Лабораторная работа №4	M3138	2023
OpenMP	Селезнев Дмитрий Александрович	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: C++ gcc17. Стандарт OpenMP 2.0.

Описание OpenMP

OpenMP – открытый стандарт распараллеливания программ на языках C, C++ и Fortran.

В своем решении я использовал директиву parallel, которая создает область, которую может выполнять несколько потоков.

Также я использовал директиву for, которая относится к группе директив распределяющих работу между потоками, в данном случае разбивает цикл for на итерации, которые выполняются в разных потоках. Причем я также указывал флажок nowait, который означает, что потоки не синхронизируются в конце цикла.

Директива critical из группы конструкций для синхронизации потоков. Конкретно эта директива указывает, что нужно выполнить соответствующий код последовательно, то есть только одним потоком за раз.

Ещё использовалась функция omp_set_num_threads для того чтобы указать на сколько потоков нужно распараллелить код.

Также для экспериментальной части потребовался schedule. Он есть нескольких типов:

static – итерации делятся на блоки размера chunk_size и статически назначаются потокам циклическим образом в порядке номера потока.

dynamic – итерации делятся на блоки размера chunk_size, каждый из которых назначается потоку, который свободен.

guided – итерации делятся на блоки уменьшающегося (экспоненциально) размера, размера большего chunk_size, кроме, возможно, последнего.

Описание кода

Конструкции из предыдущего пункта использовались только в функции parallel, реализацию которой можно увидеть ниже.

```
void parallel(int pos) {
    double tend_start = omp_get_wtime();
    int num_of_threads;
#pragma omp parallel
        {
            size_t colors_i[num_of_colors + 1];
            for (int i = 0; i <= num_of_colors; i++) colors_i[i] = 0;
#pragma omp for nowait
            for (int i = pos; i < length; i++) colors_i[arr[i]]++;
#pragma omp critical
            for (int i = 0; i <= num_of_colors; i++) if (flag_for_test) colors[i] +=
colors_i[i];
        }
    for (int i = 0; i <= num_of_colors; i++) pref_sum_p[i + 1] = pref_sum_p[i] +
colors[i];
    for (int i = 0; i <= num_of_colors; i++) pref_sum_average[i + 1] =
pref_sum_average[i] + (double)colors[i] * (double)i;
    double mx = -INF;
#pragma omp parallel
    {
        num_of_threads = omp_get_num_threads();
        double max_i = -INF;
        int t0, t1, t2;
        for (int f0 = 0; f0 <= num_of_colors - 3; f0++) {
            for (int f1 = f0 + 1; f1 <= num_of_colors - 2; f1++) {
#pragma omp for nowait
                for (int f2 = f1 + 1; f2 <= num_of_colors - 1; f2++) {
                    double num = check(f0, f1, f2);
                    if (max_i <= num) {
                        max_i = num;
                        t0 = f0;
                        t1 = f1;
                        t2 = f2;
                    }
                }
            }
        }
#pragma omp critical
        {
            if (mx < max_i) {
                mx = max_i;
                thresholds0 = t0;
                thresholds1 = t1;
                thresholds2 = t2;
            }
        }
    }
#pragma omp parallel
    {
        int color;
#pragma omp for
        for (int i = pos; i < length; i++) {
            if (arr[i] <= thresholds0) color = 0;
            else if (arr[i] <= thresholds1) color = 84;
```

```
            else if (arr[i] <= thresholds2) color = 170;
            else color = 255;
            if (flag_for_test) arr[i] = color;
        }
    }
    double time_ = omp_get_wtime() - tend_start;
    cout << thresholds0 << " " << thresholds1 << " " << thresholds2 << "\n";
    sum_time += time_;
    printf("Time (%i thread(s)): %f ms\n", num_of_threads, time_ * 1000);
}
```

Листинг 1 – Функция parallel

Как видно все разбиения на потоки заключаются в распараллеливании циклов for. Причем видно что в первых двух for пришлось использовать critical, так как возможна ситуация когда несколько потоков одновременно будут обращаться к одной и той же ячейке памяти. Первый блок распараллеливания отвечает за подсчет гистограммы, второй за вычисление порогов, третий за перезапись картинки. В коде присутствует тройной вложенный for, из которых я распараллелил внутренний, так как внутри есть if, из-за которого некоторые итерации этого for дольше других, следовательно, если распараллелить внешний for, то менее равномерно распределяются действия между потоками. Конечно, на распределение итераций потокам тоже уходит время, но экспериментально было выяснено, что распараллелить внутренний for быстрее.

Стоит отметить, что critical работает долго, но так как в коде он исполняет очень небольшое количество операций, то это незначительно влияет на время, зато работает стабильно.

По заданию нужно для каждой тройки чисел f0, f1, f2 посчитать межкластерную дисперсию и найти такую тройку, что дисперсия максимальна. Вычисление дисперсии происходит в функции check и нескольких вспомогательных функциях.

```
double get_dispersion(int a, int b) {
    double u = (pref_sum_average[b + 1] - pref_sum_average[a]);
    return u * u / (double)(pref_sum_p[b + 1] - pref_sum_p[a]);
}

double check(int a, int b, int c) {
    return (get_dispersion(0, a) + get_dispersion(a + 1, b))
    + (get_dispersion(b + 1, c) + get_dispersion(c + 1, (int)num_of_colors));
}
```

Листинг 2 – Вычисление дисперсии

Как видно здесь просто происходит подсчет формул указанных в файле[3]. Разве что я соптимизировал подсчет суммы, подсчитав префиксные суммы, теперь подсчет каждой дисперсии работает за O(1). Также я соптимизировал вычисления тем, что не использую множитель N(то есть

количество пикселей), это сделано для ускорения вычислений, так как для поиска порогов не требуется знать точный максимум, главное чтобы значения всё еще правильно относились к друг другу, а множитель N, есть у всех.

Результат работы

Мой процессор: Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz

77 130 187

Time (8 thread(s)): 17.000032 ms

Рисунок 1 – Модифицированная тестовая картинка

Экспериментальная часть

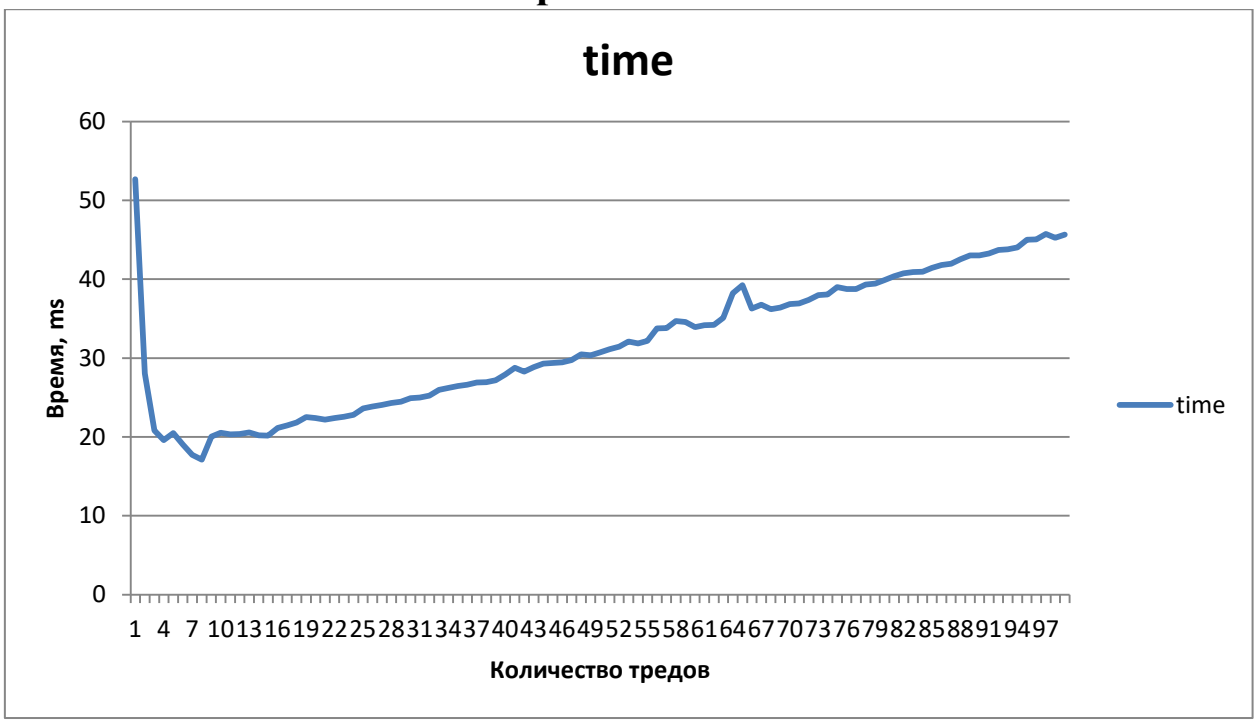

График 1 – Зависимость времени работы от количества тредов

На моем процессоре согласно[4] – 8 потоков, и, как можно заметить, именно на этом количестве потоков достигается пиковое значение (программа отрабатывает за 17ms), потом же время начинает расти.

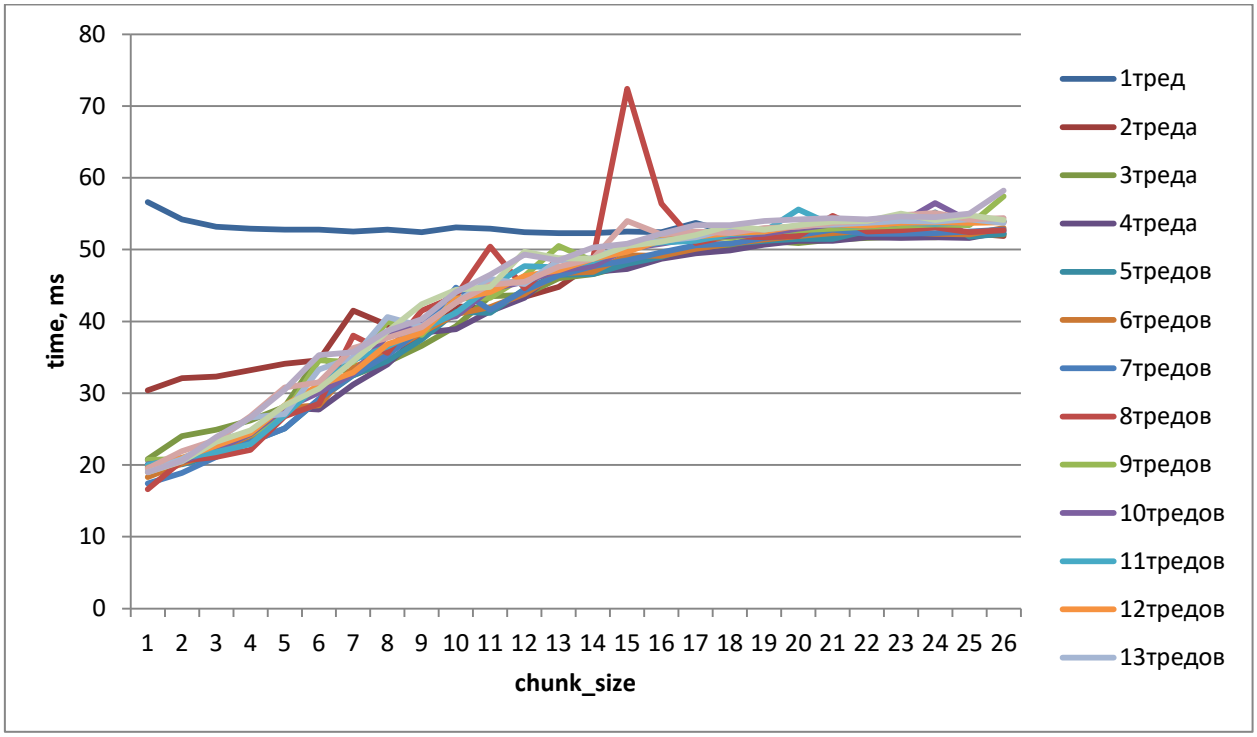

График 2 – Зависимость времени от chunk size, schedule(static)

На графике каждая линия отвечает за количество тредов, при которых получились искомые данные. Как видно с ростом chunk size время увеличивается вне зависимости от количества тредов, следовательно, лучше использовать chunk size = 1. Скорее всего, это происходит от того, что при меньших chunk_size итерации между тредами распределяются более равномерно. Стоит заметить, что деления внизу графика даны в десятках.

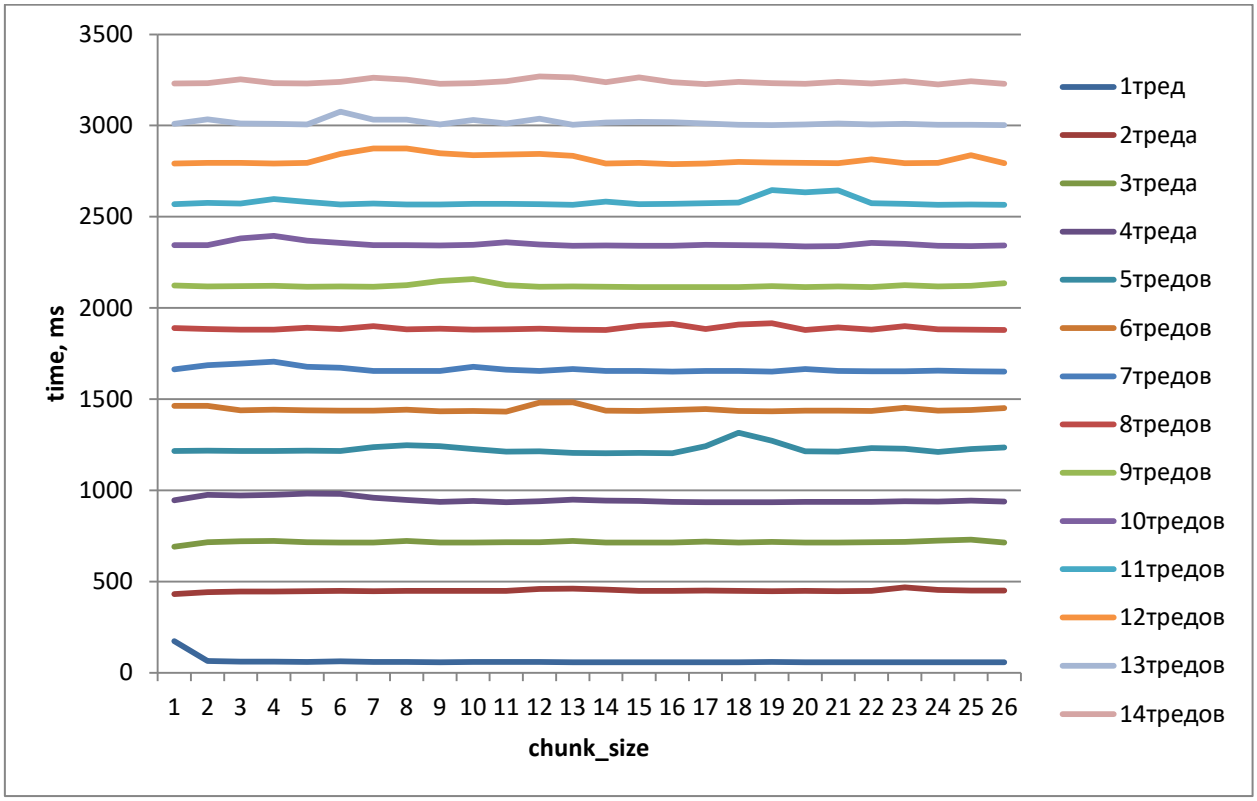

График 3 - Зависимость времени от chunk size, schedule(dynamic)

Как видно из графика в рамках одного треда скорость работы почти не отличается, но с ростом количества тредов растет и время работы.

Список источников

1) https://ru.wikipedia.org/wiki/OpenMP
2) https://learn.microsoft.com/ru-ru/cpp/parallel/openmp/openmp-c-and-cpp-application-program-interface?view=msvc-170
3) https://drive.google.com/file/d/122YfkZEbzzHFtoSx-LwwUHpg7rmqiNVw/view
4) https://nanoreview.net/ru/cpu/intel-core-i5-10300h#:~:text=Core%20i5%2010300H%20-%20процессор,4817.%20Passmark%20CPU%20(одноядерный)%202589

5)

Листинг кода