浙江大学实验报告

专业：计算机科学与技术
姓名：吴同
学号：3170104848
日期：2020 年 4 月 1 日

课程名称：并行计算与多核编程　指导老师：楼学庆　电子邮件：wutongcs@zju.edu.cn
实验名称：OpenMP 编程　实验类型：综合型　联系电话：18888922355

一、 实验目的

- 利用 OpenMP 编程计算圆周率
- 比较不同线程下的程序性能

二、 实验原理

1. 蒙特卡洛方法计算圆周率

蒙特卡洛方法是一种基于概率论的计算圆周率的方法。其理论基础为：以 $(0,0),(0,1),(1,0),(1,1)$ 四个点构成的正方形中随机投入粒子，粒子落在位于第一象限的四分之一单位圆内的概率为 $\frac{\pi}{4}$，即部分单位圆与正方形的面积比。由此，当进行足够多次的随机试验后，可认为粒子落入四分之一单位圆内的频率即为概率，由此即可求得 π 的近似值。

2. OpenMP

OpenMP 是由一系列 #paragma 指令组成，这些指令控制程序如何以多线程运行。所有的 OpenMP 指令都是以 #pragma omp 开头，换行符结束。几乎所有的的指令都只与指令后面的那段代码相关。

OpenMP 中的 for 指令用于告诉编译器，拆分接下来的 for 循环，并分别在不同的线程中运行不同的部分。num_threads 指令来指定线程数，可以指定该代码块用几个线程来运行。

reduction 指令用于在线程汇合到主线程时，将各个线程中的一个变量汇合到一起，其语法是：reduction(operator:list)。

三、 实验过程

首先调用 omp_get_max_threads() 函数，获取该计算机系统中最多可使用的线程数。接下来从 1 个线程开始，进行不同线程数下的蒙特卡洛方法圆周率计算。

在每次计算开始前，分别调用 clock() 函数和 omp_get_wtime() 函数记录起始时间，在每次计算结束后，再调用这两个函数计算结束时间。通过将两个时间作差，即可得出程序所用时间。两种方法所不同的是，调用 clock() 函数将得到的是 CPU 总共的耗时，为多个线程所用 CPU 用时之和。调用 omp_get_wtime() 函数记录的时间则为程序的实际用时。将二者比较，即可得出多线程的加速比。

在每次求圆周率的计算中，生成两个 0-1 之间的随机数，如果以这两个数构成的坐标在四分之一的单位圆内，则计数器加一。迭代一亿次后，计算落在四分之一圆内的坐标占正方形的面积比，即可求出圆周率。

程序的代码如下：

```
#include <iostream>
#include <random>
#include <ctime>
#include <omp.h>

const long long iteration = 100000000;

int main()
{
   std::default_random_engine e;
   std::uniform_real_distribution<double> u(0.0, 1.0);

   int maxThreads = omp_get_max_threads();
   for (int i = 1; i <= maxThreads; ++i)
   {
      long long count = 0;
      clock_t startClock = clock();
      double startWall = omp_get_wtime();
      std::cout<<"Number of threads: "<<i<<'\n';

      #pragma omp parallel for reduction(+:count) num_threads(i)
      for (int i = 0 ; i < iteration; ++i)
      {
         double x = u(e);
         double y = u(e);
         if (x * x + y * y < 1)
         {
            ++count;
         }
      }
      double endWall = omp_get_wtime();
      clock_t endClock = clock();
      std::cout<<"Time on clock: "<<(double)(endClock - startClock) /
          CLOCKS_PER_SEC<<'\n';
      std::cout<<"Time on wall : "<<endWall - startWall<<'\n';
      double pi = 4 * double(count) / double(iteration);
      std::cout<<"PI = "<<pi<<'\n';
   }
}
```

四、 实验结果

测试环境:

- CPU：Intel(R) Core(TM) i5-7267U CPU @ 3.10GHz
- 操作系统：macOS Catalina 10.15.4

程序运行在双核 CPU 上，线程数分别为 1、2、3、4，得到测试结果如下：

运行时间 (s) 线程数 / 数据规模	1	2	3	4
程序实际用时	2.52591	1.27926	0.993198	0.861137
线程用时之和	2.42318	2.45026	2.70444	2.81751
加速比	0.959330	1.91537	2.72296	3.27185

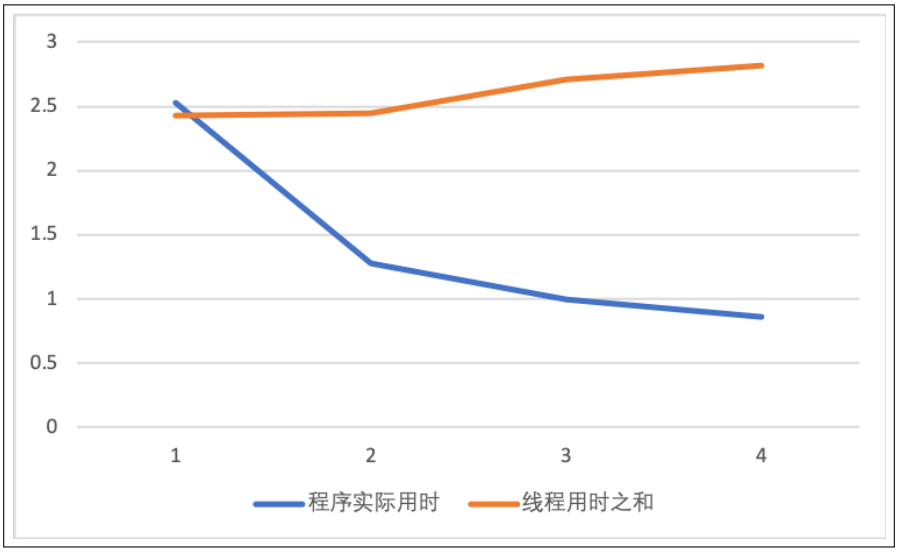

图 1: 程序用时随线程数变化图

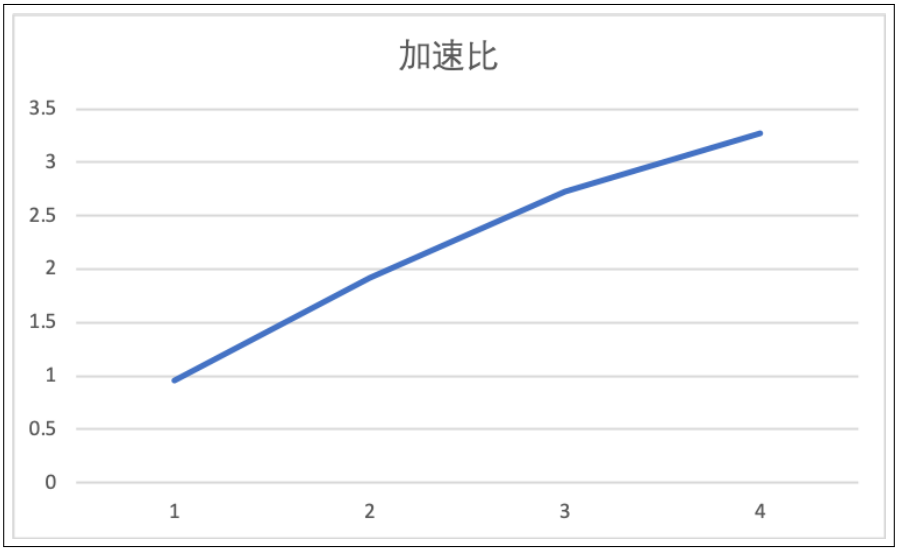

图 2: 加速比随线程数变化图

五、 分析讨论

由测试数据可知，当所使用的线程数增加时，程序的实际用时逐渐降低，各个线程的用时之和略有增加。分析加速比的变化，当线程数为 1、2、3 时，加速比随线程数线性增长，但三线程和四线程之间加速比增长缓慢，这是由于测试所用的 CPU 是双核四线程的，程序无法真正在四线程上同时运行。从线程总用时随线程数增长也可看出，线程切换过程存在一些开销。但由于不同线程能够在相当多的时间内同时运行在 CPU 上，所以多线程仍可体现出明显的加速效果。

与直接使用线程库相比，使用 OpenMP 进行多线程编程简洁、优雅得多。