Лабораторная работа №3

"Реализация алгоритма с использованием технологии *OpenMP*"

Выполнил студент группы Б20-505
Сорочан Илья

Московский Инженерно-Физический Интститут
Москва 2023

1 Рабочая среда

Технические характеристики:
CPU: *6-core AMD Ryzen 5 4500U*
Kernel: *5.15.85-1-MANJARO x86_64*
Mem: *7303.9 MiB*

Используется:
Компилятор: *GCC 12.2.0*
OpenMP: *4.5*

2 Сортировка Шелла

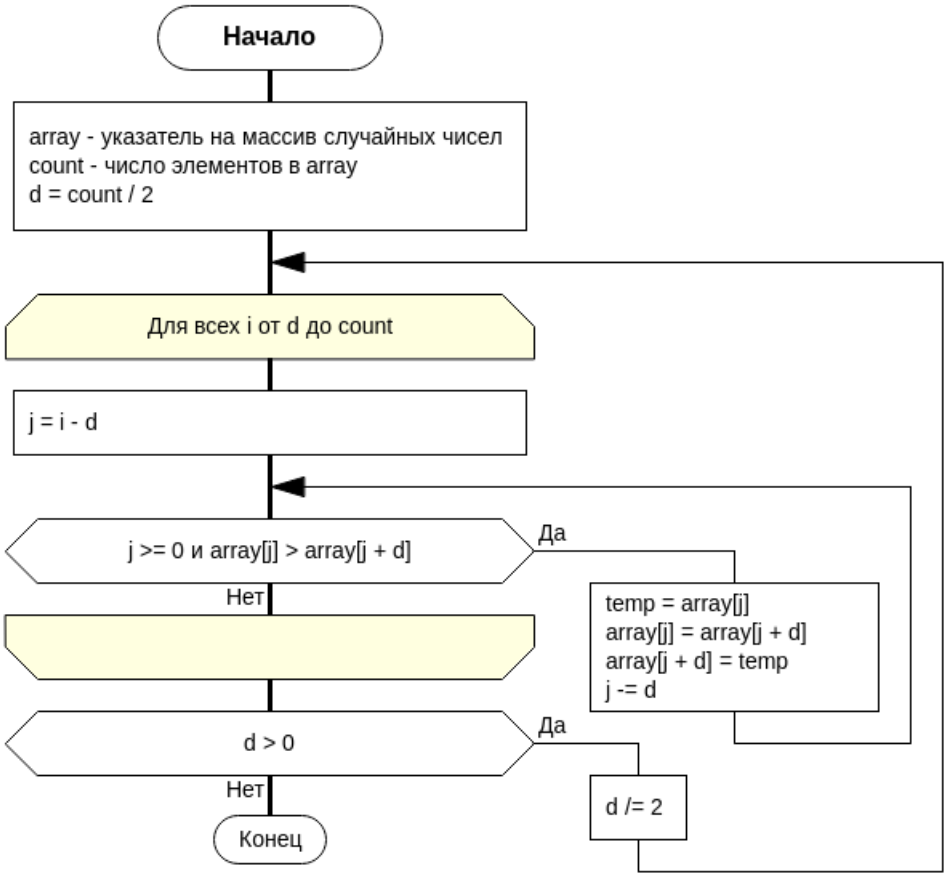
2.1 Принцип работы

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). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).

Для определённости будет рассматриваться классический вариант, когда изначально $d = \frac{n}{2}$ и уменьшается по закону $d_{i+1} = \frac{d_i}{2}$, пока не достигнет 1. Здесь n обозначает длину сортируемого массива.

Тогда в худшем случае сортировка займет $O(n^2)$.

Блок схема сортировки Шелла:



2.2 Параллелизация

Как и в предыдущих лабораторных, в первую очередь следует попробовать сделать параллельным цикл.

Задаем число потоков и общие переменные через *omp parallel*. Однозначно общими должны быть массив и его длинна.

Так как внутренний цикл по i по сути затрагивает только d-е элементы относительно i-го, то:

```
#pragma omp parallel num_threads(THREADS) shared(array, count) default(none)
    for (int d = count / 2; d > 0; d /= 2) {
```

```
    const int cd = d;
    #pragma omp for
    for (int i = cd; i < count; ++i) {
        for (int j = i - cd; j >= 0 && array[j] > array[j + cd]; j -= cd) {
            int temp = array[j];
            array[j] = array[j + cd];
            array[j + cd] = temp;
        }
    }
}
```

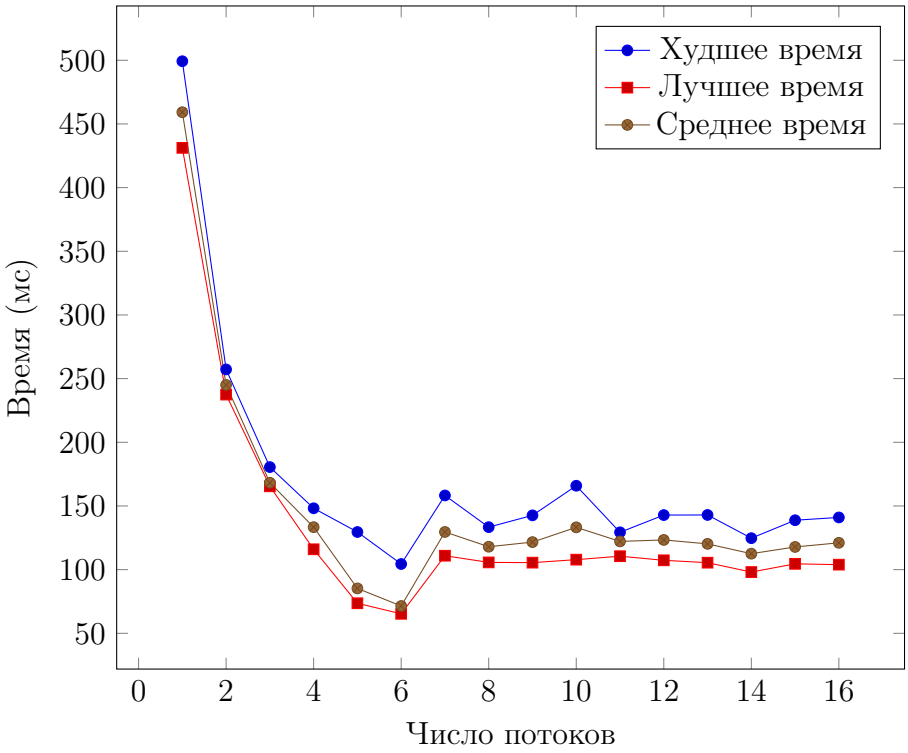
Здесь так же видно, что d вынесена в константу cd. Это сделано для того, что бы *OpenMP* не принял меры предосторожности в цикле по i. Он может это сделать так как d меняется во внешнем цикле, но он не знает меняется ли во внутреннем.

3 Экспериментальные данные

На каждое число потоков отводилось 10 запусков. Так же число элементов в массиве было уменьшено с *10000000* до *1000000*.

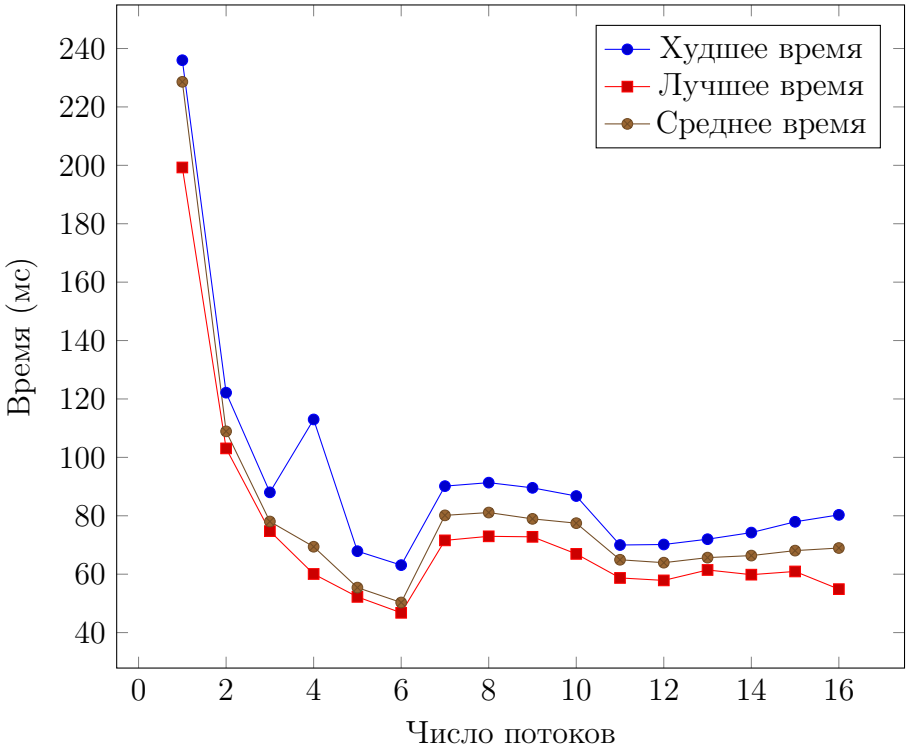
3.1 Время выполнения

Для начала я решил взглянуть не только на среднюю скорость выполнения, но и на крайние варианты:

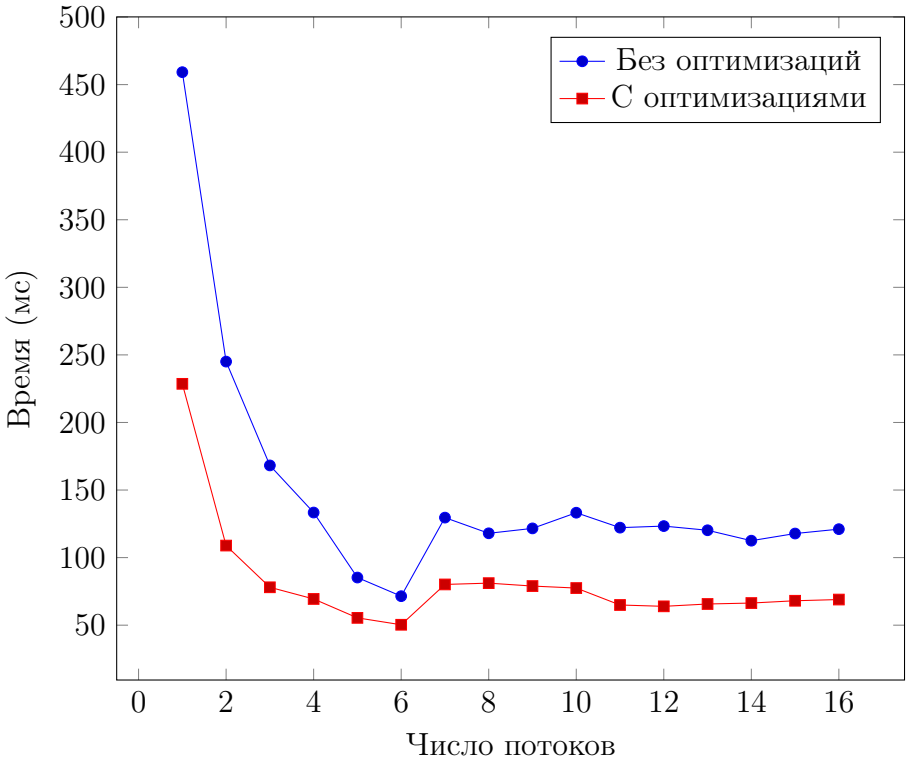


Крайне заметно, что многопоточная программа работает куда быстрее обычной. При этом уже с 2-х потоков виден прирост практически в 2 раза. Однако далее он становится все незначительнее.

Рассмотрим теперь данные с оптимизацией:

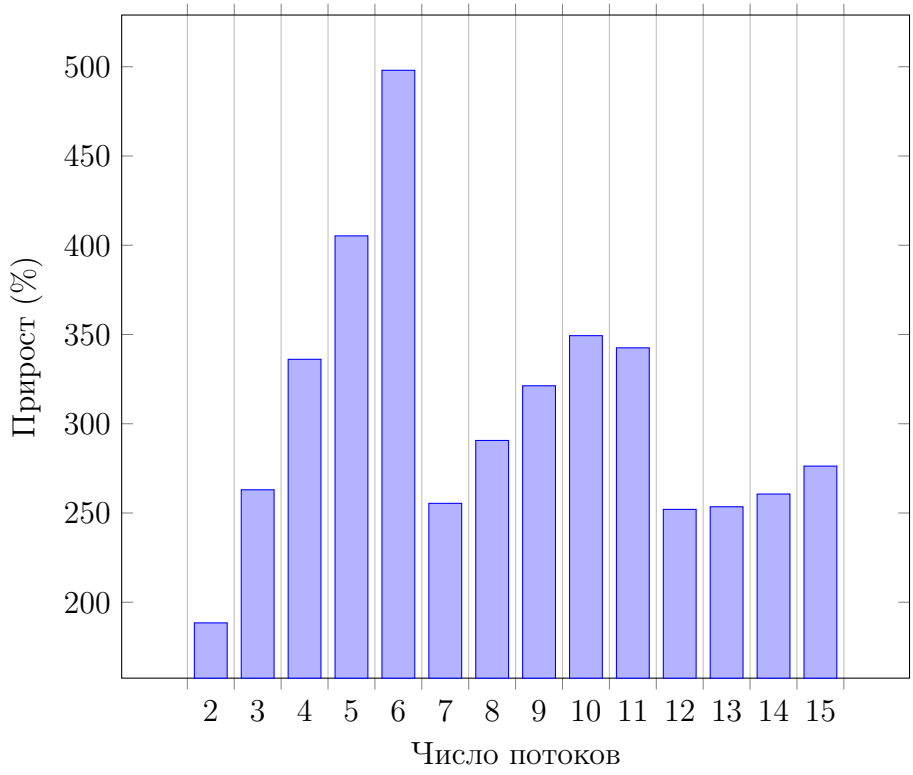


Как видно на графике выше, повышение числа потоков лишь увеличивает среднее время исполнения. При этом заметна общая тенденция: максимальная эффективность достигается при 6-ти потоках:

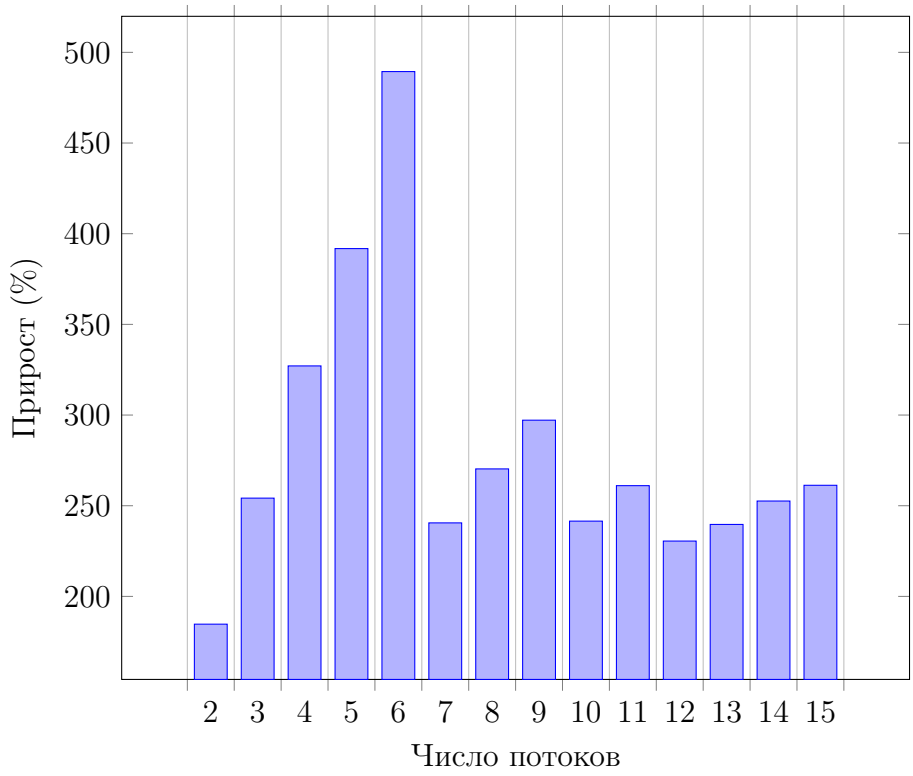


3.2 Прирост производительности

В целом с увеличением числа потоков производительность растет. Рассмотрим ускорение многопоточной программы относительно однопоточной. Для не оптимизированной сборки:



Для оптимизированной сборки:



4 Заключение

В данной работе было исследовано ускорение, получаемое при использовании нескольких потоков в задании о сортировке массива сортировкой Шелла. Была усовершенствована предоставленная программа и собранны данные. Так же был написан скрипт, подсчитывающий прирост производительности относительно одного потока. Оформлен отчет.

В ходе работы было выяснено, что в применение нескольких потоков крайне положительно влияет на итоговую производительность. Из 30 многопоточных сборок только 2 превосходили обычную менее чем в 2 раза. При этом наблюдался прирост вплоть до 5-ти раз.

Приложение А

Использованные программные коды

Для проверки версии *OpenMP* использовался следующий код:

```
// Print openmp version

#include <stdio.h>

#if _OPENMP == 200505
#define _OPENMP_VERSION "2.5"
#elif _OPENMP == 200805
#define _OPENMP_VERSION "3.0"
#elif _OPENMP == 201107
#define _OPENMP_VERSION "3.1"
#elif _OPENMP == 201307
#define _OPENMP_VERSION "4.0"
#elif _OPENMP == 201511
#define _OPENMP_VERSION "4.5"
#elif _OPENMP == 201811
#define _OPENMP_VERSION "5.0"
#elif _OPENMP == 202011
#define _OPENMP_VERSION "5.1"
#else
#define _OPENMP_VERSION "unknown"
#endif

int main(int argc, char** argv) {
    printf("OpenMP Version: %s\n", _OPENMP_VERSION);

    return 0;
}
```

Для измерения времени исполнения алгоритма использовался следующий код (выводит *csv* в стандартный вывод):

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

const int N = 1000000;
const int MAX_THREADS = 16;
const int RUNS_PER_THREAD = 10;

void randArr(int *array, int size) {
    for (int i = 0; i < size; ++i)
        array[i] = rand();
}

// run algo and return time elapsed
double run(const int threads, int *array, const int size) {
    double start = omp_get_wtime();
    for (int d = size / 2; d > 0; d /= 2) {
        const int cd = d;
        #pragma omp parallel for num_threads(threads) shared(array, size, cd) default(none)
        for (int i = 0; i < cd; ++i) {
            // insertion sort
            for (int j = cd + i; j < size; j += cd) {
                int key = array[j];
                int k = j - cd;

                while (k >= i && array[k] > key) {
                    array[k + cd] = array[k];
                    k -= cd;
                }
                array[k + cd] = key;
            }
        }
    }
```

```
    double end = omp_get_wtime();
    return (end - start) * 1000;
}

int main(int argc, char **argv) {
    // set constant seeds
    int seed[MAX_THREADS];
    for (int i = 0; i < MAX_THREADS; ++i)
        seed[i] = rand();


    int *array = (int *)malloc(N * sizeof(int));

    puts("Threads,Worst (ms),Best (ms),Avg (ms)");

    for (int threads = 1; threads < MAX_THREADS + 1; ++threads) {
        double sum = 0, max_time = -1, min_time = 100000;
        for (int i = 0; i < RUNS_PER_THREAD; ++i) {
            // gen array with special seed
            srand(seed[i]);
            randArr(array, N);

            // calc value
            double time = run(threads, array, N);
            if (time > max_time)
                max_time = time;
            if (time < min_time)
                min_time = time;
            sum += time;
        }

        printf("%d,%.3f,%.3f,%.3f\n", threads, max_time, min_time, sum / RUNS_PER_THREAD);
    }

    free(array);

    return 0;
}
```

Для вычисления эффективности многопоточной программы по отношению к однопоточной использовался следующий скрипт:

```
import csv, sys

if len(sys.argv) < 3:
    exit(1)

filein = open(sys.argv[1], "r")
fileout = open(sys.argv[2], "w")

reader = csv.reader(filein)
writer = csv.writer(fileout)

# skip header
header = reader.__next__()
writer.writerow([header[0], "Efficiency"])

# get first one
first_avg = reader.__next__()[-1]
# writer.writerow(["1", "100"])
first_avg = float(first_avg)

for row in reader:
    avg = float(row[-1])
    relative = "{:.3f}".format(100 * first_avg / avg)
    writer.writerow([row[0], relative])

filein.close()
fileout.close()
```

Приложение Б

Таблицы с практическими результатами

Таблица без оптимизаций:

Threads	Worst (ms)	Best (ms)	Avg (ms)
1	499.25	431.22	459.21
2	257.23	237.46	245.03
3	180.53	165.43	168.21
4	148.19	115.99	133.36
5	129.55	73.6	85.25
6	104.41	65.31	71.46
7	158.26	110.94	129.59
8	133.37	105.69	117.99
9	142.63	105.5	121.62
10	165.88	107.84	133.22
11	129.33	110.56	122.17
12	142.85	107.34	123.32
13	142.94	105.46	120.24
14	124.71	98.1	112.52
15	138.81	104.56	117.81
16	140.99	103.91	121.05

Таблица с оптимизациями:

Threads	Worst (ms)	Best (ms)	Avg (ms)
1	235.99	199.27	228.59
2	122.15	103.07	108.91
3	88.02	74.75	78.04
4	112.99	60.07	69.4
5	67.88	52.23	55.41
6	63.1	46.78	50.31
7	90.16	71.58	80.15
8	91.35	72.97	81.11
9	89.57	72.8	78.94
10	86.76	66.95	77.47
11	69.99	58.72	64.94
12	70.17	57.87	63.94
13	71.98	61.44	65.67
14	74.24	59.84	66.37
15	77.94	60.94	68.07
16	80.31	54.88	68.96

Таблица сравнений без оптимизаций:

Threads	Efficiency
2	188.44
3	263
4	336.1
5	405.25
6	498.01
7	255.4
8	290.64
9	321.27
10	349.35
11	342.52
12	252.03
13	253.49
14	260.62
15	276.27
16	280.81

Таблица сравнений с оптимизациями:

Threads	Efficiency
2	184.69
3	254.19
4	327.09
5	391.79
6	489.4
7	240.54
8	270.3
9	297.17
10	241.51
11	261.07
12	230.51
13	239.68
14	252.59
15	261.27
16	258.34