Лабораторная работа №4	M3136	2023
OpenMP	Багринцев Михаил Алексеевич	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: Необходимо написать многопоточную программу на C/C++ с использованием OpenMP 2.0. Код был написан на языке C++ 17-ой версии и скомпилирован с помощью GNU C17 (GCC) version 11.2.0 (x86_64-w64-mingw32).

Описание:

Необходимо написать программу, решающую задачу одного из вариантов. Помимо написания программы, необходимо провести замеры времени работы на вашем компьютере и привести графики времени работы программы (некоторые графики из следующих подпунктов можно объединить в один):

1. при различных значениях числа потоков при одинаковом параметре schedule* (без chunk_size);
2. при одинаковом значении числа потоков при различных параметрах schedule* (с chunk_size);
3. с выключенным openmp и с включенным с 1 потоком.

В каждом варианте результат работы программы выводится в выходной файл, а время работы программы - в поток вывода (stdout). Формат вывода (в формате Си): `“Time (%i thread(s)): %g ms\n”`. Время работы выводится только в консоль. В данном случае имеется в виду время работы алгоритма без времени на считывание данных и вывод результата.

Вариант: Easy. В этом варианте необходимо было реализовать расчёт площади круга методом Монте-Карло. Что из себя представляет вычисление площади методом Монте-Карло? Это когда мы имеем одну фигуру (внешнюю), площадь которой мы знаем и которая точно покрывает другую (внутреннюю) - площадь, которой мы и хотим найти и мы рандомно проставляем много точек внутри первой фигуры. Также нам нужно уметь определять, попала ли точка внутрь внутренней фигуры или нет, так как её площадь мы будем считать как отношение точек, попавших в неё, к общему числу точек умножить на площадь внешней фигуры.

Описание конструкций OpenMP

OpenMP - это API, поддерживающее программирование мультипроцессорных вычислений над общей памятью на языках C/C++/Fortran. Поддерживается openMP на многих ISA (x86, RISC-V…) и операционных система (macOS, Windows, FreeBSD, Linux…). OpenMP состоит из

- набора директив компилятора

```
#pragma omp parallel default(none)
#pragma omp for
```

- функций

```
omp_get_thread_num()
omp_get_num_threads()
```

- переменных окружения

```
OMP_SCHEDULE
OMP_NUM_THREADS
```

OpenMp - это реализация классической модели параллелизма Fork-join.

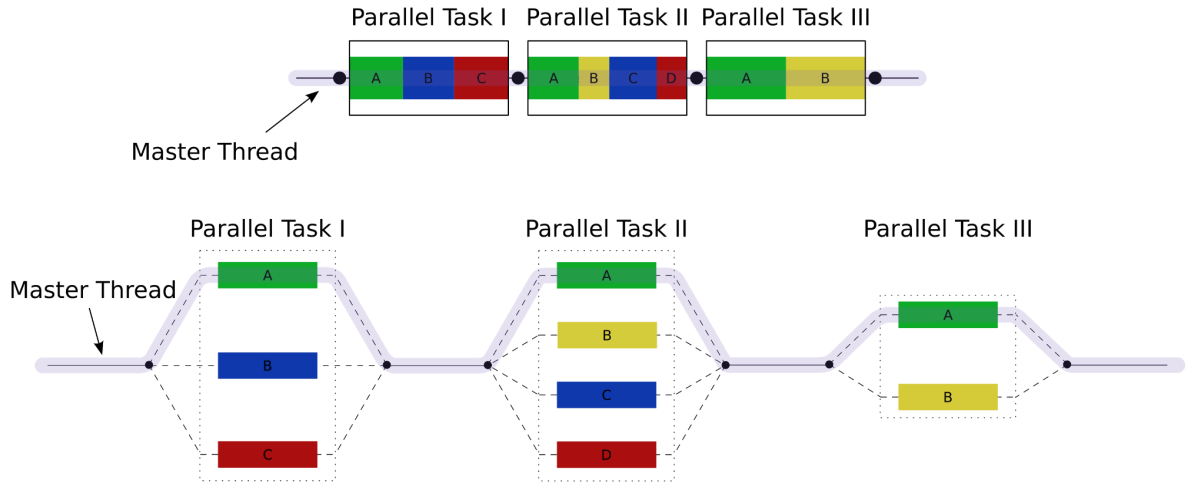

Рисунок 1 - Fork-join model

Эта модель параллелизма, при которой основной поток (Master Thread) разветвляется на необходимое количество потоков, между которыми уже делится выполнение задачи. По окончанию выполнения задачи, потоки сливаются в исходный Master Thread. Процессы разветвления и слияния называются fork и join соответственно.

Самая простая директива openMP - parallel. Она определяет регион блок кода, который будет выполнен в несколько потоков. У неё существуют аргументы, которые в документации называются clause:

- if (scalar-expression) - условие на распараллеливание
- private (variable-list) - список локальных переменных для каждого потока
- firstprivate (variable-list) - список локальных переменных с инициализацией
- default (shared | none) - область видимости переменных, не указанных явно
- shared (variable-list) - список общих на все потоки переменных
- copyin (variable-list) - список глобальных переменных, которые станут локальными для потоков и будут инициализированы значениями в master thread.

- reduction (operator: variable-list) - применяет оператор к результатам локальных переменных из списка и общий результат записывается в переменную master thread (после join).
- num_threads (integer-expression) - явное указание количества потоков, на которые распараллеливается master thread. Если переменная не указана, то количество потоков устанавливается через omp_set_num_threads() или переменную OMP_NUM_THREADS.

Распределение работы

- **for** конструкция определяет цикл, итерации которого будут выполняться параллельно. У for директивы есть интересный пункт - schedule. У schedule также есть параметры - тип и размер куска, на которые делится цикл (chunk_size). Типы могут быть одним из
    - static - куски (chunks) последовательно распределяются между потоками. Сначала chunk загружается в первый поток, потом следующий chunk - во второй поток и так до n-ого потока (chunk - число, определяющее количество итераций цикла). Потом, если цикл не заполнился первый поток нагрузят следующим куском и так далее. После распределения начинается выполнение итераций.
    - dynamic - здесь куски загружаются в потоки, когда те освобождаются. То есть если какой-то поток будет быстрее выполнять свои куски, то он суммарно выполнит большее их количество.
    - guided - dynamic тип с особенностью, заключающейся в том, что размер куска (chunk_size) не равен тому, который мы указали, а не превышает его и может быть

уменьшен операционной системой.

- runtime - выбирает режим, указанный в переменной окружения OMP_SCHEDULE (static по умолчанию).

Также директива **for** имеет параметр nowait, которой используется, если мы хотим, чтобы треды, выполнившие свою часть цикла не ждали остальных, а шли выполнять параллельный код дальше.

В своей программе я использовал функции установки количества потоков, получения номера потока (thread) и получения количества потоков, которое нам выделила система.

```
omp_set_num_threads(num_of_threads);
omp_get_thread_num() # thread id
omp_get_num_threads() # number of threads
```

Также для распараллеливания небольшого участка кода я использовал директиву parallel с соответствующими clause-ами:

```
#pragma omp parallel default(none) shared(hits, attempts,
radius_power, radius, threads_num)
{
...
}
```

Здесь написано, что мы хотим выполнить параллельно участок кода в фигурных скобках с количеством потоков, указанным функцией omp_set_num_threads. Ещё мы указали с помощью default clause, что только переменные, находящиеся в shared являются общими.

Директива, которую я тоже оставил не без внимания - critical.

```
#pragma omp critical
```

Она делает безопасную операцию, то есть в каждый момент времени только один поток выполняет код в критической области.

Описание работы написанного кода

Если опустить детали обработки ошибок, то программа проверяет количество потоков, переданное в качестве аргумента командной строки и устанавливает его с помощью функции omp_set_num_threads, если оно не равно -1.

```
#pragma omp parallel default(none) shared(hits, attempts,
radius_power, radius, threads_num)
    {
        double x, y;
        const int thread_id = omp_get_thread_num();
        long long hits_in_thread = 0;
        if (thread_id == 0) {
            threads_num = omp_get_num_threads();
        }

        mt19937_64 gen(time(nullptr) ^ (thread_id <<
5));
        uniform_real_distribution<double> dis(-radius,
radius);
#pragma omp for schedule(dynamic, 1000) nowait
        for (int i = 0; i < attempts; ++i) {...}
#pragma omp critical
        hits += hits_in_thread;
    }
```

Каждый тред считает количество попаданий в локальной переменной hits_in_thread и потом мы безопасно прибавляем подсчитанные значения в критической области. Опущенный в данном куске for вставлен ниже. Когда работает Master Thread, он записывает в разделяемую переменную количество тредов, на которое разделился основной поток (система может выделить меньше тредов, чем мы запросили). Эта переменная используется для вывода логов. Каждый тред имеет свой генератор рандомных чисел, так как для начального значения

мы используем номер треда и время.

```
#pragma omp for schedule(dynamic, 1000) nowait
          for (int i = 0; i < attempts; ++i) {
               x = dis(gen);
               y = dis(gen);
               if (x * x + y * y <= radius_power) {
                    hits_in_thread++;
               }
          }
```

Выше представлен цикл, которой собственно считает количество попаданий внутрь круга. В качестве фигуры, в которой содержится наш круг был выбран квадрат со сторонами удвоенного радиуса круга. Параметры schedule могут быть изменены для оптимизации. Помимо schedule, был использован параметр nowait, потому что после этого for у нас идёт критическая область, в которой мы суммируем локальные переменные в глобальную.

Результат работы программы

Для тестов возьмём три два значения attempts - с github и 100000000. Значение schedule установим dynamic, chunk_size = 1000.

Input	Кол-во тредов	Лог (среднее значение времени)
1 100000	-1	Time (1 thread(s)): 0.00900006 ms.
	1	Time (1 thread(s)): 0.00900006 ms.
	2	Time (2 thread(s)): 0.00499988 ms.
	4	Time (4 thread(s)): 0.00300002 ms.
	8	Time (8 thread(s)): 0.00199986 ms.
	16	Time (16 thread(s)): 0.00199986 ms.

1 100000000	-1	Time (1 thread(s)): 8.48 ms.
	1	Time (1 thread(s)): 8.545 ms.
	2	Time (2 thread(s)): 4.342 ms.
	4	Time (4 thread(s)): 2.263 ms.
	8	Time (8 thread(s)): 1.365 ms.
	16	Time (16 thread(s)): 1.082 ms.

CPU	AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz

Экспериментальная часть

1. Различные значениях числа потоков при одинаковом параметре schedule* (без chunk_size).

   Параметр schedule поставим runtime.

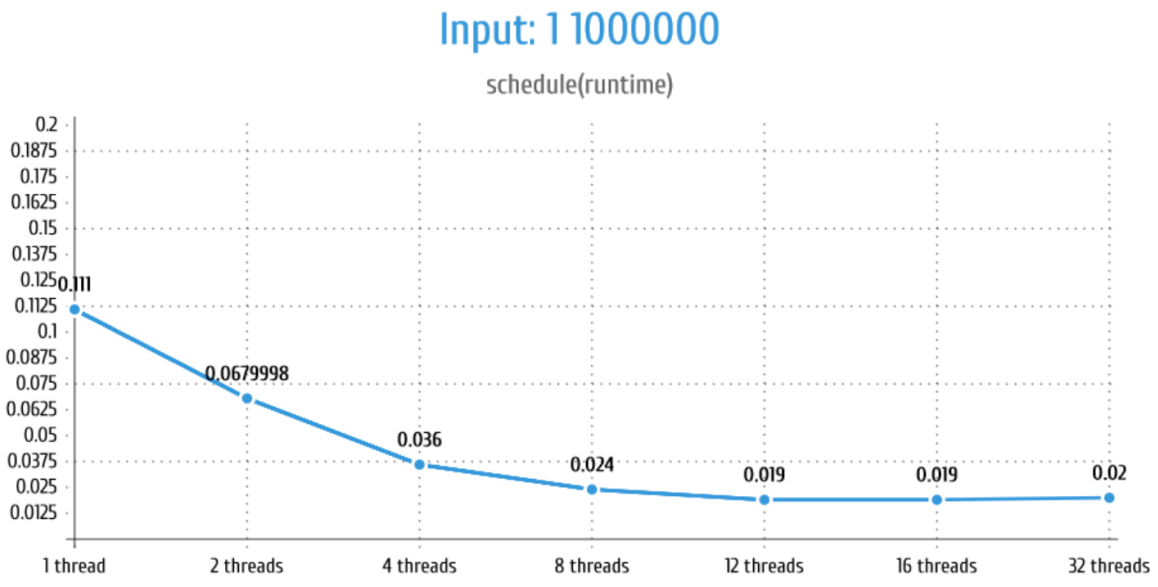

Рисунок 2 - График для различного числа потоков при input: 1 1000000.

Прирост скорости с увеличением потоков очевиден. Но прирост наблюдается только пока мы используем настоящие потоки. То есть в случае моего процессора это количество ядер умноженное на 2 (Multi-threading) - 12 потоков. При двенадцати потоках мы получили прирост производительности приблизительно в 5.5 раз. Но если увеличивать количество потоков дальше, то мы будем получать уменьшение производительности, так как больше времени будет уходить на переключение между разными потоками.

2. Одинаковое значение числа потоков при различных параметрах schedule* (с chunk_size).

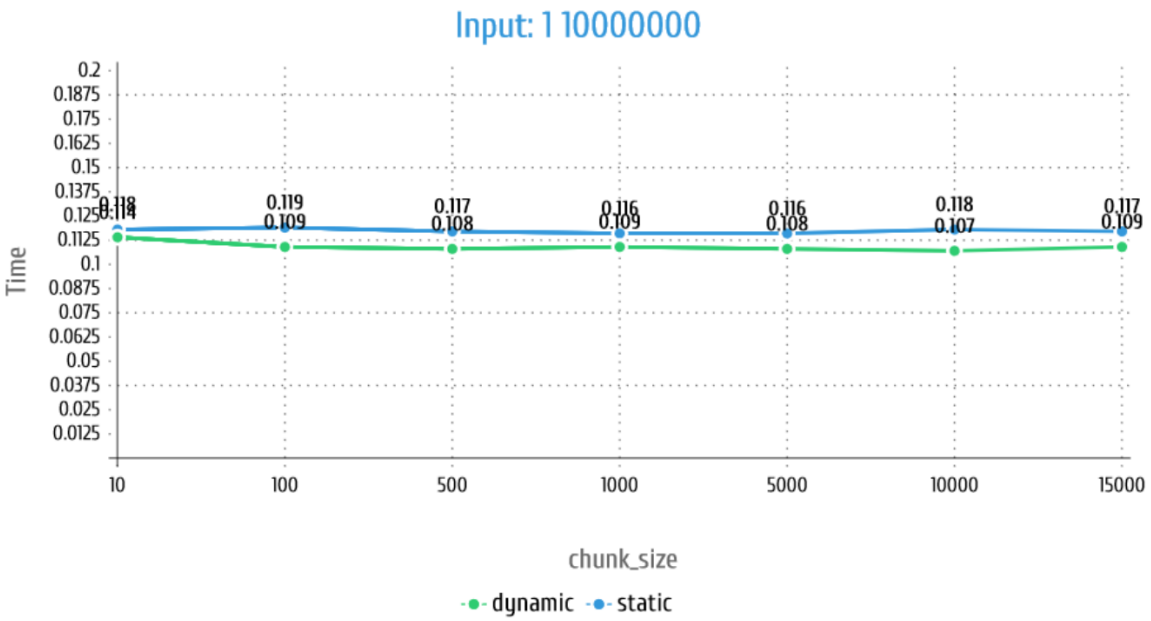

Рисунок 3 - График для различных параметров schedule на 12 потоках.

Из приведённого выше графика мы видим, что время работы цикла в dynamic и static режимах имеют совсем небольшую разницу. Это связано с тем, что в цикле, который мы распараллеливаем, все итерации занимают одинаковое время. Действительно, на каждой итерации каждый поток просто считает 2 следующих псевдо-рандомных числа и возводит их в квадрат - никаких неожиданностей. Динамический параметр schedule давал бы значительный прирост, если бы диапазон времени вычисления на

разных итерациях был довольно большой. Динамическое планирование показывает себя немного лучше на рисунке 3, потому что dynamic scheduling не занимается вопросами синхронизации и защитой между потоками.

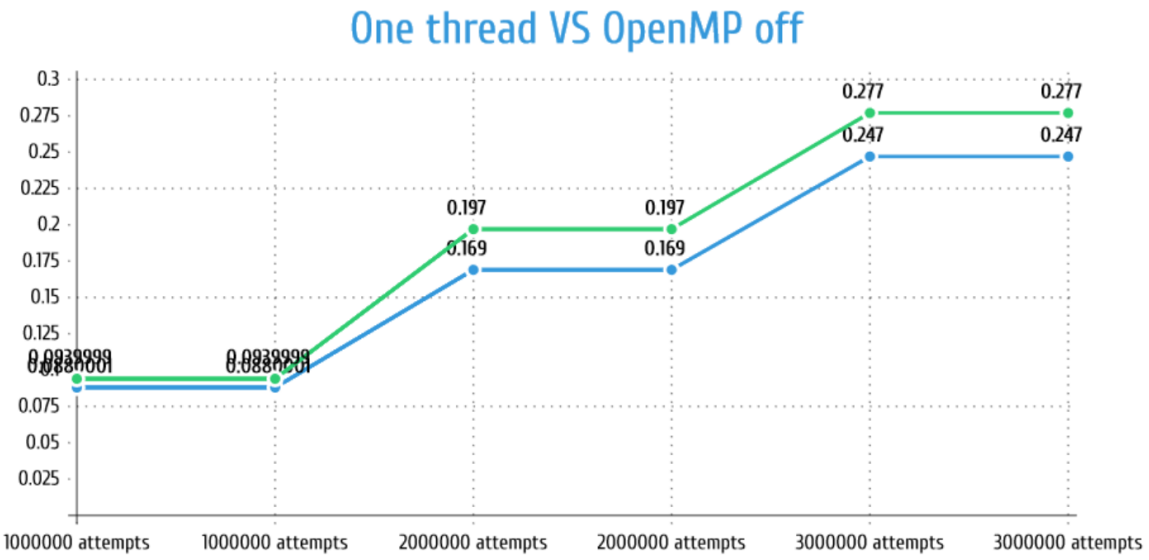

Рисунок 4 - График для сравнения работы программы с одним потоком и последовательным выполнением на разных входных данных.

Зелёная линия, отвечающая за время работы при argv[1] = 1, явно доминирует над синей линией, отвечающей за время работы при argv[1] = -1. Из этого графика становится очевидно, что выполнение программы в один поток менее эффективно, когда определённая её часть написана для нескольких потоков. Причём чем больше входные значения, тем больше разница по времени. Выполнение в один поток многопоточной части кода зачастую настолько неэффективно, что лучше выполнять программу последовательно, потому что компилятор сможет сам оптимизировать определённые места и не будет тратиться времени на выполнение команд openMP.

Список источников

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

2. https://en.wikipedia.org/wiki/OpenMP

Листинг

easy.cpp

```
#include <iostream>
#include <omp.h>
#include <random>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <cstdio>
#include <iosfwd>
#include <string>

using namespace std;

int main(int argc, char* argv[])
{
	if (argc != 4) {
	cout << "Illegal number of arguments in command line." << endl;
	cout << "You entered:" << '\t';
	for (int i{0}; i < argc; ++i) {
		cout << argv[i] << ' ';
	}
	return 1;
	}
	string s = argv[1];
	char* end_num;
	int num_of_threads = strtol(argv[1], &end_num, 10);
	if (argv[1] == end_num) {
	cout << "Cannot convert number of threads (argv[1]) to integer" <<
endl;
	return 1;
	} else if (num_of_threads < -1) {
	cout << "Number of threads (argv[1]) cannot be less than -1." <<
"Your argv[1]=" << num_of_threads << '.' << endl;
	return 1;
	}

	ifstream in_file{argv[2]};
	if (!in_file) {
	cout << "Cannot open file " << argv[2] << " for reading." << endl;
	return 1;
```

```
	}

	double radius;
	long long attempts;

	in_file >> radius;
	in_file >> attempts;

	if (!in_file) {
	cout << "Error while trying to read params from open file" << endl;
	return 1;
	}
	in_file.close();

	double start, end;
	long long hits = 0;
	const double radius_power = radius * radius;
	int threads_num;

	start = omp_get_wtime();

	if (num_of_threads == -1) {
	double x, y;
	random_device rd;
	mt19937_64 gen(rd());
	uniform_real_distribution<double> dis(-radius, radius);
	for (long long i{0}; i < attempts; ++i) {
		x = dis(gen);
		y = dis(gen);
		if (x * x + y * y <= radius_power) {
			hits++;
		}
	}
	} else {
	if (num_of_threads != 0) {
		omp_set_num_threads(num_of_threads);
	}
#pragma omp parallel default(none) shared(hits, attempts, radius_power,
radius, threads_num)
	{
		double x, y;
		const int thread_id = omp_get_thread_num();
		long long hits_in_thread = 0;
		if (thread_id == 0) {
			threads_num = omp_get_num_threads();
		}
```

```
            mt19937_64 gen(time(nullptr) ^ (thread_id << 5));
            uniform_real_distribution<double> dis(-radius, radius);
#pragma omp for schedule(dynamic, 1000) nowait
            for (int i = 0; i < attempts; ++i) {
                  x = dis(gen);
                  y = dis(gen);
                  if (x * x + y * y <= radius_power) {
                  hits_in_thread++;
                  }
            }
#pragma omp critical
            hits += hits_in_thread;
      }
      }

      end = omp_get_wtime();

      ofstream out_file{argv[3]};
      if (!out_file) {
      cout << "Cannot open file " << argv[3] << " for writing." << endl;
      return 1;
      }

      const double square_area = radius_power * 2 * 2;
      const double square = (double) hits /(double) attempts *
square_area;

      out_file << square << endl;
      out_file.close();

      printf("Time (%d thread(s)): %g ms.\n", (num_of_threads == -1) ? 1 :
threads_num, end - start);

      return 0;
}
```