Лабораторная работа №4

"Технология OpenMP. Особенности настройки"

Выполнил студент группы Б20-505
Сорочан Илья

Московский Инженерно-Физический Интститут
Москва 2023

1 Рабочая среда

Технические характеристики (вывод *inxi*):

```
CPU: 6-core AMD Ryzen 5 4500U with Radeon Graphics (-MCP-)
speed/min/max: 1396/1400/2375 MHz Kernel: 5.15.85-1-MANJARO x86_64 Up: 46m
Mem: 2689.5/7303.9 MiB (36.8%) Storage: 238.47 GiB (12.6% used) Procs: 238
Shell: Zsh inxi: 3.3.24
```

Используемый компилятор:

```
gcc (GCC) 12.2.0
```

Согласно официальной документации даная версия компилятора поддерживает *OpenMP 5.0*

2 Работа с *OpenMP*

2.1 Версия и дата принятия

Макрос *_OPENMP* является целочисленным числом и показывает дату принятия *OpenMP* в формате *yyyymm*, где *yyyy* - год принятия, а *mm* - месяц.

Даты можно посмотреть на официальном сайте *OpenMP*. но в своем коде я написал удобный макрос.

2.2 *OMP_DYNAMIC*

Переменная окружения *OMP_DYNAMIC* отвечает за динамический выбор числа потоков. Например если она имеет значение *true*, то *OpenMP* автоматически выбирает число потоков для *parallel* участков. Если же *false*,

2.3 wtick

Функция *omp_get_wtick()* возвращает количество секунд, прошедшее между тиками таймера из *omp_get_wtime()*

2.4 Вложенность

Функция *omp_get_nested()* возвращает флаг, указывающий на то включен ли вложеннный параллелилизм. Если да, то количество вложенных конструкций ограниченно числом, которое можно получить, вызвав *omp_get_max_active_levels()*.

2.5 schedule

Переменная окружения *OMP_SCHEDULE* задаёт тип распределения нагрузки и размер чанков для всех директив циклов. Тип определяет как циклы делятся на подмножества итераций размером в один чанк:

- *static* – все подмножества распределяются между потоками один раз, в самом начале;
- *dynamic* – каждый из процессов получает чанк, по его выполнении он запрашивает новый. Так продолжается пока чанки не закончатся;
- *guided* – аналогично *dynamic*, однако он не содержит чанка, размер которого меньше заданного размера чанка;

- *auto* – компилятор выбирает на свое усмотрение;
- *runtime* – выбор производится непосредственно перед выполнением цикла.

2.6 Пример использования *omp_ lock*

Замки необходимы для обеспечения выболнения промежутка кода только одним потоком. Например чтение из файла.

```
omp_lock_t writelock;

omp_init_lock(&writelock);

#pragma omp parallel for
for ( i = 0; i < x; i++ )
{
  // do something important
  omp_set_lock(&writelock);
  // do something important but only one thread access at a time
  omp_unset_lock(&writelock);
  // do another important task
}

omp_destroy_lock(&writelock);
```

2.7 Разработанный код

Для иллюстрации директив *OpenMP*, затронутых в данном разделе была разработна следующая программа:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.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\nRelease date: %d\n", _OPENMP_VERSION, _OPENMP);

    printf("\nAvaliable processors: %d\nAvaliable threads: %d\n", omp_get_num_procs(), omp_get_max_threads());

    if (omp_get_dynamic())
        puts("\nDynamic is on");
    else
        puts("\nDynamic is off");

    printf("\nOpenMP wtick: %fs\n", omp_get_wtick());

    if (omp_get_nested())
        printf("\nNested parallelism up to %d\n", omp_get_max_active_levels());
    else
        puts("Nested parallelism is off");


    omp_sched_t sched;
    int chunk_size;
    omp_get_schedule(&sched, &chunk_size);
    char *s;
    switch (sched) {
        case omp_sched_static: s = "static"; break;
        case omp_sched_dynamic: s = "dynamic"; break;
        case omp_sched_guided: s = "guided"; break;
        case omp_sched_auto: s = "auto"; break;
```

```
    }
    printf("\nOpenMP schedule: %s\nChunk size: %d\n", s, chunk_size);

    return 0;
}
```

3 Применение schedule

3.1 Исходный код

В качестве примера я взял свой код из третьей лабораторной работы, однако переработал его, что бы при компиляции можно было указывать не только число потоков, но и расписание вместе с размером чанка:

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

int main(int argc, char** argv) {
    const int count = 1000000;

    srand(920214);
    int *array = (int*)malloc(count*sizeof(int));
    for(int i=0; i<count; i++) { array[i] = rand(); }

    clock_t start = clock();

    #if THREADS == 1
    for (int d = count / 2; d > 0; d /= 2) {
        for (int i = d; i < count; ++i) {
            for (int j = i - d; j >= 0 && array[j] > array[j + d]; j -= d) {
                int temp = array[j];
                array[j] = array[j + d];
                array[j + d] = temp;
            }
        }
    }
    #else
    #pragma omp parallel num_threads(THREADS) shared(array, count) default(none)
    for (int d = count / 2; d > 0; d /= 2) {
        const int cd = d;
        #ifdef CHUNK_SIZE
        #pragma omp for schedule(SCHEDULE, CHUNK_SIZE)
        #else
        #pragma omp for schedule(SCHEDULE)
        #endif
        for (int i = cd; i < count; ++i) {
            for (int j = i - cd; j >= 0 && array[j] > array[j + cd]; j -= cd) {
                int temp = array[j];
                array[j] = array[j + cd];
                array[j + cd] = temp;
            }
        }
    }
    #endif

    clock_t end = clock();

    const double CLOCKS_PER_MS = (double)CLOCKS_PER_SEC / 1000;
    double total = (double)(end - start) / CLOCKS_PER_MS;
    printf("%.3f", total);

    free(array);

    return 0;
}
```

Скрипт так же был изменен:

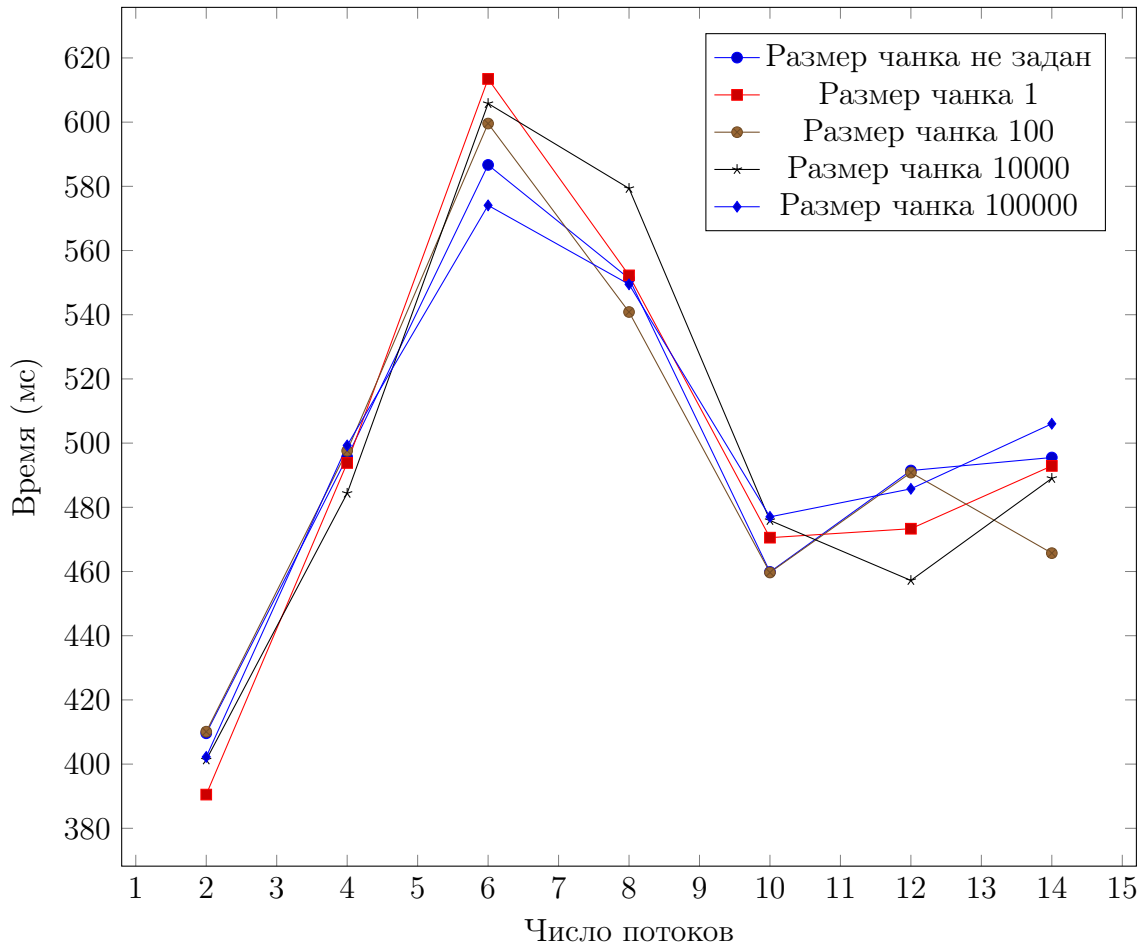
```
# This script compiles main.c with different number of threads
# and collects data to data.csv file
# format: worst,best,average
import os
import subprocess
import csv
import sys

# important constants
RUNS_PER_THREADS = 5
TESTING_THREADS = range(2, 16, 2)

# compile with threads
def compile(threads, schedule, chunk_size):
    cmd = "gcc main.c"
    if threads > 1:
        cmd += " -fopenmp -DTHREADS=" + str(threads) + " -DSCHEDULE=" + schedule
        if chunk_size:
            cmd += " -Dchunk_size=" + str(chunk_size)
    cmd += " -o main"
    os.system(cmd)

# capture worst, best and average
```

```
def run():
    data = []
    for _ in range(RUNS_PER_THREADS):
        proc = subprocess.run(["./main"], capture_output=True, text=True)
        data.append(float(proc.stdout))

    return sum(data) / len(data)

def main():
    if len(sys.argv) < 3:
        print("Specify output file, schedule and chunk size")

    out = sys.argv[1]
    schedule = sys.argv[2]
    try:
        chunk_size = sys.argv[3]
    except:
        chunk_size = None

    with open(out, "w") as file:
        writer = csv.writer(file)
        writer.writerow(["Threads", "Average time (ms)"])
        for threads in TESTING_THREADS:
            print("testing threads =", threads, "chunk size =", chunk_size)
            compile(threads, schedule, chunk_size)
            result = ["{:.3f}".format(run())]
            writer.writerow([str(threads)] + result)

if __name__ == "__main__":
    main()
```

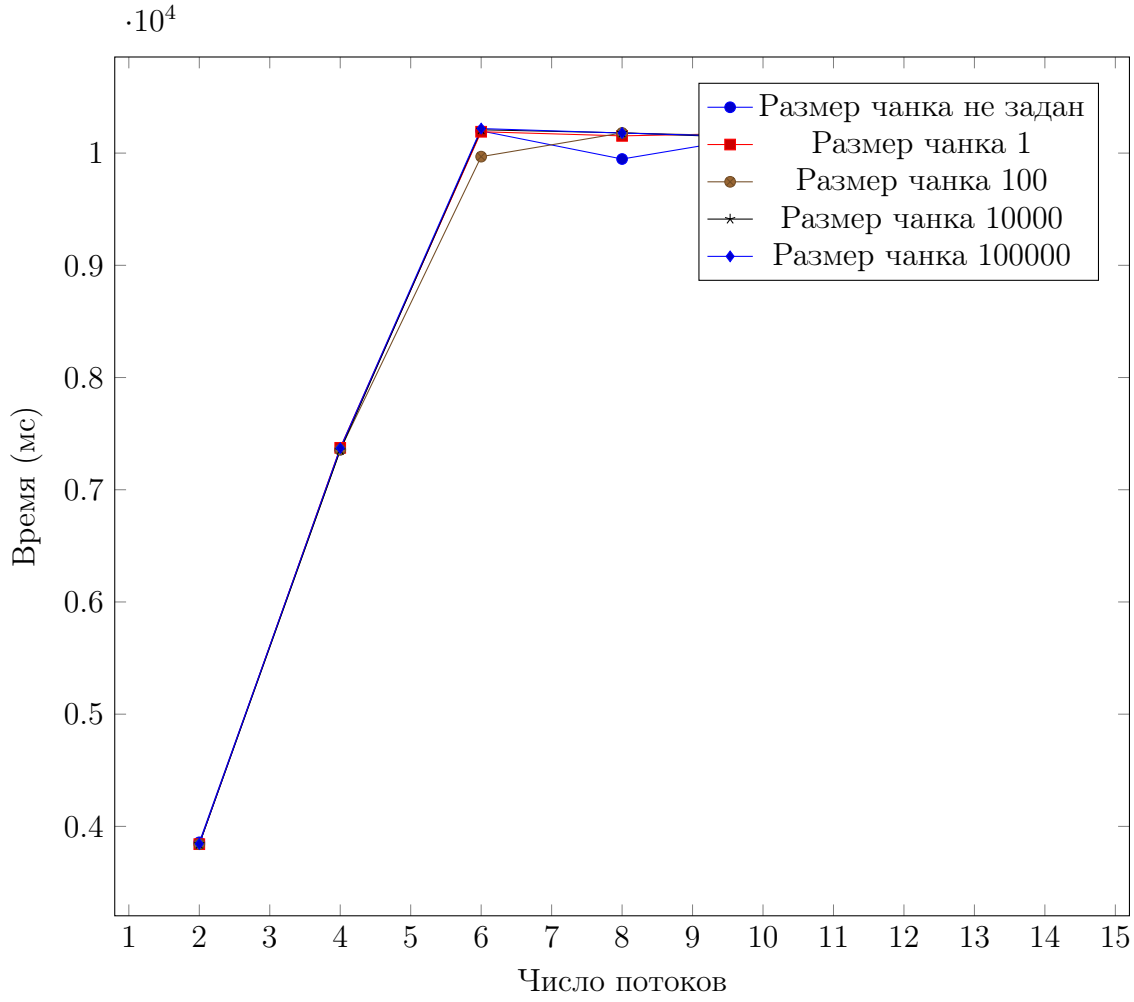
Так же в этот раз я делал не по 10, а по 5 запусков на поток. Это связано с большим измеряемым объемом данных.

3.2 Графики

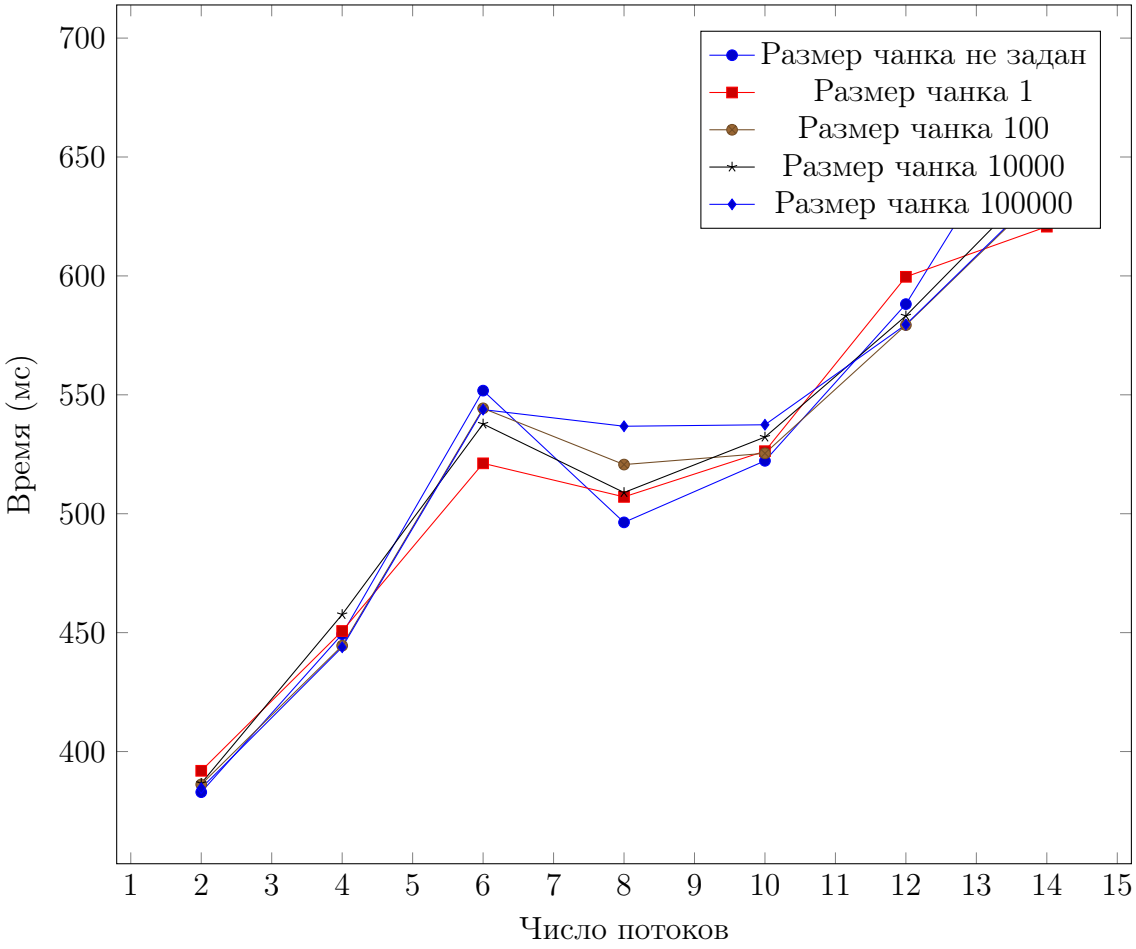
Для *schedule* = *static*:



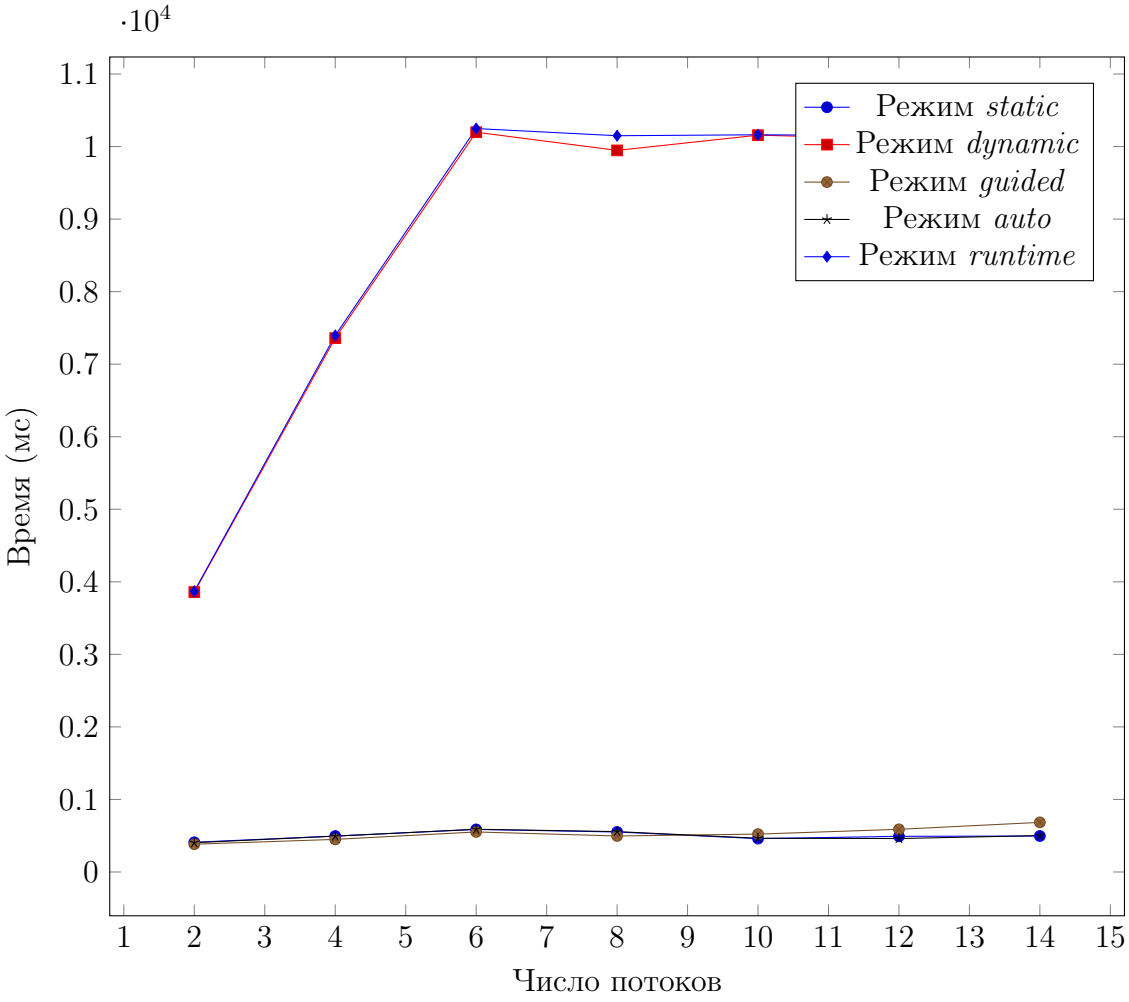
Для *schedule* = *dynamic*:



Для $schedule = guided$:



Из графиков видно, что значения по умолчанию являются оптимальными. Для остальных режимов (*auto* и *runtime*) чанки не указывались. Рассмотрим сравнение этих режимов:



4 Заключение

В данной работы были исследованы некоторые директивы *OpenMP*. Была усовершенствована программа из третьей лабораторной работы. Модифицирован скрипт и произведены замеры с различными распределениями нагрузки в цикле.

В ходе работы было выяснено, что тип распределения *static* куда быстрее *dynamic*.

Хочу заметить, что при использовании *static* многопоточная программа без оптимизаций выполняется гораздо быстрее однопоточной программы с ними (результаты в предыдущей лабораторной). Это говорит о том, что результаты многопоточных программ в третьей лабораторной работе могут быть улучшены.