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 (вывод *inxi*):

```
CPU: 6-core AMD Ryzen 5 4500U with Radeon Graphics (-MCP-)
speed/min/max: 1396/1400/2375 MHz Kernel: 5.15.85-1-MANJARO x86_64 Up: 46m
Mem: 2689.5/7303.9 MiB (36.8%) Storage: 238.47 GiB (12.6% used) Procs: 238
Shell: Zsh inxi: 3.3.24
```

Используемый компилятор:

```
gcc (GCC) 12.2.0
```

Согласно официальной документации даная версия компилятора поддерживает *OpenMP 5.0*

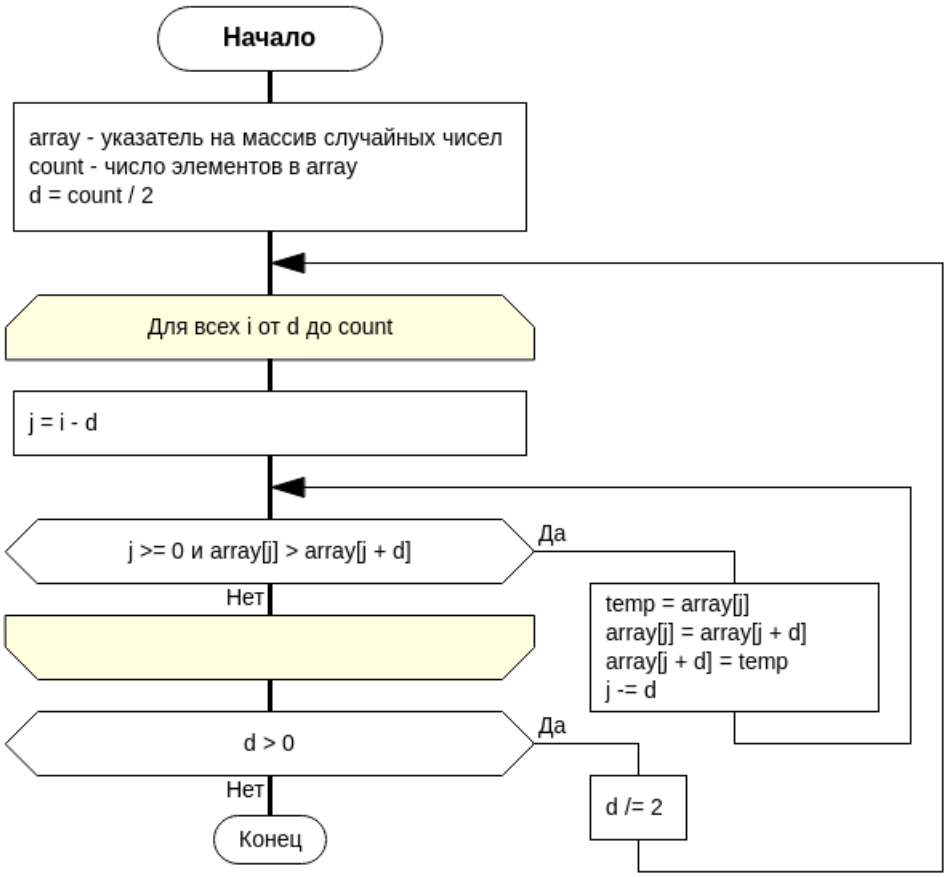
2 Реализация алгоритма в одном потоке

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 $d = \frac{count}{2}$ и уменьшается по закону $d_{i+1} = \frac{d_i}{2}$, пока не достигнет 1. Здесь $count$ обозначает длинну сортируемого массива.

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

Блок схема сортировки Шелла:



3 Параллелизация

Как и в предыдущих работах, в первую очередь параллелизируется цикл.

В первую очередь 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:

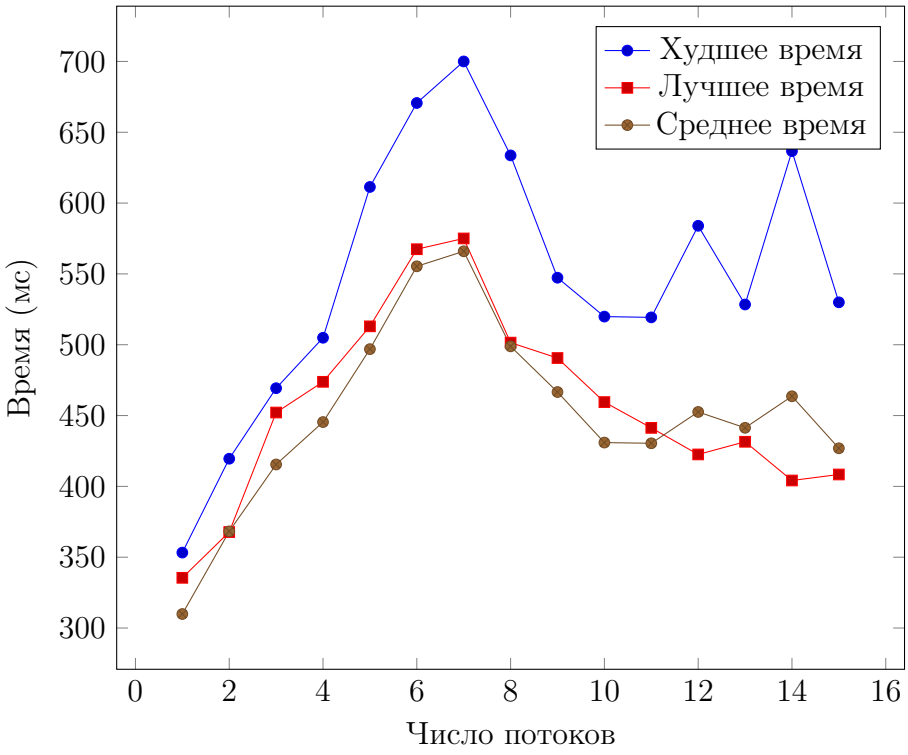
```
#pragma omp parallel num_threads(THREADS) shared(array, count) default(none)
    for (int d = count / 2; d > 0; d /= 2) {
        const int cd = d;
        #pragma omp for
        for (int i = cd; i < count; ++i) {
            for (int j = i - cd; j >= 0 && array[j] > array[j + cd]; j -= cd) {
                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.

4 Экспериментальные данные

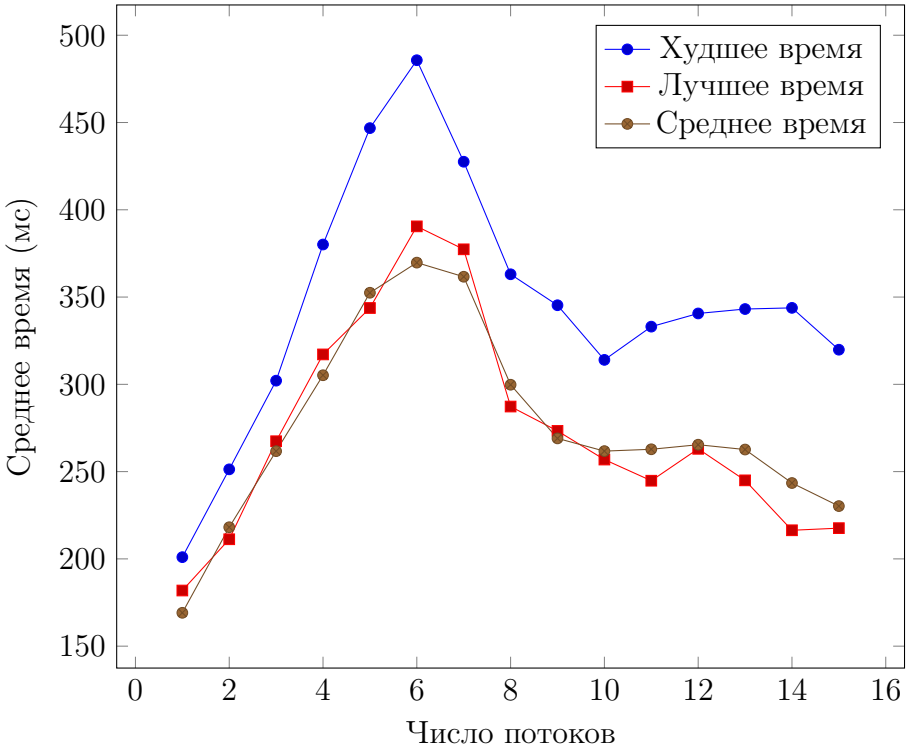
Во всех измерениях бралось 10 запусков на поток.

В этот раз я решил увеличить количество элементов и худшее время уже доходит до секунды. Однако, как видно ниже на результатах это сказалось не сильно.



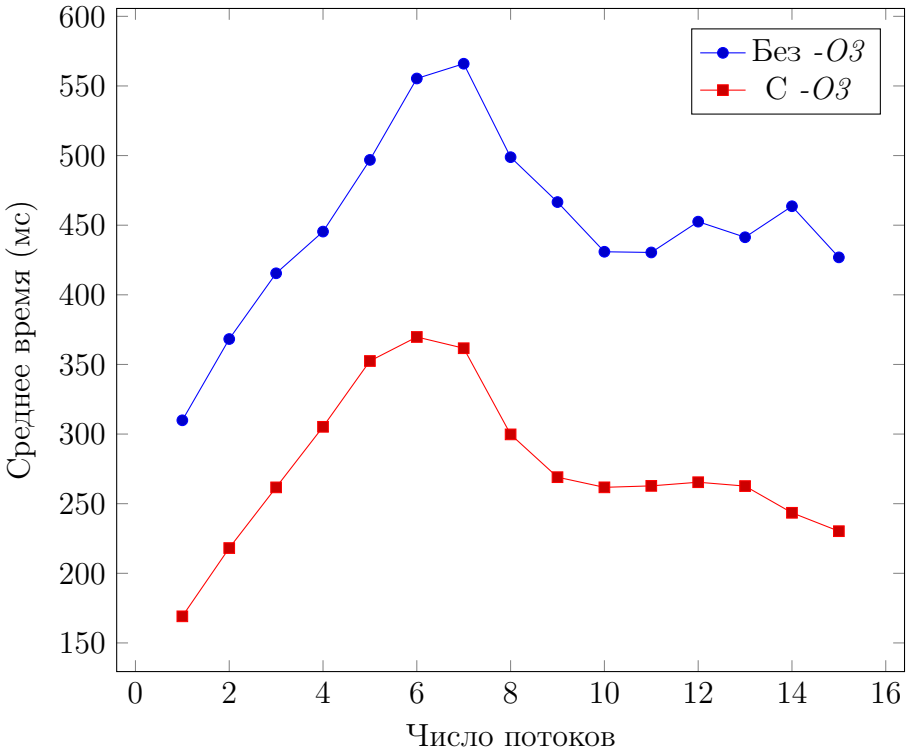
Из графика видно, что в среднем многопоточная программа работает медленнее, что на мой взгляд странно. В этой лабораторной, в отличие от предыдущих, алгоритм куда сложнее. Видимо этого недостаточно.

Ну и как уже упоминалось в прошлой работе – оптимизации компилятора. Ниже приведена таблица для компиляции с *-O3*:

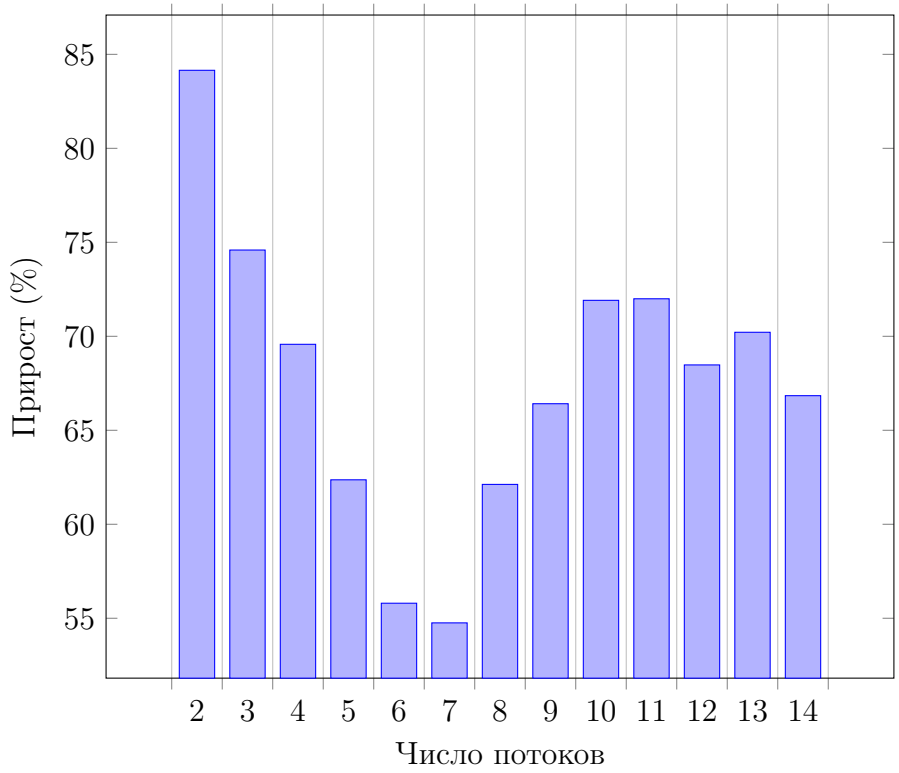


Прекрасно видно, что однопоточная программа лидирует с большим отрывом и причиной этому – оптимизации компилятора.

Стоит заметить, что многопотчные сборки так же получили прирост и общая тенденция неизменна. Я полагаю, что *OpenMP* мешает компилятору как-то все сильно оптимизировать:



Рассмотрим так же прирост, даваймый каждым числом процессоров относительно первого (берем среднее время):



Интересно, что в отличие от предыдущих исследований здесь лидируют сборки с 2 и 3 потоками.

5 Заключение

В данной работе было исследовано ускорение, получаемое при использовании многопоточности в сортировке Шелла. Была усовершенствована предоставленная программа и написан специальный скрипт, собирающие данные о нескольких запусках этой программы в один файл, попутно её перекомпилируя.

В ходе работы было выяснено, что в данной задаче применение многопоточности лишь замедлит программу. С уверенностью можно сказать, что частью причины таких результатов являются оптимизации, производимые компилятором.

С другой стороны следует отметить, что для настолько тривиальной задачи применять многопоточность нет смысла. В Экспериментальном массиве сортировалось миллион элементов и даже так, сборка без потоков и со всеми оптимизациями занимала сотые доли миллисекунд.

Приложение А

Использованные программные коды

Код без многопоточности:

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

int main(int argc, char** argv)
{
    const int count = 1000000;       ///< Number of array elements
    const int random_seed = 920214; ///< RNG seed
```

```
    int* array = 0;                    ///< The array which we are sorting

    /* Initialize the RNG */
    srand(random_seed);

    /* Generate the random array */
    array = (int*)malloc(count*sizeof(int));
    for(int i=0; i<count; i++) { array[i] = rand(); }

    clock_t start = clock();

    /* shellsort */
    for (int d = count / 2; d > 0; d /= 2) {
        for (int i = d; i < count; ++i) {
            for (int j = i - d; j >= 0 && array[j] > array[j + d]; j -= d) {
                int temp = array[j];
                array[j] = array[j + d];
                array[j + d] = temp;
            }
        }
    }

    clock_t end = clock();

    const double CLOCKS_PER_MS = (double)CLOCKS_PER_SEC / 1000;
    double total = (double)(end - start) / CLOCKS_PER_MS;
    printf("%.3f", total);

    free(array);

    return 0;
}
```

Код с многопоточностью:

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

int main(int argc, char** argv)
{
    const int count = 1000000;       ///< Number of array elements
    const int random_seed = 920214; ///< RNG seed

    int* array = 0;                  ///< The array which we are sorting

    /* Initialize the RNG */
    srand(random_seed);

    /* Generate the random array */
    array = (int*)malloc(count*sizeof(int));
    for(int i=0; i<count; i++) { array[i] = rand(); }

    clock_t start = clock();

    /* shellsort */
    #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;
```

```
            }
        }
    }

    clock_t end = clock();

    const double CLOCKS_PER_MS = (double)CLOCKS_PER_SEC / 1000;
    double total = (double)(end - start) / CLOCKS_PER_MS;
    printf("%.3f", total);

    free(array);

    return 0;
}
```

Для сборки данных использовался следующий скрипт:

```
# This script compiles main.c with different number of threads
# and collects data to data.csv file
# format: worst,best,average
import os
import subprocess
import csv
import sys

# important constants
RUNS_PER_THREADS = 10
THREADS_LIMIT = 16

# compile with threads
def compile(threads):
    if threads <= 1:
        os.system("gcc main.c -o main")
    else:
        os.system("gcc threaded.c -fopenmp -DTHREADS=" + str(threads) + " -o main")

def compile_opt(threads):
    if threads <= 1:
        os.system("gcc main.c -O3 -o main")
    else:
        os.system("gcc threaded.c -O3 -fopenmp -DTHREADS=" + str(threads) + " -o main")

# capture worst, best and average
def run():
    data = []
    for _ in range(RUNS_PER_THREADS):
        proc = subprocess.run(["./main"], capture_output=True, text=True)
        data.append(float(proc.stdout))

    worst = data[0]
    best = data[0]
    s = 0
    for val in data[1:]:
        if val > worst:
            worst = val
        if val < best:
            best = val
        s += val
    return (worst, best, s / len(data))

def main():
    if len(sys.argv) < 2 or sys.argv[1] != 'opt':
        comp = compile
    else:
        comp = compile_opt

    if len(sys.argv) >= 3:
        file = sys.argv[2]
    else:
        file = "data.csv"

    with open(file, "w") as data:
        writer = csv.writer(data)
        writer.writerow(["Threads", "Worst (ms)", "Best (ms)", "Average (ms)"])
        for threads in range(1, THREADS_LIMIT):
            print("Testing threads =", threads)
            comp(threads)

            writer.writerow([str(threads)] + ["{:.3f}".format(val) for val in run()])

if __name__ == "__main__":
    main()
```

Для вычисления относительного прироста производительности использовался следующий скрипт:

```
# make csv comparacent
import csv
import sys

def main():
    if len(sys.argv) < 3:
```

```
        print(sys.argv[0], "input", "output")

    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()


if __name__ == "__main__":
    main()
```

Приложение Б

Таблицы с теоритическими и практическими результатами

Таблица без оптимизаций:

Threads	Worst (ms)	Best (ms)	Average (ms)
1	353.26	335.45	309.89
2	419.52	367.76	368.25
3	469.31	452.14	415.46
4	504.92	473.83	445.41
5	611.38	512.99	496.89
6	670.66	567.43	555.36
7	700	575.03	565.96
8	633.66	501.49	498.85
9	547.31	490.61	466.61
10	519.86	459.5	430.92
11	519.32	441.32	430.42
12	583.96	422.54	452.54
13	528.41	431.5	441.35
14	636.67	404.16	463.62
15	529.94	408.4	426.9

Таблица с оптимизациями:

Threads	Worst (ms)	Best (ms)	Average (ms)
1	200.99	181.88	169.11
2	251.3	211.25	218.13
3	302.1	267.4	261.72
4	380.1	317.14	305.21
5	446.79	343.73	352.47
6	485.69	390.5	369.72
7	427.53	377.36	361.64
8	363.06	287.28	299.77
9	345.32	273.31	269.03
10	314.01	256.91	261.75
11	333.05	244.78	262.79
12	340.61	263.06	265.41
13	343.14	244.99	262.65
14	343.81	216.39	243.46
15	319.84	217.61	230.25

Таблица сравнений:

Threads	Efficiency
2	84.15
3	74.59
4	69.57
5	62.37
6	55.8
7	54.75
8	62.12
9	66.41
10	71.91
11	72
12	68.48
13	70.21
14	66.84
15	72.59