Лабораторная работа №4	группа 05	2022
OpenMP	Москаленко Т.Д.	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: работа выполнена на C++. Стандарт OpenMP 2.0. Компилятор, на котором работал:

```
Apple clang version 14.0.0 (clang-1400.0.29.202)
```

Описание конструкций OpenMP для распараллеливания команд

OpenMP *(Open Multi-Processing)* — открытый стандарт для распараллеливания программ на языках C, C++ и Фортран. Позволяет писать многопоточные приложения на многопроцессорных системах с общей памятью на этих языках. Рассмотрим некоторые конструкции стандарта, нужные нам для написания лабораторной:

- Конструкция `parallel`:

```
#pragma omp parallel [clause[ [, ]clause] ...]
{
   code
}
```

  Данная конструкция создаёт блок в программе, который будет выполняться в несколько потоков.

  Все возможные значения `clause` представлены ниже:

  - `if`(scalar-expression)
  - `private`(variable-list)
  - `firstprivate`(variable-list)
  - `default`(shared | none)
  - `shared`(variable-list)
  - `copyin`(variable-list)
  - `reduction`(operator: variable-list)
  - `num_threads`(integer-expression)

  Параметр `num_threads` позволяет выставить нужное количество потоков (от 1 до 8) внутри многопоточного куска программы (в работе используется аналогичная функция `omp_set_num_threads()`).

```
#pragma omp parallel if(parallel == 1) num_threads(threads)
```

- Конструкция `for`:

```
#pragma omp for [clause[[,] clause] ... ]
for(...;...;...) {
   code
}
```

Данная конструкция позволяет распределить вычисления внутри блока `for` между потоками, по правилу, которое задаётся с помощью `clause`.

Все возможные значения `clause` представлены ниже:

- `private`(variable-list)
- `firstprivate`(variable-list)
- `lastprivate`(variable)
- `reduction`(operator: variable-list)
- `schedule`(kind[, chunk_size])
- `nowait`

В программе мы будем пользоваться только `schedule`, поэтому подробнее становимся именно на ней.

`schedule`(с англ. *план, расписание*) — параметр, который указывает, как итерации цикла `for` распределяются между потоками команды. Корректность программы не должна зависеть от того, какой поток выполняет конкретную итерацию. Значение `chunk_size`, если оно указано, должно быть положительным целым числом, не зависящим от внутреннего блока `for`.

Параметр `kind` в нашей лабораторной используется один из следующих:

- `static` — итерации разделены на фрагменты размера `chunk_size`. Фрагменты статически назначаются потокам в циклическом порядке по номерам. Если размер `chunk_size` не указан, все итерации делятся на фрагменты примерно одинакового размера. Каждому потоку назначается один фрагмент.
- `dynamic` — этот тип работает также как статический, отличается лишь распределением фрагментов между потоками. Фрагменты упорядочены, и распределяются по порядку. Каждый новый фрагмент первому освободившемуся потоку. Если размер `chunk_size` не указан, по умолчанию он равен 1.

- Конструкция `atomic`:

```
#pragma omp atomic
   expression-stmt
```

Данная конструкция гарантирует, что определенное место в памяти (место, где лежит переменная x) обновляется только одним потоком в конкретный момент времени. В качестве `expression-stmt` может быть одно из следующих выражений

- `x binop= expr`
- `x++`
- `++x`
- `x--`
- `--x`

- Функция `omp_get_wtime()`:

  Данная функция возвращает значение типа `double` равное прошедшему времени настенных часов (wall clock) в секундах с некоторого «времени в прошлом». Конкретное «время в прошлом» не определено строго, но оно гарантированно не изменится во время выполнения программы.

Описание работы написанного кода

Рассмотрим начальную часть функции `main`

```
if (argc != 4) {
   cerr << "incorrect␣arg␣count\n";
   return 1;
}
double r;
long long N;

FILE *input = fopen(argv[2], "rt");
if (!input) {
   cerr << "can't␣open␣input␣file\n";
   return 1;
}
if (fscanf(input, "%lf␣%lld\n", &r, &N) != 2 || !feof(input)) {
   cerr << "incorrect␣format\n";
   return 1;
}
assert(r > 0);
assert(N > 0);
```

Здесь происходит проверка на корректность введённых аргументов.
Рассмотрим дальнейший блок `if`

```
int threads = atoi(argv[1]);
long long sum_hit = 0;
double time_start;
if (threads > 0 && threads <= omp_get_max_threads()) {
```

```
   omp_set_num_threads(threads);
#pragma omp parallel
   {
      ...
   }
} else if (threads == 0) {
#pragma omp parallel
   {
      ...
   }
} else if (threads == -1) {
   ...
} else {
   cerr << "incorrect param threads (argv[1])\n";
   return 1;
}
```

Мы получаем входной параметр `threads`.

- `threads` $\in \{1, 2, \ldots,$ `max_threads`$\}$ — выставляем соответствующее значение потока и запускаем `#pragma omp parallel`.
- `threads` $= 0$ — значение потока выставляется автоматически `#pragma omp parallel`.
- `threads` $= -1$ — программа работает последовательно в один поток.
- `threads` $\notin \{-1, 0, 1, \ldots,$ `max_threads`$\}$ — завершаем работу программы

Рассмотрим внутренность первого блока `if`

```
omp_set_num_threads(threads);
time_start = omp_get_wtime();
#pragma omp parallel
{
   unsigned int rnd = (time(NULL) * 1103515245 + omp_get_thread_num() * 12345);
   long long hit = 0;
   double x;
   double y;
#pragma omp for schedule(static)
   for (long long i = 0; i < N; i++) {
      rnd = rnd * 1103515245 + 12345;
      x = (2 * (double) (rnd) / (double) 4294967295 - 1) * r;
      rnd = rnd * 1103515245 + 12345;
      y = (2 * (double) (rnd) / (double) 4294967295 - 1) * r;
      hit += x * x + y * y < r * r;
   }
#pragma omp atomic
   sum_hit += hit;
}
```

Мы выставляем количество потоков из входных данных, засекаем время. Внутри параллельного блока каждому потоку мы сопоставляем переменные `rnd, hit, x, y`. После объявления переменных мы создаём блок `for`, в котором реализован сам метод Монте-Карло подсчёта площади окружности. В этом методе мы случайно выбираем `N` точек в квадрате `r` × `r`.

Метод работает так: Долю точек, расстояния которых до точки (0,0) меньше чем `r` мы делим на все точки. По закону больших чисел чем больше будет `N`, тем ближе будет получившееся число к π. Чтобы получить площадь нужно получившееся отношение умножить на r^2

- `rnd` — отвечает за псевдо-случайную переменную. Каждый шаг, когда нам нужно будет вычислить рандомное значение, мы будем воспринимать `rnd` как начальные данные в линейном конгруэнтным генераторе с параметрами 1103515245 и 12345 (строчки 11 и 13).

- `hit` — отвечает за количество попаданий во внутреннюю часть круга. Обновляется для каждого потока отдельно (строчка 15). После выполнения цикла значение в данной переменной добавляется к значению общей переменной `sum_hit` (строчка 18).

- `x, y` — отвечают за координаты псевдо-случайной точки. Случайные числа от `-r` до `r`. Реализация в 12, 14 строчках. Константа 4294967295 это максимальное значение переменной типа `unsigned int`

Рассмотрим завершающий блок программы.

```
1 double time = omp_get_wtime() - time_start;
2 FILE *output = fopen(argv[3], "wt");
3 if (!output) {
4    cerr << "can't␣open␣output␣file\n";
5    return 1;
6 }
7 fprintf(output, "%lf\n", 4 * r * r * (double) sum_hit / (double) N);
8 fclose(output);
9 printf("Time␣(%i␣thread(s)):␣%g␣ms\n", threads, time * 1000);
```

В нём программа замеряет время после завершения параллельного блока, открывает файл для записи ответа, записывает данные в файл, консоль и

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

Процессор	1,4 GHz 4-ядерный процессор Intel Core i5
Файл `input.txt`:	`1 1000000`
Запуск программы:	`lab4 8 test.txt output.txt`
Вывод в консоль:	`Time (8 thread(s)): 1.523 ms`
Файл `output.txt`:	`3.141404`

Экспериментальная часть

В репозитории лежит ещё 4 `"*.csv"` файла, в которых содержатся сырые данные для графиков собранные с работы программы и скрипт, написанный на языке Python, который обрабатывает эти сырые данные, усредняя значения по общим параметрам.

```python
import pandas as pd

dynamic = pd.read_csv("dynamic.csv")
static = pd.read_csv("static.csv")
schedule_stat = pd.read_csv("schedule_stat.csv")
schedule_dyn = pd.read_csv("schedule_dyn.csv")

print("dynamic")
for i in range(-1, 9):
    df_filter = (dynamic["threads"] == i)
    print(i, dynamic[df_filter]["time"].mean())

print()
print("static")
for i in range(-1, 9):
    df_filter = (static["threads"] == i)
    print(i, static[df_filter]["time"].mean())

print()
print("schedule␣static")
for i in range(1, 10):
    df_filter = (schedule_stat["chunk_size"] == i)
    print(i, schedule_stat[df_filter]["time"].mean())
for i in range(10, 100, 10):
    df_filter = (schedule_stat["chunk_size"] == i)
    print(i, schedule_stat[df_filter]["time"].mean())
for i in range(100, 1001, 100):
    df_filter = (schedule_stat["chunk_size"] == i)
    print(i, schedule_stat[df_filter]["time"].mean())

print()
print("schedule␣dynamic")
for i in range(1, 10):
    df_filter = (schedule_dyn["chunk_size"] == i)
    print(i, schedule_dyn[df_filter]["time"].mean())
for i in range(10, 100, 10):
    df_filter = (schedule_dyn["chunk_size"] == i)
    print(i, schedule_dyn[df_filter]["time"].mean())
for i in range(100, 1001, 100):
    df_filter = (schedule_dyn["chunk_size"] == i)
    print(i, schedule_dyn[df_filter]["time"].mean())
```

Обе таблицы являются визуализацией результат работы скрипта.

threads	dynamic, сек	static, сек
-1	0.3616	0.3616
0	1.6541	0.3546
1	1.3585	0.3543
2	2.2190	0.1787
3	2.0853	0.1215
4	2.1091	0.0951
5	2.0057	0.0942
6	2.0773	0.1056
7	2.0528	0.0870
8	2.0272	0.0857

chunk_size	dynamic, сек	static, сек
1	2.0386	0.1067
2	1.0289	0.1013
3	0.7061	0.0962
4	0.5285	0.0964
5	0.4467	0.1048
6	0.3957	0.1087
7	0.3427	0.1065
8	0.3003	0.1059
9	0.2666	0.1034
10	0.2054	0.0907
20	0.1223	0.0843
30	0.1063	0.0861
40	0.1118	0.0914
50	0.1076	0.0964
60	0.1062	0.0974
70	0.1095	0.0971
80	0.1102	0.0975
90	0.1080	0.0979
100	0.0886	0.0854
200	0.0857	0.0850
300	0.0846	0.0850
400	0.0867	0.0856
500	0.0956	0.0945
600	0.0961	0.0965
700	0.0962	0.0986
800	0.0960	0.0975
900	0.0956	0.0959
1000	0.0868	0.0854

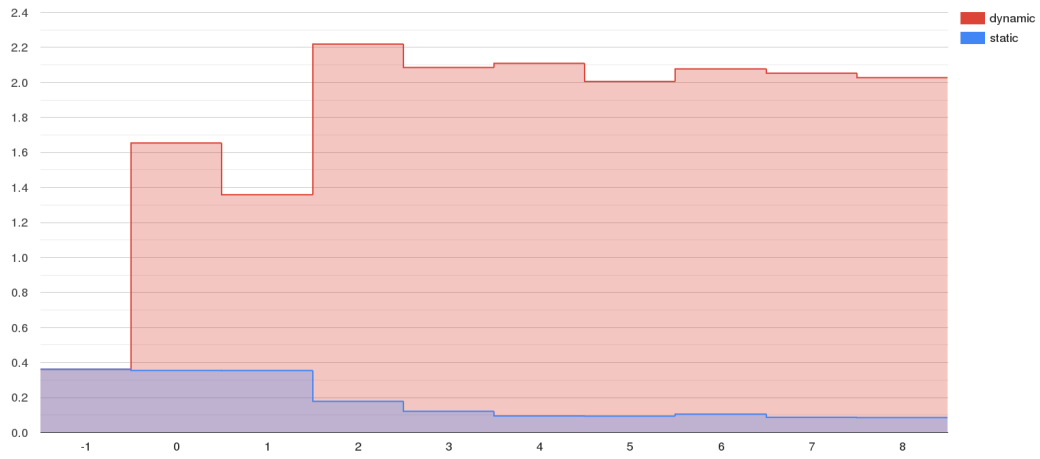

Рис. 1. threads

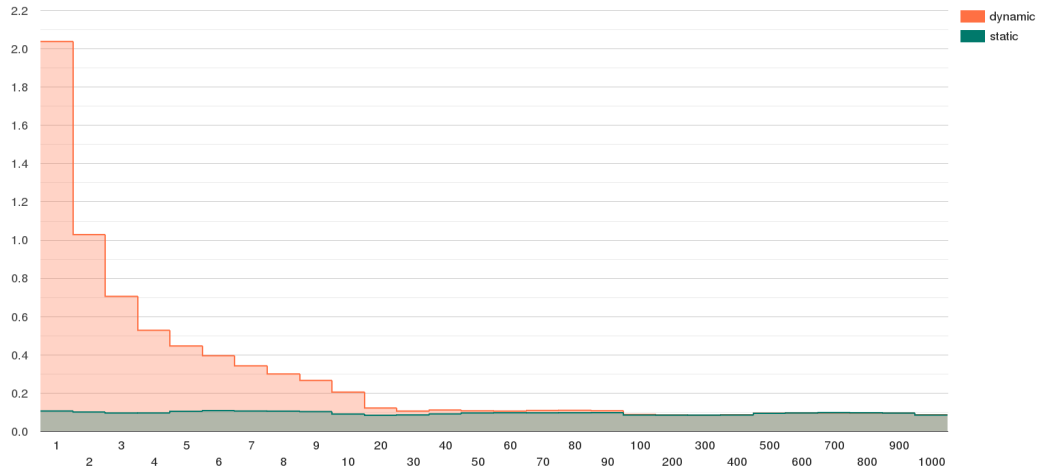

Рис. 2. schedule static

График 1 соответствует первой табличке данных График 2 соответствует второй табличке данных.

На основе этих данных мы можем сделать вывод, что `schedule(dynamic)` медленно работает без параметра `chunk_size`, так как очень много времени уходит на выбор потока, который завершился первым.

Но при большом параметре `chunk_size` `schedule(dynamic)` обгоняет даже `schedule(static)`.

Список источников

Спецификация openmp
Построение графиков
Метод Монте-Карло

Листинг кода

Листинг 1. easy.cpp

```
#include <iostream>
#include <string>
#include <cstdio>
#include <ctime>
#include <cstddef>
#include <omp.h>
#include <cassert>

using namespace std;

int main(int argc, char **argv) {
   if (argc != 4) {
      cerr << "incorrect arg count\n";
      return 1;
   }
   double r;
   long long N;

   FILE *input = fopen(argv[2], "rt");
   if (!input) {
      cerr << "can't open input file\n";
      return 1;
   }
   if (fscanf(input, "%lf %lld\n", &r, &N) != 2 || !feof(input)) {
      cerr << "incorrect format\n";
      return 1;
   }
   assert(r > 0);
   assert(N > 0);

   int threads = atoi(argv[1]);
   long long sum_hit = 0;
   double time_start;
   if (threads > 0 && threads <= omp_get_max_threads()) {
      omp_set_num_threads(threads);
      time_start = omp_get_wtime();
#pragma omp parallel
      {
         unsigned int rnd = (time(NULL) * 1103515245 + omp_get_thread_num() *
    12345);
         long long hit = 0;
```

```
41          double x;
42          double y;
43 #pragma omp for schedule(static)
44          for (long long i = 0; i < N; i++) {
45             rnd = rnd * 1103515245 + 12345;
46             x = (2 * (double) (rnd) / (double) 4294967295 - 1) * r;
47             rnd = rnd * 1103515245 + 12345;
48             y = (2 * (double) (rnd) / (double) 4294967295 - 1) * r;
49             hit += x * x + y * y < r * r;
50          }
51 #pragma omp atomic
52          sum_hit += hit;
53       }
54    } else if (threads == 0) {
55       time_start = omp_get_wtime();
56 #pragma omp parallel
57       {
58          unsigned int rnd = (time(NULL) * 1103515245);
59          long long hit = 0;
60          double x;
61          double y;
62 #pragma omp for
63          for (long long i = 0; i < N; i++) {
64             rnd = rnd * 1103515245 + 12345;
65             x = (2 * (double) (rnd) / (double) 4294967295 - 1) * r;
66             rnd = rnd * 1103515245 + 12345;
67             y = (2 * (double) (rnd) / (double) 4294967295 - 1) * r;
68             hit += x * x + y * y < r * r;
69          }
70 #pragma omp atomic
71          sum_hit += hit;
72       }
73    } else if (threads == -1) {
74       time_start = omp_get_wtime();
75       unsigned int rnd = (time(NULL) * 1103515245);
76       long long hit = 0;
77       double x;
78       double y;
79       for (long long i = 0; i < N; i++) {
80          rnd = rnd * 1103515245 + 12345;
81          x = (2 * (double) (rnd) / (double) 4294967295 - 1) * r;
82          rnd = rnd * 1103515245 + 12345;
83          y = (2 * (double) (rnd) / (double) 4294967295 - 1) * r;
84          hit += x * x + y * y < r * r;
85       }
86       sum_hit += hit;
87    } else {
88       cerr << "incorrect␣param␣threads␣(argv[1])\n";
89       return 1;
90    }
91    double time = omp_get_wtime() - time_start;
92    FILE *output = fopen(argv[3], "wt");
```

```
93      if (!output) {
94          cerr << "can't open output file\n";
95          return 1;
96      }
97      fprintf(output, "%lf\n", 4 * r * r * (double) sum_hit / (double) N);
98      fclose(output);
99      printf("Time (%i thread(s)): %g ms\n", threads, time * 1000);
100     return 0;
101 }
```