操作系统实验六报告

智能科学与技术　　2024 年 5 月 17 日

实验任务

使用 C 语言和 pthread 线程库实现一个多线程的采用蒙特卡洛随机采样算法计算 值的程序。具体要求：

1）程序需要支持命令行参数-t，用于指定程序运行时创建多少个线程用于计算；

2）了解 GCC 标准库提供的随机函数的实现，分析其如何保证多线程安全，以及对多线程程序性能的影响；

3）需结合自身硬件环境（如 CPU 核心数），分析同等总采样次数的情况下，不同的线程数，程序的运行时间开销，并分析原因。

1　实验环境

1. 电脑操作系统：Windows 11 + 阿里云 Alibaba Cloud Linux（基于 centos）+wsl（Ubuntu）

2. 编程语言：C 语言

3. 辅助工具：

（1）使用 VScode 用于编辑各种文件。

（2）使用 Xshell 7 远程连接阿里云服务器，构建 Linux 系统环境，方便使用各种转换与调试工具（gcc），使用 Xftp7 进行本机与服务器的文件传输。

（4）使用 makefile 组织和管理整个实验项目的编译、部署和运行

2　实验原理

根据实验任务中的三个要求，针对性进行学习与查询资料，得到任务相关的知识与思路：

2.1　蒙特卡洛随机采样算法

蒙特卡洛方法是一种通过随机抽样来进行数值计算的方法，其基本思想是利用大量随机样本来近似计算目标值。在计算 π 值时，蒙特卡洛方法可以通过在一个正方形内部随机投点，并统计落在四分之一圆内的点的比例来估算 π 值。

算法步骤：

1. 生成随机点：首先，在一个正方形内生成大量的随机点。这些随机点的生成可以利用计算机的伪随机数生成器来实现。

2. 判断点的位置：对于每个生成的随机点，判断其是否落在四分之一圆内。可以通过检查点到原点的距离是否小于等于半径来进行判断，这里半径为正方形边长。

3. 统计落在四分之一圆内的点的数量：对于落在四分之一圆内的点，统计其数量。

4. 计算 π 值：根据四分之一圆的面积与正方形的面积的比值，即落在四分之一圆内的点数量与总随机点数量的比值，来估算 π 值。

由于四分之一圆的面积为 $\pi/4$，而正方形的面积为 1，所以 π 的估算值可以通过以下公式计算：

$\pi \approx$4*(落在四分之一圆内的点数量)/(总随机点数量)

2.2 使用 C 语言和 pthread 线程库实现多线程计算 π 值程序

在 C 语言中，可以使用 pthread 线程库来实现多线程编程。每个线程负责执行部分的随机点生成和统计工作，最后通过主线程将各个线程的结果进行合并，得到最终的 π 值估算。

2.3 设计思路

1. 命令行参数-t 的支持：

通过解析命令行参数获取用户指定的线程数量。在程序中创建相应数量的线程来进行 π 值的计算。

2. 多线程安全的随机函数：

GCC 标准库提供的随机函数一般是线程安全的。然而，在多线程环境下，随机函数的性能可能会受到影响，可能会导致竞争和性能下降。本实验在探索过程中分别使用了 srand()+rand() 和 seed+rand_r() 两种随机函数进行实验。实验结果表明 seed+rand_r() 是正确且高效的方法，后面实验过程会以 rand_r() 为例进行实验，后续在“实验中遇到的困难与解决”部分会进行两种的随机函数的对比和原因分析。

3. 性能分析：

对于同等总采样次数的情况下，不同的线程数可能会影响程序的运行时间开销。在硬件环境允许的情况下，增加线程数量可以加快计算速度，但是线程数量过多也可能会导致线程切换开销增加，反而影响性能。根据硬件环境的 CPU 核心数等因素，选择适当的线程数量可以最大程度地利用系统资源，提高程序的运行效率。

3 实验过程

根据实验原理，我们将实验分成两部分，编写 C 语言程序以及分析性能。其中 C 语言程序可由数据段，主函数和线程函数三部分组成。

3.1 数据段

这里定义了几个重要的全局变量和一个互斥锁：

1. ‘TOTAL_POINTS‘：这是一个预定义的宏，表示进行蒙特卡洛采样的总采样点数。在这个程序中，每个线程将会在一个单位正方形内随机生成这么多个点，并统计其中落在四分之一圆内的点的数量。

2. ‘total_points_inside_circle‘：这是一个整型变量，用于记录所有线程生成的点中落在四分之一圆内的点的总数量。每个线程在完成采样后会将其落在圆内的点的数量累加到这个变量中。

3. ‘pthread_mutex_t lock‘：这是一个互斥锁，用于保护对 ‘total_points_inside_circle‘变量的访问。由于多个线程会同时访问和修改这个全局变量，为了防止竞态条件和数据不一致的情况发生，需要在对这个变量的访问过程中使用互斥锁进行同步。

```
1 #define TOTAL_POINTS 100000000 // 总采样点数
2 int total_points_inside_circle = 0;
3 pthread_mutex_t lock;
```

3.2 主函数

主函数 main 中主要执行的任务为：

1. 解析命令行参数，获取线程数
2. 创建并启动多个线程进行蒙特卡洛采样，并等待所有线程完成
3. 最后根据采样结果计算 π 的估计值

以下初始化互斥锁，创建线程，等待线程结束是多线程的经典操作，不再赘述。

```c
int main(int argc, char *argv[]) {
    int num_threads = 1;

    // 解析命令行参数，获取线程数
    if (argc > 1) {
        num_threads = atoi(argv[1]);
    }

    // 检查线程数是否合法
    if (num_threads <= 0) {
        printf("Invalid number of threads. Exiting.\n");
        return 1;
    }

    int points_per_thread = TOTAL_POINTS / num_threads; // 每个线程的采样点数

    pthread_t threads[num_threads];
    pthread_mutex_init(&lock, NULL); // 初始化互斥锁

    // 创建并启动多个线程进行蒙特卡洛采样
    for (int i = 0; i < num_threads; i++) {
        pthread_create(&threads[i], NULL, monte_carlo, &points_per_thread);
    }

    // 等待所有线程完成
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&lock); // 销毁互斥锁

    // 根据采样结果计算的估计值
    double pi_estimate = 4.0 * total_points_inside_circle / TOTAL_POINTS;
    printf("Estimated value of pi: %f\n", pi_estimate);

    return 0;
}
```

3.3 线程函数

这个函数是一个线程函数，主要用于执行蒙特卡洛随机采样的任务。具体来说，这个函数完成了以下操作：

1. 随机采样：根据传入的参数 ‘thread_points‘，确定每个线程需要进行的随机采样点数。然后，使用当前时间作为种子，利用 ‘rand_r‘函数生成随机数，并计算每个点到原点的距离的平方，判断其到原点的距离是否小于等于 1。如果是，则表明该点落在了四分之一圆内，将计数器 ‘points_inside_circle‘加 1。

2. 更新总的圆内点数：在循环结束后，将线程内统计到的落在圆内的点数 ‘points_inside_circle‘累加到全局变量 ‘total_points_inside_circle‘中。由于这个变量是全局共享的，为了保证线程安全，在更新时需要先对其进行加锁操作。

需要注意的是，在多线程环境下，这个函数会被多个线程同时调用，因此在更新全局变量时必须进行同步操作，否则可能会导致数据不一致的情况发生。

```
// 线程函数，执行蒙特卡洛随机采样
void *monte_carlo(void *arg) {
    int points_inside_circle = 0;
    unsigned int seed = time(NULL); // 使用当前时间作为种子
    int *thread_points = (int *)arg; // 每个线程的采样点数

    // 随机采样
    for (int i = 0; i < *thread_points; i++) {
        double x = (double)rand_r(&seed) / RAND_MAX; // 生成[0,1]之间的随机数
        double y = (double)rand_r(&seed) / RAND_MAX;
        double distance = x * x + y * y; // 计算点到原点的距离的平方
        if (distance <= 1) { // 若距离小于等于1，则点在圆内
            points_inside_circle++;
        }
    }

    // 更新总的圆内点数，需要加锁
    pthread_mutex_lock(&lock);
    total_points_inside_circle += points_inside_circle;
    pthread_mutex_unlock(&lock);

    pthread_exit(NULL); // 退出线程
}
```

3.4 分析性能

使用 gcc 进行编译链接，注意需要添加-pthread 参数。然后使用 time 运行可执行程序，就会输出计算得 π 的结果以及运行的时间。相关命令行：

```
gcc -o main main.c -pthread
time ./main <threads_nums>
```

4 实验结果

不断修改-t 参数（可以使用 makefile 方便实验的进行），可得以下输出（Figure 1），其中 Estimated value of pi 为计算得到的 π 值，Real 指的是总运行时间，包括实际的执行时间和系统消耗的时间。User 表示用户 CPU 时间，即程序在用户态运行的时间。System 表示系统 CPU 时间，即程序在内核态运行的时间。

```
ling@LAPTOP-OID8VMCU:/mnt/c/Users/ling xiaoli/linux$ make do
gcc -o main main.c -pthread
time -f "Real: %E\nSystem: %S\nUser: %U" ./main 1
Numbers of thread: 1
Estimated value of pi: 3.141513
Real: 0:00.96
System: 0.00
User: 0.95
time -f "Real: %E\nSystem: %S\nUser: %U" ./main 2
Numbers of thread: 2
Estimated value of pi: 3.141669
Real: 0:00.50
System: 0.00
User: 0.99
time -f "Real: %E\nSystem: %S\nUser: %U" ./main 3
Numbers of thread: 3
Estimated value of pi: 3.141943
Real: 0:00.34
System: 0.00
User: 1.00
```

Figure 1: 运行结果

接下来，记录 <threads_nums> 从 1 到 18 的值的实验结果，绘制成折线图（Figure 2）研究分析同等总采样次数的情况下，不同的线程数，程序的运行时间开销。

由图像 Figure 2 可以看出，随着线程数的增加，实际运行总时间 RealTime 逐步下降，这是因为多线程可以实现并行计算，将工作分配给多个线程来执行，从而加快了程序的执行速度。但是随着线程数的增加，下降的幅度逐步减小，在线程数达到 16 时运行时间不降反升（为了更直观地显示这一变化，对 8-15 的数据进行了平滑处理），随后在一个值附近震荡。通过查阅资料，推测这是因为当线程数增加时，线程间的竞争和调度开销可能会增加，甚至可能会出现线程间的争用和资源竞争，导致性能反而下降。

通过下面指令，可以查询得到本机的逻辑 CPU 数目（processor 数）为 16，经过查阅资料，关于具体在多线程程序中设置线程数多大的问题，对计算密集型的程序有的给出的建议是 processor count + 1，有的建议是 processor count 的 1.5 倍，都是实验过程中积累的经验值，但仍以针对各自的 CPU 进行实测为准。在该实验过程中，在线程数达到 16 时运行时间不降反升，大致符合以上查询信息。(下面 Figure 3 的 2-核 CPU 在线程数达到 3 左右就开始稳定震荡于一个值，也佐证了以上结论的有效性)

```
cat /proc/cpuinfo | grep "processor" | wc -l
```

同时，本实验还研究了在 2-核 CPU 上进行实验的结果，并将 16-核 CPU 的实验结果绘制在一起进行比较（Figure 3）。由 Figure 3 可以看出 16-核 CPU 的充分优势，说明即使线程数相同，不同核心的 CPU 上运行相同任务的运行时间也可能会有差异，这取决于诸多因素，包括核心性能、负载情况、内存访问和调度器的影响等。

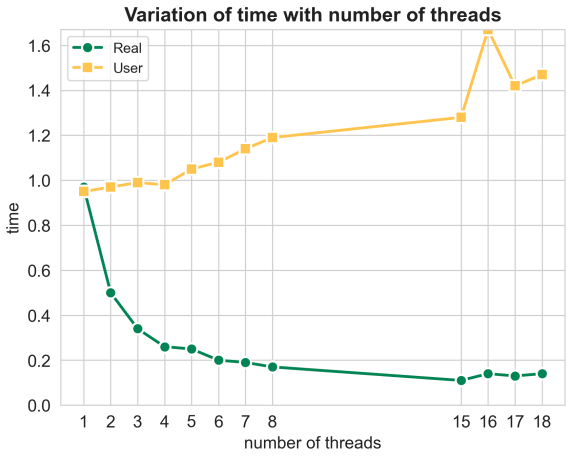

Figure 2: 不同的线程数的运行时间开销

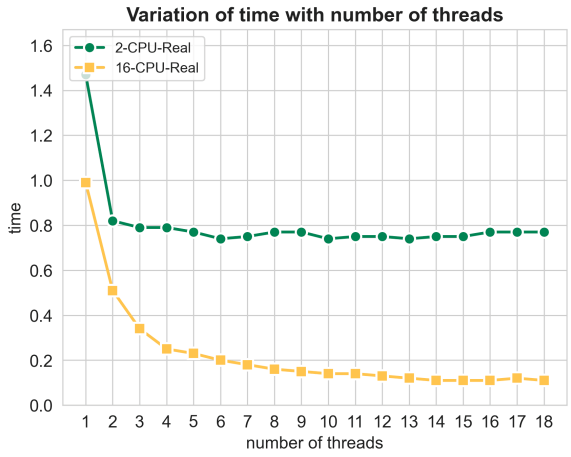

Figure 3: 2 核-CPU 与 16-核 CPU 运行时间对比

5 实验中的困难与解决

在实验刚开始的时候，在线程函数中随机采样时使用的是 srand(time(NULL))+rand() 的组合：

```
void *monte_carlo(void *arg) {
    ...
    double x = (double)rand() / RAND_MAX;
    double y = (double)rand() / RAND_MAX;
    ...
}
int main(int argc, char ** argv)
{
    ...
    srand(time(NULL));
    ...
}
```

但是发现使用多线程进行实验反而要比单线程要慢很多，输出如 Figure 4 所示，但线程数为 2 时比线程数为 1 时运行时间多了好几倍。

```
ling@LAPTOP-OID8VMCU:/mnt/c/Users/ling xiaoli/linux$ time ./main 1
Numbers of thread: 1
Estimated value of pi: 3.141533

real    0m2.525s
user    0m2.511s
sys     0m0.000s
ling@LAPTOP-OID8VMCU:/mnt/c/Users/ling xiaoli/linux$ time ./main 2
Numbers of thread: 2
Estimated value of pi: 3.142087

real    0m20.000s
user    0m19.340s
sys     0m19.748s
```

Figure 4: 使用 rand() 多线程比单线程要慢

通过查阅官方文档和相关资料，得知了出现该情况的原因：计算机的随机数都是由伪随机数，即是

由小 M 多项式序列生成的，其中产生每个小序列都有一个初始值，即随机种子。使用 srand() 和 rand() 组合时，进程中只生成一个随机数生成器 (种子)，所有线程共享该生成器。每次调用 rand() 都会修改生成器的一些参数，比如当前种子的值。在单线程环境下，这是可行的。但在多线程环境下，可能会导致临界区问题。为了解决这个问题，可能会采用互斥量等方法来解决临界区问题。如果 rand() 的调用次数不多，多线程环境下问题不大。但是在蒙特卡洛法估算 π 的程序中，需要频繁调用 rand()，这可能导致每个线程频繁地对临界区进行上锁和解锁。当临界区被锁定时，其他线程无法完成 rand() 调用，从而被阻塞。这样会导致效率大幅降低，甚至可能出现使用多线程反而导致程序运行更慢的情况。

在调查原因的过程中，发现了解决的方案：rand_r 函数。rand_r 是 C 标准库提供的线程安全的随机数生成函数，其实现方式如下：

```
unsigned int rand_r(unsigned int *seed) {
    *seed = (*seed * 1103515245 + 12345) & 0x7fffffff;
    return *seed;
}
```

这个函数的参数 seed 是一个指向无符号整数的指针，用于存储随机数生成器的种子。rand_r 函数根据传入的种子生成一个伪随机数，并更新种子的值，以便下次调用时生成不同的随机数。结合官方文档的代码和对 rand() 的对比与分析，可以得到以下对 rand_r() 可行性的分析：

1. 保证多线程安全的机制：

（1）每个线程拥有自己的种子：rand_r 函数的种子参数是一个指针，每个线程都可以拥有自己的种子。这样，不同线程之间生成的随机数序列是相互独立的，不会相互影响。

（2）避免竞态条件：由于每个线程都有自己的种子，因此不会出现多个线程同时修改同一个种子的情况。这样可以避免竞态条件，保证了函数的线程安全性。

2. 对多线程程序性能的影响：

（1）性能表现：由于 rand_r 函数是线程安全的，因此在多线程环境下可以安全地并发调用。相比于 rand 函数，rand_r 函数在多线程环境下的性能通常更好，因为它每个线程生成了它自己独有的随机数生成器，不会有临界段问题，就不需要额外的同步开销来保证线程安全。

（2）并发度：由于每个线程都有自己的种子，并且生成随机数的过程是独立的，因此可以实现较高的并发度。多个线程可以同时调用 rand_r 函数生成随机数，不会相互阻塞，从而提高了程序的并行性能。

以上分析了 rand() 以及 rand_r() 如何保证多线程安全，以及对多线程程序性能的影响。并通过对比实验结果，可以得到 rand_r() 方法的有效性。

6 参考文章

1. rand() 和 rand_r() 的区别
2. cpu 核心数与线程数
3. 从伪随机数的产生到高大上的蒙特卡洛算法（C 语言实现）