Лабораторная работа №2

"Выделение ресурса параллелизма. Технология 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*

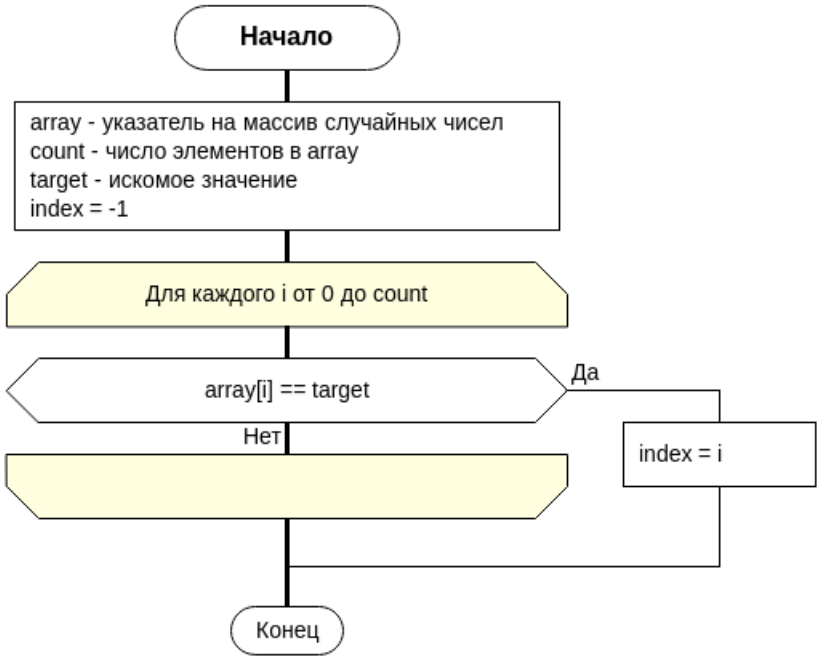
2 Анализ алгоритма

Данный алгоритм ищет индекс заданного элемента в массиве со случайно сгенерированными значениями.
Временная сложность:

- В лучшем случае – $O(1)$;
- В хужшем – $O(count)$, где $count$ количество элементов в массиве.

Хочу так же заметить, что хоть и маловероятно, но программа может не найти элемент с заданным значеним. Данный случай никак не обрабатывается выданном коде. В параллелизованом производится замер времени, соответсвенно результат вычислений не представляет интереса.

Блок схема поиска элемента:



3 Параллелизация

Очевидно, что наиболее тяжелым структурным элементом является цикл алгоритма.
Для начала зададим несколько опций параллелизации с помощью *omp parallel*:

- *num_threads* - число используемых потоков;
- *shared(array, count, index, target)* - общая для всех потоков память (переменные). Сюда включены массив, его размер, индекс искомого элемента (для сохранения результата) и искомое значение соответсвенно;
- *default(none)* - локальность всех переменных, не указанных в *shared*.

Перед самим циклом поставим *omp for* для распределения его итераций между потоками. При этом не забудем заменить *break* на *omp cancel for*. Эта директива прервет исполнение всех потоков, если мы найдем искомый элемент.

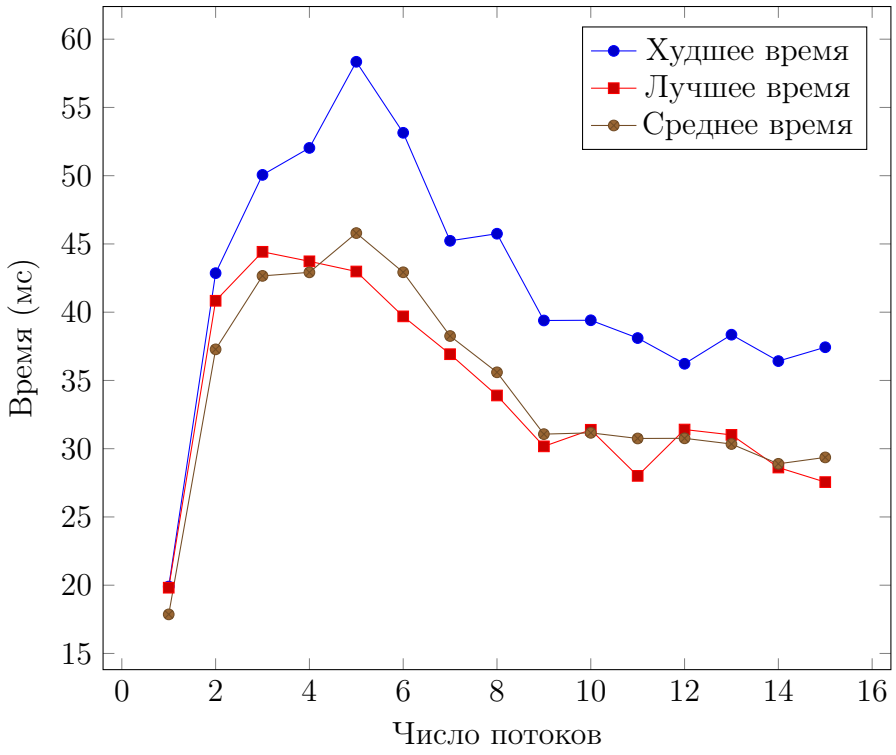
Важно заметить, что для работы *omp cancel for* при запуске необходимо устанавливать значение переменной окружения *OMP_CANCELLATION=true*.

В итоге каждый поток будет по сути придерживатся той же блок схемы, за исключением двух деталей:

- i меняется от $\frac{count}{threads} \cdot tid$ до $\frac{count}{threads} \cdot (tid + 1)$, где $count$ число элементов в массиве, $threads$ – число потоков;
- При обнаружении искомого элемента все потоки прерываются.

4 Экспериментальные данные

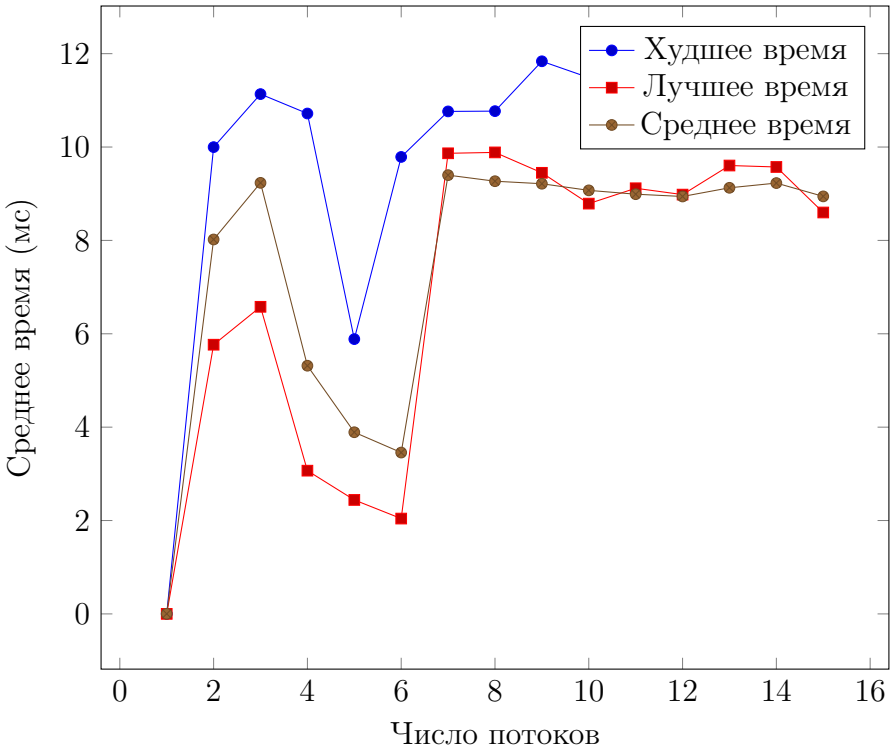
Во всех измерениях бралось 10 запусков на поток.



Из графика видно, что в среднем многопоточная программа работает медленнее. Я могу выделить две основные причины.

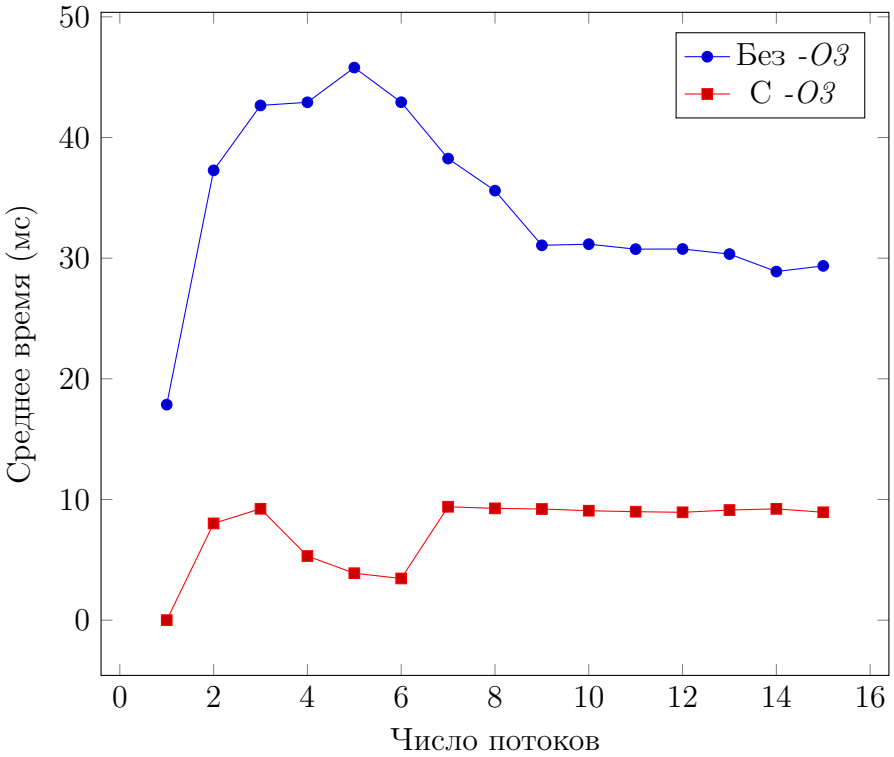
Во-первых представленная задача проста в вычислительном плане. Вполне возможно, что инициализация работы с потоками и их прерывание занимают слишком много времени для такой тривиальной задачи.

Во-вторых оптимизации компилятора. Я провел повторные тесты с добавлением флага *-O3*:

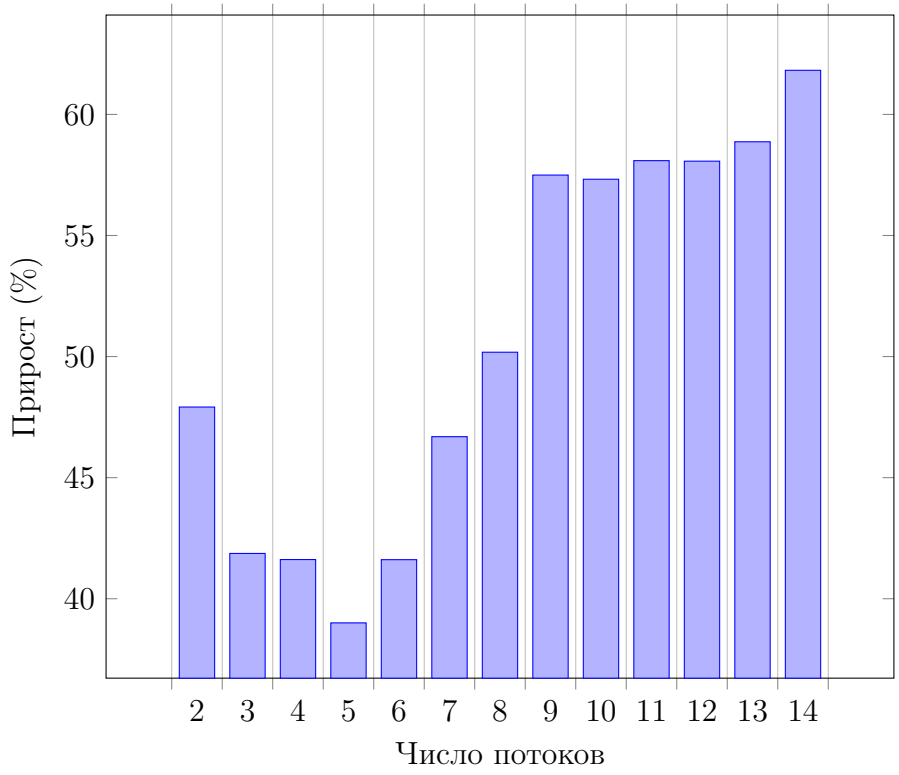


Прекрасно видно, что однопоточная программа лидирует с большим отрывом и причиной этому – оптимизации компилятора.

Для сравнения вот среднее время с *-O3* и без него:



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



5 Заключение

В данной работе было исследовано ускорение, получаемое при использовании многопоточности в задании о нахождении максимума. 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 стоит отметить, что вычислительная сложность программы низка, а соответсвенно инициализация потоков не выгодна.

Приложение А

Использованные программные коды

Оригинальный предоставленный код:

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

int main(int argc, char** argv)
{
    const int count = 10000000;      ///< Number of array elements
    const int random_seed = 920214; ///< RNG seed
    const int target = 16;           ///< Number to look for

    int* array = 0;                  ///< The array we need to find the max in
    int  index = -1;                 ///< The index of the element we need

    /* 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(); }

    /* Find the index of the element */
    for(int i=0; i<count; i++)
    {
        if(array[i] == target)
```

```
        {
            index = i;
            break;
        }
    }

    printf("Found occurence of %d at index %d;\n", target, index);
    return(0);
}
```

Код без многопоточности:

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

int main(int argc, char** argv)
{
    const int count = 10000000;      ///< Number of array elements
    const int random_seed = 920214; ///< RNG seed
    const int target = 16;           ///< Number to look for

    int* array = 0;                  ///< The array we need to find the max in
    int  index = -1;                 ///< The index of the element we need

    /* 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();
    /* Find the index of the element */
    for(int i=0; i<count; i++)
    {
        if(array[i] == target)
        {
            index = i;
            break;
        }
    }
    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);

    return 0;
}
```

Доработанный код:

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

int main(int argc, char** argv)
{
    const int count = 10000000;      ///< Number of array elements
    const int random_seed = 920214; ///< RNG seed
    const int target = 16;           ///< Number to look for

    int* array = 0;                  ///< The array we need to find the max in
    int  index = -1;                 ///< The index of the element we need
```

```
    /* 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();
    /* Find the index of the element */

    #pragma omp parallel num_threads(THREADS) shared(array, count, index, target) default(none)
    {
        #pragma omp for
        for(int i=0; i<count; i++) {
        if(array[i] == target) {
            #pragma omp critical
            index = i;
            #pragma omp cancel for
        }
    }
    }
    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);

    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, env={"OMP_CANCELLATION": "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:

```python
# 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], header[-1]])

    # 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	19.9	19.82	17.86
2	42.86	40.84	37.28
3	50.06	44.42	42.66
4	52.04	43.73	42.92
5	58.34	42.98	45.8
6	53.14	39.69	42.92
7	45.23	36.92	38.26
8	45.75	33.9	35.6
9	39.39	30.17	31.07
10	39.41	31.39	31.16
11	38.11	28.01	30.75
12	36.22	31.4	30.76
13	38.35	31.01	30.34
14	36.42	28.62	28.89
15	37.43	27.55	29.36

Таблица с оптимизациями:

Threads	Worst (ms)	Best (ms)	Average (ms)
1	$2 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	$2 \cdot 10^{-3}$
2	10	5.77	8.02
3	11.13	6.58	9.23
4	10.72	3.07	5.32
5	5.89	2.44	3.89
6	9.79	2.04	3.46
7	10.76	9.87	9.4
8	10.77	9.89	9.27
9	11.84	9.45	9.21
10	11.49	8.79	9.07
11	11.19	9.12	8.99
12	10.51	8.98	8.94
13	10.79	9.6	9.13
14	11.06	9.57	9.23
15	11.15	8.6	8.94

Таблица сравнений:

Threads	Efficiency
2	47.92
3	41.87
4	41.62
5	39
6	41.61
7	46.69
8	50.18
9	57.5
10	57.32
11	58.09
12	58.07
13	58.87
14	61.82
15	60.83