中山大学计算机学院本科生实验报告

课程名称：并行程序设计与算法

实验	**Pthreads并行方程求解及蒙特卡洛**	专业（方向）	计算机科学与技术
学号	21307035	姓名	邓栩瀛
Email	dengxy66@mail2.sysu.edu.cn	完成日期	2024.4.15

1、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，结合数据及任务之间的依赖关系，及实验计时，分析其性能。

一元二次方程：为包含一个未知项，且未知项最高次数为二的整式方程式，常写作$ax^2+bx+c=0$，其中x为未知项，a,b,c为三个常数。

一元二次方程的解：一元二次方程的解可由求根公式给出：

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a} \tag{1}$$

输入：a,b,c三个浮点数，其的取值范围均为[-100, 100]

输出：方程的解x_1,x_2，及求解所消耗的时间t

要求：使用Pthreads编写多线程程序，根据求根公式求解一元二次方程。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，讨论其并行性能。

2. 蒙特卡洛方法求π的近似值

基于Pthreads编写多线程程序，使用蒙特卡洛方法求圆周率求π近似值。

蒙特卡洛方法与圆周率近似：蒙特卡洛方法是一种基于随机采样的数值计算方法，通过模拟随机时间的发生，来解决各类数学、物理和工程上的问题，尤其是直接解析解决困难或无法求解的问题。其基本思想是：当问题的确切解析解难以获得时，可以通过随机采样的方式，生成大量的模拟数据，然后利用这些数据的统计特性来近似求解问题。在计算圆周率π值时，可以随机地将点撒在一个正方形内。当点足够多时，总采样点数量与落在内切圆内采样点数量的比例将趋近于，$\frac{\pi}{4}$可据此来估计π的值。

输入：整数n，取值范围为[1024, 65536]

问题描述：随机生成正方形内的n个采样点，并据此估算π的值。

输出：总点数n，落在内切圆内点数m，估算的π值，及消耗的时间t。

要求：基于Pthreads编写多线程程序，使用蒙特卡洛方法求圆周率π近似值，讨论程序并行性能。

2、实验过程和核心代码

执行命令

```
clang++ -std=c++11 -pthread main.cpp -o main.out
```

核心代码

1.一元二次方程求解

数据结构定义

```
struct ThreadData {
    double a, b, c;
    double x1, x2;
};
```

解一元二次方程函数定义

```
void* quadraticEquation(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    double a = data->a;
    double b = data->b;
    double c = data->c;

    double discriminant = b * b - 4 * a * c;
    if (discriminant >= 0) {
        data->x1 = (-b + sqrt(discriminant)) / (2 * a);
        data->x2 = (-b - sqrt(discriminant)) / (2 * a);
    }

    pthread_mutex_lock(&mutex1);
    threads_completed++;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex1);

    return NULL;
}
```

每个线程都调用quadraticEquation函数来解一元二次方程,共享同一个 data 对象的地址

```
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_create(&threads[i], NULL, quadraticEquation, (void*)&data);
    }
```

使用互斥锁和条件变量实现线程同步机制

```
    pthread_mutex_lock(&mutex1);
    while (threads_completed < NUM_THREADS) {
        pthread_cond_wait(&cond, &mutex1);
    }
    pthread_mutex_unlock(&mutex1);
```

主线程使用pthread_join函数等待所有线程结束，确保所有线程都已经执行完毕，然后解锁互斥锁

```
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], NULL);
    }
```

2.蒙特卡洛方法求π的近似值

数据结构定义

```
struct ThreadData {
    int totalPoints;
    int pointsInsideCircle;
};
```

生成随机点并估计π的函数

```
void *monteCarlo(void *threadarg) {
    ThreadData *data;
    data = (ThreadData *) threadarg;

    int totalPoints = data->totalPoints;
    int pointsInsideCircle = 0;

    unsigned int seed = time(NULL);
    for (int i = 0; i < totalPoints; ++i) {
        double x = (double) rand_r(&seed) / RAND_MAX;
        double y = (double) rand_r(&seed) / RAND_MAX;

        if (x * x + y * y <= 1.0)
            pointsInsideCircle++;
    }

    data->pointsInsideCircle = pointsInsideCircle;
    pthread_exit(NULL);
}
```

创建线程

```
    for (t = 0; t < NUM_THREADS; ++t) {
        threadData[t].totalPoints = totalPoints / NUM_THREADS;
        rc = pthread_create(&threads[t], NULL, monteCarlo, (void *) &threadData[t]);
        if (rc) {
            std::cerr << "Error" << rc << std::endl;
            exit(-1);
        }
    }
```

连接线程

```
    for (t = 0; t < NUM_THREADS; ++t) {
        rc = pthread_join(threads[t], &status);
        if (rc) {
            std::cerr << "Error" << rc << std::endl;
            exit(-1);
        }
    }
```

计算所有线程的圆内的点数之和

```
    int totalInsideCircle = 0;
    for (t = 0; t < NUM_THREADS; ++t) {
        totalInsideCircle += threadData[t].pointsInsideCircle;
    }
```

计算π的近似值

```
    double pi = 4.0 * totalInsideCircle / totalPoints;
```

3、实验结果

1.一元二次方程求解

线程数	1	2	4	6	8	16
时间(s)	0.000243	0.000526	0.000820	0.000450	0.000693	0.001726

实验结果分析：

1. 随着线程数量的增加，运行时间并非总是线性减少的。
2. 在一定范围内，增加线程数量可以加速计算过程，但是增加过多线程会导致性能下降，可能是因为线程创建和管理的开销、线程间的竞争和同步开销等原因。

3. 选择合适数量的线程能够在充分利用多核处理器性能的同时避免过多的线程开销。

2.蒙特卡洛方法求π的近似值

线程/n	1024	2048	4096	16384	65536
1	8.3e-05	0.000121	0.000171	0.000507	0.001917
2	8.7e-05	8.5e-05	0.000137	0.000361	0.001055
4	7.1e-05	9.6e-05	0.000233	0.000391	0.000573
8	0.000192	0.000204	0.000214	0.000263	0.000749
16	0.000442	0.000379	0.000379	0.000478	0.000630

实验结果分析：

1. 随着线程数量的增加，程序的运行时间逐渐减少，因为更多的线程可以并行处理任务，从而提高整体的计算速度。
2. 随着线程数量的增加，运行时间的减少速度逐渐放缓，这表明并行计算的性能提升并不是线性的，增加更多的线程并不一定会线性地减少运行时间。
3. 在某些情况下，增加线程数量并没有带来预期的性能提升，甚至可能导致性能下降，可能是由于线程管理开销增加导致的性能下降。
4. 对于不同规模的问题，最佳的线程数量可能会有所不同。例如，对于小规模的问题，使用更多的线程并不会显著提高性能，而对于大规模的问题，增加线程数量可以带来明显的性能提升。

4、实验感想

1. 这个实验进一步加深了我对Pthreads编程以及并行计算的实践操作，更加深入地理解了并行计算的原理和实现方式。
2. 性能分析对于优化并行程序至关重要，仅仅增加线程数量并不能保证性能的提升，还需要结合实际情况进行分析和调整。
3. 在编写多线程程序时，需要考虑线程的调度和同步机制，以避免竞争条件和死锁等问题。
4. 选择合适的线程数量、优化算法和数据结构，以及避免不必要的同步和通信操作都是提高并行计算性能的重要因素。