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:
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 Анализ алгоритма

2.1 Принцип работы

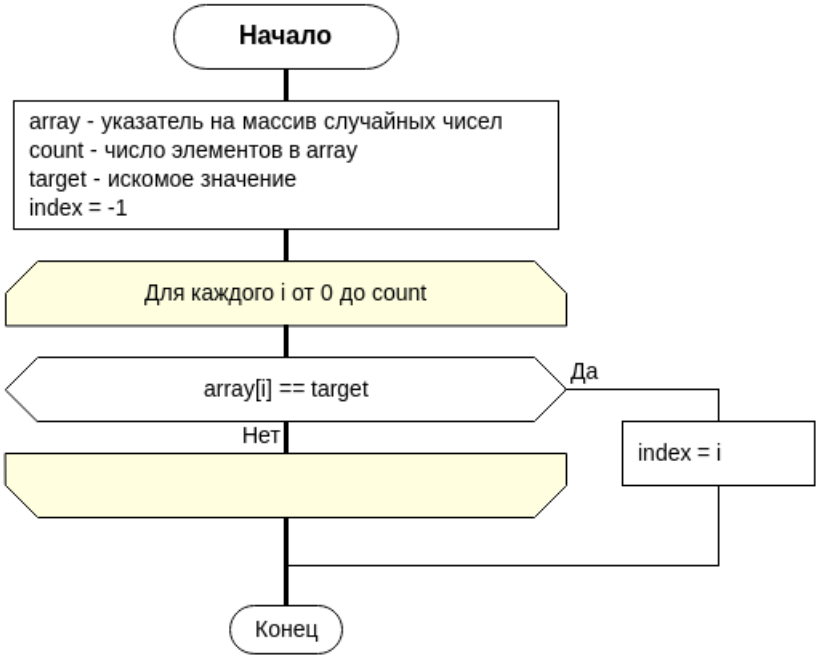
Алгоритм является поиском элемента в массиве. Программа:

1. Выделяет память под массив, инициализирует генератор случайных значений;
2. Заполняет массив случайными числами;
3. Ищет элемент с определенными настройками *OpenMP*.

Результатом поиска является индекс элемента в массиве. Поиск осуществляется последовательно.

Временная сложность алгоритма $O(n)$, где n – число элементов в массиве.

Блок-схема алгоритма выглядит следующим образом:



2.2 Параллелизация

Аналогично предыдущей лабораторной работе алгоритм можно параллелизовать, распределив итерации между потоками. Единственное существенное отличие – если элемент был встречен, то необходимо в тот же момент выйти из цикла.

Рассмотрим задаваемые опции параллелизации:

- *num_ threads* - число используемых потоков;
- *shared(array, N, 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*.

Для преждевременного прерывания поискового цикла (аналог *break*) будет использоваться *omp cancel for*. Эта директива прервет все потоки как только искомый элемент будет найден. Для работы директивы *omp cancel for* может потребоваться установка переменной окружения *OMP_ CANCELLATION=true*.

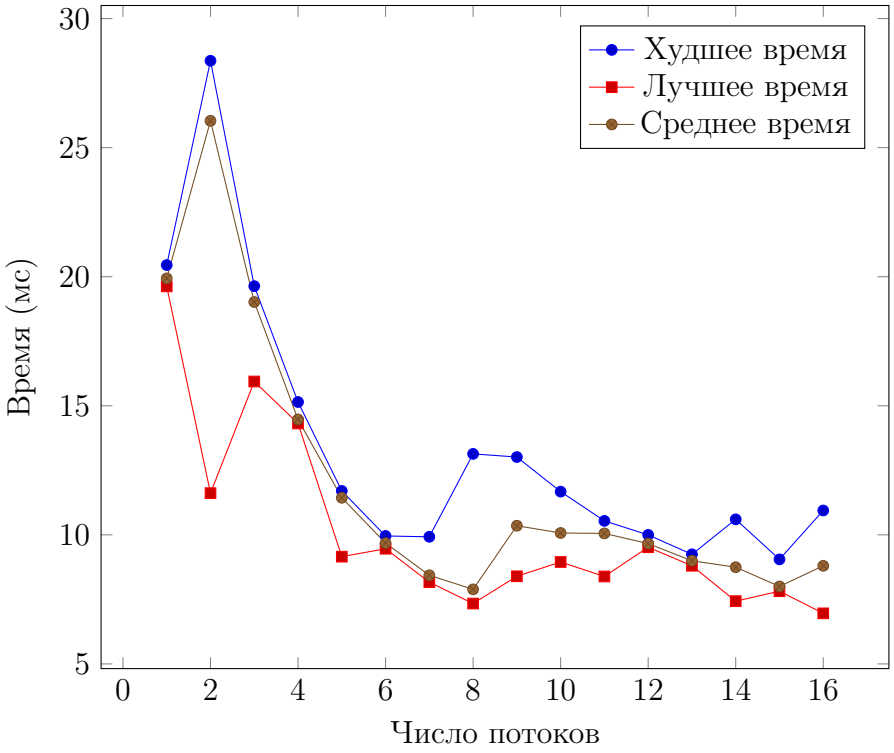
Также так как используется общая переменная *index*, в операциях с ней следует использовать *omp critical*.

3 Экспериментальные данные

На каждое число потоков отводилось 20 запусков.

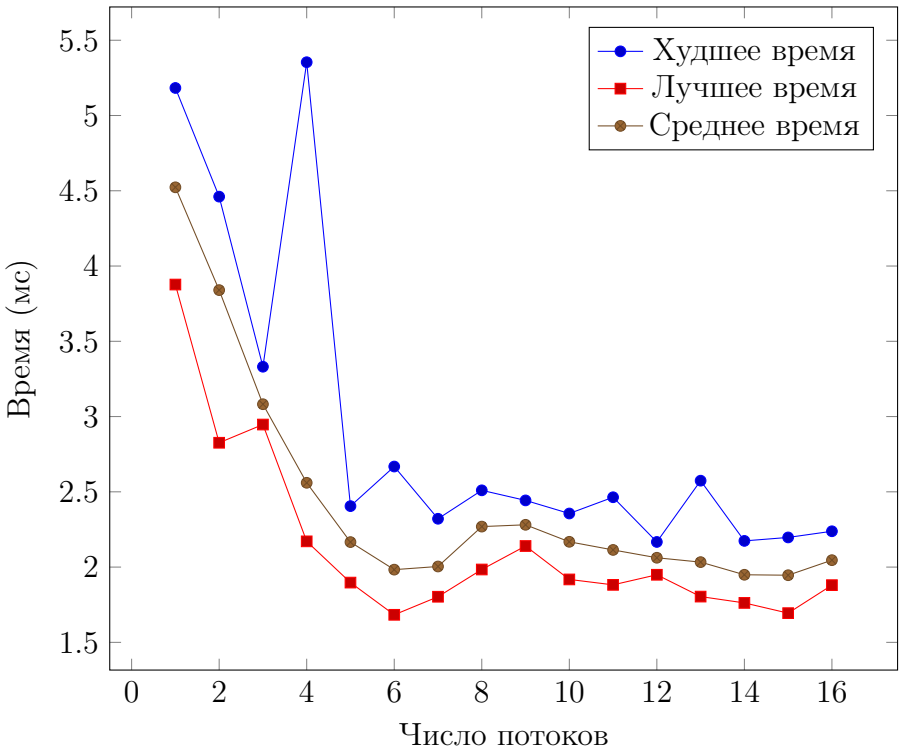
3.1 Время выполнения

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



Крайне заметно, что многопоточная программа работает куда быстрее её конкурентов. Единственным исключением является запуск при 2-х потоках.

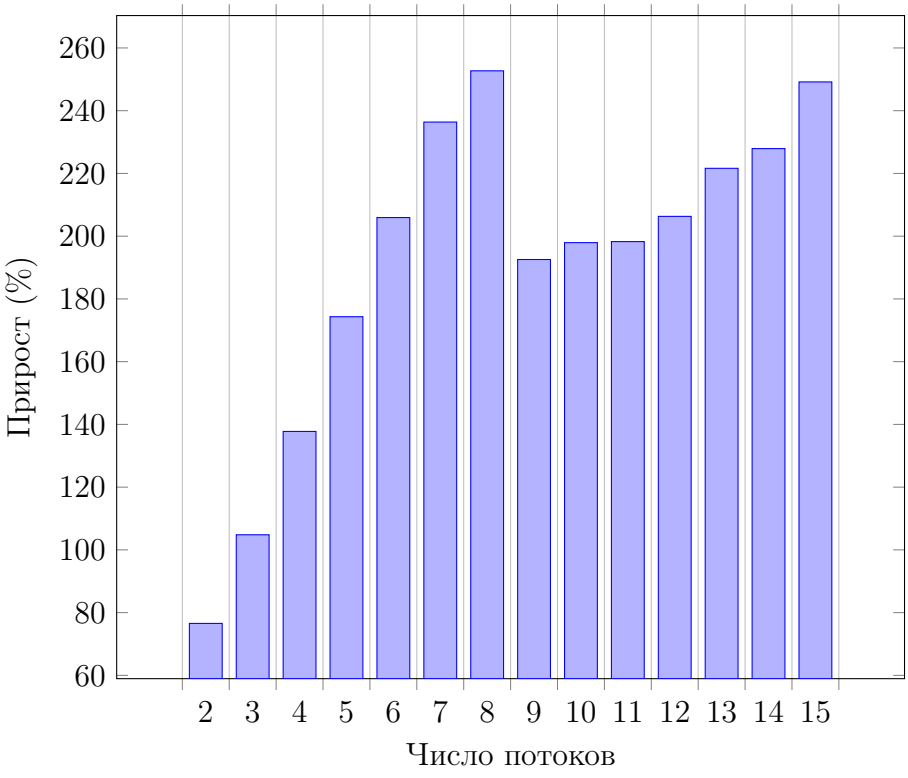
Аналогично первой лабораторной рассмотрим теперь данные с оптимизацией:



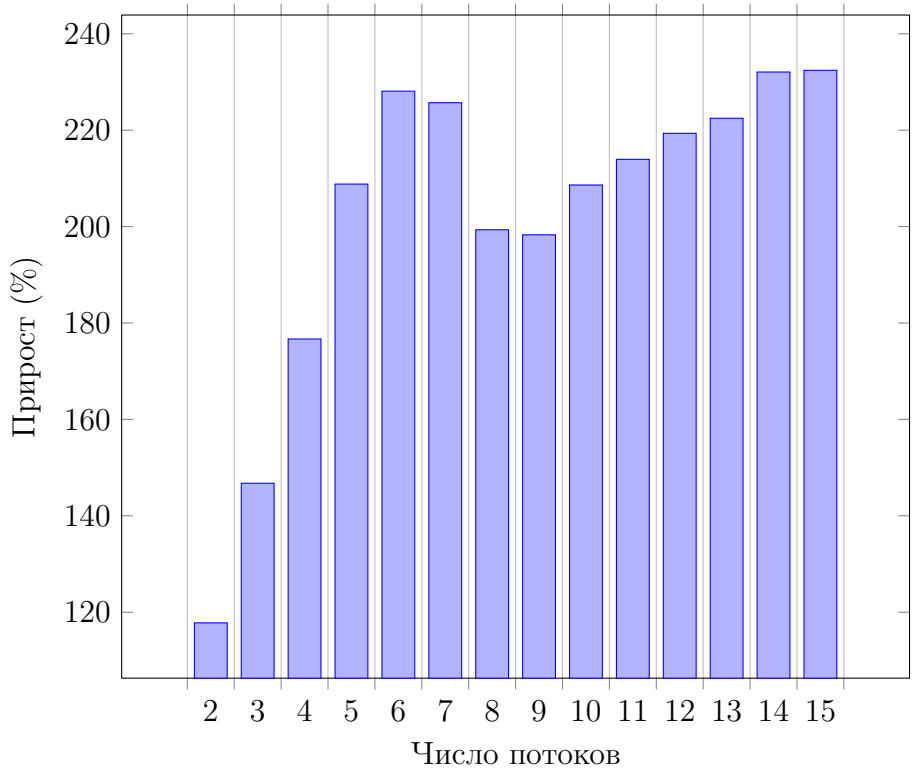
Как видно на графике выше, повышение числа потоков уменьшает среднее время исполнения.

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:


240
220
200
180
160
140
120
Прирост (%)
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 нескольких потоков в задании о поиске элемента. 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 одна была медленнее однопоточной. При этом наблюдался прирост вплоть до 2-х с половиной 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 = 10000000;
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, const int target) {
    double start = omp_get_wtime();;
    int index = -1;
    #pragma omp parallel num_threads(threads) shared(array, size, index, target) default(none)
    {
        #pragma omp for
        for(int i = 0; i < size; ++i) {
            if(array[i] == target) {
                #pragma omp critical
                index = array[i];
                #pragma omp cancel for
            };
        }
    }
    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, 16);
            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	20.45	19.63	19.94
2	28.37	11.62	26.04
3	19.64	15.94	19.02
4	15.15	14.32	14.47
5	11.7	9.16	11.44
6	9.96	9.46	9.68
7	9.93	8.17	8.43
8	13.14	7.34	7.89
9	13.01	8.4	10.35
10	11.67	8.95	10.07
11	10.54	8.39	10.06
12	10	9.52	9.66
13	9.25	8.8	9
14	10.6	7.43	8.75
15	9.05	7.82	8
16	10.94	6.96	8.8

Таблица с оптимизациями:

Threads	Worst (ms)	Best (ms)	Avg (ms)
1	5.18	3.88	4.52
2	4.46	2.83	3.84
3	3.33	2.95	3.08
4	5.35	2.17	2.56
5	2.41	1.9	2.17
6	2.67	1.68	1.98
7	2.32	1.8	2
8	2.51	1.98	2.27
9	2.44	2.14	2.28
10	2.36	1.92	2.17
11	2.46	1.88	2.11
12	2.17	1.95	2.06
13	2.57	1.8	2.03
14	2.17	1.76	1.95
15	2.2	1.69	1.95
16	2.24	1.88	2.05

Таблица сравнений без оптимизаций:

Threads	Efficiency
2	76.56
3	104.82
4	137.76
5	174.33
6	205.93
7	236.38
8	252.71
9	192.56
10	197.94
11	198.27
12	206.31
13	221.63
14	227.92
15	249.17
16	226.55

Таблица сравнений с оптимизациями:

Threads	Efficiency
2	117.79
3	146.76
4	176.68
5	208.82
6	228.09
7	225.7
8	199.34
9	198.29
10	208.63
11	213.96
12	219.35
13	222.48
14	232.07
15	232.43
16	221.07