中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

实验	一元二次方程求解 蒙特卡洛方法求圆周率近似	专业(方向)	计算机科学与技术
学号	--	姓名	--
Email	--	完成日期	2025.4.19

实验目的

1. 一元二次方程求解
   - 使用Pthread编写多线程程序, 求解一元二次方程组的根
   - 求根公式的中间值由不同线程计算, 并使用条件变量识别何时线程完成了所需计算
   - 分析程序并行性能
2. 蒙特卡洛方法求π的近似值
   - 使用Pthread创建多线程, 并行生成正方形内n个随机点
   - 统计落在正方形内切圆点数, 估计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

实验过程和核心代码

一元二次方程求解

使用结构体存储方程的系数以及中间值, 并且作为参数传递给线程:

```
struct EquationData {
    double a, b, c;
    double b_square, ac4, b_square_sub_ac4, sqrt, a2;
    int count1, count2;
};
```

定义两个条件变量和对应的两个互斥锁

```
// 使用cond_1时需要的锁
pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;
// 使用cond_2时需要的锁
pthread_mutex_t mutex_2 = PTHREAD_MUTEX_INITIALIZER;
//
pthread_cond_t cond_1 = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_2 = PTHREAD_COND_INITIALIZER;
```

第一步, 创建两个线程分别计算b^2和$4ac$, 两个线程计算完毕后都会将`EquationData`中的`count1`加一, 并且调用`pthread_cond_signal(&cond_1)`来通知主线程自己完成了计算. 当某一个线程计算完毕时, 主线程被唤醒, 但此时`count1`<2, 不满足循环跳出的条件, 再次进入阻塞状态, 直到晚计算出结果的线程计算完毕, 主线程被再次唤醒, 此时count1 = 2, 跳出循环继续执行

```
// 创建线程
pthread_create(&threads[0], NULL, compute_b_square, static_cast<void*>(&para));
pthread_create(&threads[1], NULL, compute_ac4, static_cast<void*>(&para));

// 等待 b^2 和 4ac 计算完毕
pthread_mutex_lock(&mutex_1);
// 当某一个线程计算完毕时, 主线程被唤醒, 但此时count1<2, 再次进入阻塞状态
// 知道晚计算出结果的线程计算完毕, 主线程被再次唤醒, 此时count1 = 2, 可以继续执行
while(para.count1 < 2) {
    pthread_cond_wait(&cond_1, &mutex_1);
}
pthread_mutex_unlock(&mutex_1);
```

第二步, 主线程根据第一步的结果计算$b^2 - 4ac$是否大于0, 如果小于0说明此方程无实数解, 直接退出. 若大于0, 则继续创建两个线程, 分别计算$\sqrt{b^2 - 4ac}$和$2a$的值. 同上, 两个线程计算完毕后都会将 `EquationData` 中的 `count2` 加一, 并且调用 `pthread_cond_signal(&cond_2)` 来通知主线程自己完成了计算.

```
para.b_square_sub_ac4 = para.b_square - para.ac4;
if(para.b_square_sub_ac4 < 0) {
    printf("No real roots\n");
    return 0;
}
pthread_create(&threads[2], NULL, compute_sqrt, static_cast<void*>(para));
pthread_create(&threads[3], NULL, compute_a2, static_cast<void*>(para));
// 等待 sqrt 和 2a 计算完毕
pthread_mutex_lock(&mutex_2);
while(para.count2 < 2) {
    pthread_cond_wait(&cond_2, &mutex_2);
}
pthread_mutex_unlock(&mutex_2);
```

最后, 主线程根据中间值将两个解计算出来, 并且回收线程, 打印结果, 销毁互斥锁和条件变量.

```
double x1 = (-para.b + para.sqrt) / para.a2;
double x2 = (-para.b - para.sqrt) / para.a2;
// 回收线程
for(int i = 0; i < 4; ++i) {
    pthread_join(threads[i], NULL);
}

printf("x1 = %.4f\nx2 = %.4f\n", x1, x2);
printf("parallel running time: %.4f s\n", elapsed * 1e-9);
pthread_mutex_destroy(&mutex_1);
pthread_mutex_destroy(&mutex_2);
pthread_cond_destroy(&cond_1);
pthread_cond_destroy(&cond_2);
```

为了对比并行求解的性能, 这里对计算串行求解的时间也进行记录

```
// 串行
begin = chrono::high_resolution_clock::now();
para.b_square = para.b * para.b;
para.ac4 = 4 * para.a * para.c;
para.b_square_sub_ac4 = para.b_square - para.ac4;
para.sqrt = sqrt(para.b_square_sub_ac4);
para.a2 = 2 * para.a;
x1 = (-para.b + para.sqrt) / para.a2;
x2 = (-para.b - para.sqrt) / para.a2;
end = chrono::high_resolution_clock::now();
elapsed = chrono::duration_cast<chrono::nanoseconds>(end - begin);
printf("x1 = %.4f\nx2 = %.4f\n", x1, x2);
printf("serial running time: %.4f s\n", elapsed * 1e-9);
```

蒙特卡洛方法求π的近似值

设置一个随机数生成器, 随机返回一个[0-1)内的double值

```
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> distrib(0, 1);

inline double static getADouble() {
    return distrib(gen);
}
```

这里为了避免每次调用都重新生成一个随机数生成器, 直接在全局定义了一个生成器

获取程序运行输入的线程数和采样点数量
创建相应线程数量, 执行蒙特卡洛方法

```
void* MonteCalro(void* args) {
    int* para = (int*) args;
    int cnt = 0;
    double x, y;

    for(int i = 0; i <  *para; ++i) {
        x = getADouble();
        y = getADouble();
        if(pow(pow(x, 2) + pow(y, 2), 0.5) <= 1)
            cnt++;
    }
    pthread_mutex_lock(&lock);
    ans += cnt;
    pthread_mutex_unlock(&lock);
    return NULL;
}
...
int n_each_thread = sampling_num / thread_num;
pthread_t* threads = new pthread_t[thread_num];
// 创建线程
for(int i = 0; i < thread_num; ++i) {
    pthread_create(&threads[i], NULL, MonteCalro, (void*)&n_each_thread);
}
```

这里直接将全局计数设置为全局变量, 使用锁来保证线程之间写ans互斥

计算精度以及运行时间

```
auto begin = chrono::high_resolution_clock::now();
// 创建线程...
// 回收线程...
auto end = chrono::high_resolution_clock::now();
auto elapsed = chrono::duration_cast<chrono::nanoseconds>(end - begin);

ans = ans / sampling_num * 4;

double true_ans = M_PI;
fprintf(fp, "ans: %f\n", ans);
fprintf(fp, "true ans: %f\n", true_ans);
fprintf(fp, "acc: %.2f%\n", (true_ans - abs(ans - true_ans)) / true_ans * 100);
fprintf(fp,"running time: %.4f s\n", elapsed * 1e-9);
```

编写脚本一键执行

```
program_name="montecalro"

./$program_name 1 1024
./$program_name 1 2048
...
...
./$program_name 16 32768
./$program_name 16 65536
```

实验结果

虚拟机处理器个数: 4

一元二次方程求解

某次运行结果如下

```
a: 54.8995, b: -57.0655, c: -68.8298
x1 = 1.7542
x2 = -0.7147
parallel running time: 0.0002 s
x1 = 1.7542
x2 = -0.7147
serial running time: 0.0000 s
```

多次运行后, 发现并行的时间都大致为 `0.0002s` , 而串行的时间都大致为 `0.0000s`

串行计算的运行时间更短. 这是由于方程求解的中间计算都比较简单, 而并行求解需要考虑线程创建的开销, 还有参数传递, 互斥锁和条件变量带来的耗时. 这些开销已经远超并行计算中间值带来的收益, 所以运行速度反而更慢.

蒙特卡洛方法求π的近似值

表格中两行分别表示运行时间, 精度
时间单位: s

线程数	采样点数量						
	1024	2048	4096	8192	16384	32768	65536
1	0.0002 92.76%	0.0005 96.24%	0.0007 96.15%	0.0009 96.10%	0.0017 96.24%	0.0031 95.60%	0.0062 95.54%
2	0.0002 92.63%	0.0003 94.25%	0.0005 96.33%	0.0008 96.25%	0.0016 96.71%	0.0031 96.18%	0.0060 97.32
4	0.0002 93.38%	0.0003 97.05%	0.0004 95.46%	0.0007 94.65%	0.0014 94.39%	0.0027 94.60%	0.0050 94.79%
8	0.0004 94.87%	0.0005 94.87%	0.0005 93.25%	0.0008 94.58%	0.0015 93.53%	0.0028 95.00%	0.0052 93.61%
16	0.0007 92.38%	0.0007 94.62%	0.0006 96.15%	0.0009 94.79%	0.0014 95.22%	0.0028 95.02%	0.0051 95.03%

可以看到在线程数为4时有最好的性能
各线程的平均精度如下:

线程数	**平均精度**
1	95.52
2	95.67
4	94.90
8	94.24
16	94.74

可以看到精度与线程数并没有什么相关性, 总体平均精度都在94%以上

实验感想

本次实验使用多线程来进行一元二次方程的求解和蒙特卡洛方法. 发现当任务非常简单, 计算量非常少时, 使用多线程反而导致程序的运行速度更慢. 而在数据量足够多时才能发挥多线程的优势. 所以在解决问题时不能盲目的使用多线程, 要根据问题的实际情况来进行抉择.