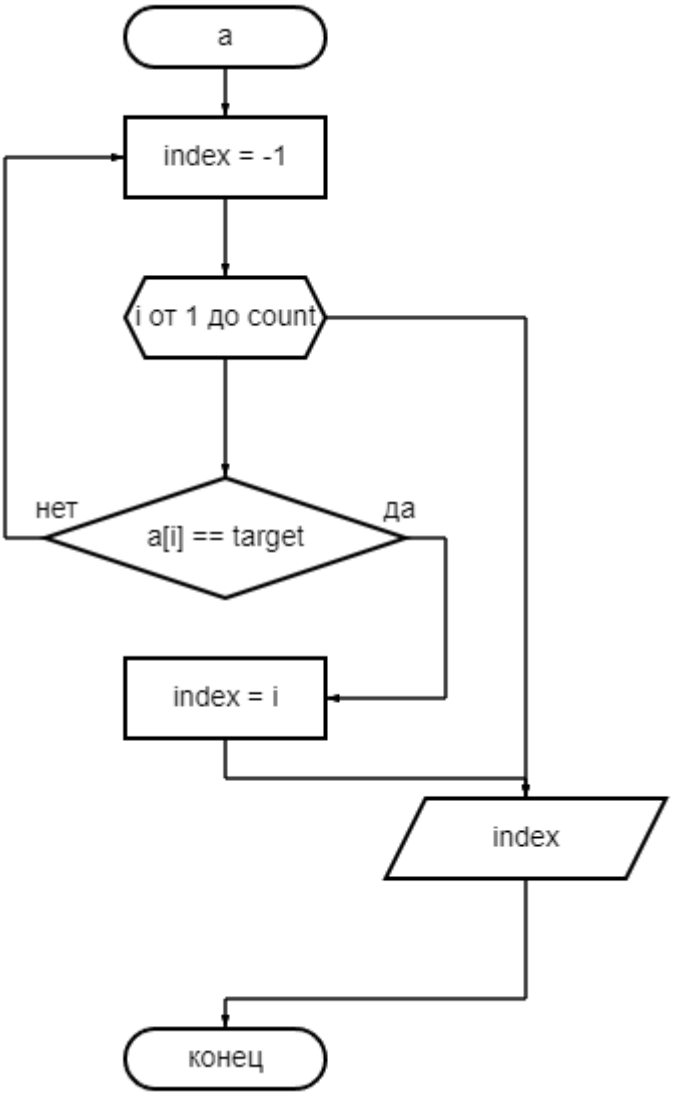
• *reduction(оператор:список)* -задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждом потоке; локальные копии инициализируются соответственно типу оператора; над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску

• *default(private|firstprivate|shared|none)* – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс private, firstprivate или shared соответственно; none означает, что всем переменным в параллельной области класс должен быть назначен явно

#pragma omp for

• *for* - Используется для распределения итераций цикла между различными потоками


начало
Randomize
i от 1 до count
a[i] = rand()
a


a
index = -1
i от 1 до count
нет
a[i] == target
да
index = i
index
конец

3. Код

1) lab1.c

```c
#include "app.h"

void f(int rand_seed, float *times)
{
    const int count = 1000000;

    int *array = calloc(count, sizeof(int));
    int threads = THREADS;
    int n = 100;
    int target = 1;
    int index = -1;

    double start_time = 0, end_time = 0;
    srand(rand_seed);

    for (int i = 0; i < count; i++)
        array[i] = rand() % count;

    for (int i = 0; i < threads; i++)
    {
        int cnt = count;

        start_time = omp_get_wtime();
        #pragma omp parallel num_threads(i + 1) shared(array, cnt, i, target, index) default(none)
        {
            #pragma omp for
            for (int j = 0; j < cnt; j++)
                if (array[j] == target)
                {
                    index = j;
                    cnt = 0;
                }


        }
            end_time = omp_get_wtime();
            times[i] = end_time - start_time;
    }

    free(array);
}
```

2) main.c

```c
#include "app.h"

int main(int argc, char **argv)
{
    printf("OpenMP: %d; \n======\n", _OPENMP);

    int iter = 10;
    int threads = THREADS;
    int seed = 93932;
    FILE *file = fopen("experiment.txt", "w");

    fwrite(&threads, sizeof(int), 1, file);
    fwrite(&iter, sizeof(int), 1, file);

    float *times = 0;

    for (int i = 0; i < 10; i++)
    {
        printf("------Iteration number: %d\n", i);
        times = (float*)calloc(threads, sizeof(float));
        f(seed + i, times);
        fwrite(times, sizeof(float), threads, file);
        free(times);
    }
    fclose(file);
    return 0;
}
```

3) plot_and_table.py

```python
from pwn import *
from prettytable.colortable import ColorTable, Themes
import matplotlib.pyplot as plt

def inp():
    data = open('experiment.txt', 'rb')
    try:
        num_of_threads = u32(data.read(4))
        threads = [i+1 for i in range(num_of_threads)]
        iterations = u32(data.read(4))
        time = [[] for i in range(num_of_threads)]
        for i in range(iterations * num_of_threads):
            time[i % num_of_threads].append(float(struct.unpack('f',data.read(4))[0]))
    finally:
        data.close()
        return time, threads

def plots(times, threads):
    time_average = [sum(k)/len(k) for k in times]
    expected_time = [time_average[0]/(k + 1) for k in range(len(threads))]
    plt.title('Execution time', fontsize=20)
    plt.plot(threads, expected_time, 'r--')
    plt.plot(threads, time_average, 'b')
    plt.xlabel('Threads')
    plt.ylabel('Time')
    plt.grid(1)
    plt.legend(['Expected time', 'Experimental time'])
    plt.show()

    s = [(sum(times[0])/len(times[0]))/(sum(k)/len(k)) for k in times]
    expected_s = [time_average[0]/k for k in expected_time]
    plt.title('Acceleration', fontsize=20)
    plt.plot(threads, expected_s, 'r--')
    plt.plot(threads, s, 'b')
    plt.xlabel('Threads')
    plt.ylabel('Acceleration')
    plt.grid(1)
    plt.legend(['Expected acceleration', 'Experimental acceleration'])
    plt.show()

    e = [s[k]/(k + 1) for k in range(len(s))]
    expected_e = [expected_s[k]/(k + 1) for k in range(len(s))]
    plt.title('Efficiency', fontsize=20)
    plt.plot(threads, expected_e, 'r--')
    plt.plot(threads, e, 'b')
    plt.xlabel('Threads')
    plt.ylabel('Efficiency')
    plt.grid(1)
    plt.legend(['Expected efficiency', 'Experimental efficiency'])
    plt.show()

def table(times, threads):
    table = ColorTable(theme=Themes.OCEAN)
    table.field_names = ['Thread'] + [i+1 for i in range(len(times[0]))]
    for i in range(len(threads)):
        times[i].insert(0, i+1)
    for i in range(len(times)):
        for j in range(len(times[i])):
            times[i][j] = round(times[i][j], 5)
    table.add_rows(times)
    print(table)

if __name__ == '__main__':
    exp = inp()
    plots(exp[0], exp[1])
    table(exp[0], exp[1])
```

4. Графики и таблица

• **Время от числа потоков** – теоретически функция имеет вид:

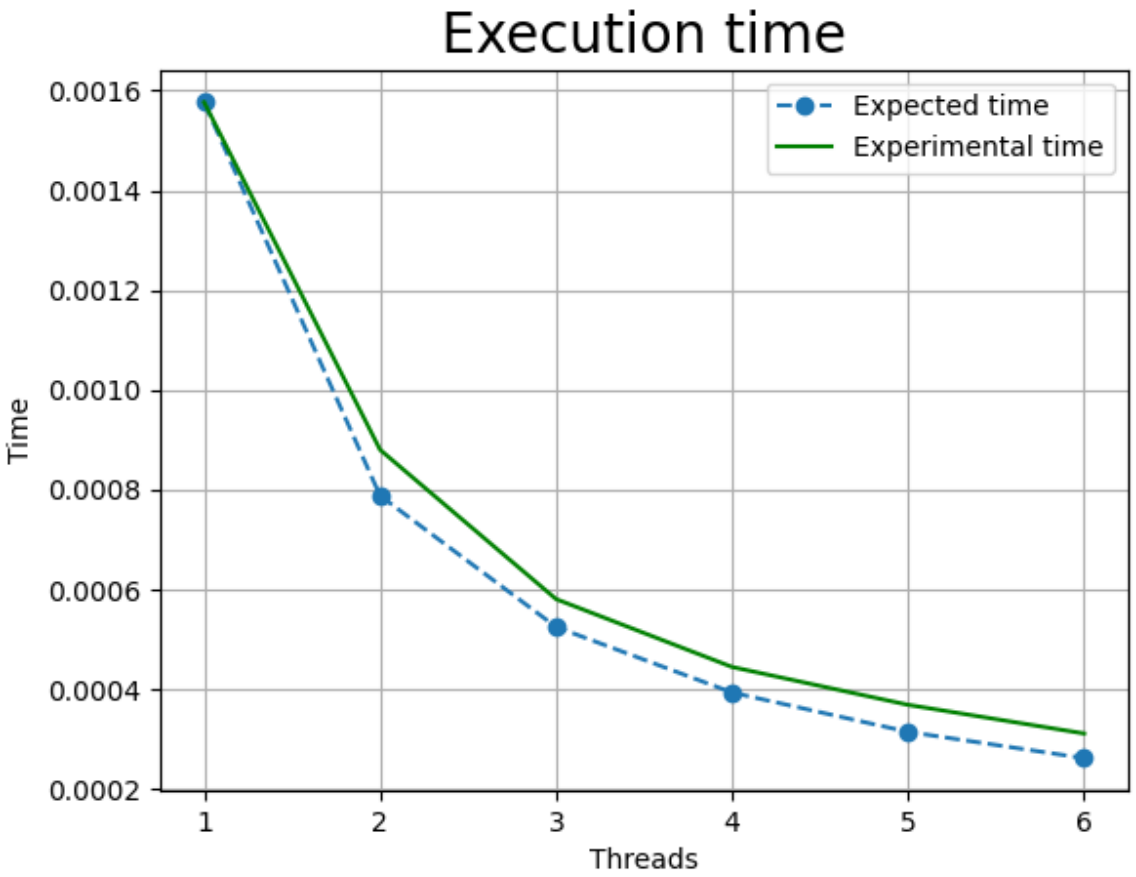
$$T_P = \alpha T_1 + \frac{(1 - \alpha)T_1}{p}$$

где α - доля последовательных операций в алгоритме, T_1 - время работы на одном потоке, а p - количество потоков. Однако в нашем случае (α = 0) эту формулу можно упростить до:

$$T_P = \frac{T_1}{p}$$

Экспериментальный результат был усреднен по 10 итерациям на случайных входных данных.

График времени от числа потоков

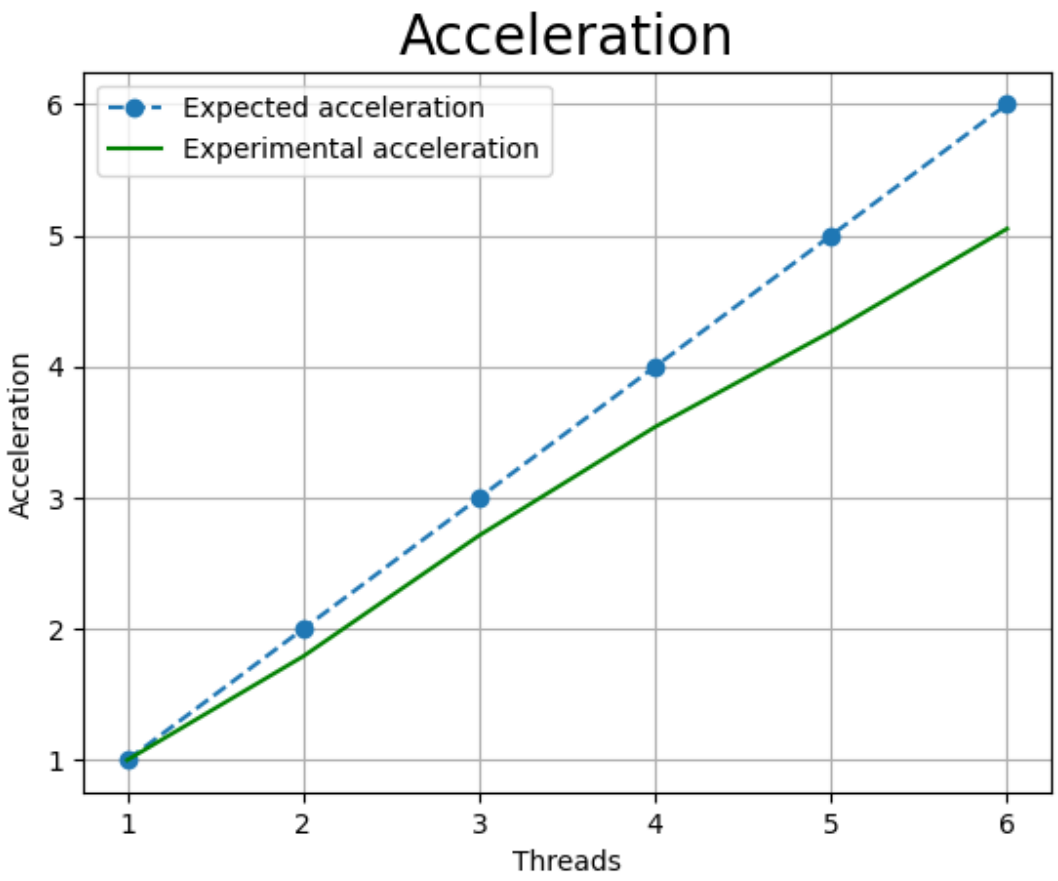


- **Ускорение от числа потоков** – ускорением параллельного алгоритма называют отношение времени выполнения лучшего последовательного алгоритмам к времени выполнения параллельного алгоритма:

$$S = \frac{T_1}{T_p}$$

где T_1 - время работы на одном потоке, а T_p - время работы алгоритма на p потоках. Экспериментальный результат был усреднен по 10 итерациям на случайных входных данных.

График ускорения от числа потоков



• **Эффективность от числа потоков** – параллельный алгоритм может давать большое ускорение, но использовать для этого множество процессов неэффективно. Для оценки масштабируемости параллельного алгоритма используется понятие эффективности:

$$E = \frac{S}{p}$$

где S - Ускорение от числа потоков, p - количество потоков. Экспериментальный результат был усреднен по 10 итерациям на случайных входных данных.

График эффективности от числа потоков

- **Таблица**

Thread	1	2	3	4	5	6	7	8	9	10
1	0.0016	0.00159	0.00161	0.00157	0.00149	0.00154	0.0016	0.00155	0.00154	0.00167
2	0.00085	0.00088	0.00094	0.00084	0.00085	0.00095	0.00085	0.00085	0.00092	0.00086
3	0.00064	0.0006	0.00057	0.00063	0.00056	0.00055	0.00056	0.00057	0.00057	0.00056
4	0.00041	0.00046	0.00047	0.00045	0.00044	0.00043	0.00043	0.00047	0.00044	0.00045
5	0.00038	0.00043	0.00036	0.00034	0.00037	0.00034	0.00034	0.00037	0.0004	0.00038
6	0.0003	0.00037	0.0003	0.00033	0.0003	0.00033	0.00032	0.00029	0.0003	0.0003

5. Заключение

В этой лабораторной работе я познакомился с основными принципами работы с **OpenMP** и приобрел базовые навыки теоретического и экспериментального анализа высокопроизводительных параллельных алгоритмов, построения параллельных программ, обнаружения ресурса параллелизма в имеющейся последовательной реализации.