Лабораторная работа №1

"Введение в паралеллельные вычисления. Технолония 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
```

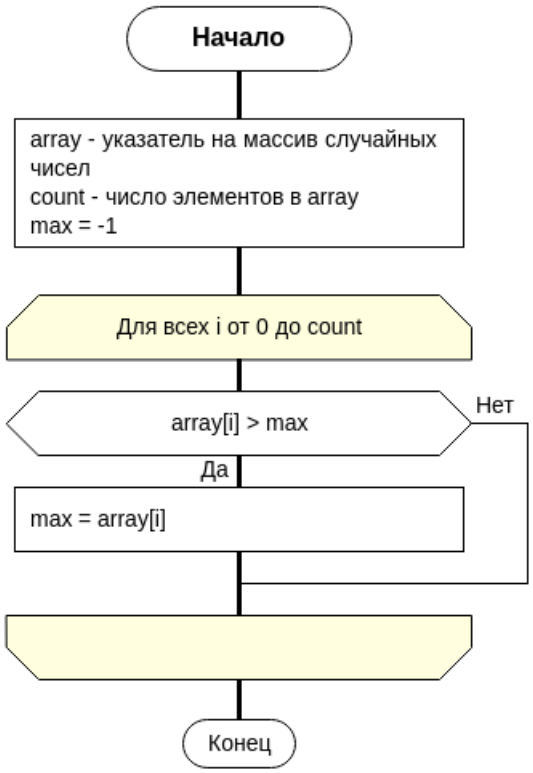
Согласно официальной документации даная версия компилятора поддерживает *OpenMP 5.0*

2 Анализ алгоритма

Данный алгоритм ищет максимум в массиве со случайно сгенерированными значениями. Длинна задается постоянной.
Временная сложность $O(\frac{count}{threads})$, где:

- *count* – число элементов в массиве;
- *threads* – число используемых потоков.

Блок схема поиска максимума:



При этом директива *omp for* распределяет итерации цикла между потоками. Если бы её не было, то благодаря *omp parallel* поиск максимума в каждом потоке был бы произведен по всему массиву.

Директива *omp parallel* задает несколько опций параллелизации:

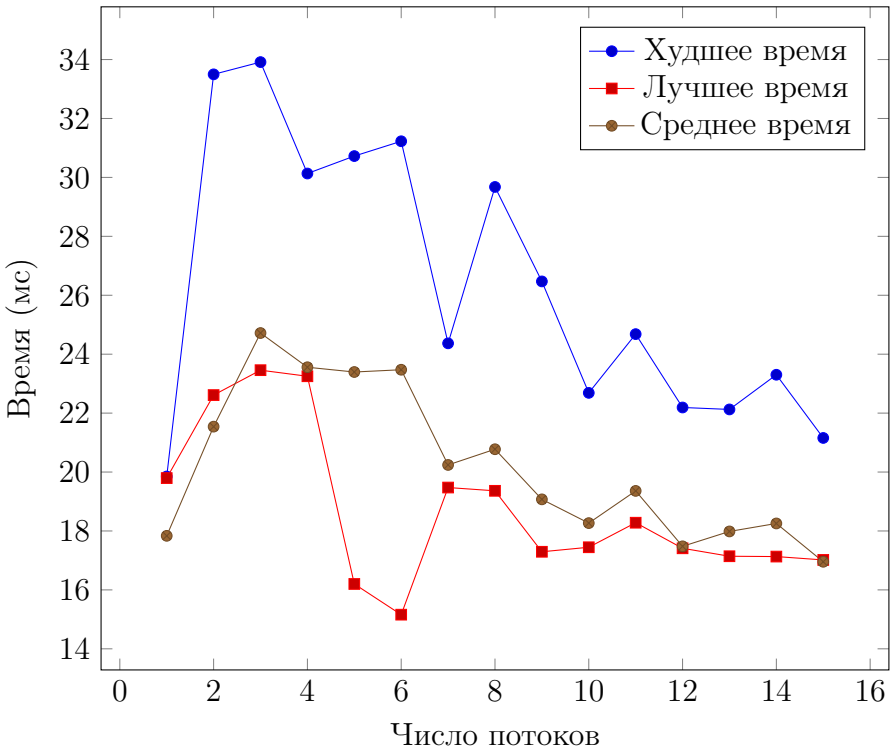
- *num_ threads* - используемых потоков;
- *shared* - общая для всех потоков память (переменные);
- *reduction* - способ объединения локальных данных потоков после окончания параллельного промежутка. В данном случае берется максимальный среди них;
- *default* - локальность перемнных по умолчанию. В данном случае по умолчанию все переменные локальные.

Без *omp parallel* программа компилируется без ошибок, однако цикл выполняется одним потоком.

Предоставленная программа была изменена с целью измерения производимости и освобождения памяти. В оригинале выделенная память не освобождалась.

3 Экспериментальные данные

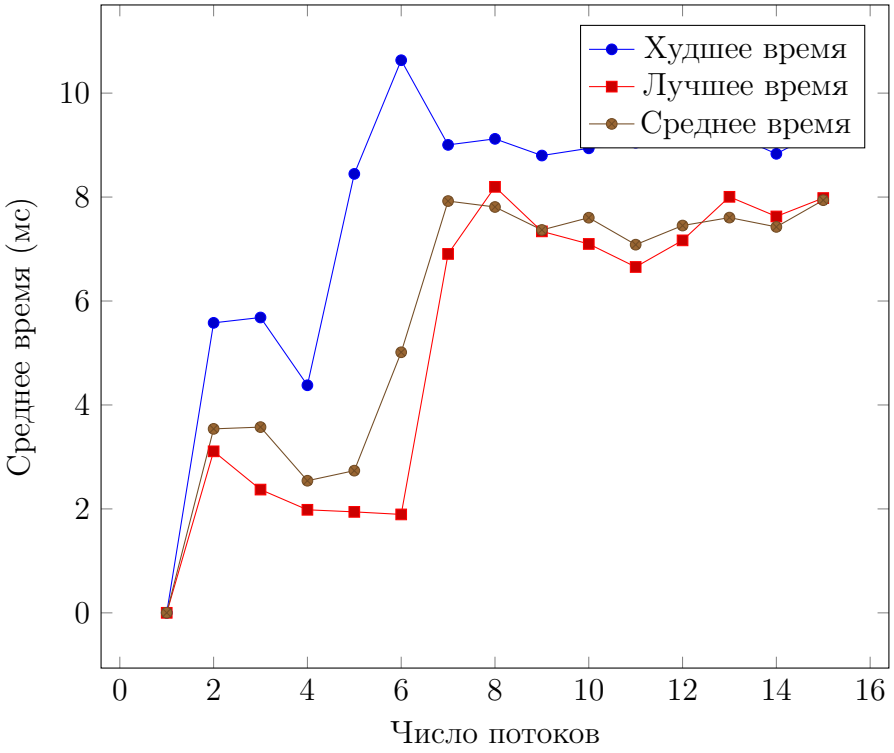
Во всех измерениях бралось 10 запусков на поток.



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

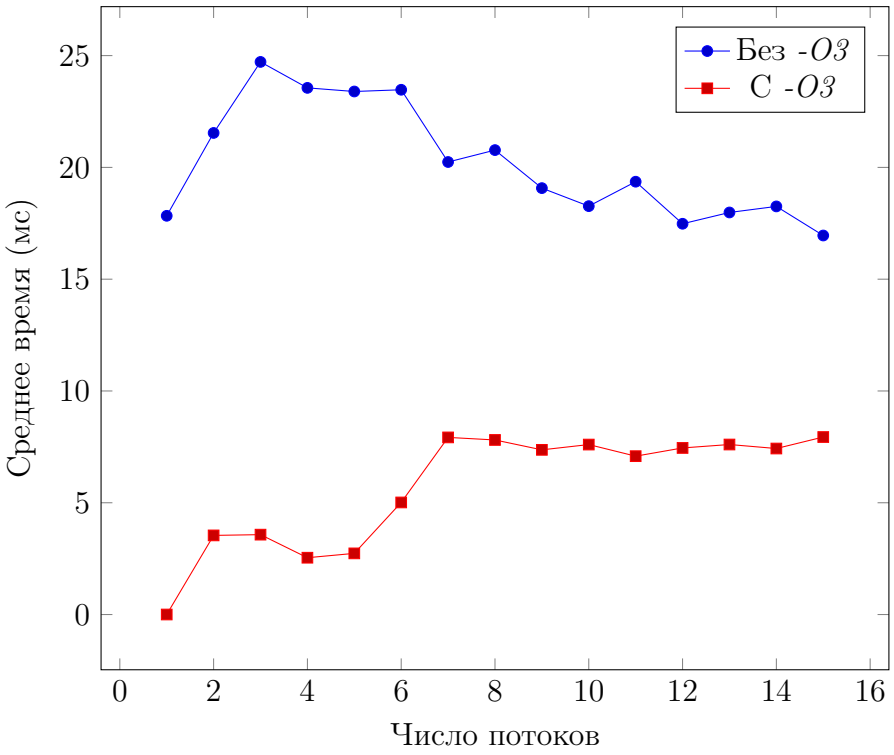
Во-первых представленная задача проста в вычислительном плане. Вполне возможно, что инициализация работы с потоками занимает слишком много времени для такой тривиальной задачи.

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

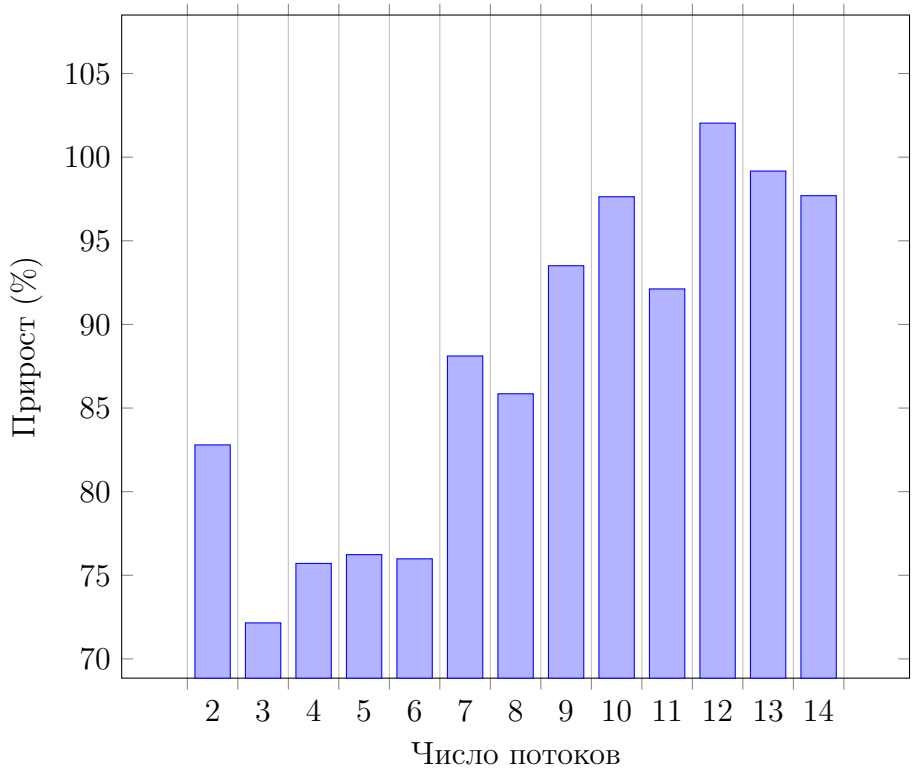


Прекрасно видно, что однопоточная программа лидирует с большим отрывом и причиной этому – оптимизации компилятора.

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



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



4 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 threads = 16;          ///< Number of parallel threads to use
    const int random_seed = 920215; ///< RNG seed

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

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

    /* Determine the OpenMP support */
    printf("OpenMP: %d;\n=======\n", _OPENMP);

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

    /* Find the maximal element */
```

```
    // #pragma omp parallel num_threads(threads) shared(array, count) reduction(max: max) default(none)
    {
        #pragma omp for
        for(int i=0; i<count; i++)
        {
            if(array[i] > max) { max = array[i]; };
        }
        printf("— My max is: %d;\n", max);
    }

    printf("=======\nMax is: %d;\n", max);
    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 = 920215; ///< RNG seed

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

    /* 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 maximal element */
    {
        for(int i=0; i<count; i++)
        {
            if(array[i] > max) { max = array[i]; };
        }
    }
    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;
}
```

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 = 920215; ///< RNG seed
```

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

    /* 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 maximal element */
    #pragma omp parallel num_threads(THREADS) shared(array, count) reduction(max: max) default(none)
    {
        #pragma omp for
        for(int i=0; i<count; i++)
        {
            if(array[i] > max) { max = array[i]; };
        }
    }
    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	19.85	19.8	17.84
2	33.5	22.61	21.54
3	33.91	23.46	24.72
4	30.13	23.25	23.56
5	30.72	16.2	23.4
6	31.23	15.16	23.47
7	24.37	19.47	20.24
8	29.68	19.36	20.77
9	26.47	17.29	19.07
10	22.69	17.45	18.27
11	24.68	18.28	19.36
12	22.19	17.41	17.48
13	22.13	17.14	17.98
14	23.3	17.13	18.26
15	21.16	17.02	16.95

Таблица с оптимизациями:

Threads	Worst (ms)	Best (ms)	Average (ms)
1	$2 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	$2 \cdot 10^{-3}$
2	5.58	3.11	3.54
3	5.68	2.37	3.57
4	4.38	1.98	2.54
5	8.45	1.94	2.74
6	10.63	1.89	5.01
7	9	6.91	7.92
8	9.12	8.2	7.81
9	8.8	7.34	7.37
10	8.94	7.1	7.6
11	9.05	6.66	7.08
12	9.83	7.17	7.45
13	9.23	8.01	7.61
14	8.83	7.63	7.43
15	9.3	7.98	7.94

Таблица сравнений:

Threads	Efficiency
2	82.8
3	72.15
4	75.71
5	76.23
6	75.98
7	88.11
8	85.85
9	93.51
10	97.64
11	92.12
12	102.04
13	99.17
14	97.7
15	105.2