Лабораторная работа №4: Технология OpenMP. Особенности настройки

ИИКС ИБ
Б20-505
Барабанов Андрей

2022

Содержание

1 Рабочая среда

- Модель процессора: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
- Объём оперативной памяти: 16,0 ГБ
- Тип оперативной памяти: DDR4
- Версия операционной системы: ManjaroLinux 22.0.0
- Разрядность операционной системы: x86_64
- Среда разработки: GCC 12.2.0
- Версия OpenMP: 4.5

```
[bar1k@manjaro ~]$ lscpu
Архитектура:             x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Порядок байт:          Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
ID прроизводителя:       GenuineIntel
  Имя модели:            Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    Семейство ЦПУ:       6
    Модель:              165
    Thread(s) per core:  2
    Ядер на сокет:       6
    Сокетов:             1
    Степпинг:            2
    CPU(s) scaling MHz:  18%
    CPU max MHz:         5000,0000
    CPU min MHz:         800,0000
    BogoMIPS:            5202,65
```

```
[bar1k@manjaro ~]$ free -h
               total        used        free      shared  buff/cache   available
Mem:            15Gi       2,2Gi        11Gi       132Mi       2,2Gi        12Gi
Swap:             0B          0B          0B
```

```
[bar1k@manjaro ~]$ cat /etc/lsb-release
DISTRIB_ID=ManjaroLinux
DISTRIB_RELEASE=22.0.0
DISTRIB_CODENAME=Sikaris
DISTRIB_DESCRIPTION="Manjaro Linux"
```

```
[bar1k@manjaro ~]$ gcc --version
gcc (GCC) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
Это свободно распространяемое программное обеспечение. Условия копирования
приведены в исходных текстах.

  Без гарантии каких-либо качеств, включая
коммерческую ценность и применимость для каких-либо целей.
```

```
[bar1k@manjaro ~]$ echo | cpp -fopenmp -dM | grep -i open
#define _OPENMP 201511
```

2 Опции, функции и переменные окружения

2.1 _OPENMP

Компилятор с поддержкой OpenMP определяет макрос **_OPENMP**, который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы. Этот макрос определён в формате **yyyymm**, где **yyyy** и **mm** – цифры года и месяца, когда был принят поддерживаемый стандарт OpenMP. Например, компилятор, поддерживающий стандарт OpenMP 4.5, определяет **_OPENMP** в **201511**.

2.2 omp_get_num_procs()

Функция **omp_get_num_procs()** возвращает количество процессоров, доступных для использования программе пользователя на момент вызова. Нужно учитывать, что количество доступных процессоров может динамически изменяться.

2.3 omp_get_max_threads()

Функция **omp_get_max_threads()** возвращает максимально допустимое число нитей для использования в следующей параллельной области.

2.4 OMP_DYNAMIC

В некоторых случаях система может динамически изменять количество нитей, используемых для выполнения параллельной области, например, для оптимизации использования ресурсов системы. Это разрешено делать, если переменная среды **OMP_DYNAMIC** установлена в **true**.
Переменную **OMP_DYNAMIC** можно установить с помощью функции **omp_set_dynamic()**.
Узнать значение переменной **OMP_DYNAMIC** можно при помощи функции **omp_get_dynamic()**.

2.5 omp_get_wtick()

Функция **omp_get_wtick()** возвращает в вызвавшей нити разрешение таймера в секундах. Это время можно рассматривать как меру точности таймера.

2.6 OMP_NESTED

Параллельные области могут быть вложенными, по умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды **OMP_NESTED**.
Изменить значение переменной **OMP_NESTED** можно с помощью вызова функции **omp_set_nested()**.
Узнать значение переменной **OMP_NESTED** можно при помощи функции **omp_get_nested()**.

2.7 OMP_MAX_ACTIVE_LEVELS

Переменная **OMP_MAX_ACTIVE_LEVELS** задаёт максимально допустимое количество вложенных параллельных областей.
Значение может быть установлено при помощи вызова функции **omp_set_max_active_levels()**.
Значение переменной **OMP_MAX_ACTIVE_LEVELS** может быть получено при помощи вызова функции **omp_get_max_active_levels()**.

2.8 OMP_SCHEDULE

schedule(type[, chunk]) – опция задаёт, каким образом итерации цикла распределяются между нитями. В опции **schedule** параметр type задаёт следующий тип распределения итераций:

- **static** – блочно-циклическое распределение итераций цикла; размер блока – **chunk**. Первый блок из **chunk** итераций выполняет нулевая нить, второй блок — следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение **chunk** не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.

- **dynamic** – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает **chunk** итераций (по умолчанию **chunk=1**), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из **chunk** итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.

- **guided** – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины **chunk** (по умолчанию **chunk=1**) пропорционально количеству ещё не

распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения **chunk**.

- **auto** – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр **chunk** при этом не задаётся.

- **runtime** – способ распределения итераций выбирается во время работы программы по значению переменной среды **OMP_SCHEDULE**. Параметр **chunk** при этом не задаётся.

Изменить значение переменной **OMP_SCHEDULE** из программы можно с помощью вызова функции **omp_set_schedule()**.
При помощи вызова функции **omp_get_schedule()** пользователь может узнать текущее значение переменной **OMP_SCHEDULE**.

2.9 Locks

Один из вариантов синхронизации в OpenMP реализуется через механизм **замков (locks)**. В качестве замков используются общие целочисленные переменные (размер должен быть достаточным для хранения адреса). Данные переменные должны использоваться только как параметры примитивов синхронизации
Замок может находиться в одном из трёх состояний: **неинициализированный**, **разблокированный** или **заблокированный**. Разблокированный замок может быть захвачен некоторой нитью. При этом он переходит в заблокированное состояние. Нить, захватившая замок, и только она может его освободить, после чего замок возвращается в разблокированное состояние.
Для инициализации простого замка используется функция **omp_init_lock()**.
Функция **omp_destroy_lock()** используется для переведения простого замка в неинициализированное состояние.
Для захватывания замка используется функция **omp_set_lock()**.
Для освобождения замка используется функция **omp_unset_lock()**.

3 Обоснование использования блокировок

Пример вычислительного алгоритма, использующего **механизм неявных блокировок**: необходимо построить гистограмму длины HISTOGRAM_SIZE, причём данные берутся из массива длины ARRAY_SIZE, значения которого лежат в интервале от 0 до (HISTOGRAM_SIZE - 1).
При использовании в параллельной области опции **for** необходимо учесть, что некоторые потоки, могут одновременно прибавлять определённые значения у гистограммы.
Использование критической области не даёт выигрыш во времени, хотя устраняет ошибки в подсчёте отдельных значений у гистограммы, потому что каждый процесс будет ждать своей очереди, чтобы вступить в эту область.
Механихм неявных блокировок способен решить эту проблему, путём создания массива заммков под каждый элемент гистограммы. Таким образом, процессы будут выстраиваться в очередь только при обращении к одному и тому же элемнту, в отличие от критичсекой области.

4 Графики

4.1 schedule(static)

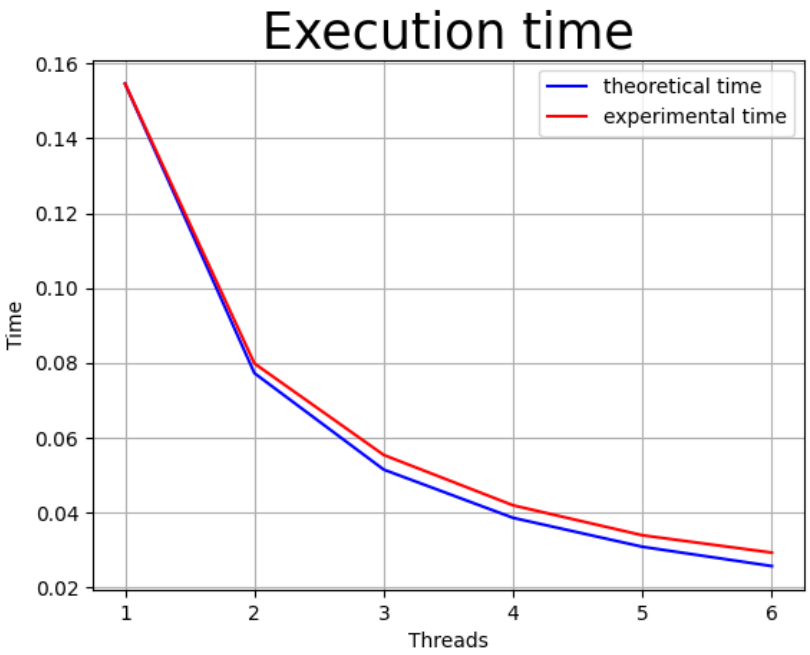

Рис. 1: Время

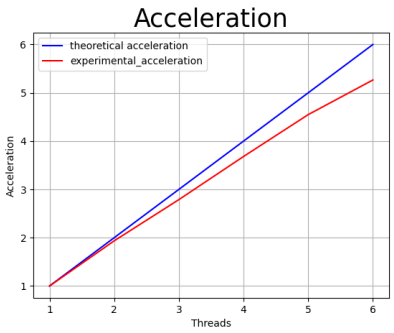

Рис. 2: Ускорение

Рис. 3: Эффективность

4.2 schedule(static, 1)

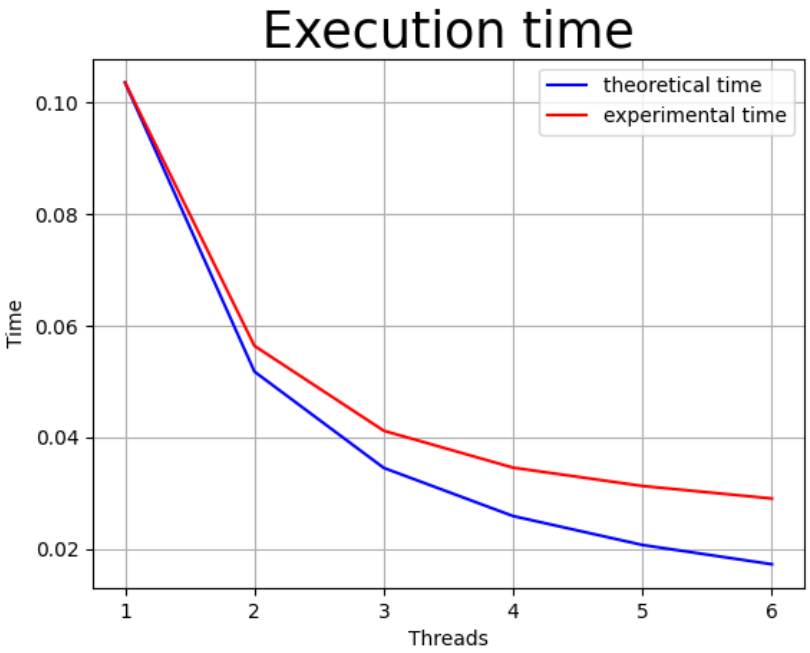

Рис. 4: Время

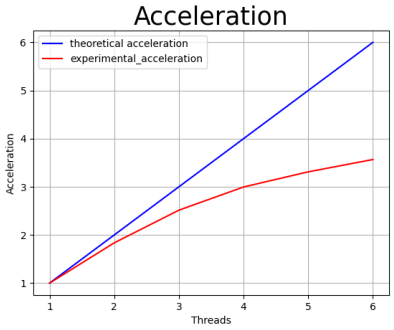

Рис. 5: Ускорение

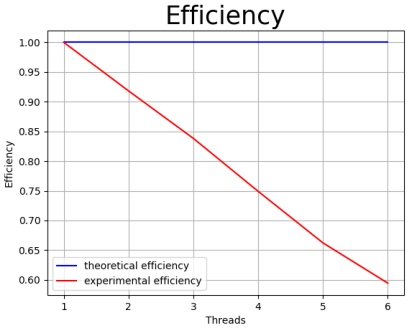

Рис. 6: Эффективность

4.3 schedule(static, 2)

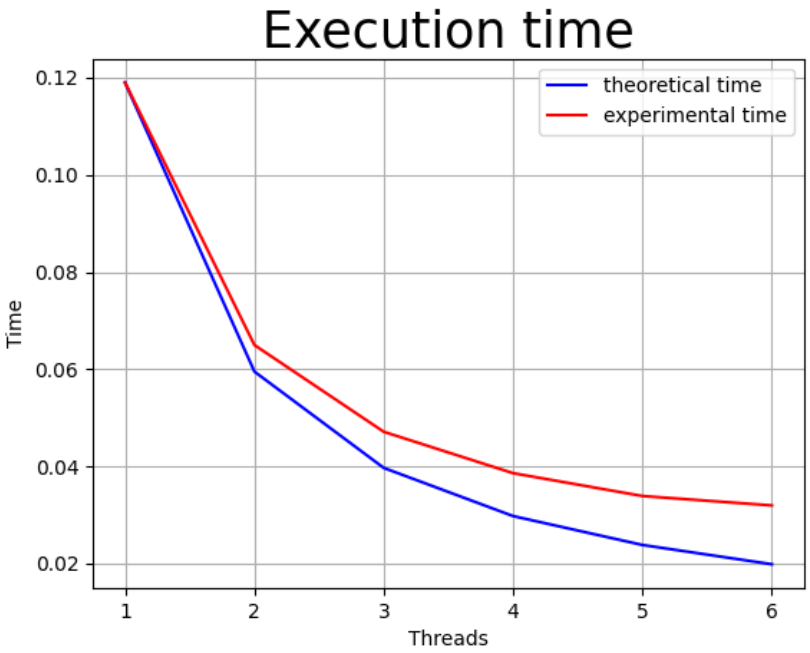

Рис. 7: Время

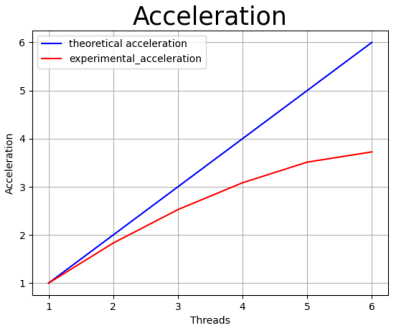

Рис. 8: Ускорение

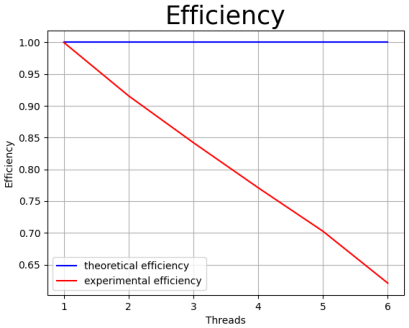

Рис. 9: Эффективность

4.4 schedule(dynamic)

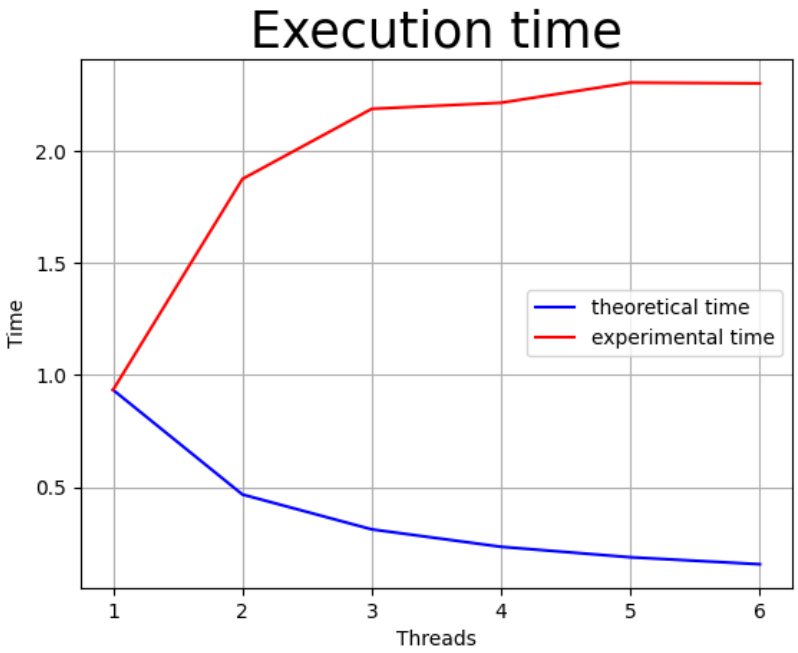

Рис. 10: Время

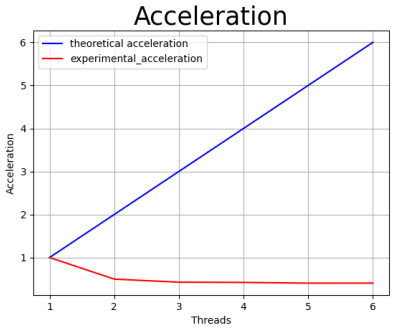

Рис. 11: Ускорение

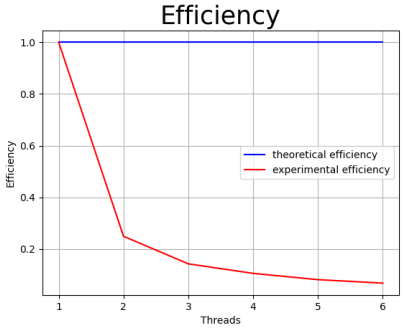

Рис. 12: Эффективность

4.5 schedule(dynamic, 2)

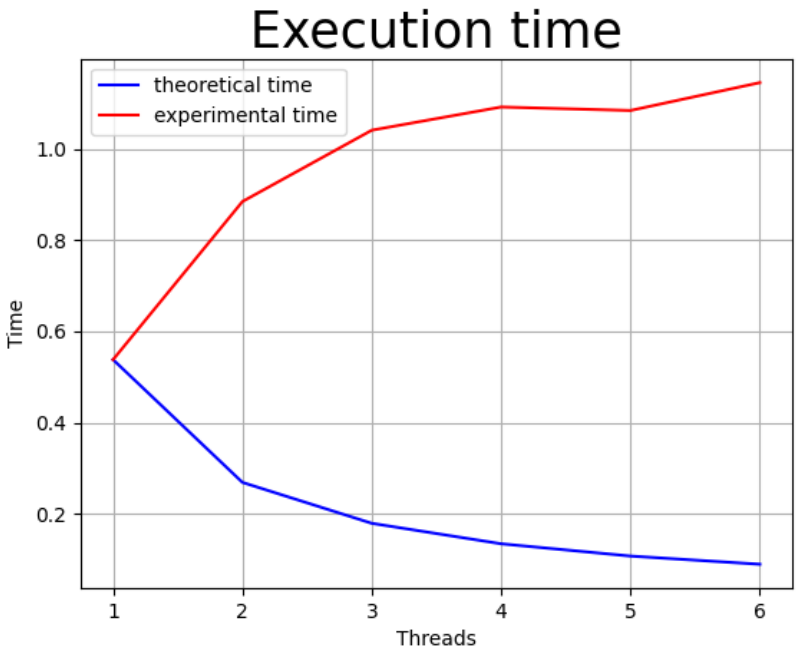

Рис. 13: Время

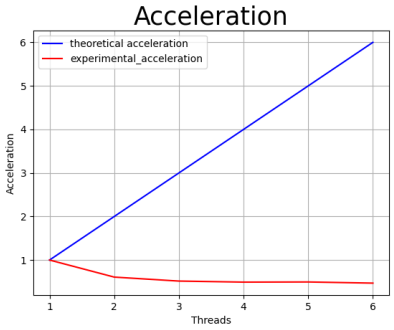

Рис. 14: Ускорение

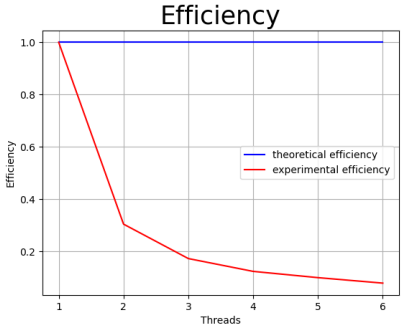

Рис. 15: Эффективность

4.6 schedule(guided)

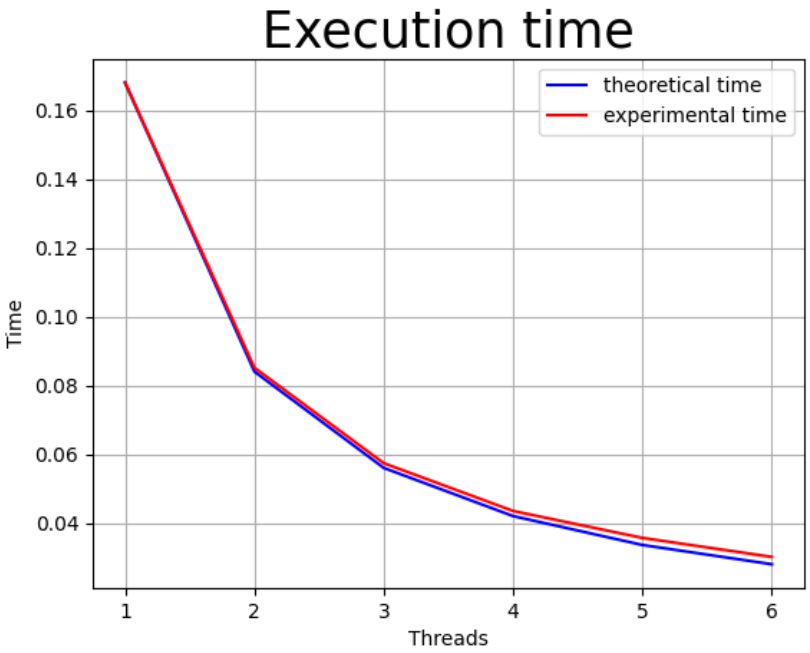

Рис. 16: Время

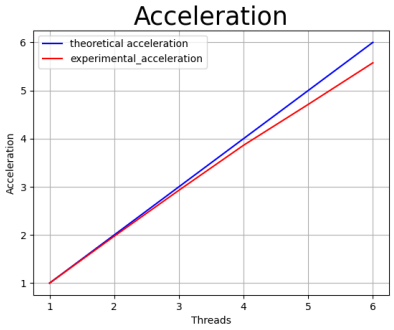

Рис. 17: Ускорение

Рис. 18: Эффективность

4.7 schedule(guided, 2)

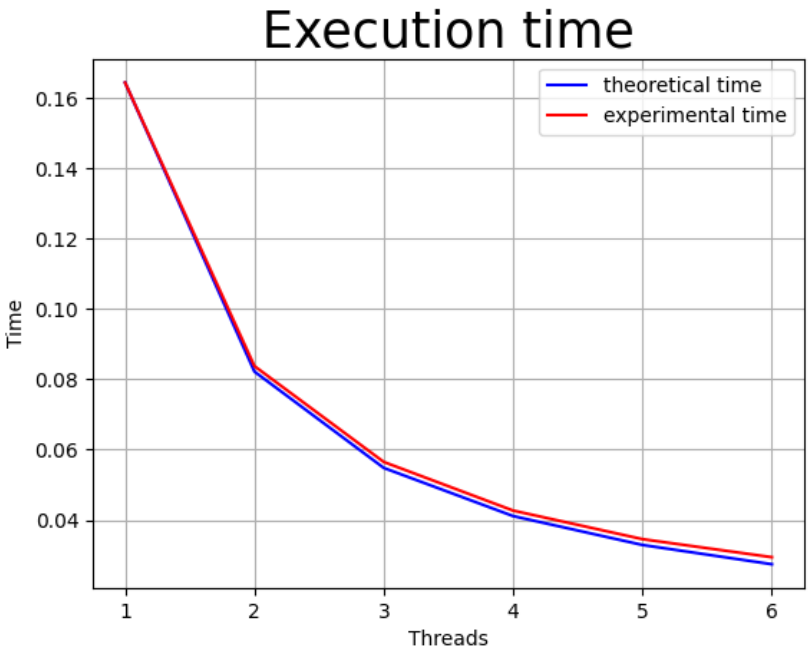

Рис. 19: Время

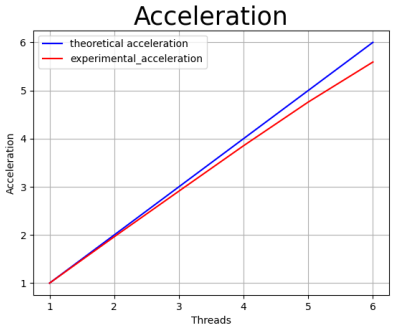

Рис. 20: Ускорение

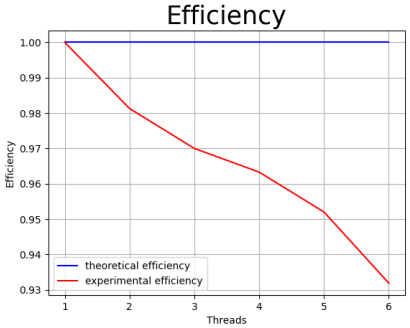

Рис. 21: Эффективность

4.8 schedule(auto)

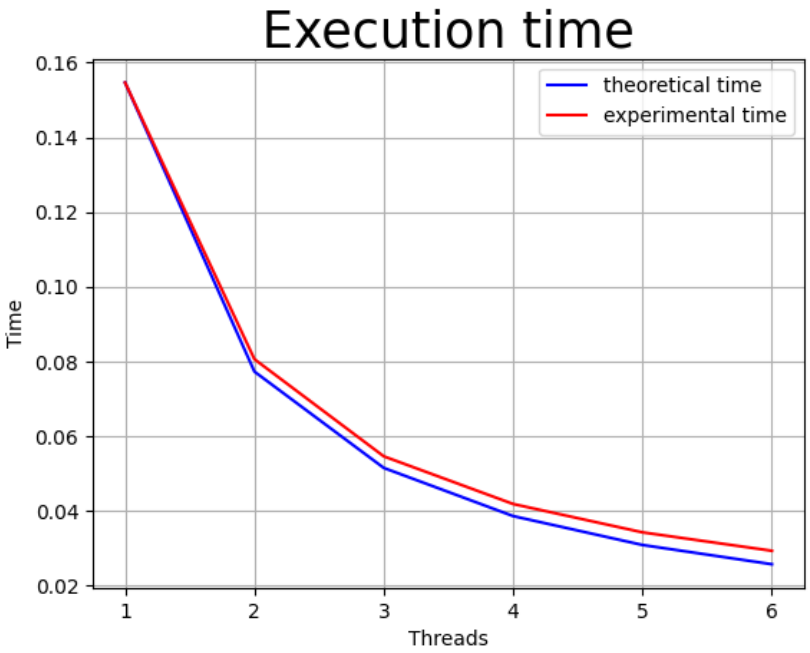

Рис. 22: Время

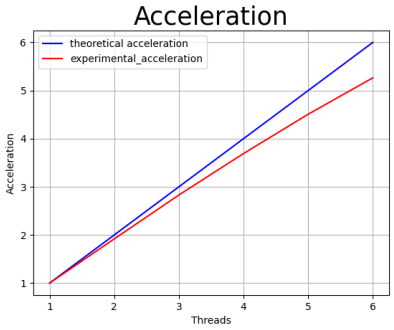

Рис. 23: Ускорение

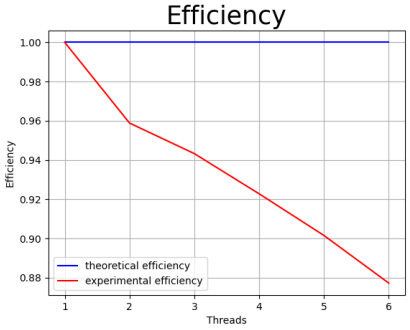

Рис. 24: Эффективность

5 Программные коды

5.1 main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

#define MAX_THREADS_NUM 6
#define ARRAY_SIZE 100000000
#define HISTOGRAM_SIZE 10
#define TESTS_NUM 10


void func(int* array, int n) {
    int max = -1;
#pragma omp parallel num_threads(n) shared(array) reduction(max : max)
    {
#pragma omp for schedule(static)
//#pragma omp for schedule(static, 1)
//#pragma omp for schedule(static, 2)
//#pragma omp for schedule(dynamic)
//#pragma omp for schedule(dynamic, 2)
//#pragma omp for schedule(guided)
//#pragma omp for schedule(guided, 2)
//#pragma omp for schedule(auto)
        for (int i = 0; i < ARRAY_SIZE; i++)
            if (array[i] > max)
                max = array[i];
    }
}

int main() {

    printf("\n_OPENMP = %d\nStandard version: 4.5\nRelease date: "
           "November 2015\n\n", _OPENMP);
    printf("Number of processors available to the device: %d\n",
           omp_get_num_procs());
    printf("Number of threads that could be used to form a new "
           "team: %d\n\n", omp_get_max_threads());


    printf("OMP_DYNAMIC = %d\n\n", omp_get_dynamic());
```

```
    printf("Precision of the timer in seconds: %.16g\n\n",
           omp_get_wtick());


    printf("OMP_NESTED = %d\n", omp_get_nested());
    printf("OMP_MAX_ACTIVE_LEVELS = %d\n\n",
           omp_get_max_active_levels());



    omp_sched_t kind;
    int chunk_size;
    const char *enum_omp_sched_t[] = {"", "static", "dynamic",
                                      "guided", "auto"};
    omp_get_schedule(&kind, &chunk_size);
    printf("OMP_SCHEDULE = (%s, %d)\n\n", enum_omp_sched_t[kind],
           chunk_size);


    int key;
    int *array = (int *) malloc(sizeof(int) * ARRAY_SIZE);
    int *histogram = (int *) malloc(sizeof(int) *
            HISTOGRAM_SIZE);
    omp_lock_t lock[HISTOGRAM_SIZE];
    for (int i = 0; i < ARRAY_SIZE; i++)
        array[i] = rand() % HISTOGRAM_SIZE;
    for (int i = 0; i < HISTOGRAM_SIZE; i++) {
        histogram[i] = 0;
        omp_init_lock(&(lock[i]));
    }
#pragma omp parallel private(key)
    {
#pragma omp for
        for (int i = 0; i < ARRAY_SIZE; i++) {
            key = array[i];
            omp_set_lock(&(lock[key]));
            histogram[key]++;
            omp_unset_lock(&(lock[key]));
        }
    }
    free(array);
    free(histogram);

    double *timing = (double *) malloc(sizeof(double) *
            MAX_THREADS_NUM * TESTS_NUM);
    array = (int *) malloc(sizeof(int) * ARRAY_SIZE);
```

```
    for (int i = 0; i < TESTS_NUM; i++) {
        srand(time(NULL));
        for (int j = 0; j < ARRAY_SIZE; j++)
            array[j] = rand() % ARRAY_SIZE;

        for (int n = 0; n < MAX_THREADS_NUM; n++) {
            double start_time = omp_get_wtime();
            func(array, n + 1);
            double end_time = omp_get_wtime();
            timing[MAX_THREADS_NUM * i + n] = end_time - start_time;
        }
    }

    FILE *fd = fopen("../timing.txt", "w");
    fprintf(fd, "%d\n", MAX_THREADS_NUM);
    fprintf(fd, "%d\n", TESTS_NUM);
    for (int i = 0; i < MAX_THREADS_NUM; i++) {
        for (int j = 0; j < TESTS_NUM; j++)
            fprintf(fd, "%lf\t", timing[MAX_THREADS_NUM * j + i]);
        fprintf(fd, "\n");
    }
    free(timing);
    free(array);

    return 0;
}
```

5.2 main.py

```
import matplotlib.pyplot as plt
from prettytable.colortable import ColorTable


def read_file():
    with open('/home/bar1k/CLionProjects/ParallelProgramming/timing.txt') \
            as file:
        threads = [x + 1 for x in range(int(file.readline()))]
        tests = int(file.readline())
        times = [round((sum([float(x) for x in file.readline().split()]) /
                        tests), 6) for _ in range(len(threads))]
    return threads, times


def draw_plots(threads, experimental_time):
    theoretical_time = [experimental_time[0] / (x + 1) for x in
                        range(len(threads))]
    plt.title("Execution time", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Time')
    plt.grid(True)
    plt.plot(threads, theoretical_time, 'b')
    plt.plot(threads, experimental_time, 'r')
    plt.legend(['theoretical time', 'experimental time'])
    plt.show()

    theoretical_acceleration = [x for x in threads]
    experimental_acceleration = [experimental_time[0] / x for x in
                                 experimental_time]
    plt.title("Acceleration", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Acceleration')
    plt.grid(True)
    plt.plot(threads, theoretical_acceleration, 'b')
    plt.plot(threads, experimental_acceleration, 'r')
    plt.legend(['theoretical acceleration', 'experimental_acceleration'])
    plt.show()

    theoretical_efficiency = [1] * len(threads)
    experimental_efficiency = [experimental_acceleration[x] / threads[x]
                               for x in range(len(threads))]
    plt.title("Efficiency", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Efficiency')
```

```
    plt.grid(True)
    plt.plot(threads, theoretical_efficiency, 'b')
    plt.plot(threads, experimental_efficiency, 'r')
    plt.legend(['theoretical efficiency', 'experimental efficiency'])
    plt.show()


def output_table(threads, times):
    table = ColorTable()
    table.field_names = ['Threads'] + [str(threads[i]) for i in
                                       range(len(threads))]
    table.add_row(['Times'] + [str(times[i]) for i in
                               range(len(times))])
    print(table)


if __name__ == '__main__':
    data = read_file()
    draw_plots(data[0], data[1])
    output_table(data[0], data[1])
```

6 Заключение

В ходе проделанной работы были изучены основные настройки среды OpenMP и связанные с ними возможности. На основании параллельного алгоритма поиска максималльного элемента из первой лабораторной работы было выяснено, что наиболее оптимальное время при изменении значений опции **schedule** достигается при:

1. **(guided)**
2. **(guided, 2)**
3. **(auto)**
4. **(static)**

Наихудший результат показали значения **(dynamic)** и **(dynamic, 2)**. Возможно, это связанно с тем, что затрачивается слишком много времени на выделение памяти для итераций.