并行程序设计与算法实验 4

实验	**Pthread实现蒙特卡洛求π**	专业	计算机科学与技术
学号	21311525	姓名	马梓培
Email	mazp@mail2.sysu.edu.cn	完成日期	2024/04/15

1. 实验要求

- 使用Pthread创建多线程，并行生成正方形内的n个随机点
- 统计落在正方形内切圆内点数，估算π的值
- 设置线程数量（1-16）及随机点数量（1024-65536）
- 分析近似精度及程序并行性能

2. 蒙特卡洛近似求π原理

借鉴自知乎.

蒙特卡洛模拟是最简单的随机抽样方式。随机抽样是一种严重依赖随机性来解决原则上确定性问题的计算算法。**蒙特卡罗采样的艺术在于我们可以用它来解决一些其他方法难以解决的数学问题。**

模拟的工作原理是在$r = 1$的平方中生成随机点。同时这个**r**是正方形内四分之一的想象圆的半径。根据点到坐标系中心的距离，可以判断该点是在假想的四分之一圆内还是在假想的四分之一圆外。这是蒙特卡罗模拟的切入点。蒙特卡罗模拟将“射击”正方形中的点，同时再想象这些点“射击”在四分之一圆的内部和外部。通过四分之一圆内外的点的比例可以确定**π**。

更正式地说：

> 四分之一的圆面积为$S1 = \frac{r^2}{4}$，正方形的面积为S2=r²
> k和m分别是通过蒙特卡罗模拟，落在四分之一的圆面积S1和正方形的面积为S2的点数。
> $\frac{S1}{S2} = r^2\pi/4/r^2 = \pi = k/m$
> $\pi = \frac{4k}{m}$

3. Pthread蒙特卡洛近似求π

代码详见附件或github.

首先，创建ThreadData结构体，方便在fork的线程中使用：

```
struct ThreadData {
    int thread_id;
    long long num_points;       // 每个线程生成的点数
    long long count_in_circle;  // 落在圆内的点数
};
```

其次，根据thread_int作为seed，随机生成不同的二维点。考虑到srand()函数生成的是[0, 1]的均匀采样，我将其-0.5，得到[-0.5, 0.5]的均匀采样。因此，圆的半径被设置为0.5。即，若生成的点到原点的距离 > 0.5，ThreadData.count_in_circle ++。

```
ThreadData* data = (ThreadData*)arg;
double x, y;
double radius = 0.5;

srand(time(NULL) ^ (unsigned int)data->thread_id);

for (int i = 0; i < data->num_points; i++) {
    x = (double)rand() / RAND_MAX; // 生成0到1之间的随机数
    y = (double)rand() / RAND_MAX;
    if (sqrt((x - 0.5) * (x - 0.5) + (y - 0.5) * (y - 0.5)) <= radius) {
        data->count_in_circle++;
    }
}
```

常规的创建线程，合并线程：

```
for (int i = 0; i < thread_cnt; i++) {
    data[i].thread_id = i;  // 分配线程ID
    data[i].num_points = points_per_thread;
    pthread_create(&threads[i], NULL, generate_points, (void*)&data[i]);
}

long long total_in_circle = 0;
for (int i = 0; i < thread_cnt; i++) {
    pthread_join(threads[i], NULL);
    total_in_circle += data[i].count_in_circle;
}
```

最后，根据公式，计算π:

```
double pi_estimate = 4.0 * total_in_circle / total_points;
```

同样的，计算时间也与上次作业相同：

```
// Barrier
pthread_mutex_lock(&mutex1);
counter++;
if (counter == thread_cnt) {
    counter = 0;
```

```
    pthread_cond_broadcast(&cond_var);
}
else {
    while (pthread_cond_wait(&cond_var, &mutex1) != 0);
}
pthread_mutex_unlock(&mutex1);
clock_gettime(CLOCK_MONOTONIC, &start_time);
...
clock_gettime(CLOCK_MONOTONIC, &end_time);
elapsed_time = (end_time.tv_sec - start_time.tv_sec);
```

4. 实验结果

N = 1024

```
❯ ./pi.out 1
Cost time: 0.000047
Estimated Pi: 3.199219
❯ ./pi.out 2
Cost time: 0.000029
Estimated Pi: 3.101562
❯ ./pi.out 4
Cost time: 0.000022
Estimated Pi: 3.058594
❯ ./pi.out 8
Cost time: 0.000012
Estimated Pi: 3.082031
❯ ./pi.out 16
Cost time: 0.000003
Estimated Pi: 3.179688
```

N = 2048

```
❯ ./pi.out 1
Cost time: 0.000061
Estimated Pi: 3.060547
❯ ./pi.out 2
Cost time: 0.000057
Estimated Pi: 3.162109
❯ ./pi.out 4
Cost time: 0.000016
Estimated Pi: 3.199219
❯ ./pi.out 8
Cost time: 0.000046
Estimated Pi: 3.070312
❯ ./pi.out 16
Cost time: 0.000012
Estimated Pi: 3.156250
```

N = 4096

```
❯ ./pi.out 1
Cost time: 0.000187
Estimated Pi: 3.125000
❯ ./pi.out 2
Cost time: 0.000120
Estimated Pi: 3.152344
❯ ./pi.out 4
Cost time: 0.000037
Estimated Pi: 3.118164
❯ ./pi.out 8
Cost time: 0.000095
Estimated Pi: 3.152344
❯ ./pi.out 16
Cost time: 0.000014
Estimated Pi: 3.159180
```

N = 8192

```
❯ ./pi.out 1
Cost time: 0.000301
Estimated Pi: 3.125488
❯ ./pi.out 2
Cost time: 0.000120
Estimated Pi: 3.197754
❯ ./pi.out 4
Cost time: 0.000123
Estimated Pi: 3.118164
❯ ./pi.out 8
Cost time: 0.000114
Estimated Pi: 3.106445
❯ ./pi.out 16
Cost time: 0.000067
Estimated Pi: 3.079590
```

N = 16384

```
❯ ./pi.out 1
Cost time: 0.000497
Estimated Pi: 3.144531
❯ ./pi.out 2
Cost time: 0.000499
Estimated Pi: 3.129150
❯ ./pi.out 4
Cost time: 0.000301
Estimated Pi: 3.131104
❯ ./pi.out 8
Cost time: 0.000056
Estimated Pi: 3.138672
❯ ./pi.out 16
Cost time: 0.000142
Estimated Pi: 3.193604
```

N = 32768

```
❯ ./pi.out 1
Cost time: 0.001188
Estimated Pi: 3.155396
❯ ./pi.out 2
Cost time: 0.000951
Estimated Pi: 3.144287
❯ ./pi.out 4
Cost time: 0.000286
Estimated Pi: 3.125977
❯ ./pi.out 8
Cost time: 0.000456
Estimated Pi: 3.140747
❯ ./pi.out 16
Cost time: 0.000082
Estimated Pi: 3.131226
```

N = 65536

```
❯ ./pi.out 1
Cost time: 0.002153
Estimated Pi: 3.156189
❯ ./pi.out 2
Cost time: 0.000963
Estimated Pi: 3.127136
❯ ./pi.out 4
Cost time: 0.000812
Estimated Pi: 3.141846
❯ ./pi.out 8
Cost time: 0.000563
Estimated Pi: 3.142517
❯ ./pi.out 16
Cost time: 0.000378
Estimated Pi: 3.131592
```

近似精度:

P\N	1024	2048	4096	8192	16384	32768	65536
1	3.199219	3.060547	3.125000	3.125488	3.144531	3.155396	3.156189
2	3.101562	3.162109	3.152344	3.197754	3.129150	3.144287	3.127136
4	3.058594	3.199219	3.118164	3.118164	3.131104	3.125977	3.141846
8	3.082031	3.070312	3.152344	3.106445	3.138672	3.140747	3.142517
16	3.179688	3.156250	3.159180	3.079590	3.193604	3.131226	3.131592

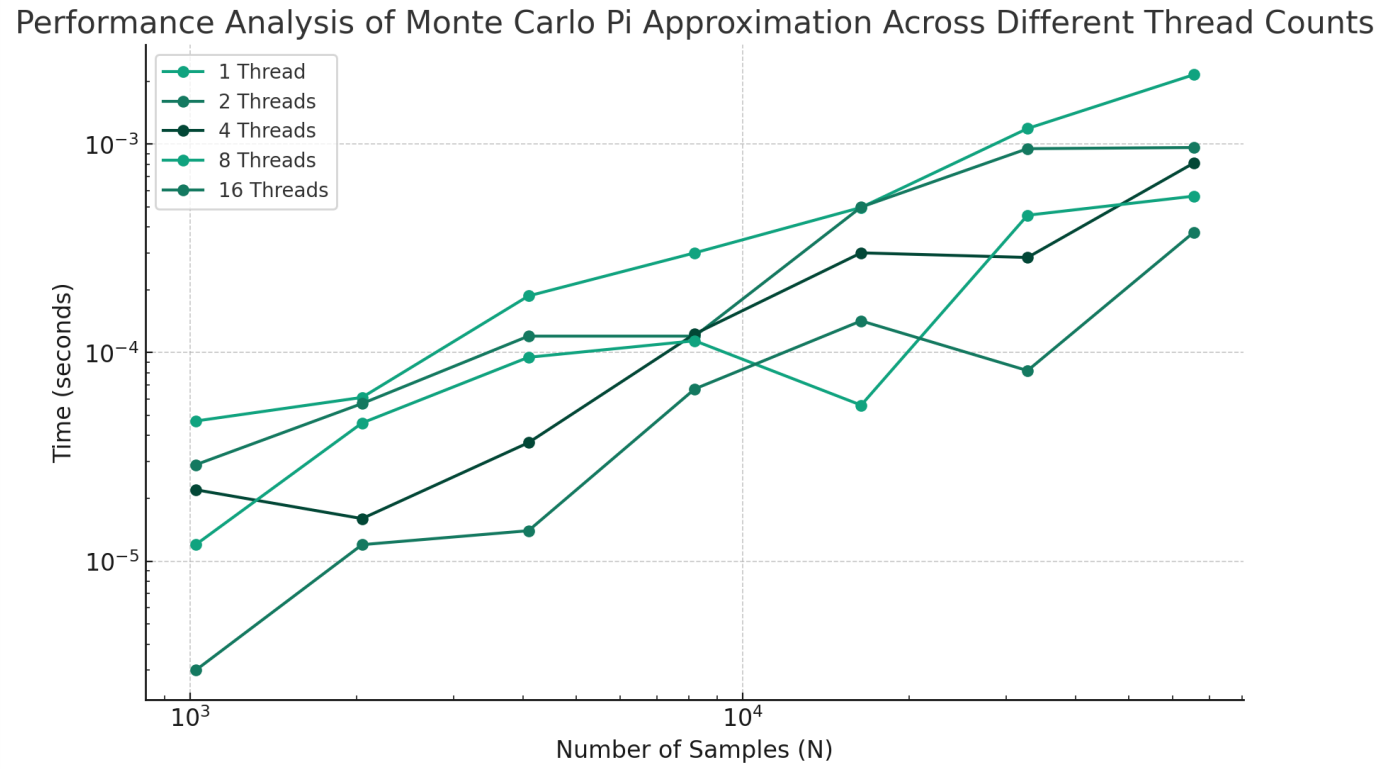
可以看到，随着采样点数N的增加，π的近似值逐渐趋近于其真实值3.14159。例如，在单线程（P=1）下，N从1024增至65536时，π的近似值从3.199219提升至3.156189。这一现象符合蒙特卡洛方法的特性，即随着样本量增加，结果的准确性提高。当然，进程数的多少并不会影响近似的精度。它仅仅是通过并行化，缩短了程序的运行时间。

并行性能：

P\N	1024	2048	4096	8192	16384	32768	65536
1	0.000047	0.000061	0.000187	0.000301	0.000497	0.001188	0.002153
2	0.000029	0.000057	0.000120	0.000120	0.000499	0.000951	0.000963
4	0.000022	0.000016	0.000037	0.000123	0.000301	0.000286	0.000812
8	0.000012	0.000046	0.000095	0.000114	0.000056	0.000456	0.000563
16	0.000003	0.000012	0.000014	0.000067	0.000142	0.000082	0.000378

随着线程数（P）的增加，计算时间普遍减少。这反映了并行计算的优势，即通过增加计算资源（线程）来分担任务，从而减少总体所需时间。例如，在N=65536的情况下，单线程（P=1）的计算时间为0.002153秒，而在16线程（P=16）的情况下减少到0.000378秒。

可视化：



5. 实验感想

在这次的并行程序设计与算法实验中，通过使用Pthread实现蒙特卡洛方法计算π的值不仅加深了我对并行计算理论的理解，也让我体会到了实际操作中的挑战与乐趣。实验中，我设置了不同的线程数量和随机点数，观察到随着采样点数的增加，π的近似值逐渐趋近于真实值，这验证了蒙特卡洛方法的有效性。此外，实验结果也展示了并行计算的强大能力，随着线程数量的增加，程序的运行时间显著减少，这反映了并行计算在处理大规模计算任务时的优势。

通过这次实验，我更加明白了理论与实践相结合的重要性。编写多线程程序时，我需要考虑各种并发问题，如数据访问冲突和资源分配，这些都是在单线程编程中不会遇到的问题。解决这些问题的过程，不仅提升了我的编程技能，也锻炼了我的问题解决能力。