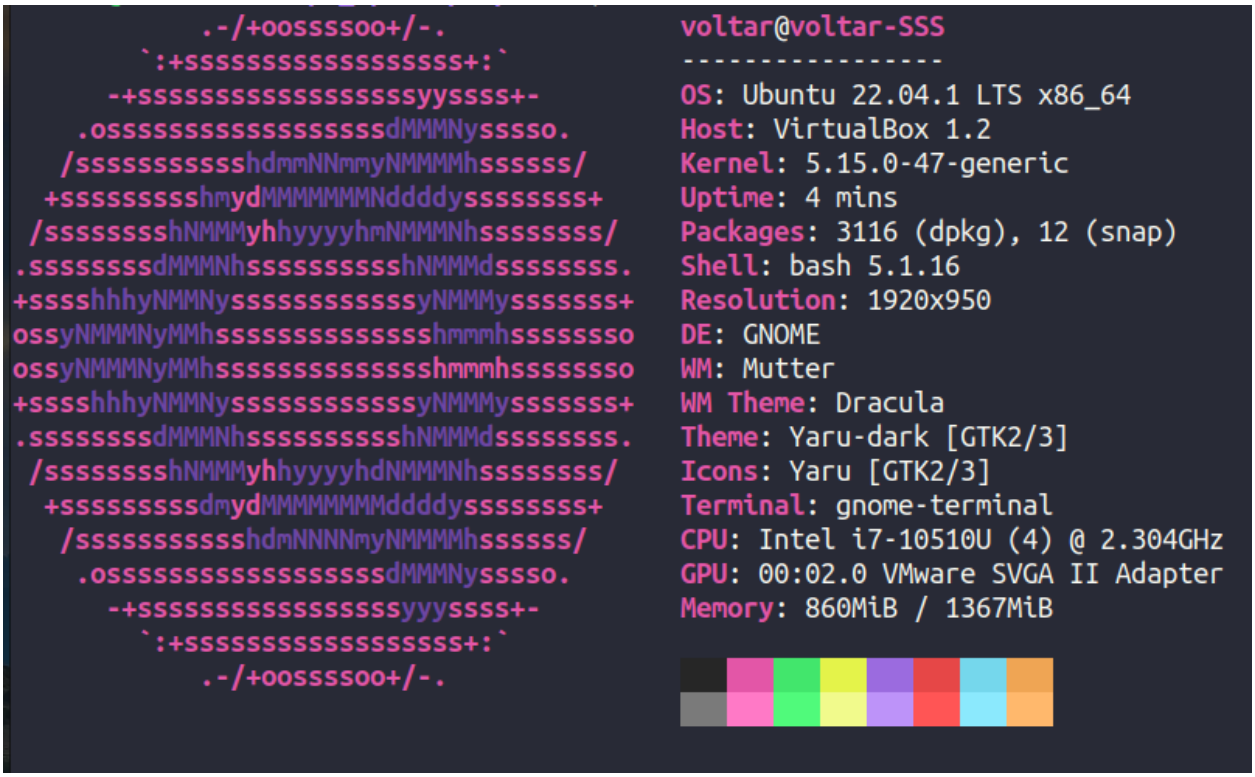
Национальный исследовательский ядерный университет “МИФИ”

Лабораторная работа №1: “Введение в параллельные вычисления. Технология OpenMp”

Зимич Григорий
Б20-505
2022 год

1 Описание используемой рабочей среды

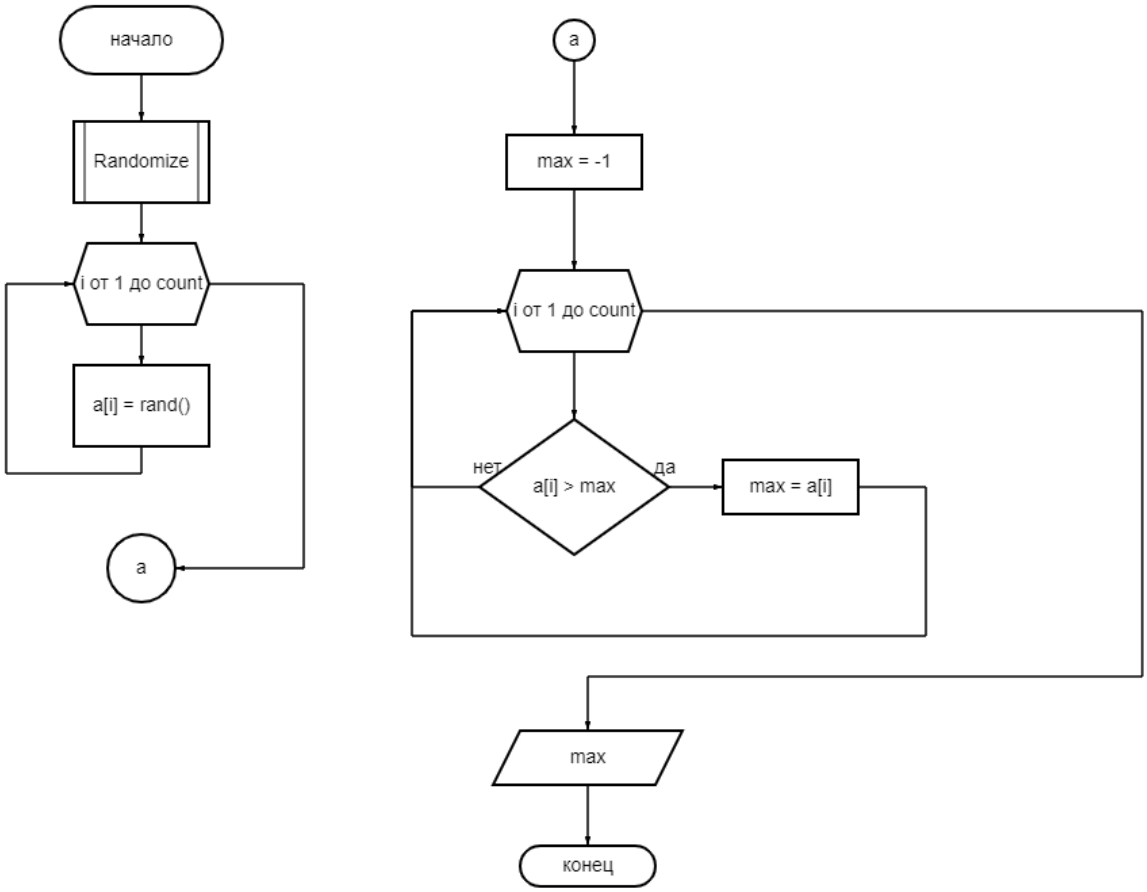



```
voltar@voltar-SSS:~$ echo |cpp -fopenmp -dM |grep -i open
#define _OPENMP 201511
```

201511, it means Nov 2015, and the version is openmp 4.5

2 Анализ алгоритма

Алгоритм работает за O(N)



Директива *parallel*

```
#pragma omp parallel num_threads(threads) shared(array, count)
   reduction(max: max) 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

3 Код

- **makefile**

```
default: build

.PHONY: build
build:
	gcc lab1.c app.c -fopenmp -o out
```

```
.PHONY: clean
clean:
	rm out
```

- **app.h**

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define THREADS 4

#pragma once

void f(int rand_seed,float *times);
```

- **app.c**

```
#include "headers/app.h"

void f(int rand_seed, float *times)
{
  const int count = 10000000;

  int *array = calloc(count, sizeof(int));
  int threads = THREADS;

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

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

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

```
    int max = -1;

    start_time = omp_get_wtime();
    #pragma omp parallel num_threads(i + 1) shared(array,count,
       i) reduction(max: max) default(none)
    {
      #pragma omp for
      for(int j = 0; j < count; j++)
      {
        if(array[j] > max) { max = array[j]; };
      }
    }
    end_time = omp_get_wtime();
    times[i] = end_time - start_time;
  }

  free(array);
}
```

- **lab1.c**

```
#include "headers/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;
}
```

- **pon.py**

```
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 ($\alpha = 0$) эту формулу можно упростить до:

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

Экспериментальный результат был усреднен по 10 итерациям на рандомных входных данных

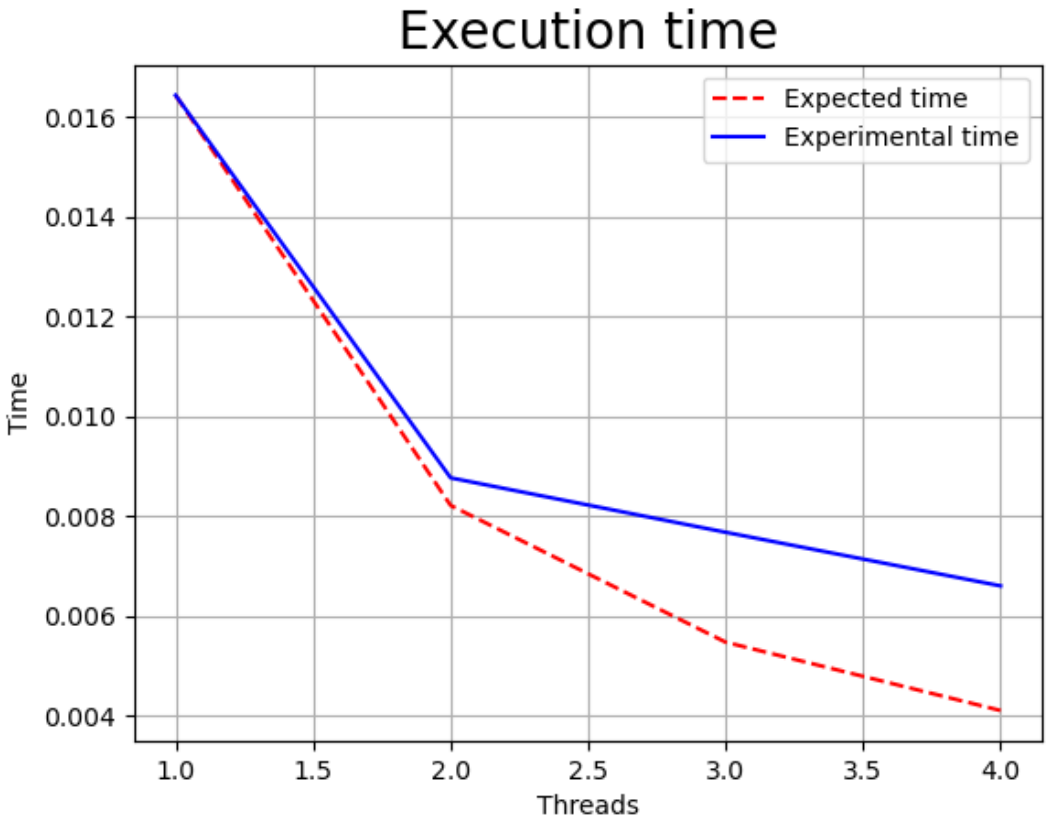


График времени от числа потоков

- **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 рандомных входных данных

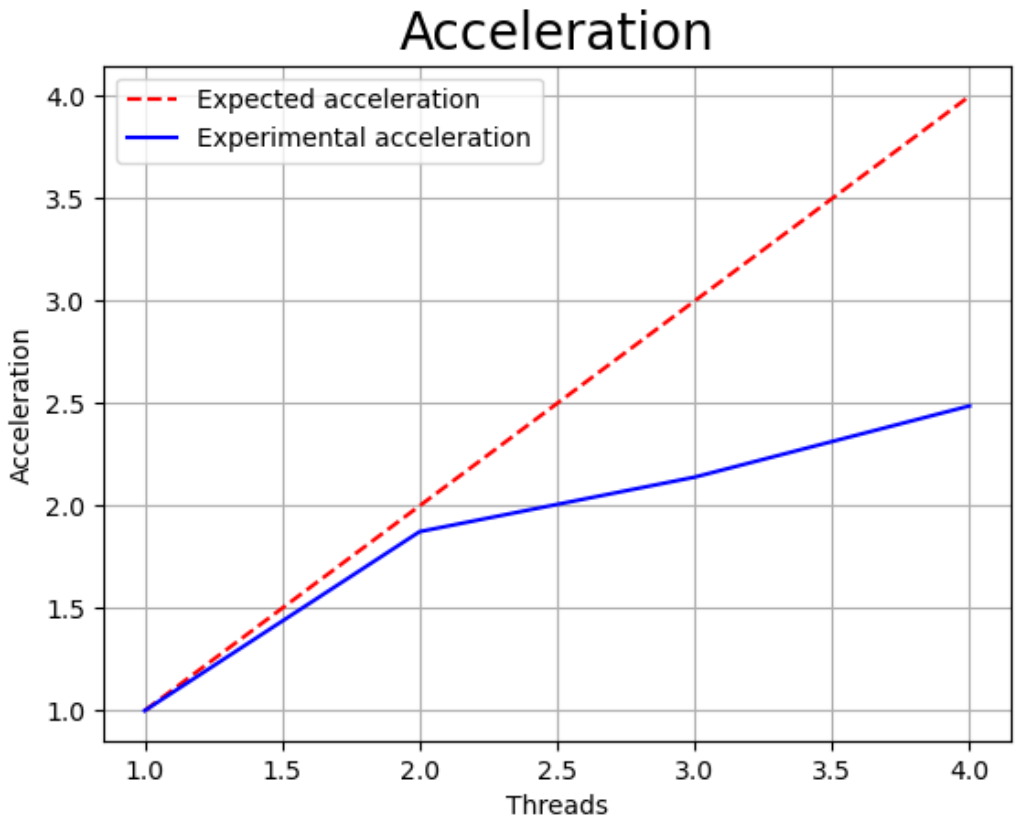


График ускорения от числа потоков

- **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 рандомных входных данных

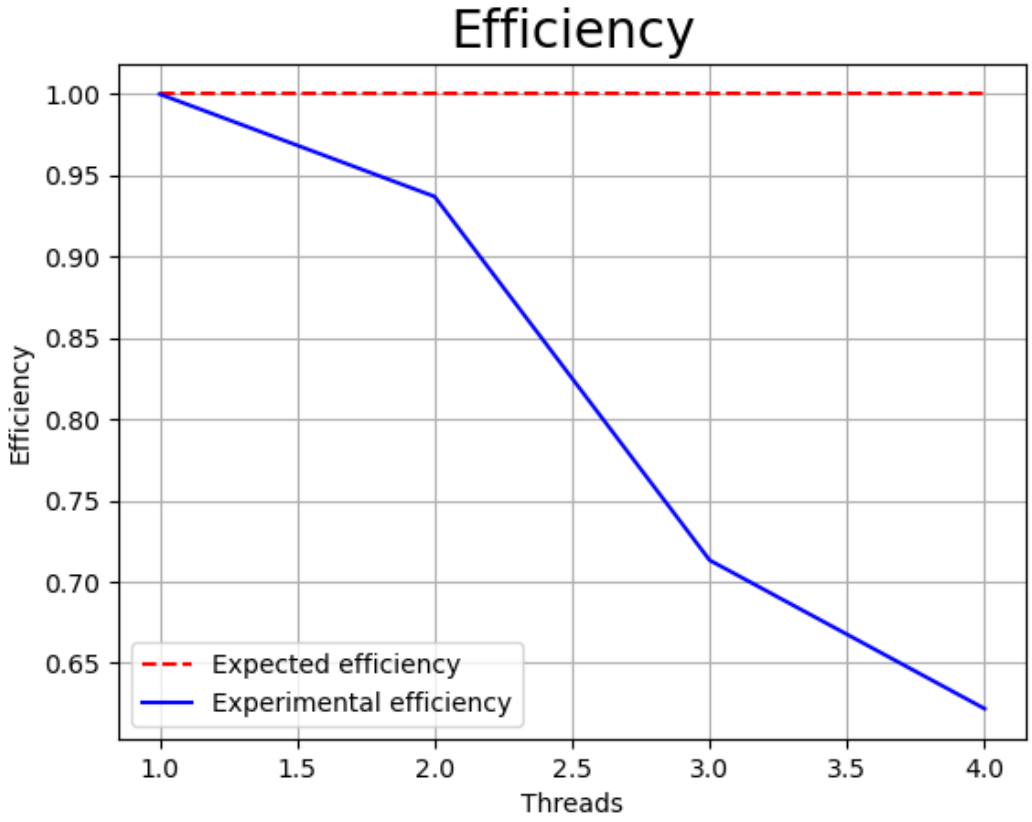


График эффективности от числа потоков

- **Таблица**

Thread	1	2	3	4	5	6	7	8	9	10
1	0.0163	0.01607	0.0163	0.01773	0.01624	0.01634	0.01615	0.01633	0.01656	0.01634
2	0.0085	0.0083	0.00829	0.00828	0.00831	0.00864	0.00818	0.00858	0.00866	0.01199
3	0.0061	0.00954	0.00997	0.00606	0.00668	0.00625	0.00697	0.00624	0.00969	0.0093
4	0.00453	0.00557	0.00571	0.00807	0.00818	0.0081	0.00818	0.00453	0.00551	0.00768

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.