Лабораторная работа №4	**M3136**	*2023*
OpenMP	**Артемьев Иван Вадимович**	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий: Язык: *C++*, компилятор: *MinGW-w64 9.0*, стандарт *OpenMP 2.0*

Описание: Выполнить задание с использованием конструкций для распараллеливания программ с использованием библиотеки openMP.

Вариант: hard.

1. Описание конструкций OpenMP для распараллеливания команд.

Мною использовались три конструкции:

pragma omp parallel
pragma omp for
pragma omp critical

Разберёмся по очереди с ними всеми:

Первая прагма *parallel* говорит компилятору что пора запускать дополнительные потоки помимо основного, и программа написанная внутри данной секции выполняется всеми потоками.

Вторая - *for,* это прагма, которая позволяет упросить программисту жизнь и с легкостью распределить цикл *for* между потоками, данная прагма назначает какой поток будет отвечает за какие итерации данного цикла. Также этой прагме можно передать множество различных параметров один из которых я использовал - *schedule*.

Schedule говорит как следует распределять итерации между потоками, ему можно задать параметры: *static, dynamic, guided, runtime*. *Static* распределять поровну итерации между потоками - используется,

когда время затрачиваемое циклом на выполнение одной итерации примерно равно. *Dynamic* распределяет итерации динамически, то есть не по количеству а по времени, из-за чего он работает дольше, но при этом он очень полезен, когда время работы итераций разнится.

Вместе со *static* и *dynamic* можно передать параметр *chunk_size*, это установка вручную на какие секции какого размера нужно делить итерации для передачи их в работу потокам.

Critical секция - секция которая не может исполняться более чем на одном потоке одновременно. С помощью *critical* секций можно обращаться к глобальным переменным не боясь возникновения *race condition.*

2. Описание работы написанного кода

Сначала происходит считывание файла, далее создаётся гистограмма - массив, хранящий сколько раз встречается каждый цвет. Далее для быстроты создаются некоторые массивы префиксных сумм. Далее с помощью трёх вложенных циклов находим пороги с наибольшей дисперсией.

Заметим, что в функции *makeTask()* не надо параллелить ни один из циклов, потому что будет происходить false sharing.

Добавим *pragma omp parallel* для нахождения трёх порогов(перед тремя вложенными циклами). Непосредтвенно перед первым *for* напишем прагму *pragma omp for* и передадим ей параметр *schedule(dynamic),* используется *dynamic,* так как время работы каждой итерации будет разное, потому что происходит много умножений, делений и обращений к памяти которые занимают различное время. Для того чтобы найти максимальную дисперсию нам необходимо найти максимальную дисперсию в каждом из потоков, а потом найти максимальную среди них. Надо делать именно так, потому что в ином случае может произойти ситуация, когда к глобальной переменной обратятся два потока

одновременно, конечно можно это исправить написав *critical* секцию, но тогда время работы значительно увеличится.

3. Результат работы программы

Процессор: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz

Вывод программы при использовании стандартного количества потоков(то есть на вход программе, в качестве аргументов командной строки, подается строка *“omp4 0 in.pgm out.pgm”*):

В консоль:

Optimal Threshold: 77 130 187

Time for default number of threads: 75.989 ms

В файл записывается картинка, которая в png формате выглядит так:

рис 1. *Результат работы программы*

4. Экспериментальная часть

Для того, чтобы получить среднее значение, я создал цикл в котором запускаю программу 500 раз. Теперь приведу таблицы со временем работы программы в зависимости от количества поток и *schedule.* Время указано в миллисекундах.

Кол-во потоков	***без chunk_size***	***chunk_size = 4***	***chunk_size = 8***	***chunk_size = 12***
8	*77.356*	*89.3*	*76.497*	*87.089*
7	*78.69*	*83.222*	*78.937*	*86.37*
6	*86.77*	*87.298*	*82.485*	*88.8*
5	*89.482*	*93.21*	*86.868*	*98.582*
4	*94.944*	*99.22*	*93.348*	*105.218*
3	*109.805*	*118.326*	*120.901*	*121.236*
2	*143.299*	*158.556*	*156.651*	*153.392*
1	*256.568*			
без openMP	*244.009*			

Таблица 1. *Время работы программы при различных параметрах*

В итоге без использования chunk_size, получаем следующий график:

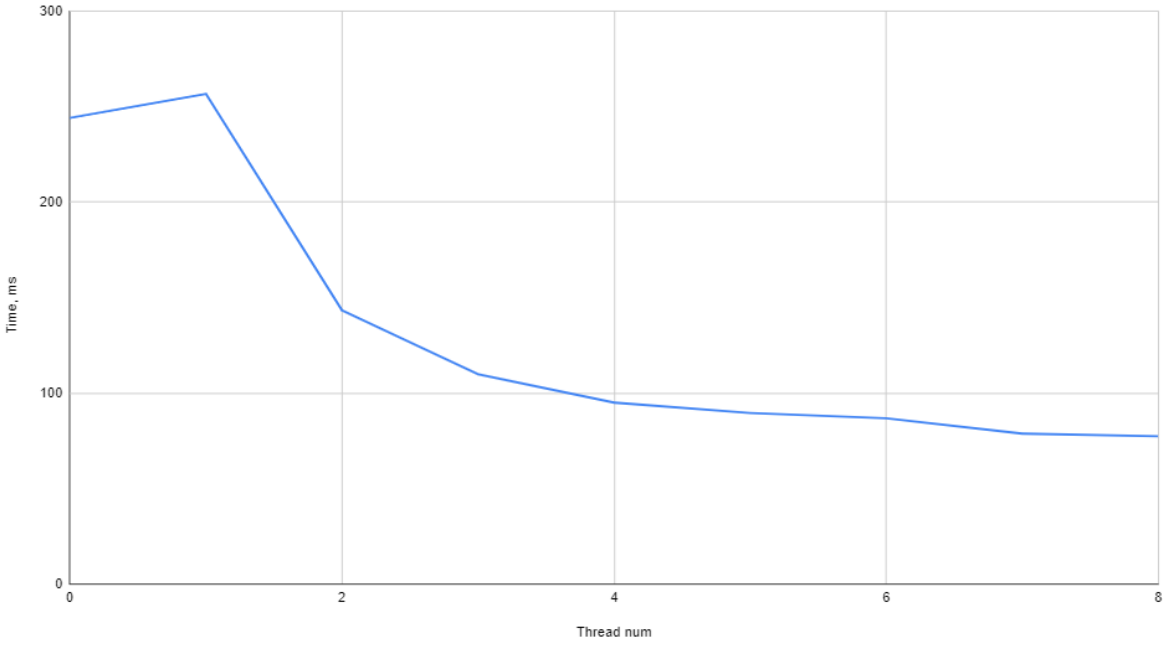

рис 2. *График отражающий зависимость времени работы от количества подключенных потоков (chunk_size не задан)*

Теперь посмотрим на время работы с различными значениями chunk_size. Мой выбор для chunk_size основан на эксперименте, в котором я запустил

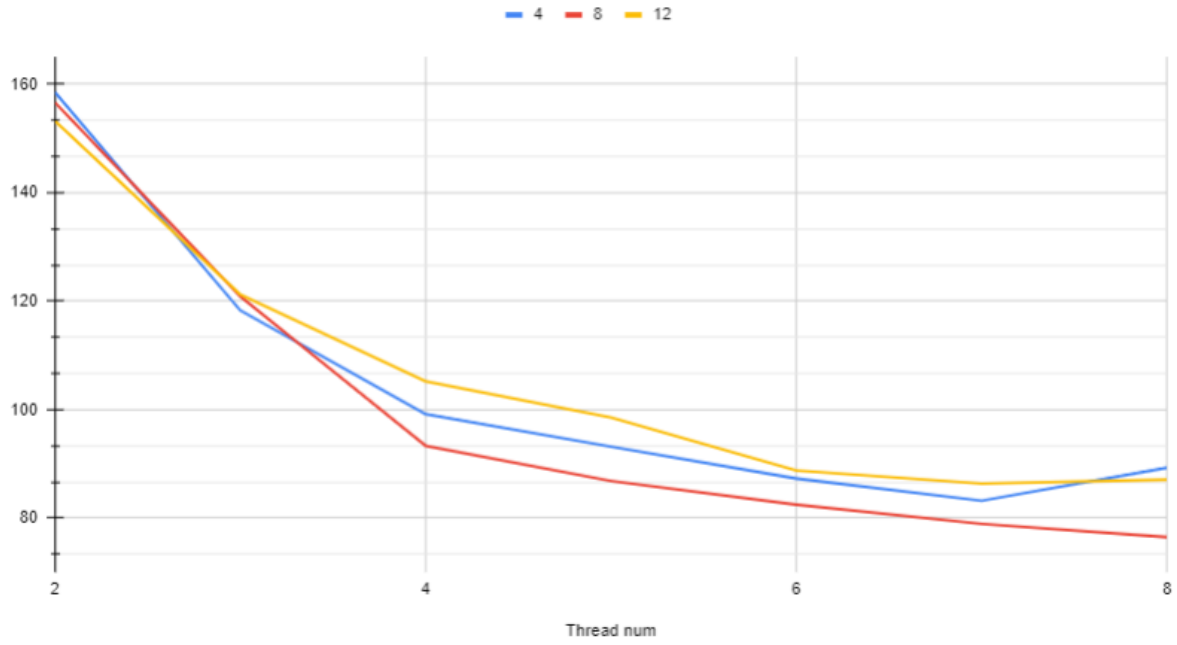

рис 3. *График отражающий зависимость времени работы от количества подключенных потоков (при различных chunk_size)*

программу на различных значениях chunk_size от 1 до 30. После эксперимента выяснилось что примерно равно время работы при значениях от 4 до 12.

5. Список источников

https://youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG

6. Листинг кода

hard.cpp

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cassert>
#include <string>
#include <vector>
#include <tuple>
#include <omp.h>

using namespace std;
const int numberOfThreshold = 3;
int numberOfThreads;
const double EPS = 1e-9;
int numRows, numCols;
int numColors, sumOfCountColors;
vector<vector<unsigned char>> colors;
vector<int> prefixCountColors;
vector<int> countColors;
vector<int> frequencies;
vector<int> prefixFrequencies;

double solveFrequency(int f0, int f1) {
     double frequency = ((prefixCountColors[f1] - prefixCountColors[f0] + countColors[f0]) * 1.0) / (sumOfCountColors);
    assert(frequency < 1 + EPS);
    return frequency;
}

double solveAvg(int f0, int f1, double q) {
    if(abs(q) < EPS) {
        return 0;
    }
        int frequency = prefixFrequencies[f1] - prefixFrequencies[f0] + frequencies[f0];
    return (frequency * 1.0) / (sumOfCountColors * q);
}
```

```
      double solveDispersion(int f0, int f1, int f2) {
         double q[numberOfThreshold + 1], m[numberOfThreshold + 1];

         q[0] = solveFrequency(0, f0);
         q[1] = solveFrequency(f0 + 1, f1);
         q[2] = solveFrequency(f1 + 1, f2);
         q[3] = solveFrequency(f2 + 1, numColors - 1);
         assert(abs(q[0] + q[1] + q[2] + q[3] - 1) < EPS);
         m[0] = solveAvg(0, f0, q[0]);
         m[1] = solveAvg(f0 + 1, f1, q[1]);
         m[2] = solveAvg(f1 + 1, f2, q[2]);
         m[3] = solveAvg(f2 + 1, numColors - 1, q[3]);

          return q[0] * m[0] * m[0] + q[1] * m[1] * m[1] + q[2] * m[2] * m[2] + q[3]
* m[3] * m[3];
      }

      tuple<int, int, int> getOptimalThresholdNoOMP() {
         tuple<int, int, int> optimalThreshold;
         double maxDispersion = 0;

         for (int f0 = 1; f0 < numColors - 3; f0++) { // [1, 252]
             for (int f1 = f0 + 1; f1 < numColors - 3 + 1; f1++) { // [2, 253]
                 for (int f2 = f1 + 1; f2 < numColors - 3 + 2; f2++) { // [3, 254]
                     double curDispersion = solveDispersion(f0, f1, f2);
                     if (curDispersion > maxDispersion) {
                         maxDispersion = curDispersion;
                         optimalThreshold = make_tuple(f0, f1, f2);
                     }
                 }
             }
         }
         cout << "Optimal Threshold: ";
          cout << get<0>(optimalThreshold) << " " << get<1>(optimalThreshold) << " "
<< get<2>(optimalThreshold);
         cout << '\n';
         return optimalThreshold;
      }

      tuple<int, int, int> getOptimalThreshold() {
         tuple<int, int, int> optimalThreshold;
         double maxDispersion = 0;
      #pragma omp parallel
      {
                tuple<double,    int,    int,    int>    optimalThresholdOnThread    =
make_tuple(0,0,0,0);
      #pragma omp for schedule(dynamic)
         for (int f0 = 1; f0 < numColors - 3; f0++) { // [1, 252]
         int thread_num = omp_get_thread_num();
             for (int f1 = f0 + 1; f1 < numColors - 3 + 1; f1++) { // [2, 253]
                 for (int f2 = f1 + 1; f2 < numColors - 3 + 2; f2++) { // [3, 254]
                     double curDispersion = solveDispersion(f0, f1, f2);
                     if (curDispersion > get<0>(optimalThresholdOnThread)) {
                            optimalThresholdOnThread = make_tuple(curDispersion, f0,
f1, f2);
                     }
                 }
             }
         }
```

```
        #pragma omp critical
        {
            if(get<0>(optimalThresholdOnThread) > maxDispersion) {
                optimalThreshold = make_tuple(get<1>(optimalThresholdOnThread),
                                              get<2>(optimalThresholdOnThread),
                                              get<3>(optimalThresholdOnThread));
                maxDispersion = get<0>(optimalThresholdOnThread);
            }
        }
        }
            cout << "Optimal Threshold: ";
             cout << get<0>(optimalThreshold) << " " << get<1>(optimalThreshold) << " "
<< get<2>(optimalThreshold);
            cout << '\n';
            return optimalThreshold;
        }

        void makeTask() {
            for(int row = 0; row < numRows; row++) {
                for(int col = 0; col < numCols; col++) {
                    countColors[(int)colors[row][col]]++;
                }
            }

            prefixCountColors[0] = countColors[0];
            for(int i = 1; i < numColors; i++) {
                prefixCountColors[i] = prefixCountColors[i - 1] + countColors[i];
            }
            sumOfCountColors = prefixCountColors[numColors - 1];

            for(int i = 0; i < numColors; i++) {
                frequencies[i] = countColors[i] * i;
            }

            prefixFrequencies[0] = frequencies[0];
            for(int i = 1; i < numColors; i++) {
                prefixFrequencies[i] = prefixFrequencies[i - 1] + frequencies[i];
            }

            tuple<int, int, int> optimalThreshold;
            if(numberOfThreads != -1) {
                optimalThreshold = getOptimalThreshold();
            } else {
                optimalThreshold = getOptimalThresholdNoOMP();
            }

            for(int row = 0; row < numRows; ++row) {
                for (int col = 0; col < numCols; ++col) {
                    if (colors[row][col] <= get<0>(optimalThreshold)) {
                        colors[row][col] = 0;
                    } else if (colors[row][col] <= get<1>(optimalThreshold)) {
                        colors[row][col] = 84;
                    } else if (colors[row][col] <= get<2>(optimalThreshold)) {
                        colors[row][col] = 170;
                    } else {
                        colors[row][col] = 255;
                    }
                }
            }
        }
```

```
int main(int argc, char** argv) {

    string inputFileName, outputFileName;

    try {
        numberOfThreads = stoi(argv[1]);
        inputFileName = argv[2];
        outputFileName = argv[3];
    } catch (const exception &e) {
        cerr << "Incorrect args " << e.what() << std::endl;
        return 1;
    }

    try {
         if(numberOfThreads > omp_get_max_threads()) cerr << "incorrect number
of threads";

        if (numberOfThreads != -1 && numberOfThreads != 0) {
            omp_set_num_threads(numberOfThreads);
        }

    } catch (const exception &e) {
        cerr << "Exception: " << e.what() << std::endl;
        return 1;
    }

    vector<vector<unsigned char>> array;
    try {
        ifstream in(inputFileName, ios_base::in | ios_base::binary);
        string inputLine;
        in >> inputLine;
        if (inputLine != "P5") cerr << "Incorrect file format";
        in >> numCols >> numRows;
        array.resize(numRows, vector<unsigned char>(numCols, 0));
        in >> numColors;
        if(numColors != 255) cerr << "Incorrect file format";
        numColors++;

        char newChar;
        in >> noskipws >> newChar;
        for(int row = 0; row < numRows; row++) {
            for (int col = 0; col < numCols; col++) {
                in >> noskipws >> array[row][col];
            }
        }

        in.close();
    } catch (const exception &e) {
        cerr << "Input file reading exception " << e.what() << std::endl;
        return 1;
    }

    const int NUM_OF_TESTS = 1;

    double sum = 0;
    for(int i = 0; i < NUM_OF_TESTS; i++) {
        colors.resize(numRows, vector<unsigned char>(numCols, 0));
        for(int row = 0; row < numRows; row++) {
            for (int col = 0; col < numCols; col++) {
                colors[row][col] = array[row][col];
            }
```

```
            }
            double tstart = omp_get_wtime();
            countColors.resize(numColors, 0);
            prefixCountColors.resize(numColors, 0);
            frequencies.resize(numColors, 0);
            prefixFrequencies.resize(numColors, 0);
            makeTask();

            double tend = omp_get_wtime();
            sum += tend - tstart;
        }

        if(numberOfThreads == 0) {
               cout << "Time for default number of threads(8): " << 1000 * sum /
NUM_OF_TESTS << " ms\n";
        } else {
             cout << "Time (" << numberOfThreads << " thread(s)): " << 1000 * sum /
NUM_OF_TESTS << " ms\n";
        }

        try {

            ofstream out(outputFileName,ios_base::out | ios_base::binary);
             out << "P5\n" << numCols << " " << numRows << '\n' << numColors - 1 <<
'\n';
            for(int row = 0; row < numRows; row++) {
                for (int col = 0; col < numCols; col++) {
                    out << colors[row][col];
                }
            }

            out.close();
        } catch (const exception &e) {
            cerr << "Input file writing exception " << e.what() << std::endl;
            return 1;
        }
    }
```