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

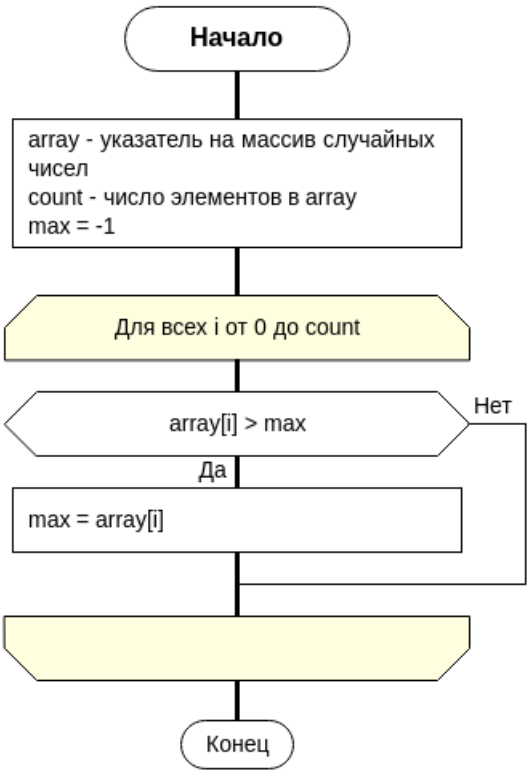
Алгоритм является поиском максимума в массиве. Программа:

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 однопоточного исполнения алгоритма 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.

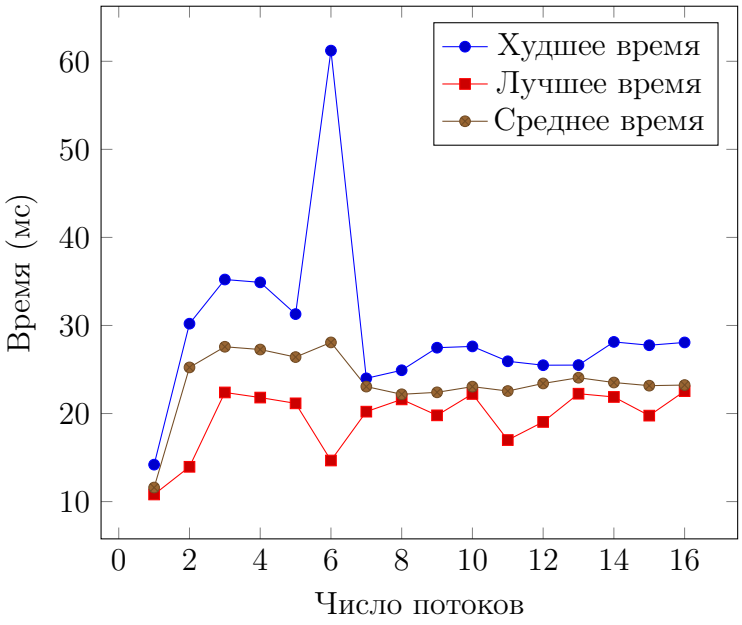
На каждое число потоков отводилось 20 запусков.

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).

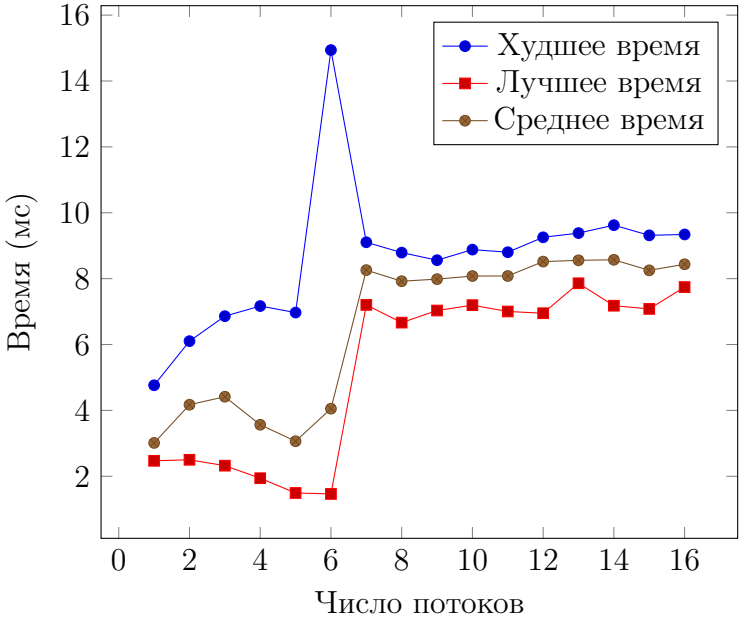
3.3 Время выполнения

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

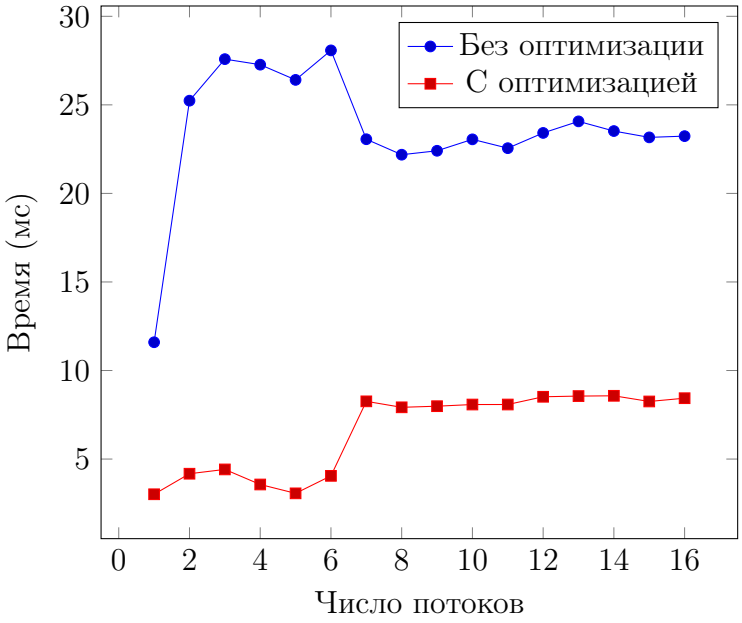


Крайне заметно, что однопотчная программа работает куда быстрее её конкурентов. Первая мысль, которая пришла мне в голову: причиной является относительная простота алгоритма. Тоесть вполне возможно, что программа тратит на подготовку к многопоточному исполнению времени больше, чем она от этого выигрывает.

Другой мыслью были оптимизации, вносимые компилятором. Возможно однопоточную программу легче оптимизировать – отсюда и разница в скорости:

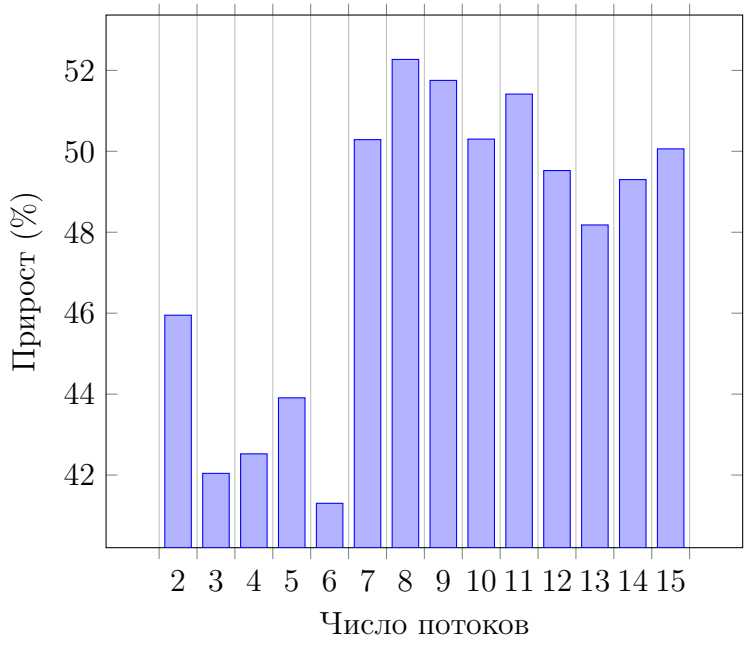


Как видно на графике выше, предположение неверное. Оптимизации уменьшают время исполнения программы вне зависимости от числа потоков. Общая тенденция остается такой же – программа замедляется при увеличении числа потоков:

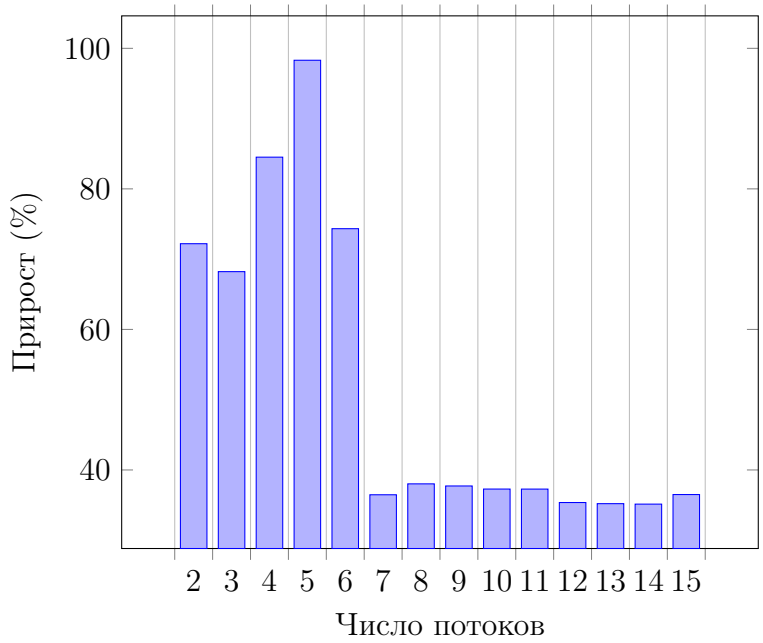


3.4 Прирост производительности

Как уже было замечено ранее, в целом с увеличением числа потоков производительность падает. Рассмотрим ускорение многопоточной программы относиельно однопоточной. Для неоптимизированной сборки:



Для оптимизированной сборки:



4 Заключение

В данной работе было исследовано ускорение, получаемое при использовании многопоточности в задании о нахождении максимума. 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) {
    clock_t start = clock();
    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];
            };
        }
    }
    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);
            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	14.19	10.81	11.6
2	30.2	13.95	25.24
3	35.22	22.39	27.58
4	34.9	21.81	27.27
5	31.29	21.16	26.41
6	61.21	14.67	28.08
7	24	20.21	23.06
8	24.91	21.61	22.19
9	27.48	19.79	22.41
10	27.62	22.23	23.05
11	25.93	16.99	22.55
12	25.49	19.05	23.42
13	25.5	22.25	24.07
14	28.13	21.88	23.52
15	27.76	19.76	23.16
16	28.07	22.55	23.23

Таблица с оптимизациями:

Threads	Worst (ms)	Best (ms)	Avg (ms)
1	4.76	2.47	3.01
2	6.1	2.5	4.17
3	6.86	2.32	4.42
4	7.17	1.94	3.56
5	6.97	1.49	3.06
6	14.94	1.46	4.05
7	9.1	7.2	8.26
8	8.79	6.67	7.92
9	8.56	7.03	7.99
10	8.88	7.2	8.08
11	8.8	7	8.08
12	9.26	6.95	8.52
13	9.38	7.86	8.56
14	9.62	7.18	8.57
15	9.31	7.08	8.25
16	9.34	7.74	8.44

Таблица сравнений без оптимизаций:

Threads	Efficiency
2	45.95
3	42.04
4	42.52
5	43.91
6	41.3
7	50.29
8	52.27
9	51.75
10	50.3
11	51.41
12	49.52
13	48.18
14	49.3
15	50.06
16	49.91

Таблица сравнений с оптимизациями:

Threads	Efficiency
2	72.2
3	68.22
4	84.51
5	98.3
6	74.33
7	36.47
8	38.02
9	37.72
10	37.28
11	37.28
12	35.37
13	35.2
14	35.14
15	36.5
16	35.7