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:
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

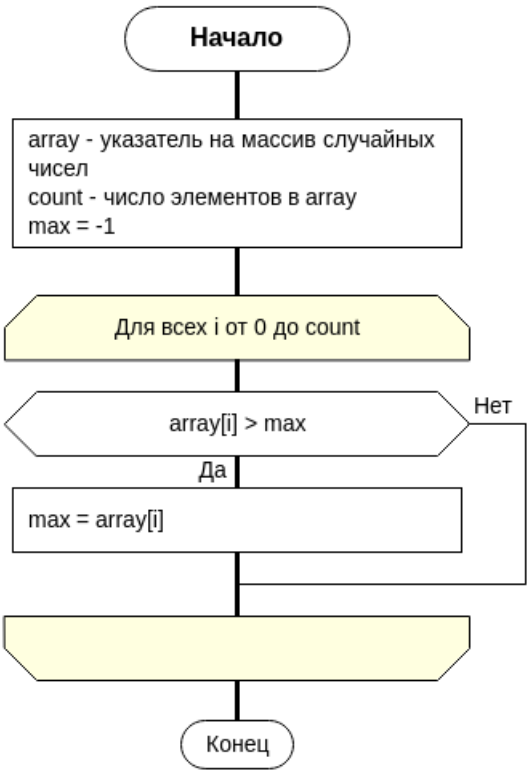
Алгоритм является поиском максимума в массиве. Программа:

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

Поиск максимума осуществляется параллельно. Массив разбивается на секции (почти) равного размера, которые распределяются между потоками. Распределение зависит от параметра *schedule*, который здесь не указан. В таком случае он определяется компилятором/ОС.
Временная сложность алгоритма $O(\frac{n}{p})$, где:

- n – число элементов в массиве;
- p – число используемых потоков.

Блок-схема алгоритма в одном потоке выглядит следующим образом:



2.2 Директивы *OpenMP*

Поясним представленные директивы *OpenMP*.
Директива *parallel* задает опции параллелизации:

- *num_ threads* – число потоков;
- *shared* – общая для потоков память;
- *reduction* – способ объединения локальных переменных в глобальную. В данном случае вычисление максимума;

- *default* – локальность переменных *по умолчанию*. В данном случае все переменные по умолчанию локальные.

Если бы данной директивы не было, то следующий за ней блок кода исполнялся бы одним потоком без участия *OpenMP*.

Директива *for* используя опции, задаваемые директивой *parallel* распределяет итерации цикла между потоками. Если бы данной директивы не было, то цикл, следующий за ней, выполнился бы во всех потоках (не было бы распределения итераций).

3 Экспериментальные данные

3.1 Модификации кода

Исходный код программы был модифицирован так, что бы:

- Освобождалась **вся** выделенная память;
- Производился запуск с несколькими потоками;
- Замерялось время, затраченное на выполнение алгоритма.

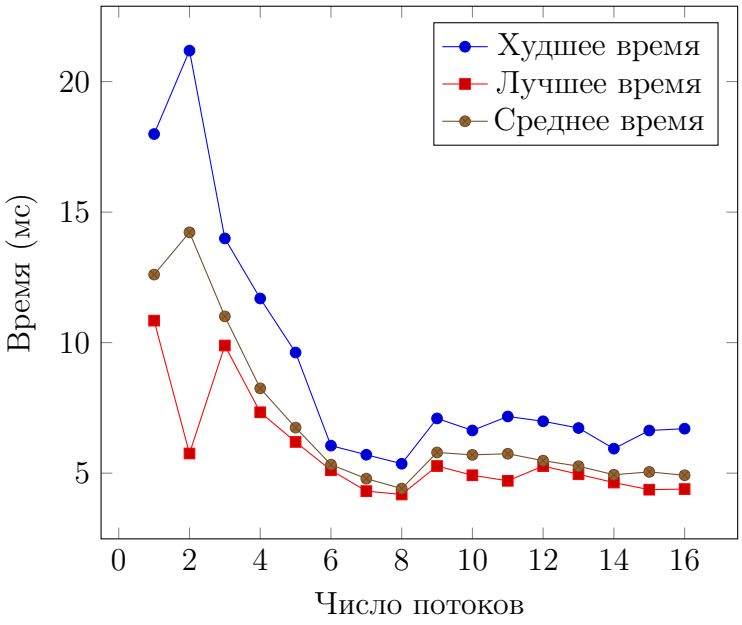
На каждое число потоков отводилось 20 запусков.

3.2 Количество операций сравнения

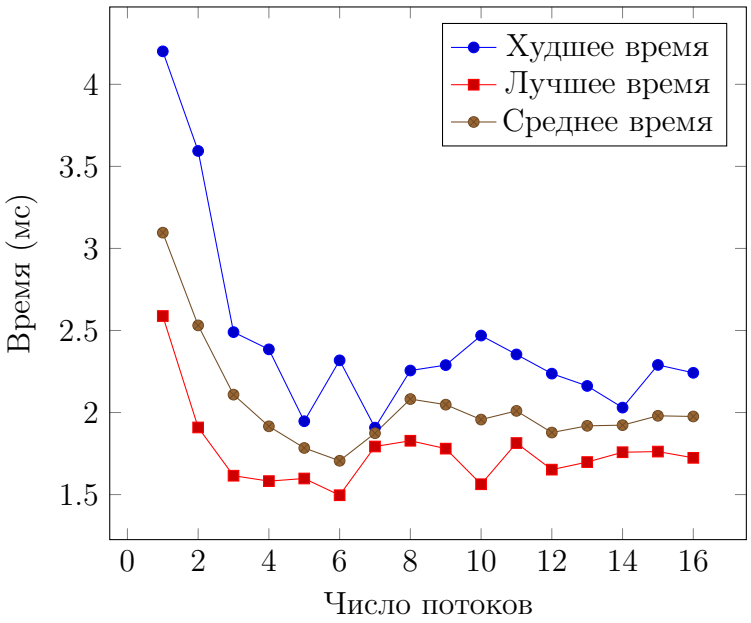
В измерении количества операций сравнения нет смысла, ведь для однопоточной программы: $O(n)$, как и для многопоточной (сравнения из *reduction* – сравнения с крайними элементами секций).

3.3 Время выполнения

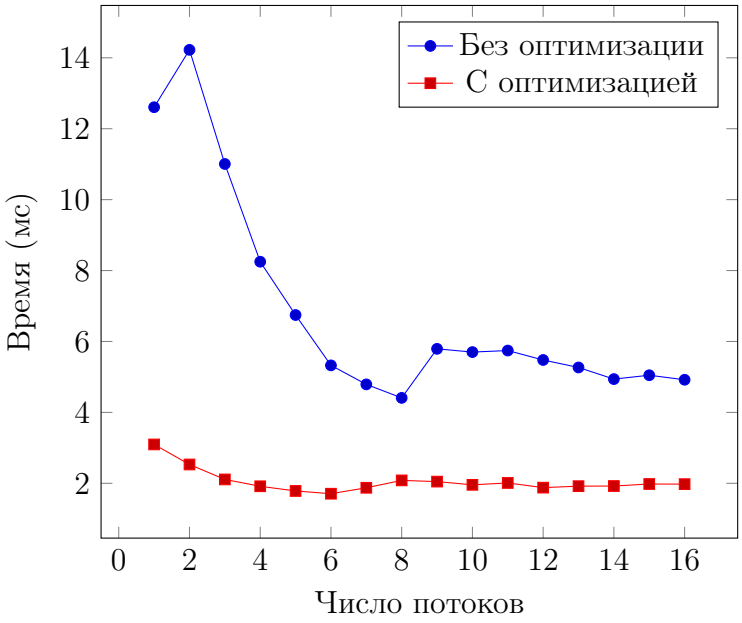
Для начала я решил взглянуть не только на среднюю скорость выполнения, но и на крайние варианты:



Многопоточная программа практически во всех случаях работает гораздо быстрее. Исключением стали 2 потока, но небольшая разница в производительности позволяет списать это на погрешность. Рассмотрим как изменится ситуация при включении оптимизаций:

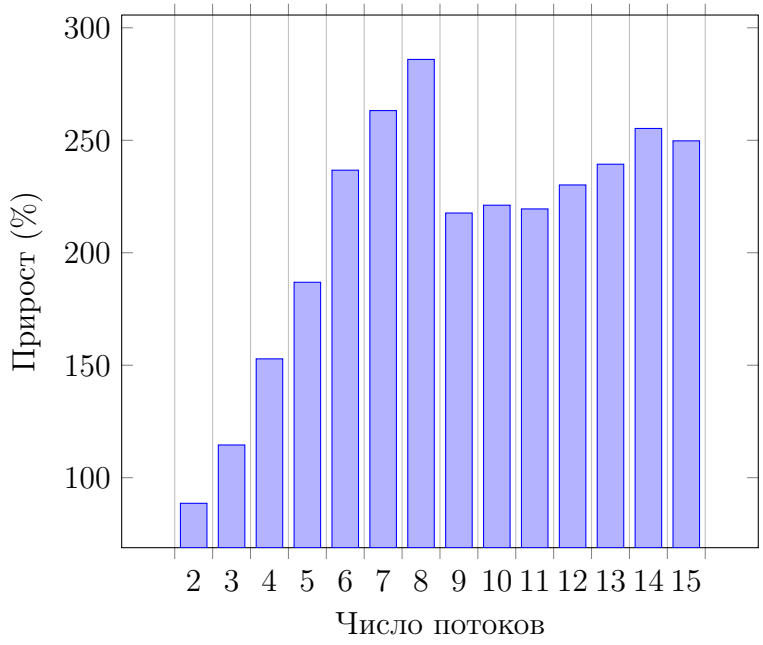


Как видно на графике выше, общая тенденция остается неизменной. Оптимизации, производимые компилятором положительно влияют на производительность вне зависимости от числа используемых потоков:

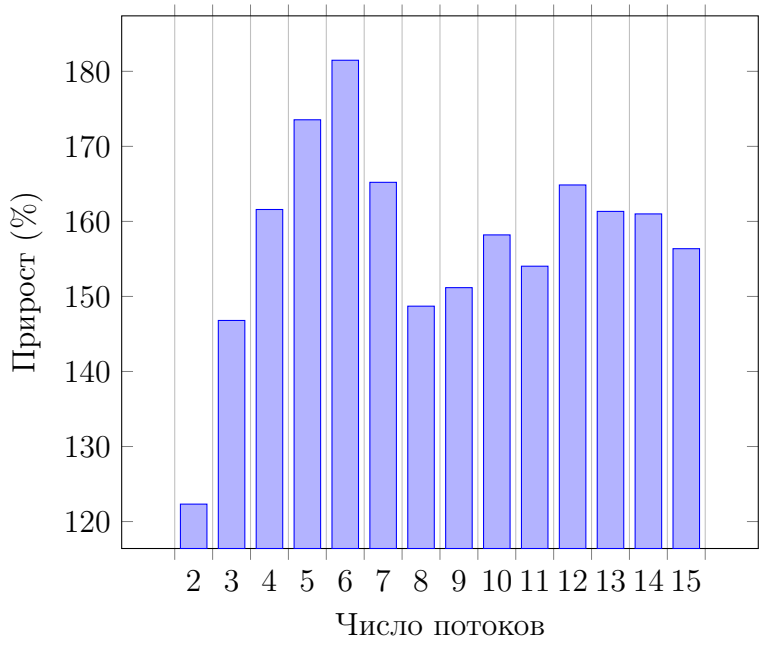


3.4 Прирост производительности

Как уже было замечено ранее, в целом с увеличением числа потоков производительность растет. Рассмотрим ускорение многопоточной программы относительно однопоточной. Для не оптимизированной сборки:



Для оптимизированной сборки:



Наибольший прирост наблюдается при отсутствии оптимизаций и практически достигает 300%!

4 Заключение

В данной работе было исследовано ускорение, получаемое при использовании нескольких потоков в задании о поиске максимума. 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 3х 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) {
    double start = omp_get_wtime();
    int max = -1;
    #pragma omp parallel num_threads(threads) shared(array, size) reduction(max: max) default(none)
    {
        #pragma omp for
        for(int i = 0; i < size; ++i) {
            if(array[i] > max) {
                max = array[i];
            };
        }
    }
    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);
            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	17.99	10.84	12.61
2	21.19	5.75	14.23
3	13.99	9.89	11.01
4	11.69	7.33	8.25
5	9.62	6.2	6.75
6	6.05	5.11	5.33
7	5.71	4.31	4.79
8	5.36	4.18	4.41
9	7.1	5.27	5.79
10	6.64	4.92	5.7
11	7.17	4.71	5.74
12	6.99	5.27	5.48
13	6.73	4.96	5.27
14	5.94	4.64	4.94
15	6.63	4.37	5.05
16	6.7	4.39	4.92

Таблица с оптимизациями:

Threads	Worst (ms)	Best (ms)	Avg (ms)
1	4.2	2.59	3.1
2	3.59	1.91	2.53
3	2.49	1.62	2.11
4	2.39	1.58	1.92
5	1.95	1.6	1.78
6	2.32	1.5	1.71
7	1.91	1.79	1.87
8	2.26	1.83	2.08
9	2.29	1.78	2.05
10	2.47	1.56	1.96
11	2.35	1.81	2.01
12	2.24	1.65	1.88
13	2.16	1.7	1.92
14	2.03	1.76	1.92
15	2.29	1.76	1.98
16	2.24	1.72	1.98

Таблица сравнений без оптимизаций:

Threads	Efficiency
2	88.63
3	114.55
4	152.83
5	186.88
6	236.71
7	263.19
8	285.94
9	217.66
10	221.14
11	219.48
12	230.14
13	239.36
14	255.25
15	249.74
16	256.24

Таблица сравнений с оптимизациями:

Threads	Efficiency
2	122.32
3	146.8
4	161.59
5	173.54
6	181.48
7	165.21
8	148.7
9	151.17
10	158.2
11	154.03
12	164.86
13	161.33
14	161
15	156.36
16	156.68