Лабораторная работа №4	M3138	2022
OpenMP	Морозов Георгий Александрович	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: работа выполнена на C++ 14. Стандарт OpenMP 2.0.

Описание

Реализация алгоритма фильтрации изображения методом Оцу для трех порогов, с использованием многопоточного программирования.

Вариант

Hard

Описание конструкций OpenMP для распараллеливания команд.

#pragma omp parallel if (N_THREADS != -1)

Данная команда создает блок, который выполняется параллельно каждым потоком. То есть код, написанный внутри этого блока выполняется n раз, где n - кол-во потоков. Если же оно равно -1, то программа выполняется без использования omp. Переменные, объявленные в этом блоке являются локальными для каждого потока.

#pragma omp for schedule(kind[, chunk_size])

Данная команда распараллеливает сразу следующий за ней for между потоками. В цикле присутствует какое-то количество итераций, которые в зависимости от аргументов schedule раздаются потокам по определенному правилу.

schedule(static):

В данном случае все итерации делятся примерно на равные чанки по кол-ву итераций во время компиляции и каждому потоку сопоставляется отдельный чанк.

schedule(static, chunk_size):

В данном случае сопоставление чанков и потоков, которыми они выполняются тоже происходит на этапе компиляции. Однако рамеры чанков определяются параметром chunk_size. Какие именно чанки сопоставляются каким именно потокам определяется по принципу round-robin. То есть первый чанк - первому потоку, второй - второму. Когда все потоки закончились (пусть их n), тогда n + 1 чанк снова отдается 1-му потоку, n + 2 - 2-му и так далее.

schedule(dynamic, chunk_size):

Если chunk_size не задан, то по умолчанию он устанавливается равным единице.

В отличие от static, в dynamic каждый конкретный чанк сопоставляется потоку уже во время исполнения. По принципу жадных

алгоритмов, когда поток освобождается, ему сопоставляются следующие chunk_size операций. Поэтому когда итерации имеют разные затраты по времени данный подход выгоднее static-а, однако когда они равнозначны, то затраты на мониторинг какому потоку отдать каждый следующий чанк только увеличивают время исполнения.

#pragma omp critical

Данная команда как раз ничего не распараллеливает, однако она нам очень полезна. Так как наша программа в какой-то момент чтобы посчитать, выполнить какие-то действия распараллеливается, каждый поток вычисляет какое-то значение и чтобы потом получить один результат, нам нужно результаты потоков как-то объединить. Однако мы не можем записывать параллельно из нескольких потоков в один адрес, так как из-за состояния гонки результат может быть непредсказуем.

Поэтому нам нужен блок critical, который находится в параллельном блоке, однако выполняется не параллельно. То есть не бывает состояния, когда этот блок кода выполняют более чем один поток одновременно.

Описание работы написанного кода.

В начале программы мы просто получаем аргументы, открываем inputFile (если такого нет в директории, бросаем ошибку). Считываем P5, width, height, 255 и далее считываем width * height байт, записывая их в inputBytesArray:

```
vector<char> inputBytesArray(width * height);
for (int i = 0; i < width * height; i++) {                    // reading bytes from file
        inputFile.read(&currentByte, 1);
        inputBytesArray[i] = currentByte;
}
```

На этом заканчивается считывание файла и начинается работа самого алгоритма, который мы должны распараллелить.

Также важно отметить наименование переменных. Если у переменной в начале стоит g (global) - то это общий ресурс для всех потоков, где и должен будет хранится в итоге полученный результат. Если у переменной в начале l (local) - то это приватная переменная для каждого потока.

```
        vector<double> gnf(L);                  // global nf - shared for each thread
#pragma omp parallel if (N_THREADS != -1)
        {
                vector<double> lnf(L);          // local nf for each thread
#pragma omp for schedule(static, 1)
                for (int i = 0; i < width * height; i+=1) {                 // filling nf array
                        // (number of occurances of each brightness in file)
                        lnf[(unsigned char)inputBytesArray[i]]++;
                }
#pragma omp critical
                {
                        for (int i = 0; i < L; i++) {
                                gnf[i] += lnf[i];
                        }
                }
        }
```

Указанный выше блок кода выполняет заполнение массива gnf размера L (в нашем случае - константа равная 256): gnf[i] - кол-во пикселей с яркостью i в изображении. Так как пиксели между собой никак не связаны, то данный процесс можно и нужно распараллелить. Однако мы не можем параллельно записывать в одну ячейку gnf, поэтому для каждого потока создадим локальный массив - lnf отвечающий за то же самое. Тогда распараллелим цикл используя #pragma omp for schedule(kind, chunk_size) и заполним массив lnf для каждого потока. После этого нам нужно объединить все данные из lnf в gnf. Но мы не можем делать это одновременно, поэтому нам нужен блок #pragma omp critical, который не будет выполнятся параллельно.

Далее просто заполняем массив частот:

```
vector<double> p(L);          // creating and filling array of frequenties
double N = width * height;
for (int i = 0; i < L; i++) {
        p[i] = gnf[i] / N;
}
```

После этого я использую оптимизацию, связанную с использованием префиксных массивов, соответственно по p[i] и i * p[i]:

```
vector<double> prefSumOfP_i(L);                    // prefix sum array of p[i]
prefSumOfP_i[0] = p[0];
for (int i = 1; i < L; i++) {
        prefSumOfP_i[i] = p[i] + prefSumOfP_i[i - 1];
}

vector<double> prefSumOfIMulP_i(L);                // prefix sum array of i * p[i]
for (int i = 1; i < L; i++) {
        prefSumOfIMulP_i[i] = p[i] * i + prefSumOfIMulP_i[i - 1];
}
```

Данные массивы понадобятся нам далее, так как нам нужны будут суммы на отрезке p[i]-ых и i * p[i] - ых. И чтобы считать такие суммы за O(1) мы и используем префиксные массивы.

Далее мы можем посчитать уже нужные нам границы gf1, gf2, gf3. И для этого также нужно хранить максимальную межкластерную дисперсию +

среднее значение яркости всего изображения в квадрате (gmaxDisp). Далее нам нужно перебрать все возможные три границы f1, f2, f3. Данный процесс тоже можно и нужно распараллелить. Распараллелим внешний цикл, каждый поток найдет свои самые выгодные границы и соответствующую им maxDisp:

```
#pragma omp for schedule(static, 1)
		for (int f1 = 0; f1 < L - 3; f1 += 1) {

			double q1 = prefSumOfP_i[f1];
			double mu1 = prefSumOfIMulP_i[f1] / q1;

			for (int f2 = f1 + 1; f2 < L - 2; f2++) {

				double q2 = prefSumOfP_i[f2] - prefSumOfP_i[f1];
				double mu2 = (prefSumOfIMulP_i[f2] - prefSumOfIMulP_i[f1]) / q2;

				for (int f3 = f2 + 1; f3 < L - 1; f3++) {
					double q3 = prefSumOfP_i[f3] - prefSumOfP_i[f2];
					double mu3 = (prefSumOfIMulP_i[f3] - prefSumOfIMulP_i[f2]) / q3;

					double q4 = prefSumOfP_i[L - 1] - prefSumOfP_i[f3];
					double mu4 = (prefSumOfIMulP_i[L - 1] - prefSumOfIMulP_i[f3]) / q4;

					double curDisp = (q1 * mu1 * mu1) + (q2 * mu2 * mu2) +
						(q3 * mu3 * mu3) + (q4 * mu4 * mu4);

					if (curDisp > maxDisp) {
						maxDisp = curDisp;
						lf1 = f1;
						lf2 = f2;
						lf3 = f3;
					}
				}
			}
		}
```

q[l, r] = q[0, r] - q[0, l - 1] (Вероятность, что случайно выбранный пиксель будем иметь яркость от l до r.

mu[l, r] = (сумма i от l до r значений i * q[i]) / q[l, r]. Но аналогично у нас уже пред посчитаны все суммы от 0 до r: i * q[i], где i пробегает все значения от 0 до a, a r пробегает все значения от 0 до 255. Тогда используя префиксный массив prefSumOfIMulP_i мы можем посчитать среднее

значение для кластера от l до r за O(1) для каждых l и r. Зафиксировав f1, f2, f3 и посчитав соответствующие q1, q2, q3, q4, mu1, mu2, mu3, mu4 вычисляем curDisp = (q1 * mu1 * mu1) + (q2 * mu2 * mu2) + (q3 * mu3 * mu3) + (q4 * mu4 * mu4). И если это значение больше maxDisp (для каждого потока - свой maxDisp), то обновляем максимум и lf1, lf2, lf3.

Также важно отметить, как расставлены скобки при вычислении curDisp. Таким образом мы распараллеливаем на уровне ILP (instruction level parallelism). Таким образом суперскаляр процессора поймет, что данные вычисления независимы и их можно распараллелить.

После этого в блоке critical мы выберем максимум и следовательно нужные нам границы, таким образом собрав данные, посчитанные каждым потоком:

```
#pragma omp critical
                {
                        if (maxDisp > gmaxDisp) {
                                gmaxDisp = maxDisp;
                                gf1 = lf1;
                                gf2 = lf2;
                                gf3 = lf3;
                        }
                }
```

После этого этапа мы посчитали нужные нам границы gf1, gf2, gf3. Все что осталось сделать, это пробежаться по нашему массиву байтов inputBytesArray и в зависимости от того, в какой промежуток попадает значение яркости каждого байта, поменять их значения на 0, 84, 170, 255. Для каждого пикселя данный процесс независим, поэтому мы его тоже распараллелим:

```
#pragma omp for schedule(static, 1)
                for (int i = 0; i < width * height; i++) {
                        int currentBrightness = (unsigned char)inputBytesArray[i];
                        if (0 <= currentBrightness && currentBrightness <= gf1) {
                                inputBytesArray[i] = 0;
                        }
                        else if (gf1 < currentBrightness && currentBrightness <= gf2) {
                                inputBytesArray[i] = 84;
                        }
                        else if (gf2 < currentBrightness && currentBrightness <= gf3) {
```

```
                    inputBytesArray[i] = 170;
                }
                else if (gf3 < currentBrightness && currentBrightness <= 255) {
                    inputBytesArray[i] = 255;
                }
            }
```

После этого осталось записать наш массив в выходной файл в формате P5:

```
    ofstream outputFile(outputFileName, std::ios::out | std::ios::binary);
    outputFile << "P5\n";
    outputFile << width << " " << height << "\n";
    outputFile << 255 << "\n";

    outputFile.write((char*)&inputBytesArray[0],
        (sizeof inputBytesArray[0]) * inputBytesArray.size());
```

Таким образом наша программа по-сути три раза распараллеливается, где один поток распускается в несколько и после в критикал блоке снова сливается в один поток.

Результат работы написанной программы с указанием процессора, на котором производилось тестирование.

Процессор: AMD Ryzen 7 4800HS with Radeon Graphics 2.90 GHz

При запуске на тестовом файле, указанном в условии:

77 130 187

Time (16 thread(s)): 67.1372 ms

Результат работы до и после:

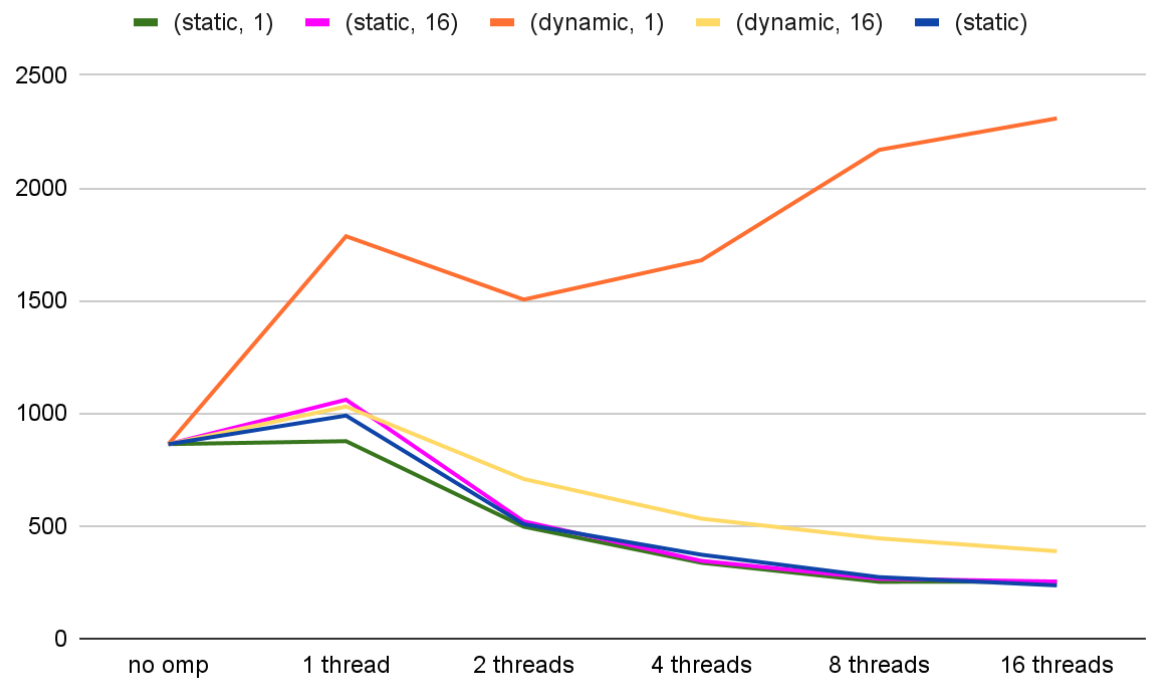
Размер файла .png исходного изображения: 243 KB

Размер файла .png преобразованного изображения: 24.3 KB

Также важно отметить, что здесь и далее программа запускается без подключенной зарядки, чтобы программа выполнялась не так быстро, чтобы повысить точность измерений.

Экспериментальная часть

Для каждого значения в графике бралось среднее из нескольких запусков. По вертикали указано количество миллисекунд. Для более точных измерений, запуски производились на изображении 8к формата.



Как видно из графика, от увеличения количества потоков почти всегда уменьшается время исполнения программы. Однако в указанном графике количество потоков увеличивается каждый раз в два раза, однако при 8 и 16 потоках время исполнения явно не уменьшилось в два раза. Это связано с тем, что задача очень мелкая и на то, чтобы хэндлить большое количество потоков уходит соразмерное задачи количество времени.

Также, при schedule(dynamic, 1) (что эквивалентно просто schedule(dynamic)) время исполнения вообще увеличивается. Это связано с тем, что во-первых каждая итерация занимает по времени примерно одинаковое количество времени и динамическое распределение итераций

по потокам во время исполнения нам не нужно. Особенно если после каждой итерации мы смотрим какой поток свободен и кому отдать текущую итерацию - это очень вредит производительности.

При schedule(dynamic, 16) некоторые из проблем dynamic в нашей задачи остаются (поэтому видно, что немного медленнее, чем static варианты), однако в разы быстрее чем schedule(dynamic, 1) так как размер чанка теперь равен 16 и мы в 16 раз реже обращаемся к состояниям потоков, чтобы узнать кому отправить текущий чанк итераций.

При static, неважно какой chunk_size, время выполнения примерно равно (с точностью до погрешности). Оно и не удивительно, так как каждый поток во время компиляции назначается к каким-то чанкам, а так как каждая итерация независима и примерно одинакова по времени другим, то в каком порядке мы их выполняем (в этой задаче) не важно.

Ниже представлена таблица, по которой строился график.

	(static, 1)	(static, 16)	(dynamic, 1)	(dynamic, 16)	(static)
no omp	862.9866	862.9866	862.9866	862.9866	862.9866
1 thread	876.079	1060.0248	1785.4875	1029.78063	989.57416
2 threads	496.52125	520.034	1504.455	708.1795	507.8358
4 threads	336.432	344.857	1679.08	532.246875	372.1054
8 threads	252.013375	265.690667	2168.84	444.7035	272.7192
16 threads	253.156889	253.0844	2308.592	387.740857	236.1266

Список источников.

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

https://www.openmp.org/wp-content/uploads/cspec20.pdf

Листинг кода.

hard.cpp

```
#include <iostream>
#include <fstream>
#include <omp.h>
#include <cstdint>
#include <vector>
#include <string>

constexpr auto L = 256;

using namespace std;

int parseToInt(string s)
{
	int res = 0;
	int n = s.length();
	int i = 0;
	bool isNegative = false;
	if (s[i] == '-') {
		isNegative = true;
		i += 1;
	}
	while (i < n)
	{
		if (s[i] > '9' || s[i] < '0') {
			throw invalid_argument("Error, Illegal argument for number of threads");
		}
		res *= 10;
		res += s[i] - '0';
		i += 1;
	}
	return isNegative ? -res : res;
}
```

```
int main(int argc, char** argv)
{
	if (argc != 4) {
		throw invalid_argument("Error, Illegal number of aruments. Expected: 4 Actual: " +
			to_string(argc));
	}
	int N_THREADS = parseToInt(string(argv[1]));
	if (N_THREADS < -1) {
		throw invalid_argument("Error, Illegal number of threads. Expected integer >= -1");
	}
	string inputFileName = string(argv[2]);
	string outputFileName = string(argv[3]);

	fstream inputFile(inputFileName, ios::in | ios::out | ios::binary);
	if (!inputFile) {
		inputFile.close();
		throw invalid_argument("Error, no such file in the directory");
	}
	char ch1, ch2;
	inputFile >> ch1 >> ch2;			// reading P5
	if (ch1 != 'P' || ch2 != '5') {
		inputFile.close();
		throw invalid_argument("Illegal file format, expected P5 pgm file");
	}
	int width, height;
	inputFile >> width >> height;		// reading width, height
	int r;
	inputFile >> r;						// reading 255
	char currentByte;
	inputFile.read(&currentByte, 1);	// reading /n
	vector<char> inputBytesArray(width * height);
	for (int i = 0; i < width * height; i++) {		// reading bytes from file
		inputFile.read(&currentByte, 1);
		inputBytesArray[i] = currentByte;
	}
	double timeStart = omp_get_wtime();
	if (N_THREADS > 0) {
		omp_set_num_threads(N_THREADS);
	}
	vector<double> gnf(L);					// global nf - shared for each thread
#pragma omp parallel if (N_THREADS != -1)
	{
		vector<double> lnf(L);				// local nf for each thread
#pragma omp for schedule(static, 1)
		for (int i = 0; i < width * height; i+=1) {		// filling nf array
			// (number of occurances of each brightness in file)
			lnf[(unsigned char)inputBytesArray[i]]++;
		}
```

```
#pragma omp critical
                {
                        for (int i = 0; i < L; i++) {
                                gnf[i] += lnf[i];
                        }
                }
        }
        vector<double> p(L);            // creating and filling array of frequenties
        double N = width * height;
        for (int i = 0; i < L; i++) {
                p[i] = gnf[i] / N;
        }
        vector<double> prefSumOfP_i(L);                 // prefix sum array of p[i]
        prefSumOfP_i[0] = p[0];
        for (int i = 1; i < L; i++) {
                prefSumOfP_i[i] = p[i] + prefSumOfP_i[i - 1];
        }

        vector<double> prefSumOfIMulP_i(L);             // prefix sum array of i * p[i]
        for (int i = 1; i < L; i++) {
                prefSumOfIMulP_i[i] = p[i] * i + prefSumOfIMulP_i[i - 1];
        }

        int gf1, gf2, gf3;
        double gmaxDisp = 0;
#pragma omp parallel if (N_THREADS != -1)
        {
                int lf1, lf2, lf3;
                double maxDisp = 0;
#pragma omp for schedule(static, 1)
                for (int f1 = 0; f1 < L - 3; f1 += 1) {

                        double q1 = prefSumOfP_i[f1];
                        double mu1 = prefSumOfIMulP_i[f1] / q1;

                        for (int f2 = f1 + 1; f2 < L - 2; f2++) {

                                double q2 = prefSumOfP_i[f2] - prefSumOfP_i[f1];
                                double mu2 = (prefSumOfIMulP_i[f2] - prefSumOfIMulP_i[f1]) / q2;

                                for (int f3 = f2 + 1; f3 < L - 1; f3++) {
                                        double q3 = prefSumOfP_i[f3] - prefSumOfP_i[f2];
                                        double mu3 = (prefSumOfIMulP_i[f3] - prefSumOfIMulP_i[f2]) / q3;

                                        double q4 = prefSumOfP_i[L - 1] - prefSumOfP_i[f3];
                                        double mu4 = (prefSumOfIMulP_i[L - 1] - prefSumOfIMulP_i[f3]) / q4;

                                        double curDisp = (q1 * mu1 * mu1) + (q2 * mu2 * mu2) +
                                                (q3 * mu3 * mu3) + (q4 * mu4 * mu4);
```

```
                                        if (curDisp > maxDisp) {
                                                maxDisp = curDisp;
                                                lf1 = f1;
                                                lf2 = f2;
                                                lf3 = f3;
                                        }
                                }
                        }
                }
#pragma omp critical
                {
                        if (maxDisp > gmaxDisp) {
                                gmaxDisp = maxDisp;
                                gf1 = lf1;
                                gf2 = lf2;
                                gf3 = lf3;
                        }
                }
        }
        printf("%u %u %u\n", gf1, gf2, gf3);
#pragma omp parallel if (N_THREADS != -1)
        {
                int id = omp_get_thread_num();
                if (id == 0) {
                        N_THREADS = omp_get_num_threads();
                }
#pragma omp for schedule(static, 1)
                for (int i = 0; i < width * height; i++) {
                        int currentBrightness = (unsigned char)inputBytesArray[i];
                        if (0 <= currentBrightness && currentBrightness <= gf1) {
                                inputBytesArray[i] = 0;
                        }
                        else if (gf1 < currentBrightness && currentBrightness <= gf2) {
                                inputBytesArray[i] = 84;
                        }
                        else if (gf2 < currentBrightness && currentBrightness <= gf3) {
                                inputBytesArray[i] = 170;
                        }
                        else if (gf3 < currentBrightness && currentBrightness <= 255) {
                                inputBytesArray[i] = 255;
                        }
                }
        }
        double timeEnd = omp_get_wtime();
        printf("Time (%i thread(s)): %g ms\n", N_THREADS, (timeEnd - timeStart) * 1000);
        ofstream outputFile(outputFileName, std::ios::out | std::ios::binary);
        outputFile << "P5\n";
        outputFile << width << " " << height << "\n";
```

```
        outputFile << 255 << "\n";

        outputFile.write((char*)&inputBytesArray[0],
                (sizeof inputBytesArray[0]) * inputBytesArray.size());
        outputFile.close();
        inputFile.close();
        return 0;
}
```