Национальный исследовательский ядерный университет “МИФИ”Лабораторная работа №2: «Выделение ресурса параллелизма. Технология OpenMP»

Шанкин Данила Б20-505

2022 год

1. Описание рабочей среды

```
danila@Danila-PK ~/Рабочий стол/parallel_lab1 neofetch                                   master
             ...-:::::-...                     danila@Danila-PK
          .-MMMMMMMMMMMMMMM-.                  ----------------
      .-MMMM`..-:::::::-..`MMMM-.              OS: Linux Mint 20.2 x86_64
    .:MMMM.:MMMMMMMMMMMMMMM:.MMMM:.            Host: NBLK-WAX9X M1020
   -MMM-M---MMMMMMMMMMMMMMMMMMM.MMM-           Kernel: 5.4.0-74-generic
 `:MMM:MM`  :MMMM:....::-...-MMMM:MMM:`        Uptime: 1 hour, 21 mins
 :MMM:MMM`  :MM:`  ``    ``  `:MMM:MMM:        Packages: 2663 (dpkg), 9 (flatpak)
.MMM.MMMM`  :MM.  -MM.  .MM-  `MMMM.MMM.       Shell: zsh 5.8
:MMM:MMMM`  :MM.  -MM-  .MM:  `MMMM-MMM:       Resolution: 1920x1080
:MMM:MMMM`  :MM.  -MM-  .MM:  `MMMM:MMM:       DE: Cinnamon
:MMM:MMMM`  :MM.  -MM-  .MM:  `MMMM-MMM:       WM: Mutter (Muffin)
.MMM.MMMM`  :MM:--:MM:--:MM:  `MMMM.MMM.       WM Theme: Mint-Y-Purple (Mint-X)
 :MMM:MMM-  `-MMMMMMMMMMMM-`  -MMM-MMM:        Theme: Mint-Y [GTK2/3]
  :MMM:MMM:`                `:MMM:MMM:         Icons: Mint-Y-Teal [GTK2/3]
   .MMM.MMMM:--------------:MMMM.MMM.          Terminal: gnome-terminal
     '-MMMM.-MMMMMMMMMMMMMMM-.MMMM-'           CPU: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx (8) @ 2.100GHz
       '.-MMMM``--:::::--``MMMM-.'             GPU: AMD ATI 03:00.0 Picasso
            '-MMMMMMMMMMMMM-'                  Memory: 2527MiB / 6951MiB
               ``-:::::-``
```

```
danila@Danila-PK ~/Рабочий стол/parallel_lab1 echo | cpp -fopenmp -dM | grep -i OPENMP     master
#define _OPENMP 201511
```

2. Анализ алгоритма

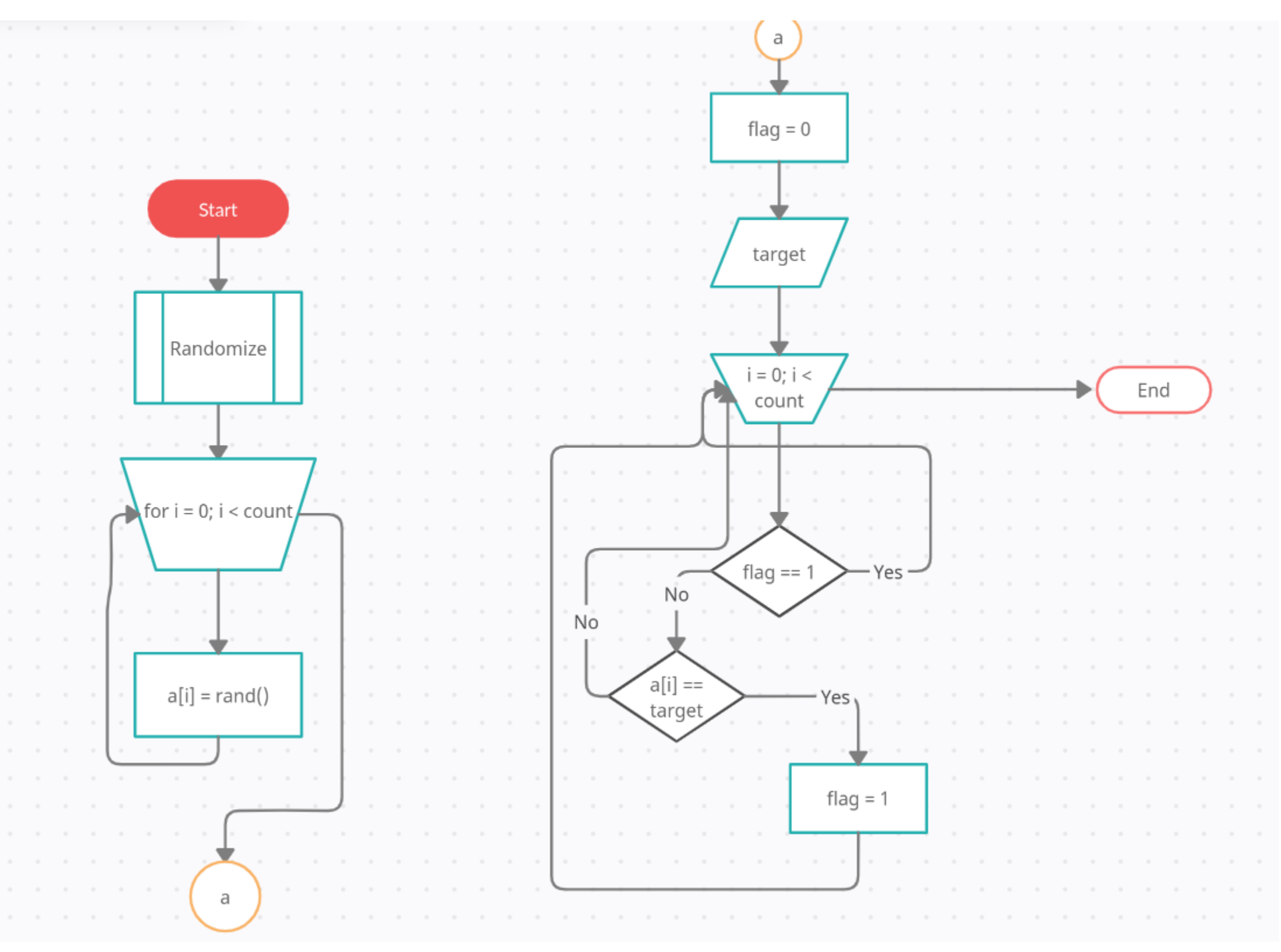


Схема алгоритма:

Алгоритм работает с максимальной ассимптотикой $O(N)$ Алгоритм работает с минимальной ассимптотикой $O(1)$

3. 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(int)` – 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(list)` – задаёт список переменных, общих для всех потоков;
- `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

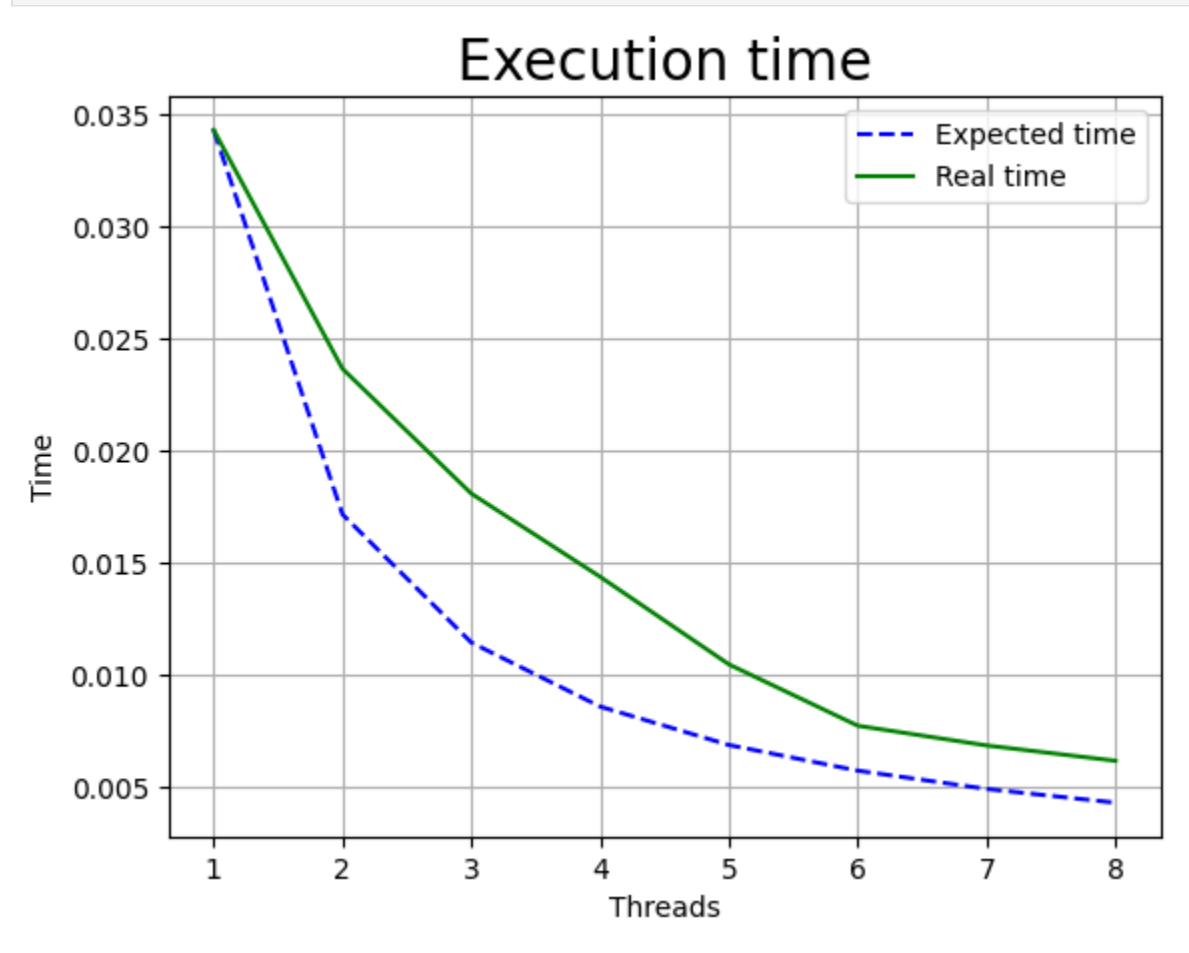
3. Код lab1.c

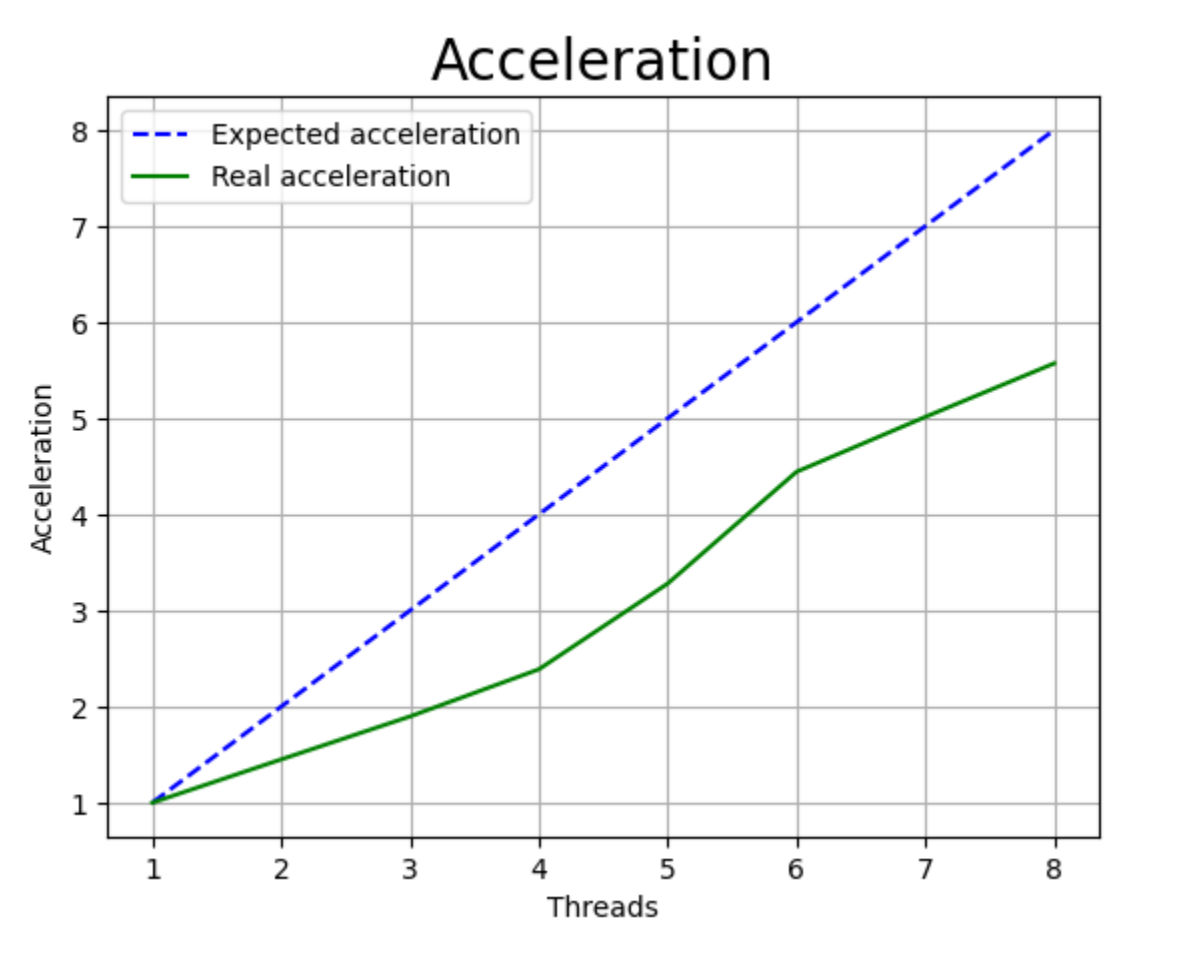
In [ ]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int threads = 8;

void function(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 target,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; }
                         }
                }
        end_time = omp_get_wtime();
        times[i] = end_time - start_time;
        }
    free(array);
}

int main(int argc, char** argv){
    int iter = 10;
    int threads = 8;
    int seed = 93932;
    FILE *file = fopen("data.txt", "w");
    fwrite(&threads, sizeof(int), 1, file);
    fwrite(&iter, sizeof(int), 1, file);
    float *times = 0;
    for(int i = 0; i < iter; i++){
        times = (float*)calloc(threads, sizeof(float));
        function(seed + i, times);
        fwrite(times, sizeof(float), threads, file);
        free(times);
    }
    fclose(file);
    return 0;
}
```

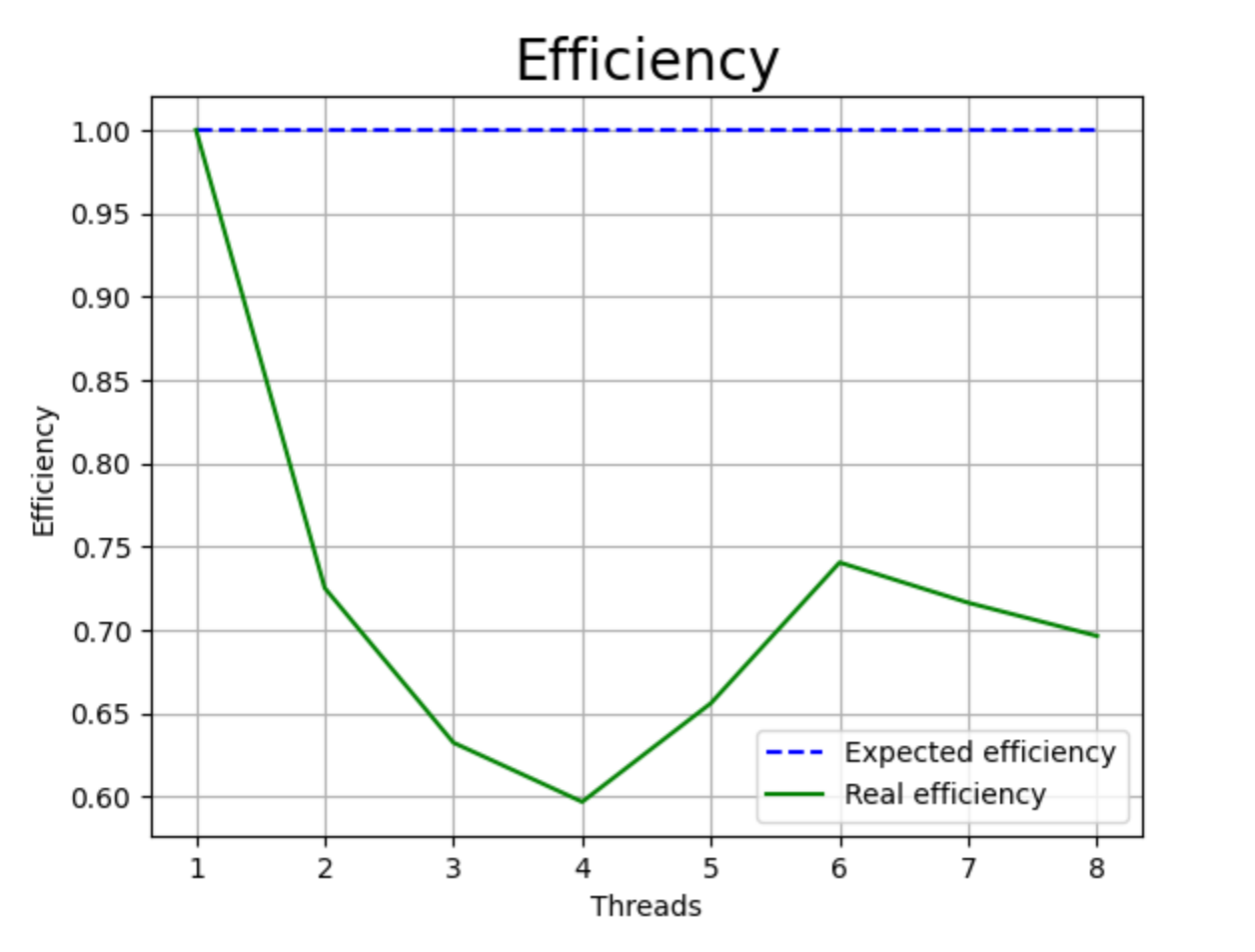
4. Код graph.py, графики и таблица

In [20]:
```
!cp 'Рабочий стол'/parallel_lab2/data.txt ./
```

In [2]:
```python
from pwn import *
from prettytable.colortable import ColorTable, Themes
import matplotlib.pyplot as plt
import numpy as np

def inp(name):
        data = open(name,'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, num_of_threads

def plots(times, threads):
    #1
        t1 = [sum(k)/len(k) for k in times] #
        expected_time = [t1[0]/(k + 1) for k in range(len(threads))]
        plt.title('Execution time', fontsize=20)
        plt.plot(threads, expected_time, 'b--')
        plt.plot(threads, t1, 'g')
        plt.xlabel('Threads')
        plt.ylabel('Time')
        plt.grid(1)
        plt.legend(['Expected time', 'Real time'])
        plt.show()
    #2
        s = [(sum(times[0])/len(times[0]))/(sum(k)/len(k)) for k in times]
        expected_s = [t1[0]/k for k in expected_time] #prefer
        plt.title('Acceleration', fontsize=20)
        plt.plot(threads, expected_s, 'b--')
        plt.plot(threads, s, 'g')
        plt.xlabel('Threads')
        plt.ylabel('Acceleration')
        plt.grid(1)
        plt.legend(['Expected acceleration', 'Real acceleration'])
        plt.show()
    #3
        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, 'b--')
        plt.plot(threads, e, 'g')
        plt.xlabel('Threads')
        plt.ylabel('Efficiency')
        plt.grid(1)
        plt.legend(['Expected efficiency', 'Real efficiency'])
        plt.show()

def table(times, threads):
        table = ColorTable(theme=Themes.DEFAULT)
        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('data.txt')
        times = exp[0]
        threads = exp[1]
        num_threads = exp[2]
        plots(times,threads)
        table(exp[0], exp[1])
```







Thread	1	2	3	4	5	6	7	8	9	10
1	0.03418	0.03418	0.03516	0.03418	0.03418	0.03418	0.03418	0.03418	0.03418	0.03418
2	0.02344	0.02344	0.02539	0.02637	0.02148	0.02246	0.02148	0.02344	0.02441	0.02441
3	0.01855	0.01855	0.0166	0.01758	0.0166	0.01953	0.01953	0.01758	0.0166	0.01953
4	0.0166	0.01465	0.01562	0.01074	0.01465	0.01465	0.01465	0.0127	0.01465	0.01465
5	0.0127	0.00879	0.01074	0.00879	0.01074	0.01074	0.00977	0.0127	0.01074	0.00879
6	0.01074	0.00781	0.00781	0.00684	0.00781	0.00684	0.00684	0.00684	0.00781	0.00781
7	0.00879	0.00586	0.00781	0.00586	0.00586	0.00879	0.00586	0.00781	0.00586	0.00586
8	0.00781	0.00586	0.00488	0.00488	0.00586	0.00977	0.00586	0.00586	0.00488	0.00586

5. Теоретический анализ

- *Время от числа потоков:* $T_p = \alpha T_1 + (1-\alpha)T_1/p$, 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: $T_p = T_1/p$. Экспериментальный результат был усреднен по 10 итерациям на рандомных входных данных
- *Ускорение от числа потоков* - Ускорением параллельного алгоритма называют отношение времени выполнения лучшего последовательного алгоритмам к времени выполнения параллельного алгоритма: $S = T_1/T_p$, 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
- *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: $E = S/p$, где S - Ускорение от числа потоков, p - количество потоков. Экспериментальный результат был усреднен по 10 итерациям на рандомных входных данных

6. Заключение В этой лабораторной работе я познакомился с новыми принципами работы с OpenMP и приобрел навыки разработки парал- лельной программы путём обнаружения ресурса параллелизма в имеющейся последовательной реализации.

Для защиты

`#pragma omp cancel construct-type-clause [ [,] if-clause]` - Конструкция `cancel` активирует отмену самой внутренней вмещающей области указанного типа. Конструкция отмены является автономной директивой.

Обновленный цикл:

```c
#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(array[j] == target) {
                  #pragma omp cancel for
              }
          #pragma omp cancellation point for
              }
}
```

In [15]:
```
!cp 'Рабочий стол'/parallel_lab2/data_cancel.txt ./
!cp 'Рабочий стол/'/parallel_lab2/data1.txt ./
```

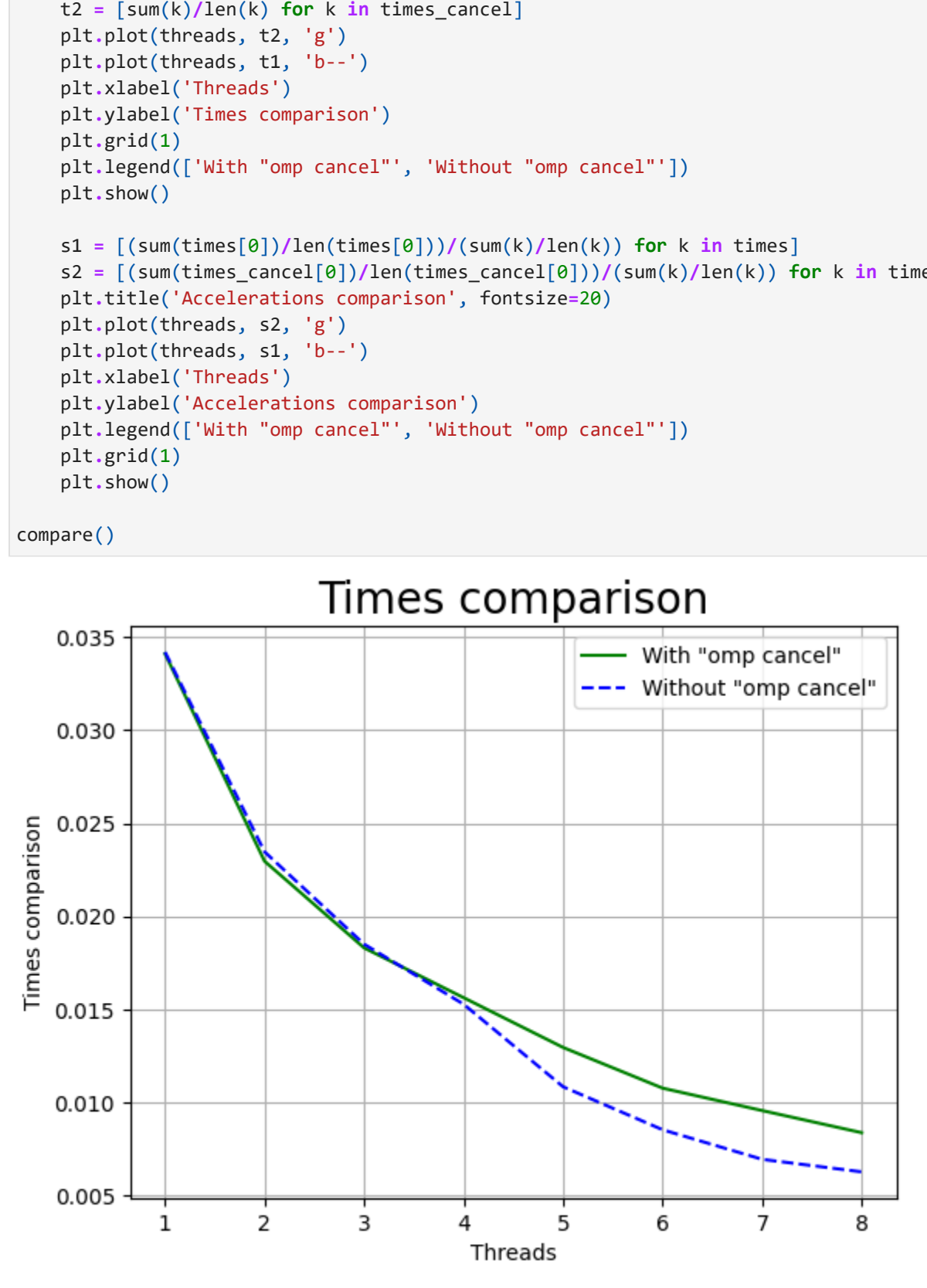
Таблица исполнения процесса с директивой

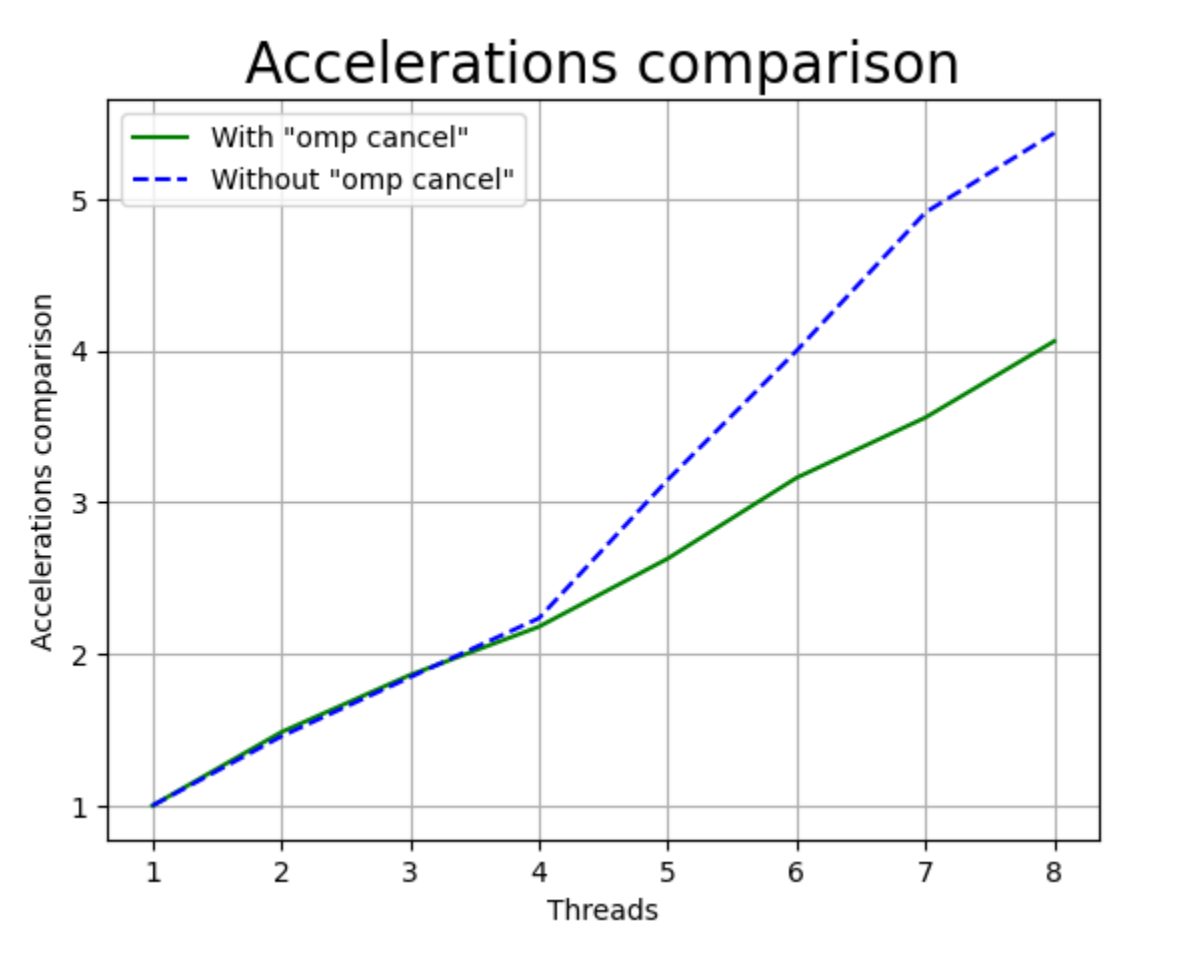
In [16]:
```python
exp_cancel = inp('data_cancel.txt')
table(exp_cancel[0], exp_cancel[1])
```

Thread	1	2	3	4	5	6	7	8	9	10
1	0.03413	0.03411	0.03403	0.03405	0.03405	0.03401	0.03413	0.03433	0.03404	0.03399
2	0.0251	0.01841	0.02521	0.02578	0.02264	0.01892	0.0234	0.02521	0.02054	0.02436
3	0.01902	0.01705	0.01897	0.01785	0.01957	0.01935	0.01596	0.01896	0.01777	0.01857
4	0.01662	0.01693	0.01303	0.01361	0.01646	0.01698	0.01566	0.01634	0.01433	0.01638
5	0.01503	0.01514	0.01049	0.01236	0.01282	0.013	0.01386	0.01358	0.01039	0.01295
6	0.01371	0.00939	0.0088	0.01284	0.00935	0.01247	0.01307	0.00874	0.01009	0.00931
7	0.00945	0.01073	0.0074	0.01148	0.00781	0.00824	0.01172	0.01041	0.00761	0.01091
8	0.00945	0.00653	0.00647	0.00922	0.00706	0.00971	0.00972	0.00647	0.00925	0.01

Сравнение времени исполнения без директивы и с директивой

In [17]:
```python
def compare():
    data = inp('data1.txt')
    data_cancel = inp('data_cancel.txt')
    times = data[0]
    times_cancel = data_cancel[0]
    threads = data[1]
    plt.title('Times comparison', fontsize=20)
    t1 = [sum(k)/len(k) for k in times]
    t2 = [sum(k)/len(k) for k in times_cancel]
    plt.plot(threads, t2, 'g')
    plt.plot(threads, t1, 'b--')
    plt.xlabel('Threads')
    plt.ylabel('Times comparison')
    plt.grid(1)
    plt.legend(['With "omp cancel"', 'Without "omp cancel"'])
    plt.show()

    s1 = [(sum(times[0])/len(times[0]))/(sum(k)/len(k)) for k in times]
    s2 = [(sum(times_cancel[0])/len(times_cancel[0]))/(sum(k)/len(k)) for k in times_cancel]
    plt.title('Accelerations comparison', fontsize=20)
    plt.plot(threads, s2, 'g')
    plt.plot(threads, s1, 'b--')
    plt.xlabel('Threads')
    plt.ylabel('Accelerations comparison')
    plt.legend(['With "omp cancel"', 'Without "omp cancel"'])
    plt.grid(1)
    plt.show()

compare()
```





Как видно из графика, при использовании данной директивы время выполнения программы уменьшилось, а ускорение, соответственно увеличилось