Национальный исследовательский ядерный университет “МИФИ”

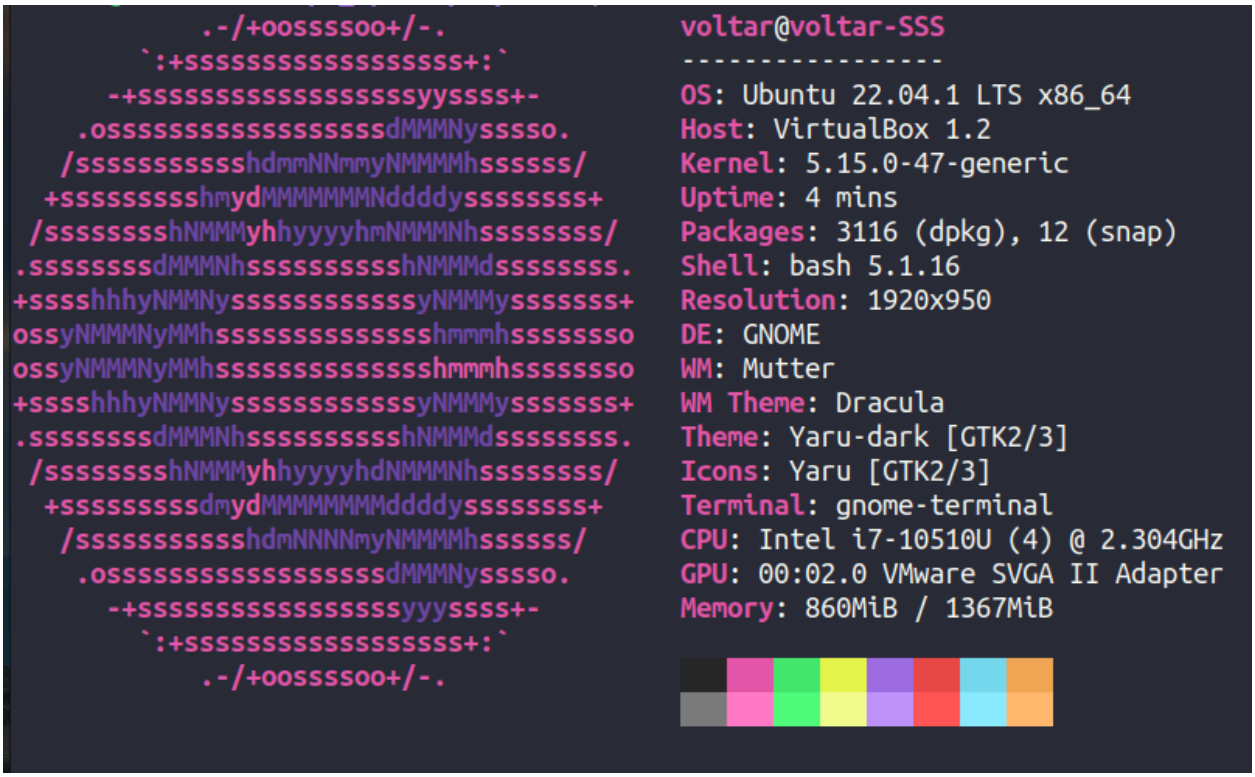
Лабораторная работа №2: “Выделение ресурса параллелизма. Технология OpenMP”

Зимич Григорий

Б20-505

2022 год

1 Описание используемой рабочей среды



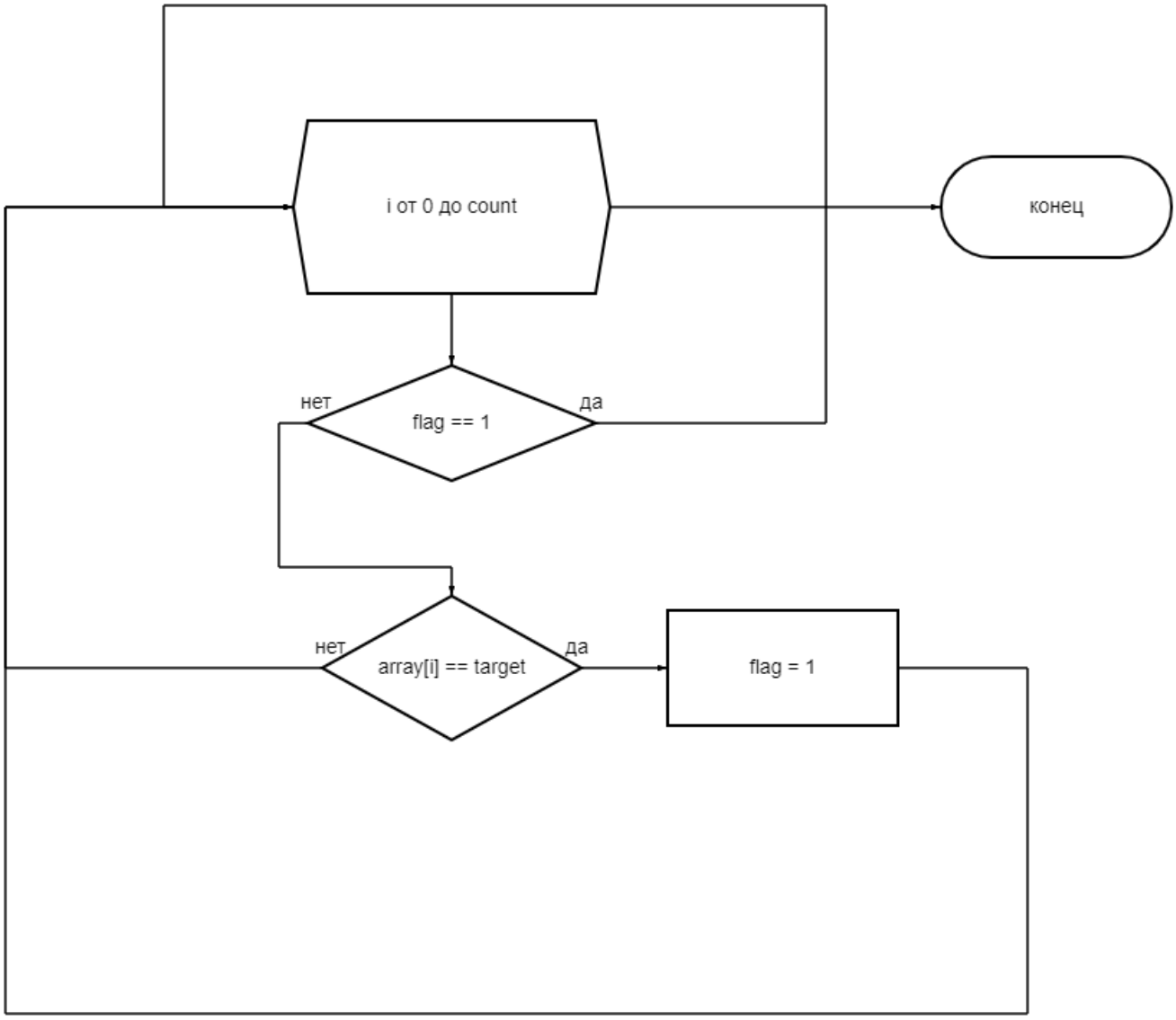

```
voltar@voltar-SSS:~$ echo |cpp -fopenmp -dM |grep -i open
#define _OPENMP 201511
```

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

2 Анализ алгоритма

Алгоритм работает за $O(N)$ в худшем случае, в лучшем $O(1)$. В среднем случае:

$$\frac{1+2+3+...+n}{n+1} = \frac{(n+1)n}{2(n+1)} = \frac{n}{2} \implies O(n)$$



Директива *parallel*

```
#pragma omp parallel num_threads(i + 1) shared(array,count, i,
   target, flag) 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;
  $array, count, i, target, flag$ - те переменные, которые нужны для реализации алгоритма в параллельной области программы
- *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 lab2.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; ///< Number of array elements

  int *array = calloc(count, sizeof(int)); ///< The array we
     need to find the max in
  int threads = THREADS;

  int target;
  int flag;

  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++)
  {
    flag = 0;
    target = rand() % count;

    start_time = omp_get_wtime();
    #pragma omp parallel num_threads(i + 1) shared(array,count,
       i, target, flag) default(none)
    {
      #pragma omp for
      for(int j = 0; j < count; j++)
      {
        if(flag) continue;
        if(array[j] == target) flag = 1;
      }
      printf("--␣My␣target␣for␣%d␣threads␣is:␣%d;\n", i + 1,
         target);
    }
    end_time = omp_get_wtime();
    times[i] = end_time - start_time;
    printf("======\ntarget␣is:␣%d␣for␣threads:␣%d,␣time:␣%f\n",
       target, i + 1, times[i]);
  }

  free(array);
}
```

- **lab2.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 < iter; 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

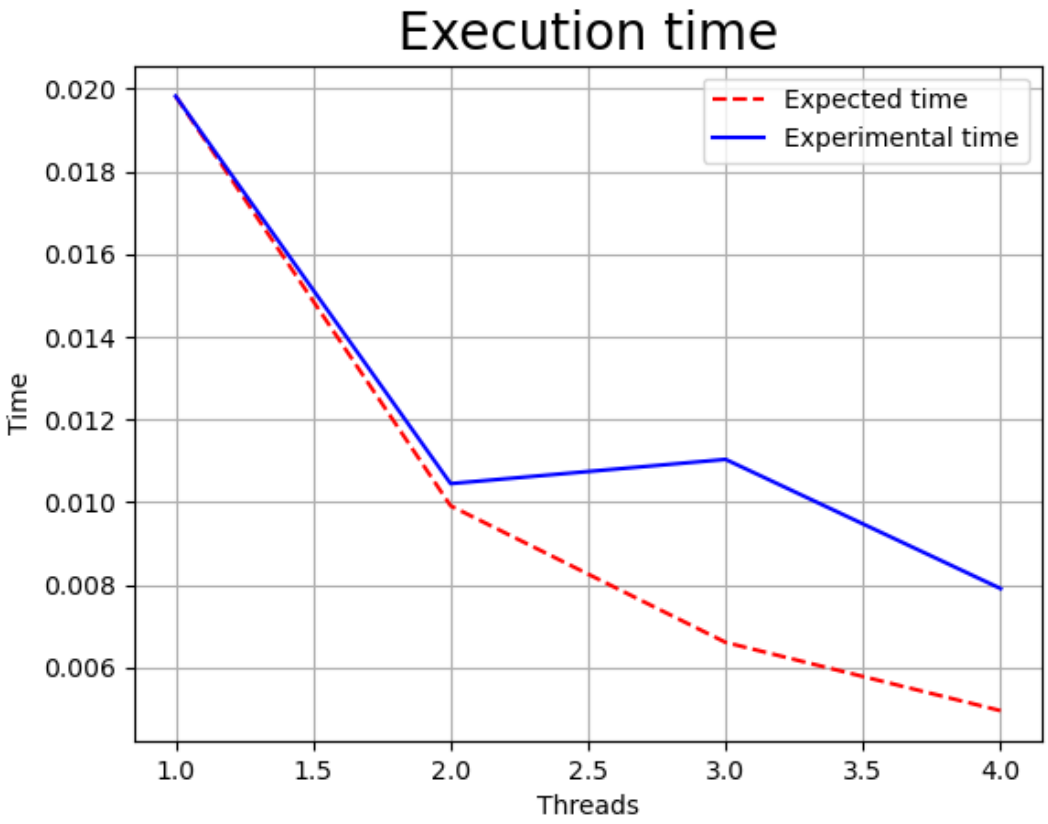


График времени от числа потоков

- **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

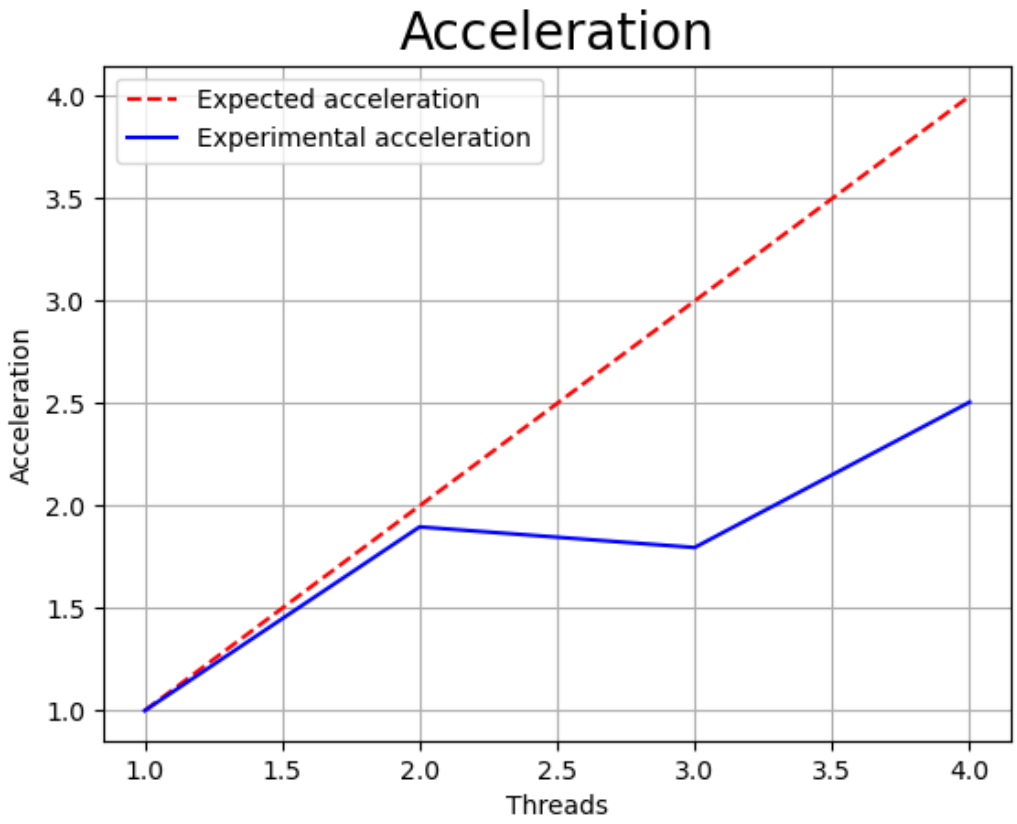


График ускорения от числа потоков

- **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

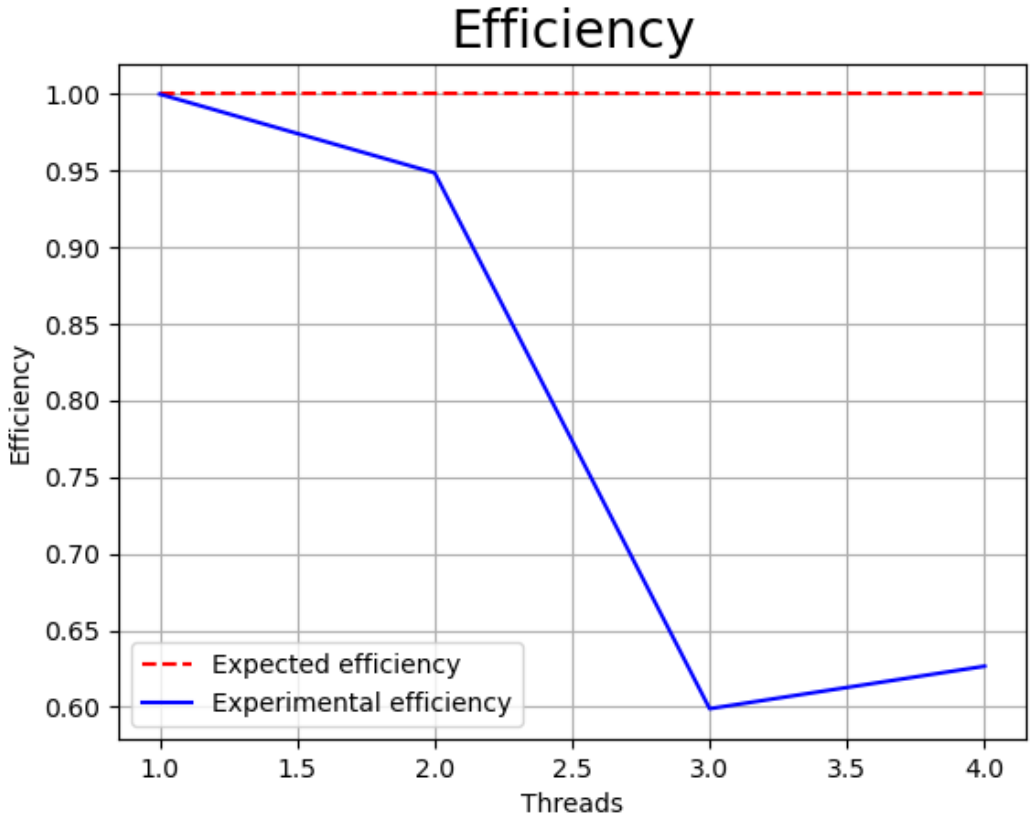


График эффективности от числа потоков

- **Таблица**

Thread	1	2	3	4	5	6	7	8	9	10
1	0.01953	0.01953	0.02002	0.02051	0.02002	0.02002	0.01953	0.01904	0.01953	0.02051
2	0.00977	0.00977	0.01025	0.01025	0.01221	0.00977	0.01025	0.01025	0.00977	0.01221
3	0.00977	0.00977	0.01123	0.0127	0.01611	0.01123	0.01123	0.00928	0.01123	0.00781
4	0.00684	0.00781	0.00732	0.0083	0.01025	0.01123	0.0083	0.00537	0.0083	0.00537

5 Заключение

В этой лабораторной работе я познакомился с новыми принципами работы с **OpenMP** и приобрел навыки разработки параллельной программы путём обнаружения ресурса параллелизма в имеющейся последовательной реализации.