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

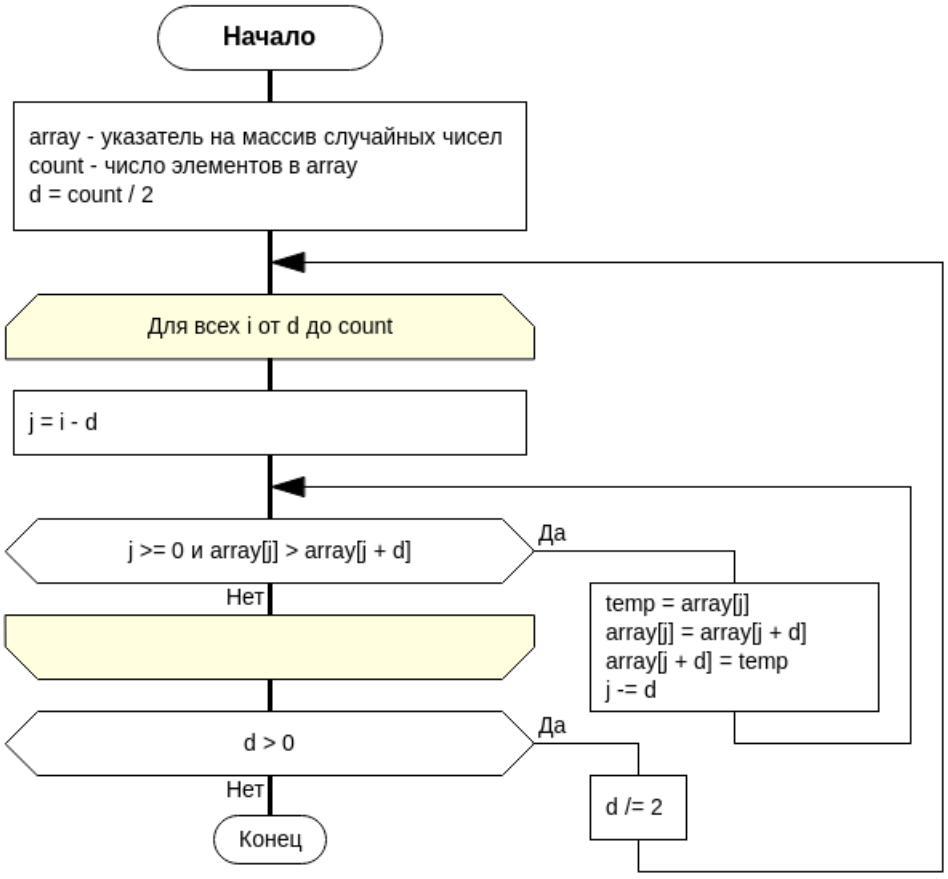
2.1 Принцип работы

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 длинну 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 параллелизируется 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 его можно параллелизировать:

```
#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;
        }
    }
}
```

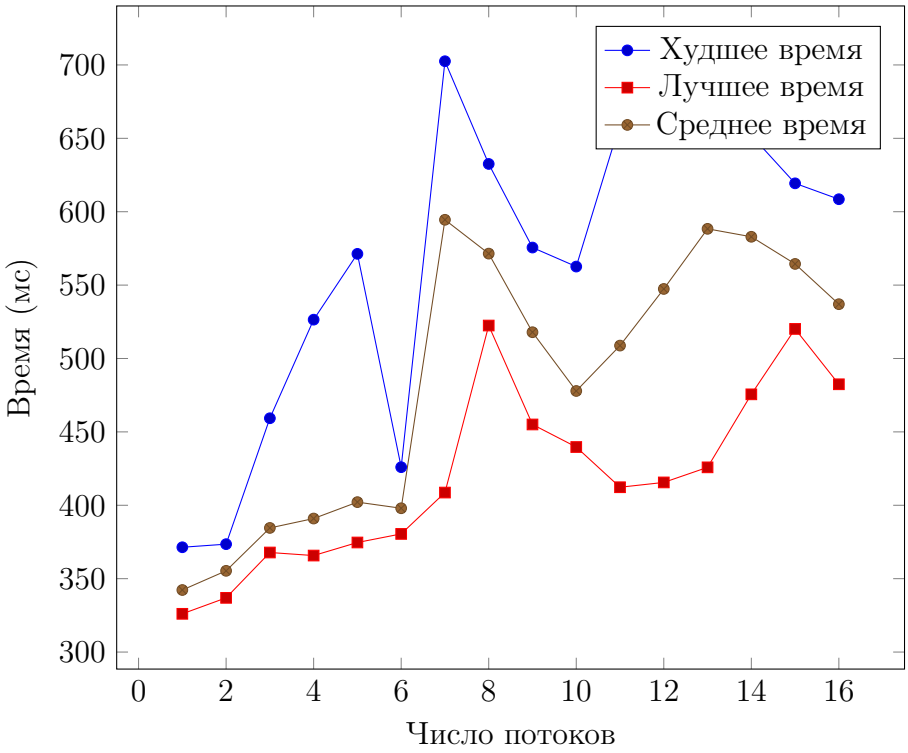
Здесь так же видно, что d вынесена в константу cd. Это сделано для того, что бы *OpenMP* не принял меры предосторожности в цикле по i. Он может это сделать так как d меняется во внешнем цикле, но он не знает меняется ли во внутреннем.

3 Экспериментальные данные

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

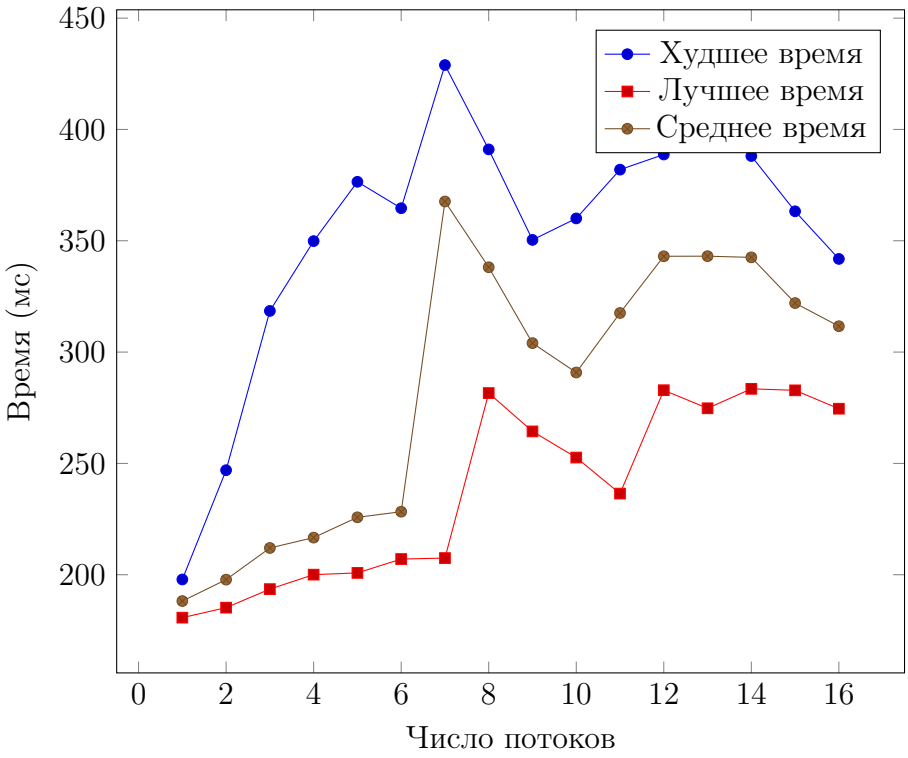
3.1 Время выполнения

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



Крайне заметно, что однопотчная программа работает куда быстрее её конкурентов. В этой лабораторной худшие запуски занимали почти секунду, так что причины такой “антипроизводительности” остаются для меня загадкой. Возможно причина кроется в распределении итераций цикла – но это тема для одной из следующих лабораторных работ.

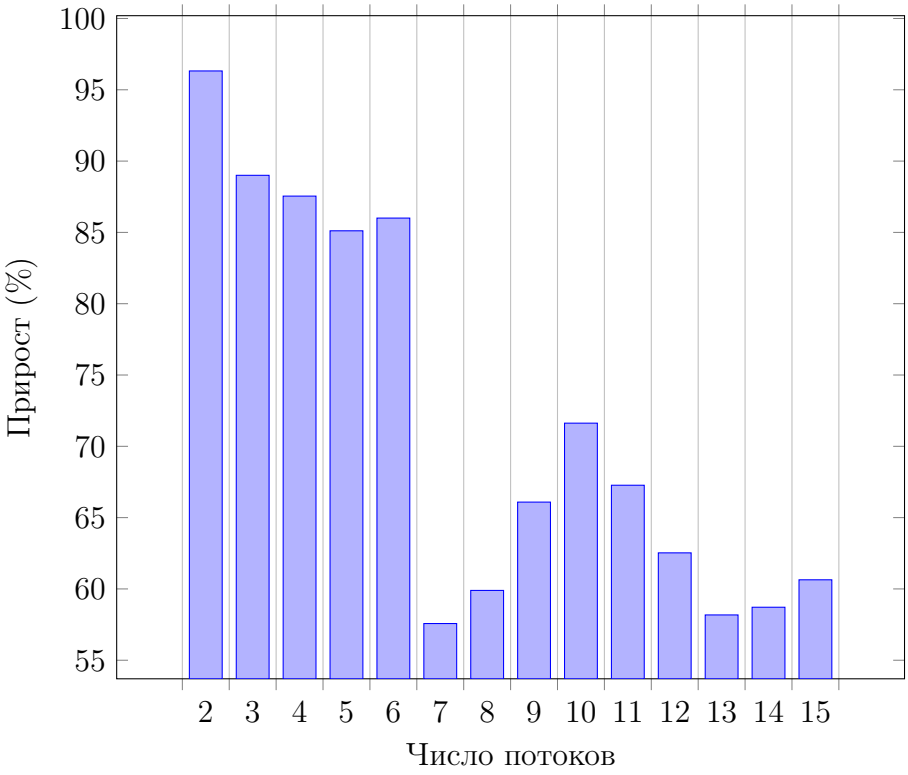
Рассмотрим теперь данные с оптимизацией:



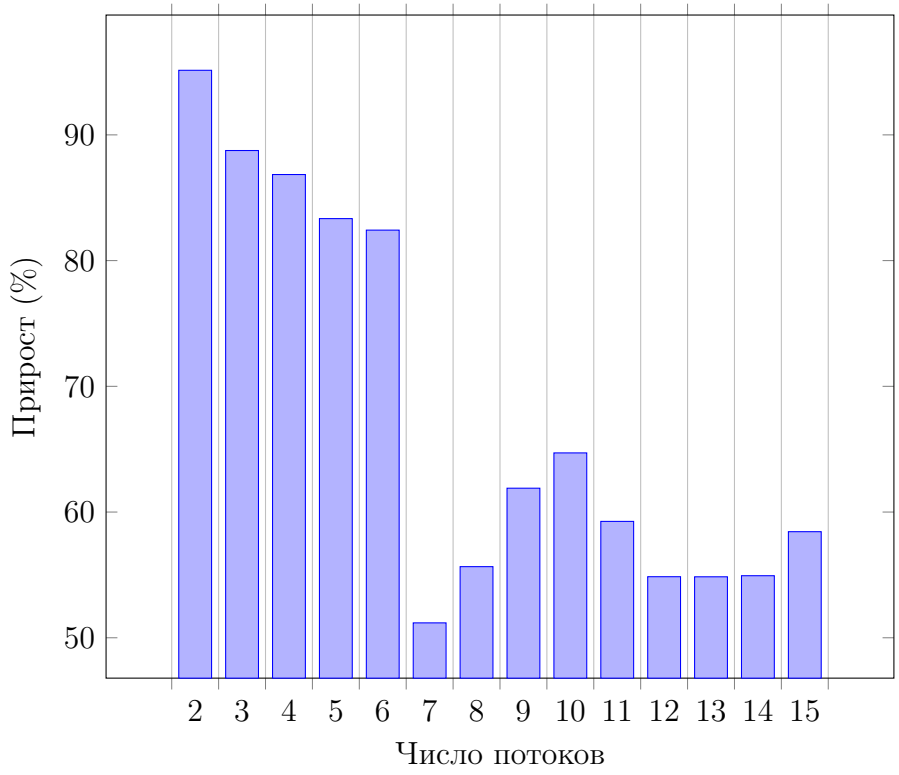
Как видно на графике выше, повышение числа потоков лишь увеличивает среднее время исполнения.

3.2 Прирост производительности

В целом с увеличением числа потоков производительность падает. Рассмотрим ускорение многопоточной программы относиельно однопоточной. Для неоптимизированной сборки:



Для оптимизированной сборки:


Прирост (%)
90
80
70
60
50
2 3 4 5 6 7 8 9 10 11 12 13 14 15
Число потоков

4 Заключение

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

В ходе работы было выяснено, что в данной задаче применение многопоточности лишь замедлит программу. Могу предположить, что это связано с распределением итераций главного цикла.

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>

const int N = 1000000;
const int MAX_THREADS = 16;
const int RUNS_PER_THREAD = 20;

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) {
    clock_t start = clock();
    int index = -1;
    #pragma omp parallel num_threads(threads) shared(array, size) default(none)
    for (int d = size / 2; d > 0; d /= 2) {
        const int cd = d;
        #pragma omp for
        for (int i = cd; i < size; ++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;
            }
        }
    }

    clock_t end = clock();
    const double CLOCKS_PER_MS = (double)CLOCKS_PER_SEC / 1000;
    return (double)(end - start) / CLOCKS_PER_MS;
}
```

```
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	371.4	326.06	342.27
2	373.5	336.94	355.35
3	459.3	367.83	384.56
4	526.45	365.75	390.97
5	571.29	374.65	402.14
6	425.96	380.49	397.97
7	702.53	408.67	594.5
8	632.55	522.46	571.45
9	575.59	455.05	517.9
10	562.63	439.69	477.87
11	660.22	412.29	508.79
12	702.1	415.57	547.38
13	665.5	425.85	588.34
14	652	475.59	582.93
15	619.34	520.15	564.43
16	608.52	482.44	537

Таблица с оптимизациями:

Threads	Worst (ms)	Best (ms)	Avg (ms)
1	197.88	180.73	188.18
2	246.96	185.22	197.8
3	318.49	193.54	212.02
4	349.84	200.04	216.68
5	376.48	200.82	225.8
6	364.62	207.05	228.3
7	428.91	207.48	367.64
8	391.04	281.6	338.1
9	350.39	264.38	304.01
10	360.04	252.64	290.83
11	381.94	236.45	317.56
12	388.74	282.87	343.04
13	413.88	274.76	343.08
14	388.06	283.5	342.54
15	363.23	282.82	322.01
16	341.84	274.57	311.62

Таблица сравнений без оптимизаций:

Threads	Efficiency
2	96.32
3	89
4	87.55
5	85.11
6	86
7	57.57
8	59.9
9	66.09
10	71.62
11	67.27
12	62.53
13	58.18
14	58.72
15	60.64
16	63.74

Таблица сравнений с оптимизациями:

Threads	Efficiency
2	95.14
3	88.76
4	86.85
5	83.34
6	82.42
7	51.19
8	55.66
9	61.9
10	64.7
11	59.26
12	54.86
13	54.85
14	54.94
15	58.44
16	60.39