Лабораторная работа №3:
Реализация алгоритма с использованием технологии OpenMP

Барабанов Андрей
Б20-505

Содержание

1 Рабочая среда

1.1 Модель процессора

```
~ 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:  56%
    CPU max MHz:         5000,0000
    CPU min MHz:         800,0000
    BogoMIPS:            5202,65
```

1.2 Версия OpenMP и GCC

```
~ echo | cpp -fopenmp -dM | grep -i open
#define _OPENMP 201511
~ gcc --version
gcc (GCC) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
Это свободно распространяемое программное обеспечение. Условия копирования
приведены в исходных текстах.

  Без гарантии каких-либо качеств, включая
коммерческую ценность и применимость для каких-либо целей.
```

2 Анализ

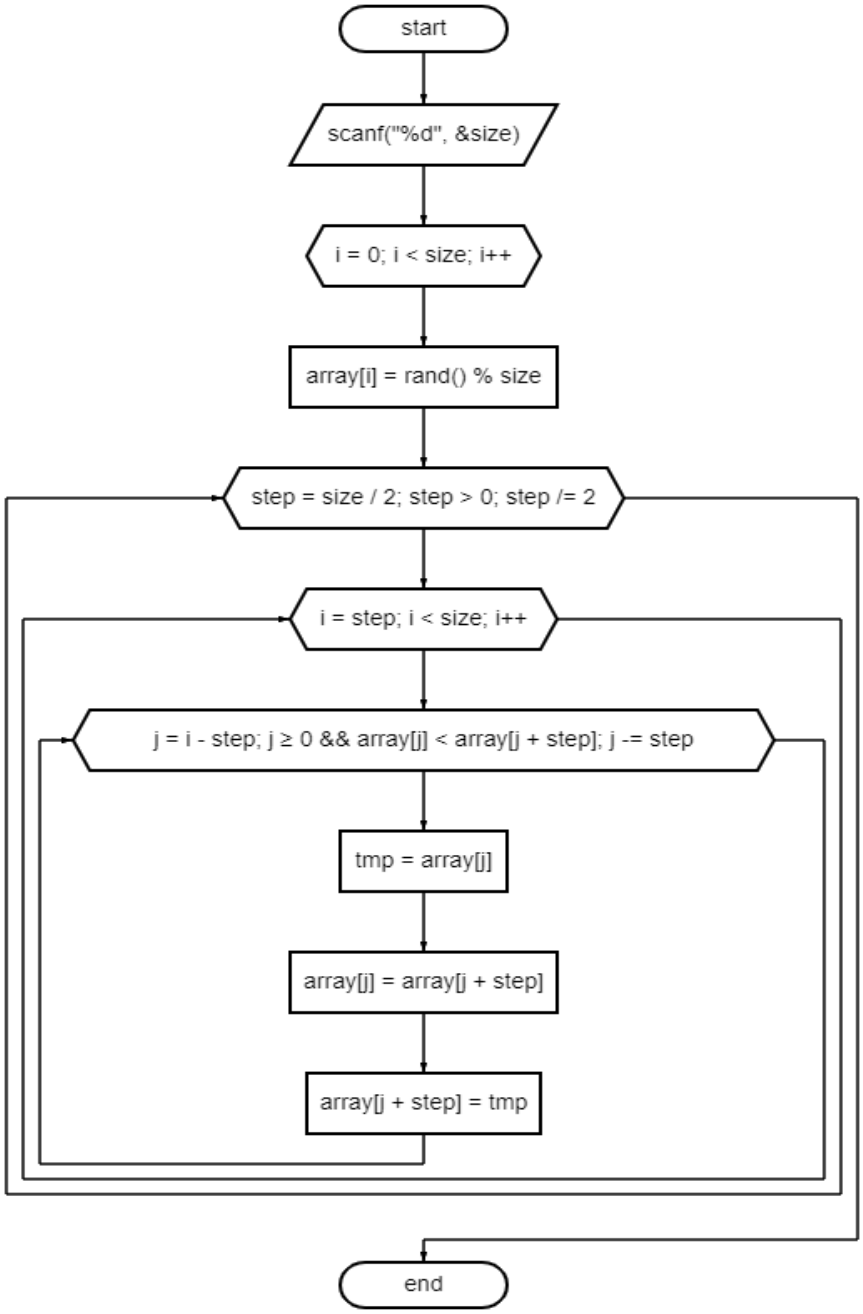
2.1 Описание сортировки Шелла

При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d. После этого процедура повторяется для некоторых меньших значений d, а завершается сортировка Шелла упорядочиванием элементов при $d = 1$ (то есть обычной сортировкой вставками).

Таким образом, имеет смысл распараллелить алгоритм после выбора определённого числа d путём сортировки чисел, отстоящих друг от друга на расстоянии d с помощью следующей директории $OpenMP$:

```
#pragma omp parallel for num_threads(threads) shared(tmp_array, size, step) default(none)
```

2.2 Блок-схема алгоритма



3 Графики

3.1 Частично отсортированный массив

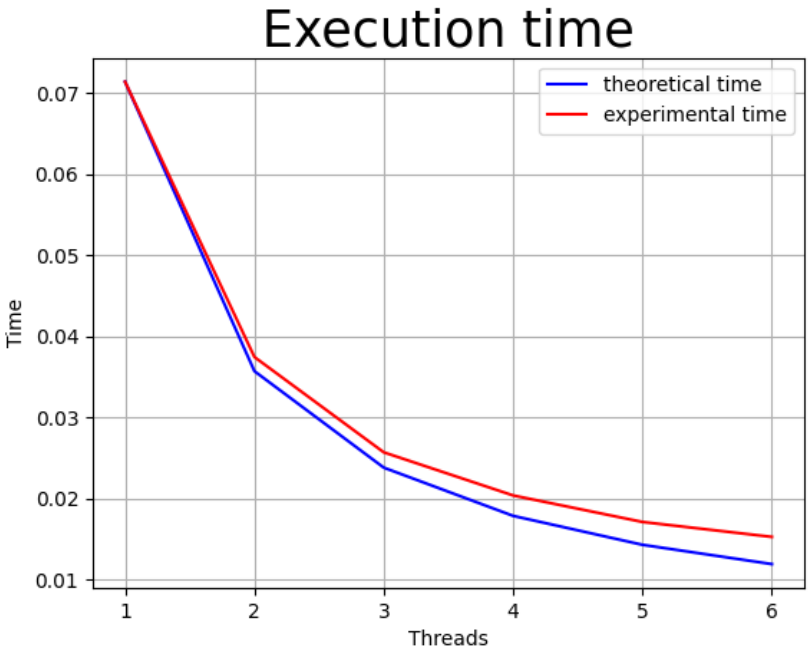


Рис. 1: Время

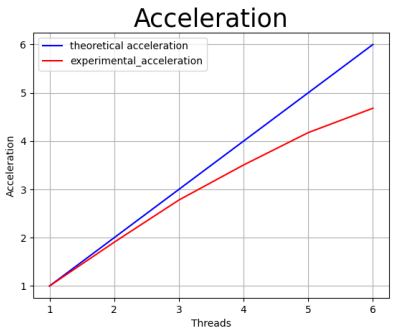


Рис. 2: Ускорение

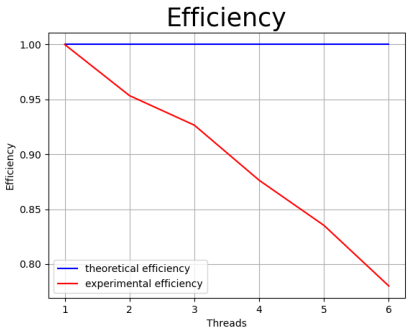


Рис. 3: Эффективность

3.2 Абсолютно неотсортированный массив

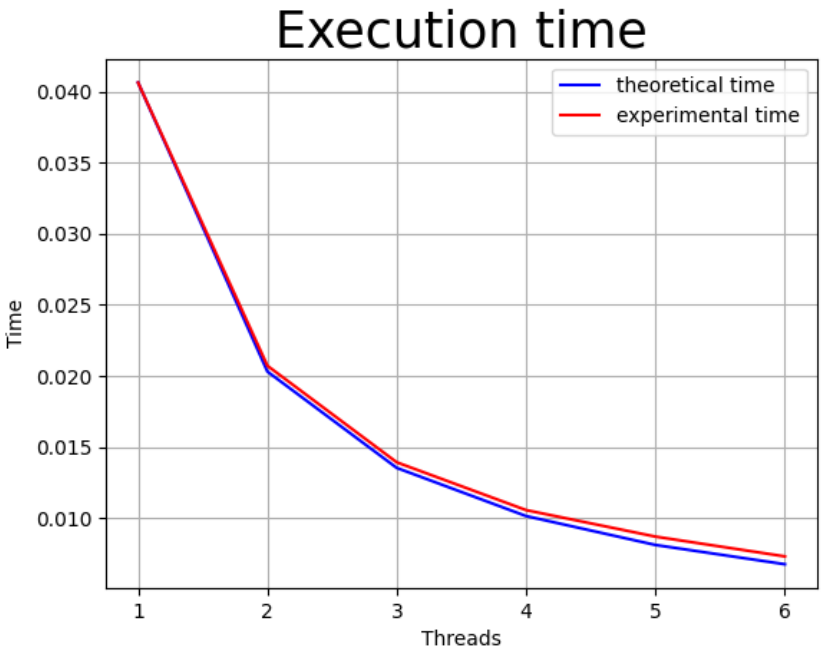


Рис. 4: Время

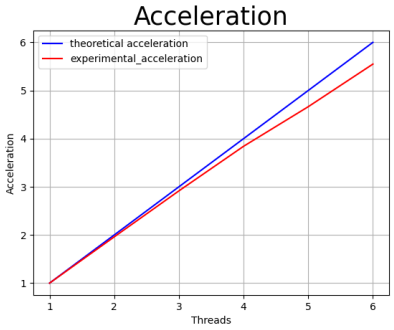


Рис. 5: Ускорение

Рис. 6: Эффективность

3.3 Массив с большим количеством одинаковых элементов

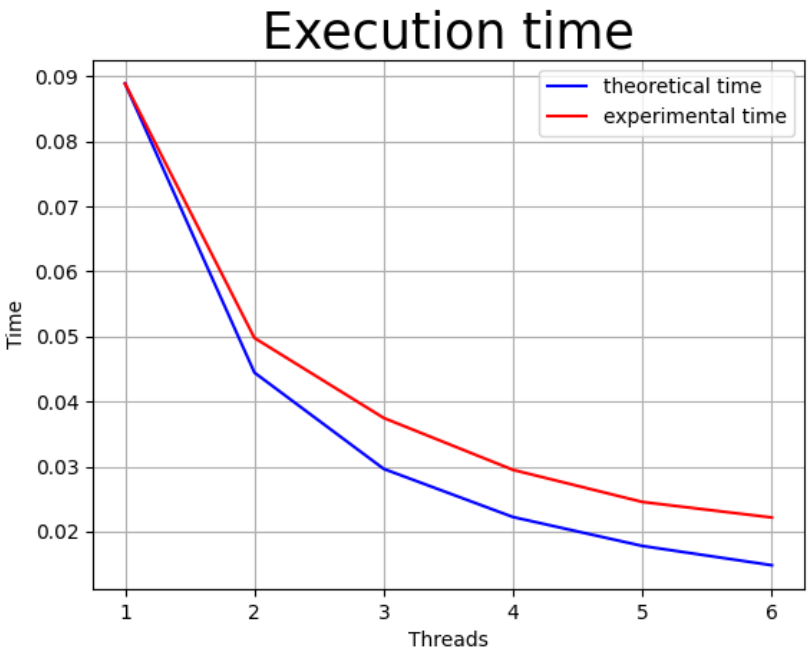


Рис. 7: Время

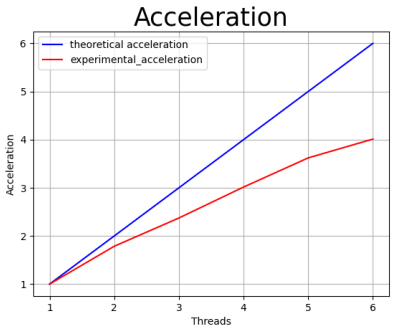


Рис. 8: Ускорение

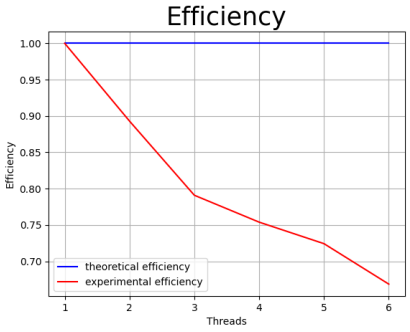


Рис. 9: Эффективность

3.4 Случайный массив

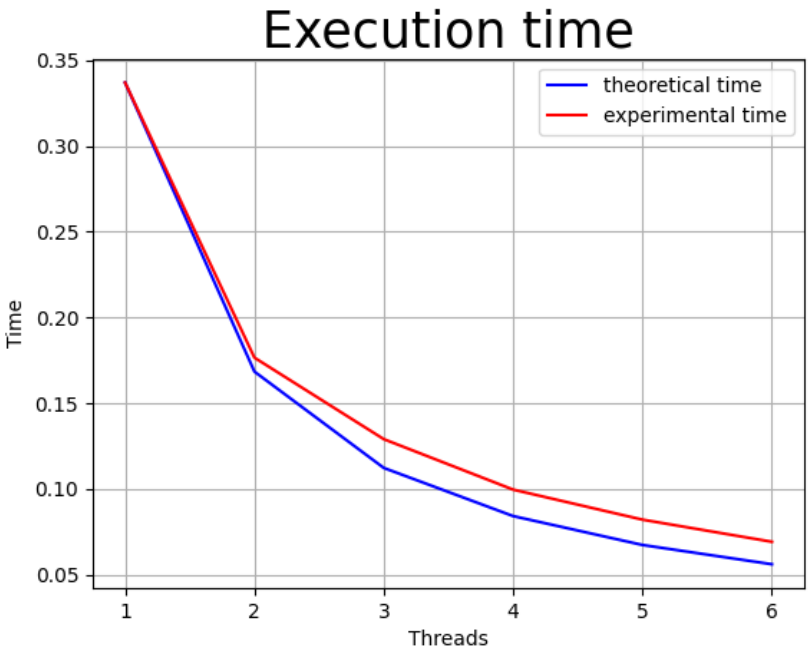


Рис. 10: Время

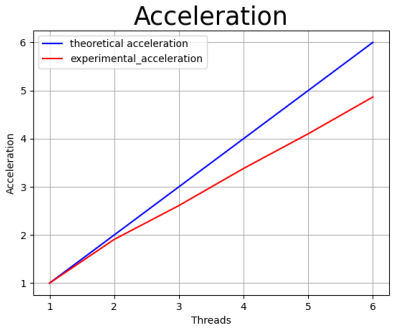


Рис. 11: Ускорение

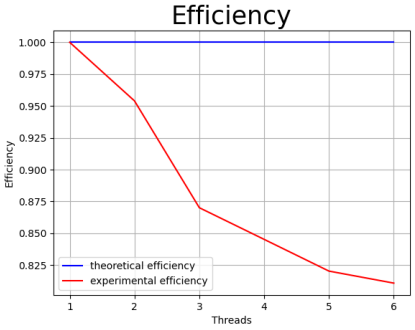


Рис. 12: Эффективность

4 Заключение

В ходе данной лабораторной работе был реализован параллельный алгоритм сортировки Шелла.

В рамках работы были рассмотрены 4 вида массивов: частично отсортированный, абсолютно неотсортированный, с большим количеством одинаковых элементов и случайный.

Из графиков следует, что параллельный алгоритм сортировки Шелла отлично справляется с частично отсортированными, абсолютно неотсортированными массивами и массивами с большим количеством одиннаковых элементов по сравнению со случайными массивами.

5 Приложение

5.1 main.c

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

#define MAX_NUM_THREADS 6

int main() {
    srand(time(NULL));
    int size, type, *array, *tmp_array;

    FILE *fd = fopen("../timing.txt", "w");
    fprintf(fd, "%d\n", MAX_NUM_THREADS);

    printf("Please type the size of array: ");
    scanf("%d", &size);
    array = (int *) malloc(size * sizeof(int));
    tmp_array = (int *) malloc(size * sizeof(int));

    printf("Please type the type of array: ");
    scanf("%d", &type);
    switch (type) {
        case 1:
            for (int i = 0; i < size; i++)
                array[i] = i;
            break;
        case 2:
            for (int i = 0; i < size; i++)
                array[i] = size - i - 1;
            break;
        case 3:
            for (int i = 0; i < size; i++)
                array[i] = rand() % 10;
            break;
        default:
            for (int i = 0; i < size; i++)
                array[i] = rand() % size;
    }

    for (int threads = 1; threads <= MAX_NUM_THREADS; threads++) {
        for (int i = 0; i < size; i++)
            tmp_array[i] = array[i];
```

```
        double start_time = omp_get_wtime();
        for (int step = size / 2; step > 0; step /= 2) {
#pragma omp parallel for num_threads(threads) shared(tmp_array, size, step)
    default(none)
            for (int i = step; i < size; i++)
                for (int j = i - step; j >= 0 &&
                        tmp_array[j] < tmp_array[j + step]; j -= step) {
                    int tmp = tmp_array[j];
                    tmp_array[j] = tmp_array[j + step];
                    tmp_array[j + step] = tmp;
                }
        }
        double end_time = omp_get_wtime();
        fprintf(fd, "%lf\n", (end_time - start_time));
    }

    fclose(fd);
    free(array);
    free(tmp_array);

    return 0;
}
```

5.2 main.py

```
import matplotlib.pyplot as plt
from prettytable.colortable import ColorTable


def read_file():
    with open('/home/bar1k/CLionProjects/ParallelProgramming/timing.txt') \
            as file:
        threads = [x + 1 for x in range(int(file.readline()))]
        times = [float(x) for x in file.read().split()]
    return threads, times


def draw_plots(threads, experimental_time):
    theoretical_time = [experimental_time[0] / (x + 1) for x in
                        range(len(threads))]
    plt.title("Execution time", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Time')
    plt.grid(True)
    plt.plot(threads, theoretical_time, 'b')
    plt.plot(threads, experimental_time, 'r')
    plt.legend(['theoretical time', 'experimental time'])
    plt.show()

    theoretical_acceleration = [x for x in threads]
    experimental_acceleration = [experimental_time[0] / x for x in
                                 experimental_time]
    plt.title("Acceleration", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Acceleration')
    plt.grid(True)
    plt.plot(threads, theoretical_acceleration, 'b')
    plt.plot(threads, experimental_acceleration, 'r')
    plt.legend(['theoretical acceleration', 'experimental_acceleration'])
    plt.show()

    theoretical_efficiency = [1] * len(threads)
    experimental_efficiency = [experimental_acceleration[x] / threads[x]
                               for x in range(len(threads))]
    plt.title("Efficiency", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Efficiency')
    plt.grid(True)
    plt.plot(threads, theoretical_efficiency, 'b')
```

```
    plt.plot(threads, experimental_efficiency, 'r')
    plt.legend(['theoretical efficiency', 'experimental efficiency'])
    plt.show()


def output_table(threads, times):
    table = ColorTable()
    table.field_names = ['Threads'] + [str(threads[i]) for i in
                                       range(len(threads))]
    table.add_row(['Times'] + [str(times[i]) for i in range(len(times))])
    print(table)


if __name__ == '__main__':
    data = read_file()
    draw_plots(data[0], data[1])
    output_table(data[0], data[1])
```