中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab4-Pthreads 并行方程求解与蒙特卡洛

姓 名 李源卿

学 号 22336128

学 院 计算机学院

专 业 计算机科学与技术

2025 年 4 月 16 日

1 实验目的

- 深入理解 Pthreads 同步机制：条件变量、互斥锁
- 评估多线程加速性能

2 实验内容

- 多线程一元二次方程求解
- 多线程圆周率计算

2.1 方程求解

- 使用 Pthread 多线程及求根公式实现并行一元二次方程求解。
- 方程形式为 $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$，其中判别式的计算、求根公式的中间值分别由不同的线程完成。
- 通过条件变量识别何时线程完成了所需计算，并分析计算任务依赖关系及程序并行性能。

2.2 蒙特卡洛求圆周率 pi 的近似值

- 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），观察对近似精度及程序并行性能的影响。

3 实验代码说明

3.1 方程求解

我认为对于求解一元二次方程，使用多线程颇有种杀鸡用牛刀的感觉，但是在写程序的过程中可以深化我们对于条件变量的理解，熟悉条件变量的使用。
与实验文档中的不同，为了能多练习一次使用条件变量，我没有仅仅使用三个线程去完成，而是创建了五个线程，工作流如下：
时间步 1：thread 0: 计算 b^2　　thread 1: 计算 $4 * a * c$
时间步 2：thread 2 : 计算 δ

时间步 3：thread 3: 计算 x_1　thread 4: 计算 x_2

线程 2 需要等待线程 0 和线程 1 的结果，这里就需要用到条件变量，而线程 3 和线程 4 需要等待线程 2 的结果，这里也需要用到条件变量。

为了实现线程 0，1，2 的同步，我设置了一个条件变量，一个互斥量，以及一个计数器：

```
pthread_mutex_t mutex0 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond0 = PTHREAD_COND_INITIALIZER;
int cnt = 0;
```

然后将 cnt 作为临界资源，线程 0 和线程 1 计算好之后就会把 cnt 加一，线程 2 则一直等待 cnt 变成 2：

```
void* calculate_b_2(void* arg){
    b_2 = b * b;
    //计算完结果之后将cnt+1
    pthread_mutex_lock(&mutex0);
    cnt++;
    pthread_mutex_unlock(&mutex0);
    //thread 0和1都计算好了，就唤醒thread 2。
    if(cnt == 2)pthread_cond_broadcast(&cond0);
    return NULL;
}

void* calculate_ac_4(void* arg){
    ac_4 = 4 * a *c;
    //计算完结果之后将cnt+1
    pthread_mutex_lock(&mutex0);
    cnt++;
    pthread_mutex_unlock(&mutex0);
    //thread 0和1都计算好了，就唤醒thread 2。
    if(cnt == 2)pthread_cond_broadcast(&cond0);
    return NULL;
}

void* calculate_delt(void* arg){
    //等待条件满足（thread 0，1的中间结果计算完毕）
    pthread_mutex_lock(&mutex0);
    while (cnt != 2)
    {
        pthread_cond_wait(&cond0, &mutex0);
    }
```

```
    pthread_mutex_unlock(&mutex0);
    //计算delt
    delt=b_2 - ac_4;
    // delt =b * b - 4 * a * c;
    //唤醒线程3和4
    pthread_mutex_lock(&mutex);
    condition = true;
    pthread_mutex_unlock(&mutex);
    pthread_cond_broadcast(&cond);
    return NULL;
}
```

后面同步 thread 3,4,5 的方法也类似，为了简洁一些，就不放在报告里了。

3.2 蒙特卡洛求圆周率 pi 的近似值

大体思路很明确：就是把任务总量 ÷ 线程数，平均分到每个线程，然后每个线程在自己内部统计落在 1/4 圆的点数，最后汇总。我在汇总的时候没有使用 mutex 上锁，而是运用了线程函数返回结果的机制：

```
void* local(void* arg) {
    ThreadArgs *args = (ThreadArgs*) arg;
    int *local_sum = malloc(sizeof(int));
    *local_sum = 0;

    for (int i = 0; i < args->iterations; i++) {
        double x, y;
        drand48_r(&args->rand_state, &x);  // 生成[0,1)的随机数
        drand48_r(&args->rand_state, &y);
        *local_sum += (x*x + y*y) <= 1.0;
    }

    pthread_exit(local_sum);
}
...
// 回收结果
in_cycle_sum = 0;
for (int i = 0; i < num_threads; i++) {
    int *local_sum;
    pthread_join(threads[i], (void**)&local_sum)
    in_cycle_sum += *local_sum;
```

```
22      free(local_sum);
23  }
```

这里的随机数生成器的选取和并行性能有很大关联，这一点在结果分析中会细说。

4 实验结果与分析

4.1 方程求解

- 分析不同线程配置下的求解时间，评估并行化带来的性能提升。
- 对比单线程与多线程方案在处理相同方程时的表现，讨论可能存在的瓶颈或优化空间。

4.1.1 不同线程配置求解时间对比

我另外实现了三个线程进行方程求解，在 my_solve1.c 中。并分别进行了测试，结果如下：
(其中 s 是五个线程的版本，s1 是三个线程的版本)
取 10 次实验结果的平均值后，五线程版本的平均耗时为 0.0003224s，三线程的平均耗

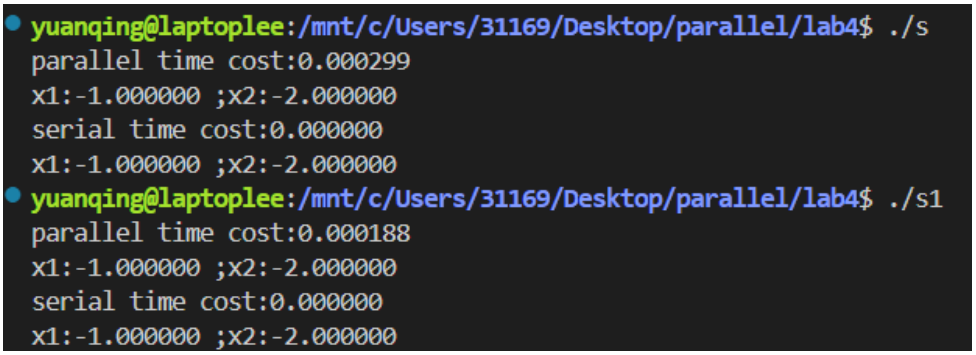

图 1: fig:s-5 threads;s1-3 threads

时为 0.0001936s。而串行版本的耗时则极短，与并行版本的时间消耗不在一个数量级。

4.1.2 可能的瓶颈和优化空间

并行版本慢的原因在于并行计算中间量所节省的时间并无法弥补同步线程的开销。对临界区上锁会使得操作系统进入内核态然后切换至用户态，导致时间开销增大，我认为这是三线程版本慢于串行版本的原因；而五线程慢于三线程版本，除了同步线程的开销更大的原因之外，还有我的电脑是四个物理核的原因，五个线程的一定会涉及到线程的调度，也存在一定的开销。

该问题的优化空间:

- 开根的结果 $\sqrt{\Delta}$ 可以复用（但我尝试了一下发现作用不大）。
- 减少线程数，合并轻量任务：将 calculate_b_2 和 calculate_ac_4 合并到一个线程中，避免两次线程创建和锁操作。
- 由于目前只解一个方程，所以并行性能没有体现出来，当方程数增多时，我们可以让两个线程先并行计算中间变量 b_2 和 ac_4，然后再由一个线程根据已经得出的中间变量 b_2 和 ac_4 算 $\sqrt{\Delta}$，最后两个线程再利用 $\sqrt{\Delta}$ 并行计算 x_1 和 x_2。这样的流水线处理应该可以有不错的性能。

4.2 蒙特卡洛方法求圆周率

- 比较不同线程数量和随机点数量下圆周率估计的准确性和计算速度。
- 讨论增加线程数量是否总能提高计算效率，以及其对圆周率估计精度的影响。
- 提供实验数据图表，展示随着线程数和随机点数的变化，计算效率和精度的趋势。
- 分析实验过程中遇到的问题，如同步问题、负载不均等，并提出相应的解决策略。

4.2.1 随机点数和线程数对计算效率和估算精度的影响

我用 python 读取实验结果只做了下面的图表。图 2 和图 3 反映了不同数量线程下误差随着随机点数的变化的趋势（如果觉得图 2 有点乱可以看图 3）。
可以看到，随着随机点数的增加，使用不同线程数目的程序的结果误差都在下降。但是使用线程数目的增加对于准确率（误差）几乎没有影响。在随机点数超过 60000 之后，使用不同线程数目的程序的结果误差都相差不大，并且误差和线程数目没有相关性。
这说明使用的线程数目对程序的结果的准确性没有太多影响，但是使用的随机点数的数目对程序的结果的准确性有很大影响，随机点数越多，结果越准确。
（效率计算：$T_{serial}/(T_{parallel} * thread\_num)$）
图四反映了使用不同数目的线程的程序效率随着随机点数的变化。可以看到，在使用同等数量的随机点数的情况下，使用的线程增加，效率下降；且使用任何数量的线程的程序，随着随机点数数量的增加，其效率都在提升。
这说明线程数增多，效率下降；且随着随机点数的增加，效率上升，这说明这是一个弱可拓展性的程序。
图五和六反映了使用不同线程在不同随机点数下的加速比变化。可以看到，四线程的程序表现最优秀，这是因为物理核的数目是四核，线程数目超过四只会徒增线程的调度成本，线程数不足四核可能会让 cpu 的核空闲，所以四核的表现是最好的。

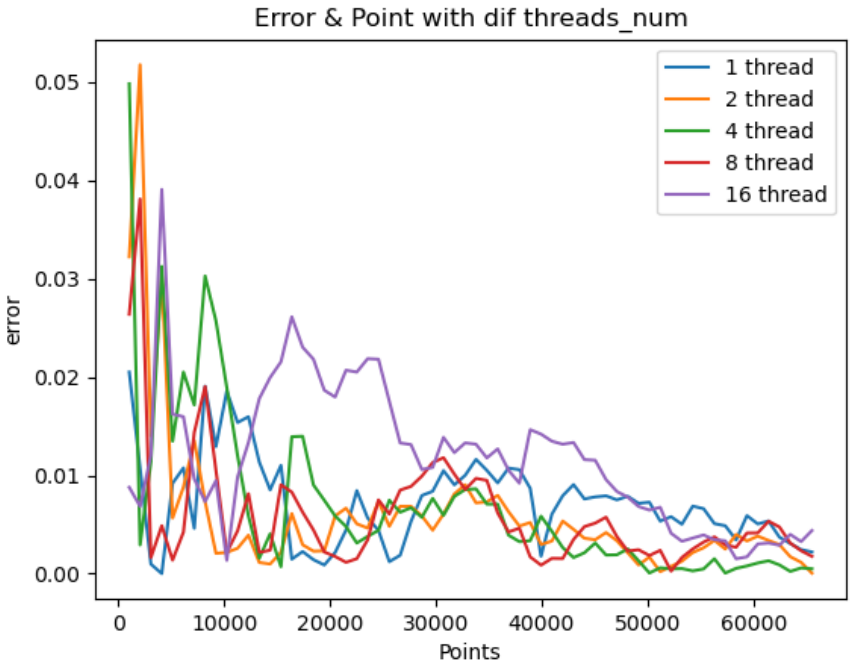

图 2: 误差随随机点数的变化

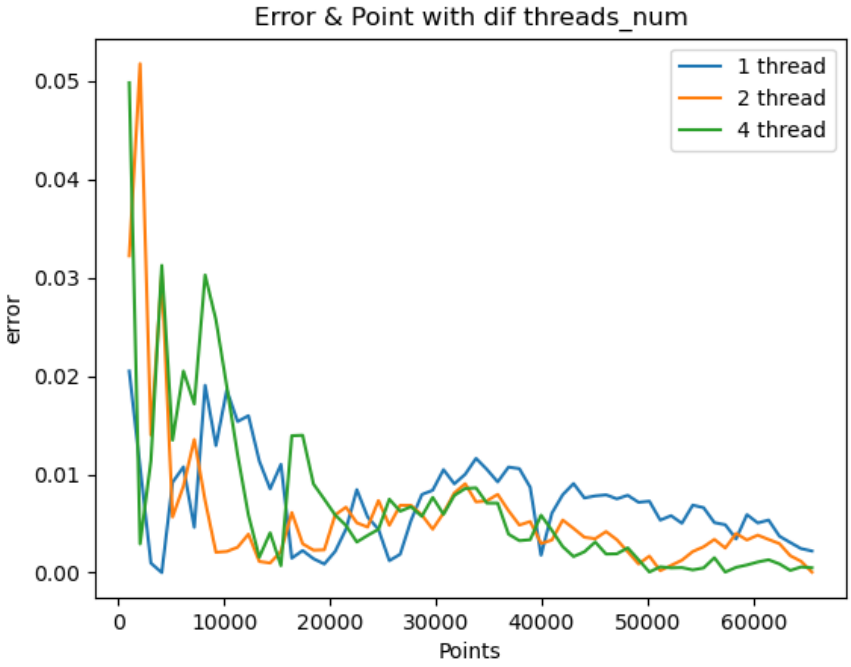

图 3: 误差随随机点数的变化

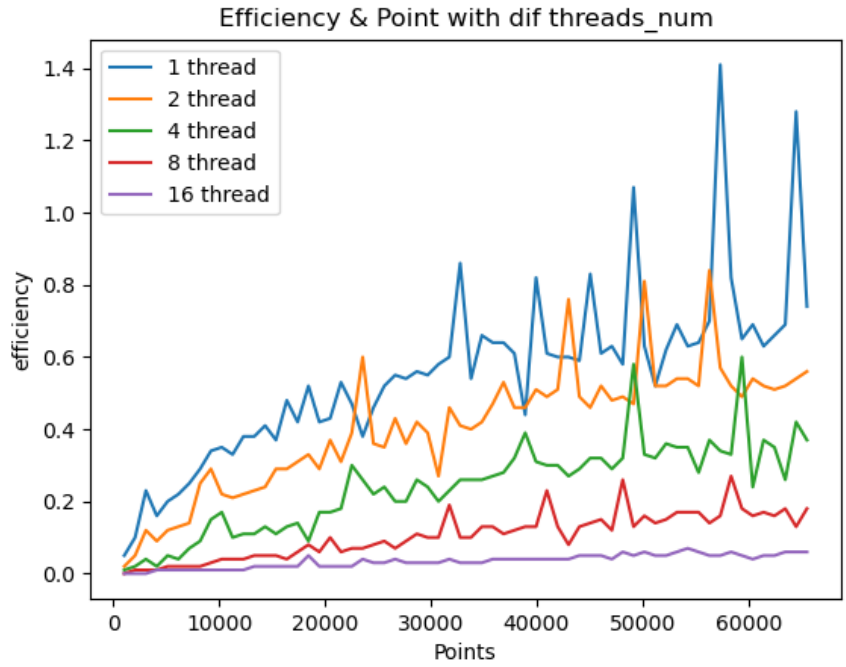

图 4: 效率随随机点数的变化

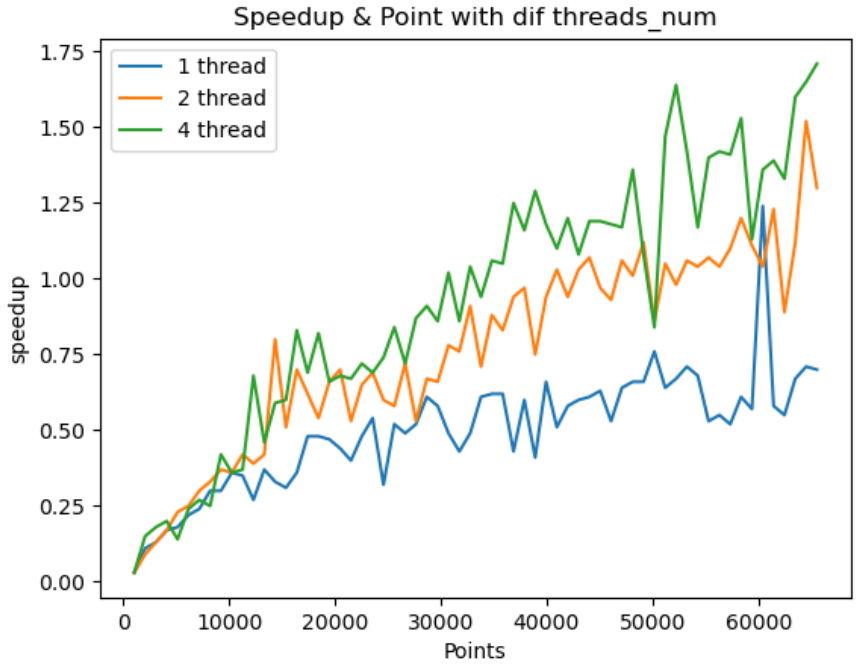

图 5: 效率随随机点数的变化

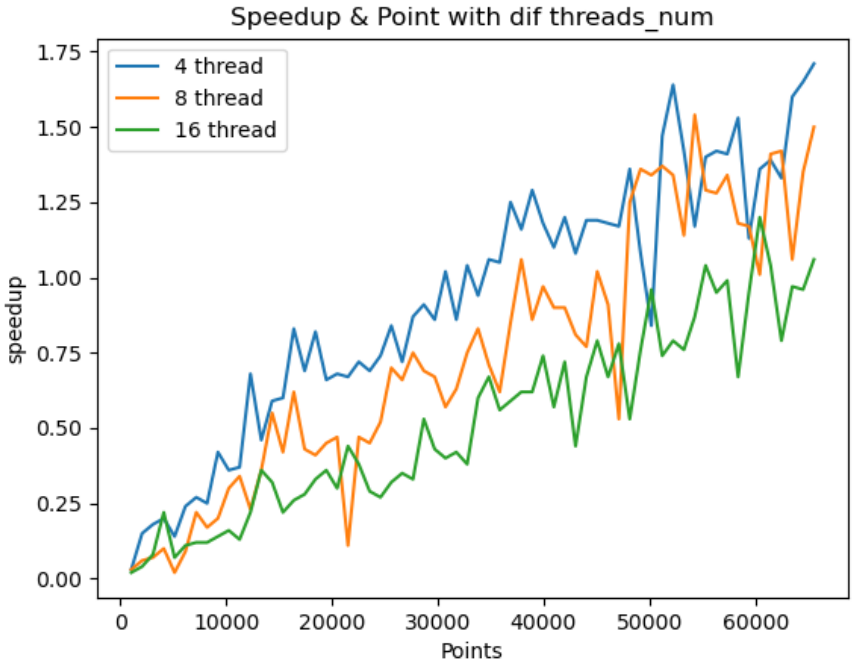

图 6: 效率随随机点数的变化

4.2.2 遇到的问题和解决办法

线程安全的随机数生成器

我一开始使用的是 stdlib 中的 rand() 函数，但是这个函数不是可重入的或线程安全的。即可能会出现竞争条件，该函数不适用于并行计算的场景，在 stackoverflow 中的帖子中提到：不同的线程得到了相同的数字。在我们的计算任务中，这会使得我们的每个线程都在计算相同的随机序列，会降低我们程序的准确性。另一方面，用这个函数会导致我的程序跑的很慢，2 线程版本几乎是 1 线程版本的九倍时间。

于是我换成了线程安全的 drand48_r 。

（参考：stackoverflow；知乎）

伪共享与数据对齐

换成了线程安全的 drand48_r 之后，并行版本的速度确实有所提升，但是仍然不如串行版本。这跟我的线程参数有很大关系：

```
//我的线程参数结构体:
struct ThreadArgs{
    long long iterations;          // 本线程需要计算的迭代次数
    struct drand48_data rand_state;  // 线程独立的随机数状态
};
//线程参数申明
struct ThreadArgs args[num_threads];
```

这样看起来似乎没有问题,但其实有比较大的性能上的隐患。ThreadArgs 中的 rand_state 是 drand48 的种子，同时也记录了随机数生成器的中间状态，也就是说，rand_state 会经常被修改。那么，假设线程 0 的 rand_state 被修改，如果线程 1 的 rand_state 和线程 0 的 rand_state 在同一行 cache line，那么线程 1 的 rand_state 就会被标记失效，如果线程 1 调用随机数生成函数的时候就会 cahce miss。这就造成了伪共享。解决办法就是把线程变量做一个内存对齐，这会保证每个线程参数结构体在不同的 cahce line，就能解决上述问题，得到正常的并行程序的性能。