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

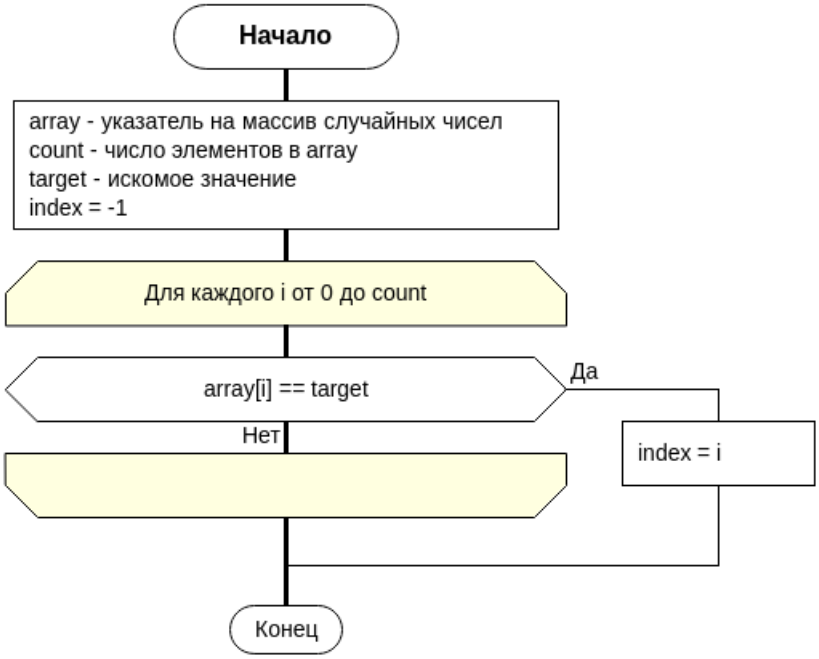
Алгоритм является поиском элемента в массиве. Программа:

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*.

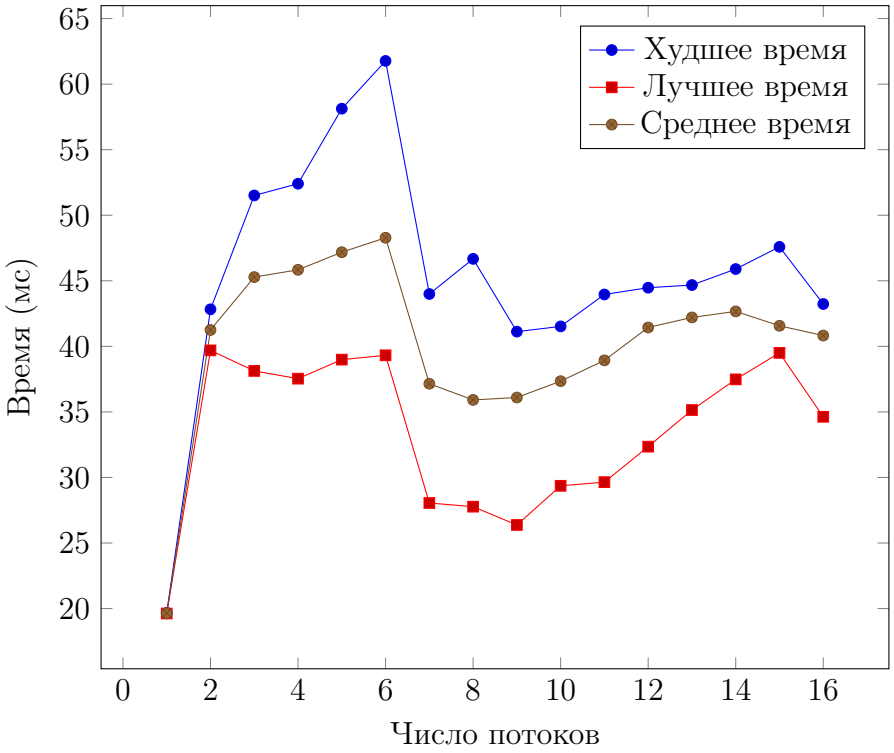
Также так как используется общая переменная *index*, в операциях с ней следует использовать *omp critical*.

3 Экспериментальные данные

На каждое число потоков отводилось 20 запусков.

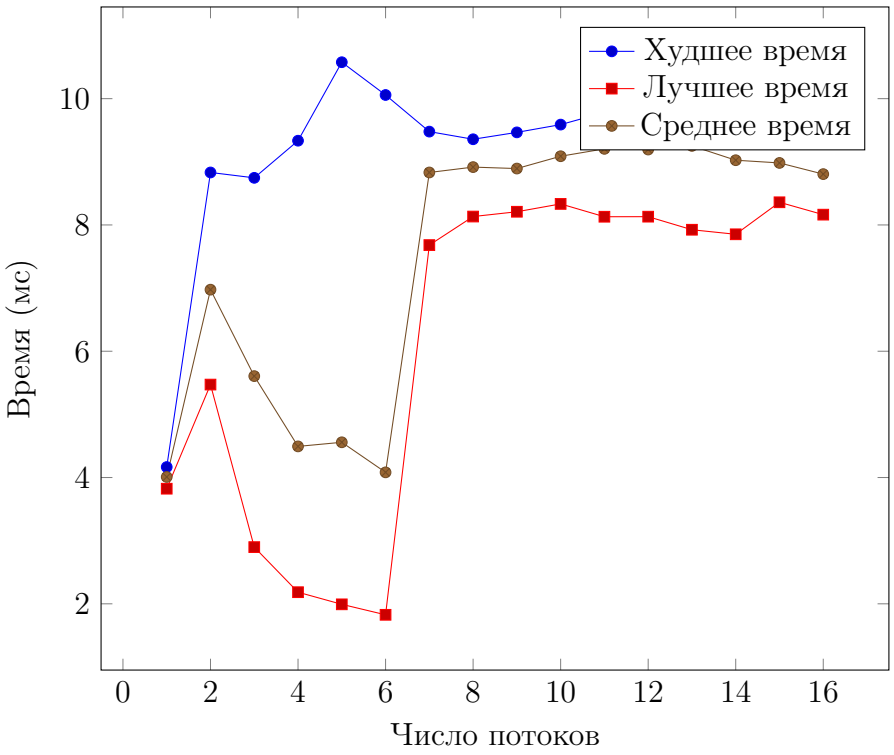
3.1 Время выполнения

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



Крайне заметно, что однопотчная программа работает куда быстрее её конкурентов. Возможно здесь, как и в первой лабораторной причиной является относительная простота алгоритма. Тоесть программа тратит на подготовку к многопоточному исполнению времени больше, чем она от этого выигрывает.

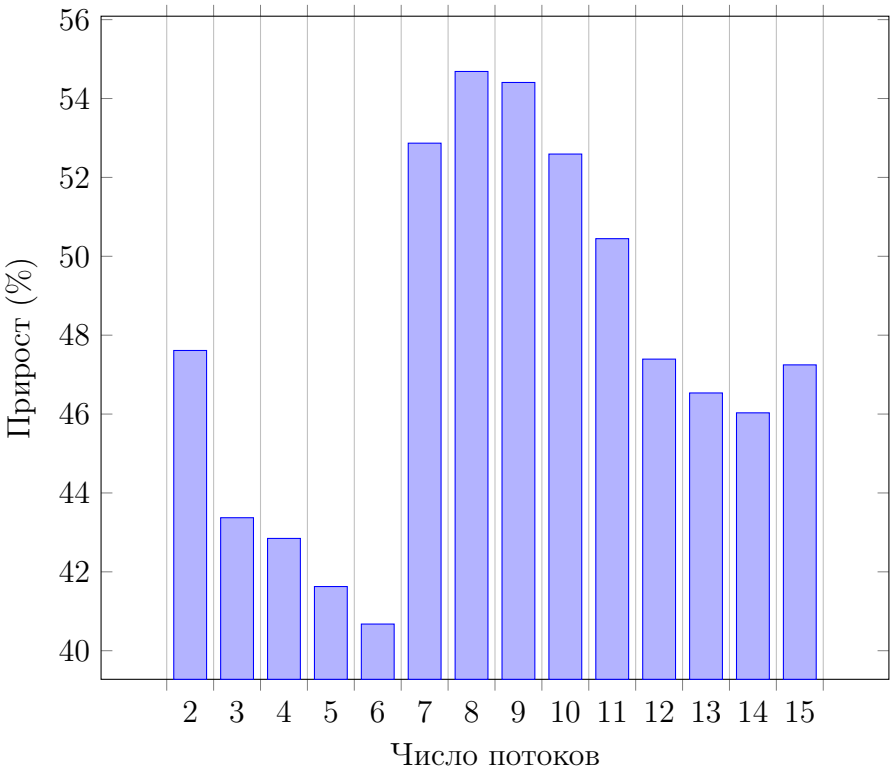
Аналогично уже упомянутой первой лабораторной рассмотрим теперь данные с оптимизацией:



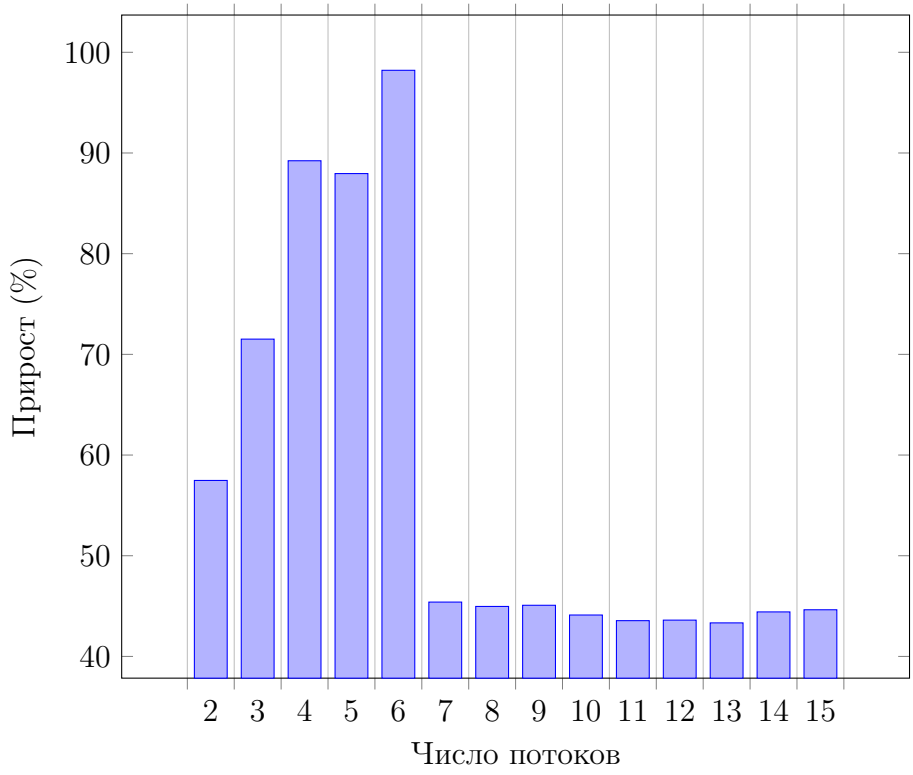
Как видно на графике выше, повышение числа потоков лишь увеличивает среднее время исполнения.

3.2 Прирост производительности

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



Для оптимизированной сборки:


100
90
80
70
60
50
40
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 многопоточности в задании о поиске элемента. 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 тем, что инициализация работы с несколькими потоками занимает больше времени, чем получается “выйграть” за ее счет.

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 = 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) {
    clock_t start = clock();
    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
            };
        }
    }
    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, 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	19.67	19.62	19.64
2	42.83	39.69	41.25
3	51.51	38.13	45.28
4	52.4	37.54	45.84
5	58.13	38.99	47.18
6	61.77	39.32	48.28
7	43.99	28.05	37.15
8	46.67	27.77	35.91
9	41.12	26.38	36.1
10	41.52	29.36	37.34
11	43.96	29.64	38.93
12	44.47	32.35	41.44
13	44.68	35.14	42.21
14	45.9	37.48	42.67
15	47.59	39.5	41.57
16	43.23	34.63	40.82

Таблица с оптимизациями:

Threads	Worst (ms)	Best (ms)	Avg (ms)
1	4.17	3.82	4.01
2	8.83	5.47	6.98
3	8.75	2.9	5.61
4	9.33	2.18	4.49
5	10.58	1.99	4.56
6	10.06	1.83	4.08
7	9.48	7.68	8.83
8	9.36	8.13	8.92
9	9.47	8.21	8.89
10	9.59	8.33	9.09
11	9.78	8.13	9.21
12	9.69	8.13	9.19
13	9.75	7.92	9.25
14	9.53	7.85	9.03
15	9.63	8.36	8.98
16	9.43	8.16	8.81

Таблица сравнений без оптимизаций:

Threads	Efficiency
2	47.61
3	43.37
4	42.85
5	41.63
6	40.68
7	52.87
8	54.69
9	54.41
10	52.59
11	50.45
12	47.39
13	46.54
14	46.03
15	47.25
16	48.11

Таблица сравнений с оптимизациями:

Threads	Efficiency
2	57.48
3	71.51
4	89.23
5	87.96
6	98.21
7	45.4
8	44.96
9	45.08
10	44.11
11	43.55
12	43.61
13	43.33
14	44.42
15	44.63
16	45.53